Conception et réalisation d'un intergiciel schizophrène pour la mise en oeuvre de systèmes répartis interopérables

Thomas Quinot

▶ **To cite this version:**

Thomas Quinot. Conception et réalisation d'un intergiciel schizophrène pour la mise en oeuvre de systèmes répartis interopérables. Réseaux et télécommunications [cs.NI]. Télécom ParisTech, 2003. Français. NNT : 2003PA066273 . tel-00003307

HAL Id: tel-00003307
https://pastel.hal.science/tel-00003307

Submitted on 5 Sep 2003

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.

École doctorale d'informatique, télécommunications et électronique de Paris

Thèse de doctorat de l'Université Paris VI — Pierre-et-Marie-Curie

Spécialité : Informatique

présentée par

Thomas Quinot

pour obtenir le grade de
Docteur de l'Université Paris VI — Pierre-et-Marie-Curie

Conception et réalisation d'un intergiciel schizophrène pour la mise en œuvre de systèmes répartis interopérables

soutenue publiquement le 24 mars 2003
devant le jury composé de

Christian Queinnec	Professeur à l'Université Paris VI	*président*
Michel Riveill	Professeur à l'Université de Nice	*rapporteur*
Edmond Schonberg	Professeur à l'Université de New York	*rapporteur*
Philippe Waroquiers	Responsable de projet TACT, Eurocontrol	*examinateur*
Fabrice Kordon	Professeur à l'Université Paris VI	*directeur de thèse*
Laurent Pautet	Maître de conférences à l'École nationale supérieure des télécommunications	*directeur de thèse*

Remerciements

Les trois années de recherche dont ce mémoire est l'aboutissement ont été riches d'enseignements, de moments palpitants — parfois aussi chaotiques. Ces travaux n'auraient pu arriver à leur terme sans l'apport de tous ceux qui ont, à un titre ou un autre, contribué à leur succès. Je remercie en particulier :

Michel Riveill, professeur à l'Université de Nice, et Edmond Schonberg, professeur à New York University, pour l'intérêt qu'ils ont porté à mes travaux, et pour l'honneur qu'ils me font d'être rapporteurs de ce mémoire ;

Pierre Cointe, professeur à l'École des mines de Nantes, Christian Queinnec, professeur à l'Université Pierre et Marie Curie, et Philippe Waroquiers, responsable du projet TACT à l'Organisme central de gestion des courants de trafic aérien (CFMU) d'Eurocontrol, pour l'honneur qu'ils me font d'avoir accepté d'être membres de mon jury ;

Fabrice Kordon, professeur à l'Université Pierre et Marie Curie, et Laurent Pautet, maître de conférences à l'École nationale supérieure des télécommunications, pour la confiance qu'ils m'ont accordée en encadrant cette thèse ;

L'ensemble des permanents et des thésards du département Informatique et réseaux de l'École nationale supérieure des télécommunications, pour leur accueil et tous les bons moments partagés ;

Les étudiants de l'École nationale supérieure des télécommunications qui, au cours de projets, de stages et de mémoires, ont contribué au développement de PolyORB ;

Pierre Beyssac, Olivier Hainque et Christian Queinnec, pour leur relecture attentive du manuscrit et leurs remarques judicieuses qui m'ont permis de l'améliorer, tant dans la forme que dans le fond.

Cette thèse n'aurait pu aboutir sans la présence de quelques personnes qui n'ont jamais cessé d'y croire, tant lorsque les travaux progressaient que dans les périodes de doute, et qui m'ont témoigné de leur amitié et de leur soutien. Merci à Samuel Tardieu et Nadine Richard.

Merci tout particulièrement à Muriel Epstein, pour avoir été chaque jour à mes côtés.

Résumé

Le développement d'applications réparties nécessite aujourd'hui l'intégration de composants existants d'origines diverses. Cet assemblage est effectué au moyen d'intergiciels, qui réalisent les abstractions d'un modèle de répartition et permettent aux composants applicatifs d'interagir.

La multiplication des intergiciels de répartition introduit un paradoxe. S'ils permettent de s'affranchir de l'hétérogénéité entre ordinateurs et entre environnements d'exploitation, ils introduisent de nouvelles incompatibilités entre modèles de répartition, ainsi qu'entre différentes mises en œuvre d'un même modèle. Ces incompatibilités résultent de la variété des interfaces qui peuvent être utilisées entre des composants applicatifs situés sur un calculateur et l'intergiciel de ce calculateur d'une part, et entre les intergiciels situés sur différents calculateur d'autre part.

Ces incompatibilités doivent être résolues pour permettre l'intégration de nouveaux composants ou le partage d'informations et de ressources entre des applications utilisant des technologies de répartition différentes. Des logiciels servant de passerelle peuvent être mis en place entre les différents modèles de répartition, mais cette solution est limitée par l'explosion combinatoire induite par l'ajout de nouveaux modèles. Il est donc nécessaire de formuler de nouveaux moyens pour rendre les systèmes répartis interopérables en intégrant cet objectif dans l'architecture des intergiciels.

Des architectures d'intergiciels génériques ont été proposées, qui introduisent la notion de personnalité. Une personnalité est une variante particulière des interfaces offertes par un intergiciel ; un intergiciel générique peut être instancié en fonction d'un modèle de répartition pour réaliser une personnalité ou une autre à partir d'un ensemble d'outils de base invariant.

À partir des architectures d'intergiciels génériques, nous identifions les fonctions récurrentes qu'ils réalisent. Nous proposons de considérer ces fonctions de façon neutre et abstraite, indépendamment du choix d'une personnalité. Il est alors possible de mettre en œuvre ces fonctions sous forme d'une couche neutre de répartition. Cette couche neutre est ensuite dotée de personnalités, qui sont des façades offrant des vues particulières des mêmes fonctions générales.

La séparation entre couche neutre et composants de personnalités permet de

découpler les deux aspects des personnalités : une facette applicative, présentée aux objets de l'utilisateur, et une facette protocolaire, présentée aux autres intergiciels. Ce découplage permet à un composant développé pour un modèle de répartition donné de fournir un service à des composants utilisant d'autres modèles, et réciproquement d'utiliser les services fournis par ces objets. Enfin, la composition de deux personnalités protocolaires permet la mise en place dynamique de passerelles permettant l'interaction entre deux objets tiers ne disposant pas de protocoles communs.

La détermination des fonctions fondamentales de l'intergiciel et leur isolement dans des modules indépendants permet également de rendre celui-ci adaptable à des besoins et à des environnements variés.

Nous définissons ainsi la notion d'architecture d'intergiciel *schizophrène*, architecture permettant la coexistence de plusieurs personnalités dans un même intergiciel, et leur interaction efficace.

À partir de l'architecture que nous avons définie et des besoins que nous avons dégagés, nous proposons une spécification et une mise en œuvre complète d'un intergiciel schizophrène. Notre intergiciel, PolyORB, réalise le cœur neutre prévu par l'architecture schizophrène.

Nous avons également réalisé des personnalités applicatives issues de divers modèles de répartition : CORBA, l'annexe des systèmes répartis d'Ada 95 et une personnalité d'intergiciel de passage de messages (un MOM) basée sur JMS. Enfin, nous avons produit deux personnalités protocolaires : GIOP et SOAP.

La mise en œuvre de ces personnalités a permis de valider la faisabilité de l'architecture, et de confirmer qu'elle permet l'interopérabilité entre modèles de répartition. Un même objet peut utiliser ou fournir des services à d'autres objets réalisés au moyen de modèles différents. Des passerelles sont établies dynamiquement pour mettre en relation des objets tiers lorsque ceux-ci ne disposent pas d'une personnalité protocolaire commune.

Notre intergiciel se caractérise par une forte configurabilité, qui se révèle essentiellement dans le cadre d'applications réparties temps réel. Il offre notamment des profils de fonctionnalités différents en fonction des besoins de l'application et des contraintes de l'environnement. En particulier, trois profils de parallélisme et de synchronisation ont été réalisés : un profil sans tâches ni synchronisation, un profil permettant l'utilisation sans restriction des tâches Ada, et un profil pour les applications temps réel basé sur la spécification Ravenscar. Ce dernier profil n'utilise aucune construction non déterministe du langage.

Nous présentons enfin une analyse des performances de notre prototype, afin d'une part d'évaluer le coût de son architecture générique et les gains apportés par la factorisation du code neutre entre les différentes personnalités, et d'autre part de déterminer l'impact de l'interopérabilité entre personnalités sur les performances d'une application répartie.

Sommaire

1 Introduction générale **19**
1.1 Paradoxe de l'intergiciel 20
1.2 L'interopérabilité, pourquoi ? 20
1.3 L'interopérabilité, comment ? 21
1.3.1 Passerelles entre intergiciels 21
1.3.2 Intergiciels interopérables 22
1.4 Objectifs 22
1.5 Plan du mémoire 23

I Étude et conception **25**

2 Problématique — axes d'investigation **27**
2.1 Systèmes répartis 28
2.1.1 Définition 28
2.1.2 Terminologie 29
2.2 Modèles de répartition et classification 29
2.2.1 Passage explicite de messages 29
2.2.2 Appels de sous-programmes à distance 30
2.2.3 Objets répartis 32
2.2.4 Mémoire partagée répartie 34
2.2.5 Discussion 36
2.2.6 Extensions aux modèles de répartition 38
2.3 Interopérabilité des systèmes répartis 39
2.3.1 Définition 39
2.3.2 Incompatibilités entre modèles de répartition 40
2.3.3 Incompatibilités entre mises en œuvre d'un modèle 40
2.3.4 Synthèse 40
2.4 Intergiciels de répartition 41
2.4.1 Rôle de l'intergiciel 41
2.4.2 Limites introduites par l'intergiciel 43
2.4.3 Architectures génériques d'intergiciel 45

2.4.4 Synthèse 47
2.5 Intergiciels pour les objets répartis 47
2.5.1 Fonctions spécifiques aux objets répartis 48
2.5.2 Relations avec les autres modèles de répartition 49
2.5.3 Synthèse 50
2.6 Synthèse et axes de travail 50

3 État de l'art des intergiciels pour objets répartis 53
3.1 Intergiciels génériques 54
3.1.1 Généricité par liaisons flexibles : Jonathan 54
3.1.2 Généricité par décomposition : Quarterware 56
3.1.3 Discussion 58
3.2 Intergiciels configurables 59
3.2.1 Configuration des propriétés globales de l'application : GLADE 59
3.2.2 Configuration par patrons de conception : TAO 64
3.2.3 Configuration par politiques d'adaptation d'objets : le POA de CORBA 65
3.2.4 Discussion 66
3.3 Intergiciels interopérables 67
3.3.1 Interopérabilité par partage de protocoles : LSA, SoapRMI 67
3.3.2 Interopérabilité par protocoles adaptables : TAO 70
3.3.3 Interopérabilité par génération de passerelles : CIAO . . . 74
3.3.4 Trois perspectives sur l'interopérabilité 78
3.3.5 Discussion 80
3.4 Synthèse 81
3.4.1 Généricité 81

4 Conception d'un intergiciel schizophrène 85
4.1 Intergiciel schizophrène 86
4.1.1 Motivation 86
4.1.2 Approches pour l'interopérabilité 87
4.1.3 Définition de la notion d'intergiciel schizophrène 89
4.2 Application témoin 90
4.2.1 Configuration homogène 91
4.2.2 Configuration d'interopérabilité avec protocole unique . . 92
4.2.3 Configuration d'interopérabilité avec protocoles multiples 92
4.2.4 Configuration d'interopérabilité indirecte 93
4.3 Traits architecturaux d'un intergiciel schizophrène 95
4.3.1 Découplage entre les aspects protocolaires et applicatifs . 95
4.3.2 Introduction d'une couche neutre 96
4.3.3 Architecture générale 97
4.4 Une architecture d'intergiciel schizophrène 98
4.4.1 Vue d'ensemble 98
4.4.2 Composants de support 99
4.4.3 Couche neutre 100
4.4.4 Personnalités applicatives 101
4.4.5 Personnalités protocolaire 102
4.5 Projection des fonctions sur l'architecture 104
4.5.1 Adressage 104

4.5.2 Transport . . . 105
4.5.3 Liaison . . . 106
4.5.4 Représentation . . . 107
4.5.5 Protocole . . . 109
4.5.6 Activation . . . 110
4.5.7 Exécution . . . 111
4.6 Processus de personnalisation . . . 111
4.7 Patrons de conception . . . 113
4.7.1 Patrons comportementaux . . . 113
4.7.2 Patrons de parallélisme . . . 114
4.7.3 Patrons de polymorphisme . . . 116
4.7.4 Synthèse . . . 118
4.8 Synthèse . . . 119

II Réalisation 121

5 Réalisation des services de base 123
5.1 Initialisation . . . 123
5.1.1 Élaboration d'un programme Ada 95 . . . 124
5.1.2 Gestionnaire d'initialisation . . . 125
5.2 Parallélisme et synchronisation . . . 126
5.2.1 Configurabilité du parallélisme . . . 126
5.2.2 Interface des objets primitifs . . . 127
5.2.3 Profils de parallélisme . . . 128
5.2.4 Objets de synchronisation composés . . . 130
5.3 Structures de données de haut niveau . . . 131
5.3.1 Pointeurs à comptage de références . . . 131
5.3.2 Dictionnaires . . . 134
5.4 Patrons de conception . . . 136
5.4.1 Composants . . . 136
5.4.2 Annotations . . . 138
5.5 Synthèse . . . 139

6 Réalisation d'une couche neutre de répartition 141
6.1 Noyau générique . . . 142
6.1.1 Modèle de répartition . . . 143
6.1.2 Cycle de vie d'une requête . . . 145
6.1.3 Références d'objets . . . 147
6.1.4 Ordonnanceur-orienteur . . . 150
6.1.5 Lieur . . . 153
6.2 Composants applicatifs de la couche neutre . . . 154
6.2.1 Interface abstraite des adaptateurs d'objets . . . 155
6.2.2 Adaptateurs d'objets concrets . . . 156
6.3 Composants protocolaires de la couche neutre . . . 157
6.3.1 Tampons de communication . . . 158
6.3.2 Mécanismes de transport . . . 159
6.3.3 Objets de liaison . . . 162
6.4 Synthèse . . . 164

7 Réalisation de personnalités applicatives 165
7.1 Anatomie d'une personnalité applicative . . . 166
7.1.1 Interface pour les clients . . . 167
7.1.2 Interface pour les serveurs . . . 167
7.1.3 Génération de code . . . 168
7.1.4 Descripteurs d'interface . . . 169
7.2 CORBA . . . 170
7.2.1 Compilateur IDL . . . 170
7.2.2 Bibliothèque . . . 171
7.2.3 Services . . . 171
7.3 DSA . . . 173
7.3.1 Génération de code . . . 173
7.3.2 Exécutif . . . 179
7.3.3 Services . . . 179
7.4 MOMA . . . 179
7.4.1 Intergiciels fondés sur le passage de messages . . . 180
7.4.2 Bibliothèque . . . 181
7.5 Synthèse — perspectives . . . 182

8 Réalisation de personnalités protocolaires 185
8.1 Anatomie d'une personnalité protocolaire . . . 185
8.1.1 Interface basse des sessions . . . 187
8.1.2 Interface haute des sessions . . . 187
8.1.3 Fonctions de représentation . . . 187
8.1.4 Usines d'objets de liaison . . . 188
8.1.5 Usines de profils . . . 189
8.2 CORBA . . . 189
8.2.1 Mise en œuvre de GIOP (fonction protocole) . . . 189
8.2.2 Mise en œuvre de CDR (fonction représentation) . . . 190
8.2.3 Usines de liaisons et de profils IIOP . . . 191
8.3 SOAP . . . 191
8.3.1 SOAP RPC (fonction protocole) . . . 191
8.3.2 Représentation SOAP . . . 193
8.3.3 Usine de profils SOAP . . . 193
8.4 Synthèse — perspectives . . . 195

9 Validation et mesures 197
9.1 Fonctionnalités d'intergiciel . . . 198
9.1.1 CORBA . . . 198
9.1.2 DSA . . . 200
9.1.3 MOMA . . . 204
9.2 Contribution à l'interopérabilité . . . 205
9.2.1 Interopérabilité entre mises en œuvre des objets répartis . . . 205
9.2.2 Interopérabilité avec un intergiciel existant . . . 207
9.2.3 Interopérabilité entre modèles de répartition . . . 208
9.3 Évaluation des performances . . . 209
9.3.1 Temps d'exécution . . . 209
9.3.2 Empreinte mémoire . . . 211
9.3.3 Leçons tirées des observations . . . 212
9.4 Génie logiciel et coût de personnalisation . . . 212

9.4.1 Répartition du code 212
9.4.2 Analyse sur des configurations complètes 214
9.5 Synthèse 214

10 Conclusion générale et perspectives **217**
10.1 Interopérabilité : le paradoxe de l'intergiciel 217
10.2 Architecture schizophrène 218
10.3 Réalisations 218
10.4 Perspectives 220

Bibliographie **223**

Table des figures

2.1 Appel de sous-programme 32
2.2 Abstraction dans les applications monolithiques et réparties . . . 37
2.3 Organisation d'une application répartie utilisant un intergiciel . . 42

3.1 Liaisons entre objets 55
3.2 Filtres de communication 62
3.3 Interopérabilité dans LSA 68
3.4 Interopérabilité dans SoapRMI 69
3.5 Composants de TAO pour l'adaptation de protocoles 72
3.6 Interopérabilité avec les protocoles adaptables de TAO 73
3.7 Pontage par composition de protocoles 74
3.8 Flot des données dans l'outil CIAO 76
3.9 Initialisation d'une application CIAO 77

4.1 Intergiciels à personnalités multiples 89
4.2 Objets de l'application témoin 90
4.3 Configuration homogène 91
4.4 Configuration à objets interopérables avec protocole commun . . 92
4.5 Interopérabilité par coexistence de protocoles multiples 92
4.6 Configuration à objets interopérables par passerelle 93
4.7 Traits architecturaux d'une architecture schizophrène 97
4.8 Architecture détaillée d'un intergiciel schizophrène 99
4.9 Fonction *adressage* 105
4.10 Fonction *transport* 106
4.11 Fonction *liaisons* 107
4.12 Fonction *représentation* 108
4.13 Fonction *protocole d'invocation* 109
4.14 Fonction *activation* 110
4.15 Fonction *exécution* 111
4.16 Patron « Component » 117
4.17 Patron « Annotation » 118

5.1 Héritage multiple en Ada 95 138

6.1 Composants génériques de la couche neutre 142
6.2 Cycle de vie d'une requête (rôle client) 146
6.3 Cycle de vie d'une requête (rôle serveur) 146
6.4 Création d'une référence d'objet local 149
6.5 Création d'une référence de passerelle 149
6.6 États des tâches de l'intergiciel 151
6.7 Composants applicatifs de la couche neutre 155
6.8 Composants protocolaires de la couche neutre 158
6.9 Sources d'événements asynchrones 160
6.10 Groupes de surveillance de sources d'événements asynchrones . . 161
6.11 Organisation des piles de protocole 162

7.1 Anatomie d'une personnalité applicative 166
7.2 Phases de compilation dans GNAT 174
7.3 Code généré pour une unité RCI 176
7.4 Code généré pour un type objet réparti 177
7.5 Code généré pour un type pointeur sur sous-programme distant . 178
7.6 Architecture typique d'un système réparti utilisant le passage de messages 181

8.1 Anatomie d'une personnalité protocolaire 186

9.1 Performances de GLADE et de PolyORB dans l'application BBS . 203
9.2 Envoi de message MOMA point à point 204
9.3 Envoi de message MOMA par publication et abonnement 205
9.4 Interopérabilité entre personnalités applicatives 206
9.5 Interopérabilité entre personnalités applicatives 207
9.6 Interopérabilité par passerelle dynamique 208
9.7 Interopérabilité entre modèles de répartition 209

Liste des tableaux

2.1 Besoins correspondants aux différents modèles de répartition . . 37

3.1 Propriétés remarquables des intergiciels existants 81

9.1 Tests fonctionnels CORBA : types de données 198
9.2 Tests fonctionnels CORBA : modes de passage de paramètres et appel unidirectionnel 199
9.3 Tests fonctionnels DSA : ACATS 200
9.4 Temps d'exécution de 10 000 appels de méthodes 210
9.5 Encombrement mémoire d'un serveur 211
9.6 Répartition du code dans PolyORB 213
9.7 Répartition du code dans Jonathan 213
9.8 Répartition de code sur des configurations complètes de PolyORB 214
9.9 Répartition de code sur des configurations complètes de Jonathan 215

Liste des extraits de code

1 Élaboration et dépendance sémantique 125
2 Interface des sections critiques 128
3 Interface des variables conditionnelles 129
4 Exemple d'utilisation de sections critiques et de variables conditionnelles 129
5 Interface des verrous lecteurs/écrivains 132
6 Interface des pointeurs à comptage de références 133
7 Interface des composants 137
8 Interface des annotations 138
9 Interface des usines de profils 189
10 Profil SOAP 194
11 Messages permettant la traduction entre identificateurs d'objets et URIs 195

CHAPITRE

1

Introduction générale

Sommaire

1.1 Paradoxe de l'intergiciel 20
1.2 L'interopérabilité, pourquoi ? 20
1.3 L'interopérabilité, comment ? 21
1.3.1 Passerelles entre intergiciels 21
1.3.2 Intergiciels interopérables 22
1.4 Objectifs 22
1.5 Plan du mémoire 23

A distributed system is one where I cannot get any work done because some machine I have never heard of has crashed.
— Leslie Lamport.

En reliant les ordinateurs, les réseaux de télécommunications permettent la coopération d'unités administratives ou géographiques distinctes ; ils contribuent à améliorer les performances des applications par la mise en commun de puissance de calcul, et permettent le partage des ressources des ordinateurs participants. Ils apportent aux applications un certain niveau de résistance aux défaillances en autorisant la réplication des données et des services.

La répartition introduit cependant de nouvelles contraintes. Elle engendre des interdépendances entre des composants s'exécutant sur des ordinateurs distants : les applications deviennent tributaires du bon fonctionnement des réseaux de télécommunication.

La coordination des différents programmes participant à un système réparti pose également des problèmes algorithmiques particuliers. L'hétérogénéité des architectures matérielles et logicielles au sein d'un système réparti rend nécessaire la mise en œuvre de composants d'adaptation : la répartition est une source de complexité qui s'ajoute aux problèmes propres d'une application.

1.1 Paradoxe de l'intergiciel

Historiquement, il appartenait aux développeurs d'applications de résoudre eux-mêmes les problèmes liés à la répartition : fiabilisation des communications, algorithmes répartis, hétérogénéité des environnements, etc. En particulier, ils devaient programmer directement le matériel de communication de chaque machine, s'assurer de la coordination entre composants, et choisir une représentation de l'information commune acceptée par tous les participants.

Pour faciliter la conception et le développement d'applications réparties, des *modèles de répartition* ont été proposés, c'est-à-dire des ensembles d'abstractions permettant de spécifier les interactions entre composants suivant leur organisation logique et leur déploiement plutôt qu'en termes de moyens techniques utilisés pour transporter l'information de l'un à l'autre. Par exemple, le modèle « objets répartis » décrit les interactions en termes d'appels de méthodes entre des objets, qui seront déployés sur des nœuds différents.

L'utilisation de modèles de répartition, mis en œuvre par des *intergiciels* (*middlewares*), permet donc au développeur de se concentrer sur les aspects fonctionnels de l'application. Elle affranchit le programmeur de certaines contraintes, telles que la gestion de la communication entre nœuds, ou l'hétérogénéité des architectures matérielles et logicielles.

Cependant, la variété des modèles de répartition et des intergiciels introduit un nouveau point d'incompatibilité potentiel. Certaines applications utilisent des intergiciels basés sur le passage de messages (MOM, *Message-Oriented Middlewares*), d'autres les objets distants ou la mémoire partagée répartie, par exemple.

Des incompatibilités existent également, pour un même modèle de répartition, entre les différentes mises en œuvre. Ainsi, dans le cas des objets répartis, les systèmes basés sur la norme CORBA (*Common Object Request Broker Architecture*), l'annexe des systèmes répartis d'Ada 95 (DSA, *Distributed Systems Annex*) et l'interface Java RMI (*Remote Method Invocation*) utilisent des variantes différentes du modèle et des protocoles de communication incompatibles.

Les intergiciels suppriment la dépendance d'une application répartie vis-à-vis d'un environnement matériel et d'un système d'exploitation, mais introduisent en contrepartie un cloisonnement entre modèles de répartition et entre mises en œuvre d'un même modèle. Cette situation constitue le *paradoxe de l'intergiciel* [79].

1.2 L'interopérabilité, pourquoi ?

Pour réutiliser autant que possible des logiciels existants, on souhaiterait que les services rendus par un objet soient disponibles pour des clients basés sur différents modèles de répartition. Un client désirant utiliser un service particulier aurait avantage à pouvoir l'obtenir de tout objet qui peut le lui fournir, quel que soit le modèle de répartition utilisé par chacun.

La question se pose donc des mécanismes nécessaires à la mise en relation d'objets utilisant des modèles de répartition et des intergiciels différents : il devient nécessaire d'établir des liens entre différents intergiciels : on parle de *M2M* (*Middleware to Middleware*) [6].

Supposons, par exemple, que l'on souhaite développer un service en profitant

de la sûreté du langage Ada 95 et de la facilité d'utilisation de DSA tout en bénéficiant de l'ouverture à des clients hétérogènes offerte par CORBA. On cherche alors à faire interagir des composants utilisant les deux environnements (DSA et CORBA), afin de cumuler leurs avantages. Dans le projet CIAO (*CORBA Interface for Ada 95 distributed Objects*), nous avons proposé un mécanisme de génération de passerelles statiques entre applications DSA et clients CORBA.

De même, lorsqu'un service rendu par un objet doit être disponible pour un utilisateur distant, il est nécessaire de réaliser une collaboration entre modèles de répartitions différents. Dans l'application proposée dans [13], une interface utilisateur graphique réalisée en Java utilise l'invocation de méthodes à distance (RMI) pour piloter un objet distant HPC++ fournissant un service de calcul intensif sur une machine parallèle.

Les auteurs proposent diverses solutions permettant aux objets Java de communiquer avec les objets répartis HPC++. Ils retiennent un mécanisme d'interopérabilité basé sur l'utilisation d'une bibliothèque de répartition commune entre RMI et les objets répartis HPC++.

1.3 L'interopérabilité, comment ?

Dans chacun des exemples que nous avons évoqués, des approches différentes sont proposées pour faire interagir des modèles de répartition différents. Nous avons identifié deux démarches principales :

- l'utilisation de passerelles entre modèles de répartition et entre intergiciels ;
- la réalisation d'un intergiciel capable d'offrir simultanément plusieurs modèles de répartition.

1.3.1 Passerelles entre intergiciels

Un objet qui reçoit des interactions au sein d'un modèle de répartition et les prolonge dans un autre constitue une passerelle entre les deux.

L'expérimentation menée dans CIAO nous a permis d'évaluer la faisabilité de la production automatique de passerelles rendant accessible pour des clients CORBA un service défini à l'aide de l'annexe des systèmes répartis d'Ada 95 [87, 88, 89]. Le prototype que nous avons réalisé a mis en évidence certaines limitations :

- CIAO est incapable de traduire certaines constructions évoluées de DSA qui ne se projettent pas naturellement dans le modèle de CORBA ;
- les passerelles générées ne permet l'interaction que dans un seul sens, entre des clients CORBA et des serveurs DSA uniquement. Elle ne permet pas à des clients DSA d'utiliser les services d'un objet CORBA, interdisant en particulier tout mécanismes de rappel (*callback*) entre objets ;
- l'ajout de nouveaux modèles s'accompagne d'une explosion combinatoire du nombre de passerelles nécessaires.

La première limitation est spécifique de notre mise en œuvre. Les deux autres, en revanche, sont intrinsèques à l'utilisation de passerelles. Une passerelle associe à un composant C basé sur un modèle de répartition M_1 une représentation dans un autre modèle de répartition M_2 ; cette représentation permet à des objets du modèle M_2 d'interagir avec C. Pour que C puisse lui-même interagir avec eux, il est nécessaire d'établir une deuxième passerelle,

symétrique de la précédente.

L'ajout de modèles de répartition supplémentaires entraîne une croissance quadratique du nombre de passerelles nécessaires : la combinatoire de cette approche est prohibitive.

1.3.2 Intergiciels interopérables

Une deuxième possibilité consiste à introduire de multiples modèles directement à l'intérieur de l'intergiciel, selon l'une des deux méthodes suivantes :

- l'utilisation d'un protocole fédérateur ;
- la mise en œuvre d'une architecture générique.

Interopérabilité par protocole fédérateur

Cette démarche consiste à intégrer, de manière *ad hoc*, un protocole de communication choisi comme référence commune au sein d'une application associant plusieurs modèles de répartition. Cette solution est proposée par exemple dans [13], où des objets répartis HPC++ et Java RMI coopèrent à travers le protocole spécifique de la bibliothèque de communication Nexus.

Cette approche ne peut cependant s'appliquer à l'interaction avec des objets tiers quelconques, puisqu'elle contraint tous les participants d'une application répartie à convenir d'un protocole unique.

Interopérabilité par architecture générique

Des architectures d'intergiciels génériques ont été proposées, telles que Quarterware [108] ou ACT [28]. Ces projets proposent une conception modulaire des intergiciels, qui permet l'addition ou le remplacement de modules assumant un rôle délimité au sein d'une architecture flexible.

Un même corpus de code générique peut ainsi être spécialisé pour réaliser un modèle de répartition ou un autre. Chaque variante d'un intergiciel générique constitue une *personnalité*.

Cette approche permet de factoriser certaines fonctionnalités des intergiciels entre différentes personnalités. Par ce moyen, il est possible d'assembler efficacement des intergiciels pour offrir aux composants applicatifs plusieurs modèles de répartition simultanément.

Toutefois, les intergiciels génériques existants ne proposent pas d'interopérabilité. En effet, bien qu'ils factorisent une partie de leur comportement dans des modules génériques, ils ne peuvent offrir qu'une personnalité à la fois.

1.4 Objectifs

Constatant que l'utilisation de passerelles statiques ne peut seule lever le paradoxe de l'intergiciel, nous proposons une nouvelle architecture d'intergiciel.

Celle-ci doit être capable de faire cohabiter de multiples personnalités, en les réalisant comme autant de façades autour de fonctionnalités communes réalisées au sein d'une *couche neutre* indépendante du choix d'une personnalité particulière. Les aspects présentés par l'intergiciel y sont découplés :

- l'aspect présenté aux objets applicatifs locaux constitue la *personnalité applicative* ;

– l'interface exposée aux intergiciels distants est une *personnalité protocolaire*.

Les différents modèles sont ainsi fédérés non par le choix arbitraire d'un protocole commun, mais par la couche neutre introduite dans l'architecture.

Nous prolongeons la démarche de mutualisation des intergiciels génériques en autorisant la *coexistence* de composants de multiples personnalités. Notre architecture doit être hautement configurable afin de permettre l'adaptation de ces composants de la couche neutre en fonction des besoins de l'application, et par conséquent de garantir leur réutilisabilité.

1.5 Plan du mémoire

La suite de ce mémoire est organisée en deux parties.

Étude et conception

La première partie est consacrée à l'étude théorique du problème de l'interopérabilité des systèmes d'objets répartis et à la présentation d'une solution à ce problème.

Dans le chapitre 2, nous introduisons la notion de modèle de répartition, et nous motivons la nécessité de permettre leur interopérabilité.

Le chapitre 3 situe cette problématique par rapport aux intergiciels existants. Il dégage les traits pertinents de ces projets par rapport aux besoins que nous avons identifiés.

Le chapitre 4 conclut la première partie et propose une architecture d'intergiciel innovante, qui permet à une même instance d'intergiciel d'offrir simultanément des modèles de répartition différents : l'architecture *schizophrène*.

Réalisation

La seconde partie présente l'intergiciel schizophrène que nous avons réalisé : PolyORB. Nous y motivons nos choix de réalisation, et nous dégageons les aspects innovants de notre prototype.

Le chapitre 5 introduit la bibliothèque d'outils de base et de patrons de conception inédits que nous avons conçue pour faciliter la réalisation de notre intergiciel par assemblage et configuration de composants.

La couche neutre de l'intergiciel réalise les fonctions de répartition de manière générique, indépendamment du choix d'un modèle ou de personnalités de répartition. Sa mise en œuvre fait l'objet du chapitre 6.

Les chapitres 7 et 8 présentent les personnalités protocolaires et applicatives que nous avons produites. Ces personnalités constituent des vues différentes, mais coexistantes, des fonctions réalisées par la couche neutre.

Nous avons effectué des tests fonctionnels et effectué des mesures de performances sur PolyORB. Nous en discutons les résultats dans le chapitre 9.

Le chapitre 10 conclut ce mémoire en rappelant les contributions essentielles de notre travail et les perspectives ouvertes par l'architecture que nous proposons.

Première partie

Étude et conception

CHAPITRE

2 Problématique — axes d'investigation

Sommaire

2.1 Systèmes répartis 28
2.1.1 Définition 28
2.1.2 Terminologie 29
2.2 Modèles de répartition et classification 29
2.2.1 Passage explicite de messages 29
2.2.2 Appels de sous-programmes à distance 30
2.2.3 Objets répartis 32
2.2.4 Mémoire partagée répartie 34
2.2.5 Discussion 36
2.2.6 Extensions aux modèles de répartition 38
2.3 Interopérabilité des systèmes répartis 39
2.3.1 Définition 39
2.3.2 Incompatibilités entre modèles de répartition 40
2.3.3 Incompatibilités entre mises en œuvre d'un modèle . 40
2.3.4 Synthèse 40
2.4 Intergiciels de répartition 41
2.4.1 Rôle de l'intergiciel 41
2.4.2 Limites introduites par l'intergiciel 43
2.4.3 Architectures génériques d'intergiciel 45
2.4.4 Synthèse 47
2.5 Intergiciels pour les objets répartis 47
2.5.1 Fonctions spécifiques aux objets répartis 48
2.5.2 Relations avec les autres modèles de répartition . . . 49
2.5.3 Synthèse 50
2.6 Synthèse et axes de travail 50

Après avoir défini la notion de système réparti et en avoir proposé une classification à gros grain, nous introduisons dans ce chapitre les problèmes théoriques liés à l'interopérabilité des systèmes d'objets répartis. En particulier, nous nous intéressons aux problèmes introduits par l'utilisation d'intergiciels.

Nous proposons enfin des axes d'investigation pertinents pour répondre efficacement à la nécessité d'offrir des systèmes répartis interopérables.

2.1 Systèmes répartis

De nombreuses définitions d'un système réparti ont été proposées. Intuitivement, un système réparti est un ensemble d'ordinateurs autonomes qui collaborent pour atteindre un objectif. Cependant, cette définition est très générale, et recouvre des systèmes informatiques très divers. Elle peut s'appliquer à des systèmes massivement parallèles, fortement couplés, où une batterie d'unités de calcul manipulent simultanément un ensemble de données qu'elles partagent physiquement. À l'inverse, cette définition est également susceptible de recouvrir des réseaux très peu couplés, par exemple l'Internet dans son ensemble.

2.1.1 Définition

Dans cette thèse, nous considérons des systèmes répartis faiblement couplés. Nous affinons la définition donnée en introduction, en nous référant aux critères proposés par [105].

Définition 1 (système réparti) *Un système réparti est défini par la réunion des éléments suivants :*

Plusieurs ordinateurs. *Un système réparti contient plusieurs ordinateurs, se composant chacun d'unités de calcul, de mémoire locale, éventuellement de périphériques de stockage persistant tels que des disques durs, et de voies de communication le reliant avec son environnement.*

Un réseau de communication. *Des voies de communication relient les ordinateurs participant à un système réparti, et leur permettent d'échanger des informations.*

Un état global *Les ordinateurs participant à un système réparti coopèrent pour maintenir un état global. En d'autres termes, le fonctionnement global du système réparti est défini par les contributions ordonnées et cohérentes des participants à cet état global. Cet état global n'est généralement pas accessible à la connaissance ; il est constitué de la somme des états locaux des ordinateurs participants.*

Dans un tel système réparti, chaque ordinateur effectue des traitements et échange des *messages* avec les autres membres du système réparti par le truchement d'un réseau de communication. Les calculs effectués localement, les interactions avec l'environnement et les messages émis et reçus concourent à faire évoluer l'état local, qui est une vue partielle à un instant donné de l'état global du système.

Un système réparti est donc un agencement d'ordinateurs autonomes reliés par un réseau de communication, et formant une unité fonctionnelle. Nous prenons le parti de ne considérer ici que des systèmes faiblement couplés, dans lesquels les ordinateurs participants ne partagent pas leur mémoire physique locale, mais coopèrent uniquement par échange de messages.

2.1.2 Terminologie

Dans la suite de ce document, nous qualifierons de *système réparti* tout système informatique répondant à la définition donnée plus haut.

Définition 2 (nœud) *Nous appellerons* nœuds *chacun des processus (s'exécutant sur une machine réelle ou virtuelle) qui participe à un système réparti. Un nœud possède un espace mémoire local, ainsi qu'une ou plusieurs tâches (fils d'exécution cohabitant dans l'espace mémoire local).*

2.2 Modèles de répartition et classification

Dans tous les systèmes répartis visés par la définition que nous avons donnée, des processus s'exécutant sur des calculateurs distincts coopèrent en échangeant des messages à travers un réseau de communication.

Dans cette section, nous proposons une classification des systèmes répartis en différents *modèles*.

Définition 3 (modèle de répartition) *Un modèle de répartition est un ensemble de définitions d'entités abstraites interagissantes, associé à une projection de ces entités sur les nœuds d'un système réparti.*

Nous identifions également les besoins liés à la mise en œuvre de chaque modèle, en fonction des abstractions et des possibilités d'interactions qu'il offre aux composants applicatifs.

2.2.1 Passage explicite de messages

Le modèle de répartition minimal, c'est-à-dire requérant le moins de fonctionnalités de la part du système d'exploitation, est le passage explicite de messages.

Dans ce modèle, un nœud émet un message à destination d'un autre au moyen de primitives spécialisées du système d'exploitation ou d'une bibliothèque de communication.

Toute action de communication est alors explicite. Le fait qu'une interaction entre deux modules de l'application soit une interaction locale ou un échange de messages est fixé définitivement dès la conception de l'application. Deux composants ne peuvent être affectés sur des calculateurs différents que lorsque l'interface entre ces composants a été prévue initialement pour pouvoir être mise en œuvre sous la forme d'un échange de messages.

Le passage de messages suppose l'existence des fonctionnalités suivantes :

Références L'émetteur doit disposer d'une *référence* qui désigne le destinataire, c'est-à-dire d'un élément d'information qui le nomme sans ambiguïté au sein de l'application répartie. Les mécanismes de communication utilisés doivent donc offrir un moyen d'associer un tel nom à chaque nœud de l'application répartie.

Asynchronisme Le destinataire doit se tenir prêt à recevoir les messages entrants qui lui parviennent de manière asynchrone. Il appartient au programmeur de déterminer dans quelles circonstances un message entrant peut être attendu, et à quel moment des traitements locaux peuvent se poursuivre : l'échange de messages implique une synchronisation entre émetteur et destinataire. De plus, lors d'un échange, ces rôles s'inversent

dès qu'une réponse à un message doit être transmise. Dans ce modèle, les synchronisations apparaissent explicitement dans la structure de l'application, et doivent être prévues par le programmeur.

Modalités évoluées de communication

Lorsque le mécanisme de communication utilisé se limite aux fonctionnalités offertes par un système d'exploitation conçu initialement pour des ordinateurs isolés, les primitives de communication disponibles n'offrent peu ou pas de fonctionnalités évoluées.

Dans les systèmes d'exploitation conçus d'emblée pour la répartition tels que Mach [1], en revanche, une sémantique plus évoluée est proposée : le système peut offrir des mécanismes de garantie de remise des messages (au moins une fois, au plus une fois, exactement une fois), de garantie de remise des messages diffusés vers un groupe de destinataires (proposée par Amoeba [69]), ou de sécurité (contrôle d'accès, authentification, confidentialité, non-répudiation).

De telles modalités évoluées d'échange peuvent également être proposées par des bibliothèques spécialisées telles que MPI (*Message Passing Interface* [68]) ou PVM (*Parallel Virtual Machine* [35]). Celles-ci se placent au-dessus des services de bas niveau du système d'exploitation, et permettent de faire abstraction des mécanismes de communication sous-jacents.

Prise en compte de l'hétérogénéité

S'il souhaite faire coopérer des calculateurs hétérogènes, le programmeur doit définir lui-même une représentation commune des données. Il doit également définir des fonctions de conversion traduisant les données à échanger des représentations natives propres à chaque calculateur vers la représentation commune qui a été choisie, et réciproquement.

L'opération consistant à traduire un message d'une représentation native vers la représentation choisie pour la transmission est appelée *emballage*, l'opération réciproque étant le *déballage* des données.

2.2.2 Appels de sous-programmes à distance

Un développeur d'application peut désirer un service de communication plus structurée que l'échange de messages, afin de résoudre une fois pour toutes les problèmes récurrents de représentation des données et de synchronisation.

Communication client-serveur

Les systèmes répartis basés sur le passage de message ont fait apparaître une modalité de communication récurrente. Le plus souvent, un nœud *client* émet un message à destination d'un autre nœud *serveur* en lui fournissant une information pour lui demander d'effectuer un traitement sur cette information. Le serveur effectue le traitement et renvoie une réponse au client, qui poursuit alors son traitement local.

Ce mécanisme est très similaire à un appel de sous-programme local. En effet, lors d'un tel appel, un composant appelant prépare des données, les communique à un composant appelé et suspend son exécution ; le composant appelé effectue

le traitement demandé et retourne un résultat à l'appelant, qui poursuit alors son exécution.

Il apparaît donc possible d'étendre la notion d'appel de sous-programme au cas où les composants « appelant » et « appelé » résident sur des nœuds différents d'un système réparti. On introduit alors un mécanisme d'appel de sous-programmes à distance, ou RPC (*Remote Procedure Call*) [10].

L'utilisation d'un mécanisme d'appel de sous-programmes à distance suppose préalablement l'identification des sous-programmes de l'application qui sont susceptibles d'être appelés par d'autres nœuds que celui où ils seront exécutés. Comme dans le cas du passage de messages, il doit exister sur chaque client une référence indiquant à quel serveur s'adresser pour obtenir l'exécution d'un service.

Souches et squelettes de sous-programmes

Pour être appelable à distance, un sous-programme doit satisfaire un certain nombre de contraintes. Comme dans le cas du passage de messages, les données manipulées doivent pouvoir être mises sous une représentation commune à tous les nœuds, et doivent conserver leur sémantique après transmission.

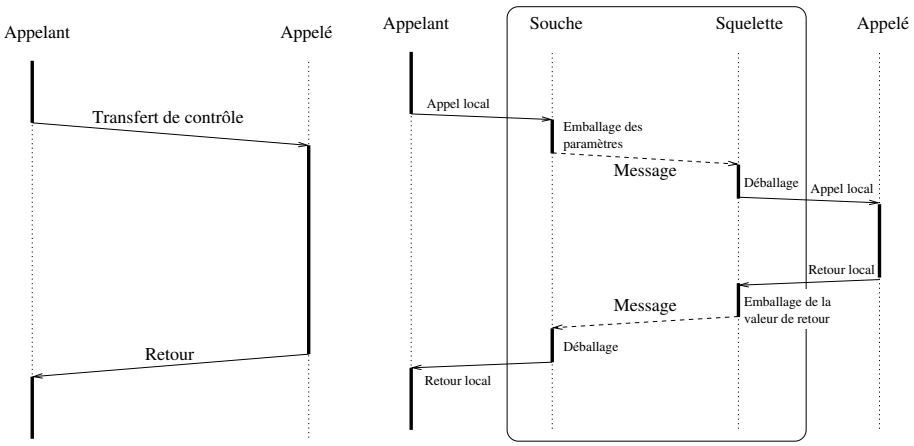
Ainsi, un sous-programme prenant en paramètre un pointeur sur une structure de données ne peut directement être considéré comme appelable à distance. En effet, un pointeur n'a de sens qu'à l'intérieur de l'espace d'adressage d'un processus. Une fois transmis sur un autre nœud que celui sur lequel il a été créé, il perd toute signification et ne peut plus être utilisé.

Les sous-programmes appelables à distance ayant été déterminés, on connaît leur *signature*, c'est-à-dire les types de leurs paramètres et des valeurs retournées. Partant de cette spécification, on peut alors générer automatiquement deux composants logiciels : une *souche* et un *squelette*.

La souche est un module qui présente à l'utilisateur une signature identique à celle du sous-programme appelable à distance. Le corps de ce module remplace le traitement réel du sous-programme par la composition et l'émission d'un message. Ce message est formé des paramètres emballés de l'appel, et de l'indication du sous-programme à exécuter. Il est envoyé au nœud sur lequel ce sous-programme réside, pour en demander l'exécution. La souche se met ensuite en attente d'un message.

Sur ce second nœud, le squelette attend continuellement l'arrivée de tels messages demandant l'exécution de sous-programmes. Lorsqu'un tel message est présenté, le squelette déballe les paramètres, et appelle le corps du sous-programme réel. Lorsque cet appel se termine, il emballe la valeur de retour dans un nouveau message, qui est envoyé au nœud demandeur. La souche récupère et déballe alors la valeur de retour et la retourne à l'appelant, dont l'exécution se poursuit (figure 2.1).

Du point de vue de l'appelant, tout se passe donc exactement comme si le sous-programme appelé était local. De même, du point de vue de la mise en œuvre du sous-programme, rien ne distingue un appel émanant d'un nœud distant d'un appel purement local. Le passage de messages sous-jacent est entièrement masqué par le code généré et les bibliothèques de communication qu'il utilise.



(a) Appel local (b) Appel à distance

FIG. 2.1 – Appel de sous-programme

2.2.3 Objets répartis

La symétrie entre appelants et appelés, brisée par le canevas de communication client-serveur de l'appel de sous-programmes à distance, peut être restaurée en introduisant le modèle « objets répartis », qui étend ce modèle aux applications orientées objets.

Programmation par objets

Dans un programme non réparti, cette dissymétrie entre appelants et appelés peut être résolue en analysant l'application comme un ensemble d'objets communiquant par appel de méthodes. Une telle analyse permet d'organiser un système en fonction des données qu'il manipule et des acteurs qui les manipulent, plutôt qu'en fonction de ses fonctionnalités. Elle a été proposée initialement dans le cadre d'applications de simulation qui devaient représenter l'interaction d'entités autonomes [19].

Encapsulation Un objet est constitué de l'encapsulation d'un *état* et de l'ensemble des sous-programmes, ou *méthodes*, susceptibles de faire évoluer cet état en conservant ses invariants. État et méthodes sont associés sous forme d'une entité indivisible et identifiable.

L'identité d'un objet le désigne sans ambiguïté au sein de l'application : à chaque objet, on peut associer une *référence* permettant d'appeler ses méthodes.

Aiguillage L'ensemble des méthodes d'un objet constitue son *interface* ; toute évolution de son état interne résulte d'un appel sur une méthode : les variables d'état ne sont pas en principe directement accessibles aux utilisateurs.

Des objets différents peuvent offrir la même interface (ils appartiennent alors à une même *classe*). Ils sont alors interchangeables vis-à-vis de leurs

utilisateurs. En ce sens, l'interface constitue le contrat qui définit les interactions possibles entre un objet et les entités qui le manipulent.

La mise en œuvre d'une interface donnée par deux objets différents peut elle-même être différente : chacun réalise un comportement particulier qui respecte la sémantique définie par l'interface. Pour les utilisateurs, cette différence éventuelle est invisible : les objets sont *polymorphes*. Lors de l'appel d'une méthode, un *aiguillage* est effectué, qui consiste à déterminer quelle mise en œuvre de la méthode doit être exécutée, en fonction de l'objet sur lequel la méthode est appelée.

Héritage Étant donnée une classe d'objets, il est possible de l'étendre, en lui ajoutant des variables d'état et des méthodes : il est possible de définir une relation d'*héritage* entre les classes d'objets. Une classe qui hérite des propriétés d'une autre est appelée *classe dérivée*, ou bien classe fille, et tous les objets appartenant à une classe dérivée appartiennent également aux classes ancêtres (puisqu'ils possèdent au moins les mêmes méthodes, les mêmes variables d'état, et que celles-ci exposent la même sémantique).

L'encapsulation des données permet de définir une frontière stricte entre l'interface d'un composant logiciel et sa mise en œuvre. Elle réduit donc l'interdépendance entre les différents composants d'une application, et facilite ainsi leur évolution et leur maintenance.

Le polymorphisme et l'héritage facilitent la réutilisation de composants, l'un en permettant la manipulation générique d'ensembles d'entités partageant un ensemble minimal de propriétés, l'autre en autorisant l'ajout de fonctionnalités à des objets existants, sans modification directe de ceux-ci.

Définition des objets répartis

Dans le modèle de programmation orienté objets, appeler une méthode d'un objet consiste à *envoyer un message* à cet objet [47]. Dans le cas d'une application répartie, un objet est une entité abstraite qui est concrétisée, à un instant donné, par un nœud précis d'un système réparti. On étend donc naturellement le modèle objets aux applications réparties en proposant la notion d'*objet réparti*.

Définition 4 (objet réparti) *Un objet réparti est un objet sur lequel il est possible de déclencher* à distance *(à partir d'un autre nœud du système réparti que celui sur lequel il réside) une des transformations définies sur l'état de l'objet, c'est-à-dire d'invoquer à distance une méthode de l'objet.*

Pour effectuer un appel de méthode à distance, le nœud appelant doit disposer d'une information identifiant l'objet cible de l'appel au sein de l'application.

Définition 5 (référence distante) *Une référence distance désignant un objet dans une application répartie est une information communicable entre nœuds de l'application, qui désigne sans ambiguïté cet objet, et permet d'effectuer des appels de méthodes à distance.*

Appel de méthode à distance

L'exécution d'un appel de méthode sur un objet distant se déroule en deux temps :

1. l'appelant utilise la référence distante désignant l'objet pour déterminer sur quel nœud celui-ci réside ;
2. il peut alors composer et émettre le message représentant l'appel de méthode, comme le ferait une souche cliente dans le cas d'un appel de sous-programme à distance.

En plus de l'identité du sous-programme appelé et des paramètres, le message doit contenir l'identité de l'objet destinataire. L'appelant réalise ainsi un premier aiguillage de l'appel, en transmettant ce message au nœud sur lequel réside l'objet. Le nœud appelé utilise ensuite l'information d'identité de l'objet destinataire pour réaliser un appel local de méthode, comme dans le cas d'un appel non réparti. Il effectue ainsi un second aiguillage de l'appel.

Une mise en œuvre du modèle « objets répartis » comporte donc en principe :

- un moyen d'affecter à un objet une référence distante. Une telle référence doit identifier l'objet au sein de l'application répartie : elle doit permettre de déterminer le nœud sur lequel il réside, ainsi que l'identité de l'objet parmi tous ceux qui résident sur le même nœud ;
- un mécanisme permettant d'exécuter un appel de méthode sur un objet connu par une telle référence, au moyen d'un double aiguillage (d'abord en fonction du nœud destinataire, puis en fonction de la classe de l'objet).

Le second aiguillage suppose que le nœud destinataire connaisse le type exact de l'objet cible. Cependant, le nœud appelant peut aussi avoir besoin de cette information. En effet, il peut utiliser cette information pour déterminer quelle interface est supportée par l'objet désigné, et quelles méthodes il peut appeler.

Les logiciels qui coopèrent au sein d'une application utilisant les objets répartis doivent donc convenir d'un modèle de types global, permettant d'associer à chaque toute référence d'objet la détermination de l'interface offerte par l'objet qu'elle désigne.

Des mécanismes de conversion de types doivent être offerts, pour permettre la conversion d'une référence d'une classe générale en une référence d'une classe dérivée (lorsque l'objet désigné le permet), et vice-versa.

Problème de l'initialisation

La description que nous avons donnée du mécanisme d'appel de méthode à distance suppose que l'appelant connaisse une référence de l'objet cible. Cependant, au démarrage de l'application, chaque nœud ne connaît que les références de ses propres objets (et non celles des objets distants).

Les systèmes répartis basés sur les objets doivent donc comporter un moyen de diffuser initialement ces références. Il peut s'agir simplement d'une représentation textuelle de ces références, qui peut être transmise d'un nœud à l'autre par un moyen externe, tel qu'une intervention manuelle de l'utilisateur. Les références peuvent également être enregistrées dans un registre centralisé, auquel les nœuds accèdent à travers un mécanisme d'appel de sous-programmes à distance.

2.2.4 Mémoire partagée répartie

La répartition peut être introduite dans la sémantique d'un programme initialement non réparti en permettant que les accès à la mémoire deviennent des accès à distance. On réalise alors un système réparti basé sur la *mémoire partagée répartie* (DSM, *Distributed Shared Memory*).

Modèles de cohérence

Dans un tel système, l'état global de l'application n'est plus défini seulement comme la somme des états locaux de ses différents nœuds, mais comme un espace de stockage virtuel accessible à tous ces nœuds. De plus, les nœuds disposent de *copies locales* de cet espace de stockage : les abstractions du modèle (les zones de mémoire partagées) ne sont plus localisées sur un nœud particulier, comme c'est le cas dans les autres modèles que nous avons évoqués, mais dupliquées sur chacun des nœuds qui y accèdent.

Les mécanismes d'échange d'information sont dissimulés par l'exécutif de répartition, et les applications ainsi réparties sont conçues selon un modèle de *partage* d'information, et non plus d'*échange* d'information. Les différents nœuds qui constituent le système réparti exécutent leur traitement de concert en manipulant les données partagées.

Les échanges de messages nécessaires sont déterminés par le choix d'un *modèle de cohérence*, un ensemble de propriétés que doivent vérifier les accès en lecture et en écriture à un objet partagé donné, et aux copies locales de cet objet dont dispose chaque participant.

Un compromis doit être déterminé entre la rigidité des contraintes imposées sur les accès par le modèle de cohérence et le volume des échanges nécessaires entre les intergiciels pour en garantir les propriétés. Les modèles de cohérence les moins contraignants peuvent autoriser des accès incohérents aux données. Ils nécessitent alors peu d'échanges, mais peuvent présenter aux différents nœuds des vues différentes, et incompatibles, de l'évolution de l'état des zones de mémoire partagées. Plus les propriétés souhaitées de compatibilité et de similitude entre ces vues sont restrictives, plus le volume d'échanges et le nombre d'opérations de synchronisation entre nœuds sont importants.

Architectures pour la mémoire partagée répartie

Le traitement parallèle d'informations partagées peut être mis en œuvre au moyen d'une architecture matérielle de partage de mémoire. Dans le cas des systèmes répartis faiblement couplés, auquel nous nous intéressons plus particulièrement, il sera simulé par un exécutif de répartition.

Des échanges de messages sont alors effectués, en fonction des accès à la mémoire réalisés par les programmes s'exécutant sur chaque nœud. La nécessité d'émettre un message lors d'un accès sur un nœud est déterminée par le modèle de cohérence choisi par le concepteur de l'application, en fonction de ses besoins particuliers.

L'envoi des messages peut être provoqué par un composant gestionnaire de mémoire inclus dans l'infrastructure matérielle [59]. Les accès à une section de mémoire partagée entre les nœuds sont alors interceptés par un gestionnaire de mémoire local, qui peut déclencher les actions de communication nécessaires pour propager les écritures et garantir la cohérence des lectures.

La communication peut également être déclenchée par des appels explicites de l'application à des primitives de synchronisation offertes par l'exécutif de répartition [4]. Dans une telle situation, il appartient au programmeur d'application de déterminer les instants auxquels la vision qu'ont les différents nœuds de la section de mémoire partagée doit être cohérente.

2.2.5 Discussion

Un parallèle apparaît entre les niveaux d'abstraction successifs rencontrés dans la programmation des applications non réparties et réparties (figure 2.2).

Dans une application non répartie, le plus faible niveau de structure correspond à des transferts de contrôle (des « sauts ») sans contrainte particulière. L'emploi de la programmation procédurale apporte une structure à ces transferts de contrôle ; la programmation par objets permet enfin d'associer aux données manipulées l'ensemble des traitements valides sur ces données, et rend possible la manipulation d'entités polymorphes.

De la même façon, dans un système réparti, le passage de messages est la modalité de communication la moins structurée. L'appel de sous-programmes à distance introduit une association entre deux messages (une requête et une réponse). Les objets répartis l'étendent naturellement et autorisent une configuration dynamique de l'application (remplacement d'une mise en œuvre par une autre en cours d'exécution, par exemple).

Dans l'appel de sous-programmes à distance, les nœuds composant l'application endossent des rôles asymétriques. Certains nœuds fournissent des sous-programmes appelables à distance et sont serveurs, d'autres y font appel et sont clients. Les objets répartis restaurent la symétrie entre nœuds de l'application : chaque nœud peut contenir des objets et des références distantes, et jouer tour à tour le rôle de client ou de serveur. La communication de références d'objets permet la configuration dynamique des échanges entre nœuds : l'organisation globale de l'application peut évoluer au cours de son exécution.

Les objets répartis permettent de choisir de façon souple la granularité avec laquelle les composants d'une application sont distribués sur différents nœuds : le développeur peut aussi bien choisir d'utiliser des objets répartis à gros grains, offrant un service complet que de nombreux petits objets manipulés par de multiples utilisateurs [57].

L'utilisation d'objets répartis nécessite néanmoins des indirections supplémentaires par rapport au simple appel de sous-programmes à distance : elle a donc un coût plus important. Elle nécessite également la mise en place de mécanismes d'initialisation de l'application permettant la diffusion initiale des références d'objets.

Passage de messages, appels de sous-programmes à distance et objets répartis s'intègrent dans une progression où les échanges de messages sont de plus en plus structurés. Cette progression reflète celle des structures de transfert de contrôle dans la programmation non répartie.

La mémoire partagée répartie, en revanche, offre un mode d'interaction entre composants structuré par les échanges de données, et dans lequel des *copies* des objets manipulés sont conservées sur différents nœuds. Elle offre un modèle de répartition transparent pour le programmeur : celui-ci peut concevoir son application sans se préoccuper de la localité des différents composants. Tout se passe comme si ceux-ci s'exécutaient sur une même machine, et accédaient aux mêmes données.

Cependant, il est délicat de choisir le modèle de cohérence nécessaire à une application donnée : il dépend essentiellement des hypothèses, habituellement implicites, faites par le programmeur au sujet des effets de bord des accès à la mémoire entre les différents composants de l'application. Le choix d'un modèle de cohérence trop restrictif est aussi préjudiciable à l'application, car les fré-

quentes synchronisations des copies locales des informations partagées peuvent avoir un impact important sur les performances globales. Enfin, si l'application peut être écrite sans connaître à l'avance la localité des données, un placement judicieux est essentiel lors du déploiement : ce choix peut avoir un impact très important sur les performances de l'application.

L'utilisation de la mémoire partagée s'apparie avec le calcul parallèle ; les problèmes de cohérence des copies d'objets sont similaires à ceux soulevés par la cohérence des caches dans les systèmes parallèles fortement couplés.

La mémoire partagée ne s'intègre pas de façon naturelle dans la progression des abstractions que nous observons dans les autres modèles de répartition. Nous prenons donc le parti de ne pas la positionner comme un élément central de notre démarche. Nous la conservons néanmoins comme perspective à nos travaux, dans la mesure où la mise en œuvre d'une mémoire partagée répartie sur une architecture faiblement couplée peut être ramenée aux échanges de messages sous-jacents, et réalisée au moyen d'une architecture d'intergiciel.

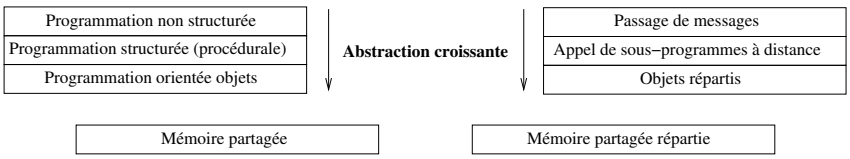


FIG. 2.2 – Abstraction dans les applications monolithiques et réparties

Le tableau 2.1 synthétise les besoins que nous avons identifiés pour chaque modèle de répartition. Ces besoins sont repérés par un signe + ; lorsqu'une fonctionnalité est nécessaire pour la mise en œuvre des extensions citées dans la section 2.2.6, elle est repérée par un *e*.

Besoin \ Modèle	Messages	RPC	Objets répartis	DSM
Transport	+	+	+	+
Références	+	+	+	
Hétérogénéité		+	+	+
Aiguillage messages		+	+	
Aiguillage appels			+	
Asynchronisme	+	*e*	*e*	
Exceptions		*e*	*e*	
Cohérence				+

TAB. 2.1 – Besoins correspondants aux différents modèles de répartition

2.2.6 Extensions aux modèles de répartition

Des fonctionnalités complémentaires peuvent être ajoutées aux différents modèles de répartition présentés dans les sections précédentes (comme nous l'avons indiqué, nous concentrons notre démarche sur les modèles « passage de messages », « appel de sous-programmes à distance » et « objets répartis »).

Exceptions

Un mécanisme d'exceptions peut être utilisé pour signaler à un composant appelant qu'un appel de sous-programme ou de méthode s'est terminé d'une façon anormale. Un message est alors émis à destination de l'appelant signalant cette condition de sortie anormale. Si le langage hôte supporte lui-même un mécanisme d'exception, ce message peut être traduit en une exception native qui sera propagée au composant applicatif appelant.

Avortement

L'utilisateur peut souhaiter annuler avant l'issue normale de son exécution un appel de méthode ou de sous-programme. En particulier, dans le cas de systèmes temps réel, il peut être amené à abandonner une opération en attente s'il détermine qu'attendre le résultat entraînerait la violation d'une échéance.

L'appel de sous-programmes à distance et le modèle objets répartis peuvent donc être enrichis par une primitive d'avortement. Cette primitive permet l'émission d'un message vers le nœud sur lequel le traitement d'un appel est en cours, pour lui demander de l'interrompre.

Comme la levée d'exceptions, l'avortement est lié au motif d'interaction par requête et réponse, et ne s'applique pas au cas du passage de messages.

Appels unidirectionnels

Lorsqu'un appel de sous-programme ou de méthode effectue un traitement, mais ne renvoie par ailleurs aucune information à l'appelant, celui-ci peut désirer poursuivre son traitement local sans attendre la confirmation de fin d'exécution de l'appel.

Le modèle d'appel à distance peut donc offrir la possibilité de marquer certains appels, ou certains sous-programmes, comme unidirectionnels. Dans ce cas, l'appelant continue son traitement dès que le message d'appel est envoyé, et l'appelé n'envoie pas de message de réponse. Si une erreur imprévue se produit et qu'une exception est levée, celle-ci sera perdue.

Un appel unidirectionnel est donc la construction qui, dans une application répartie basée sur l'appel de sous-programmes distants ou les objets répartis, se rapproche le plus d'un simple passage de message.

Transactions

Des ensembles d'échanges (messages, appels de sous-programmes ou de méthodes, actions sur un espace de stockage partagé) peuvent également être groupés au sein de transactions, afin de préserver certaines propriétés de cohérence globale du système. Les propriétés généralement offertes par les mécanismes de transactions sont :

atomicité l'ensemble des opérations constituant une transaction est soit effectué entièrement, soit pas du tout ;

cohérence les invariants du système sont globalement préservés à l'issue d'une transaction (même si des violations ont eu lieu temporairement au cours de son exécution) ;

isolation les effets d'une transaction en cours, non encore terminée, sont invisibles aux composants tiers ;

durabilité les effets d'une transaction terminée sont permanents.

2.3 Interopérabilité des systèmes répartis

Nous avons présenté quatre modèles de répartitions, dont trois (passage de messages, appel de sous-programmes à distance et objets répartis) sont fondés sur l'association d'action de répartition à des constructions de contrôle du flot d'exécution issues de la programmation structurée. Dans cette section, nous recherchons dans quelle mesure l'utilisation de ces modèles de répartition induit des incompatibilités entre composants.

2.3.1 Définition

Les composants d'une application répartie utilisent les fonctionnalités d'un modèle de répartition pour faire abstraction des détails des mécanismes de communication offerts par le réseau. Ils s'affranchissent également par ce moyen des différences entre les environnements matériels et logiciels des divers nœuds : il est possible de faire participer des ordinateurs hétérogènes à la même application répartie.

Celle-ci est donc indépendante vis-à-vis d'un environnement (matériel et système d'exploitation) particulier. En revanche, une nouvelle dépendance est introduite, source d'incompatibilités potentielles :

- deux composants basés sur des modèles différents ne peuvent interagir directement ;
- deux mises en œuvre du même modèle peuvent être incompatibles.

Cependant, la nécessité de faire interagir des composants divers dans une application répartie apparaît comme une problématique de génie logiciel.

- Lors du développement initial d'une application répartie, il est souhaitable, lorsque c'est possible, d'utiliser des modules d'utilité générale déjà existants dans le commerce, ou bien des modules spécifiques développés antérieurement.
- Au cours du cycle de vie de l'application, on peut également être amené à intégrer des nouvelles versions de ces composants « extérieurs », ou à développer des interactions avec d'autres applications qui n'auraient pas été prévues lors de la conception.

Afin de pouvoir réutiliser autant que possible des composants applicatifs, et de permettre leur intégration au sein d'applications diverses, les incompatibilités doivent donc être levées : il est nécessaire de réaliser l'interopérabilité des systèmes répartis.

Définition 6 (interopérabilité) *Deux systèmes répartis sont interopérables lorsque des entités définies dans l'un et l'autre peuvent interagir afin de coopérer à la réalisation d'un but.*

2.3.2 Incompatibilités entre modèles de répartition

Des composants prévus pour fonctionner dans des modèles différents ne peuvent interagir directement. En effet, chaque composant réparti, conçu dans le cadre d'un modèle particulier, est adapté aux abstractions de ce modèle. Celles-ci déterminent la structure des composants, ainsi que les modalités d'interactions qui leur sont accessibles.

Une première difficulté pour faire coopérer des composants issus de deux modèles différents est la nécessité de projeter les abstractions de l'un sur celles de l'autre, correspondant d'une part aux composants applicatifs et d'autre part à leurs interactions.

Plus généralement, lorsqu'un nombre quelconque de modèles doivent interagir, une projection doit être définie entre chaque couple de modèles possibles. L'ajout de nouveaux modèles entraîne donc une explosion combinatoire de la difficulté de cette projection. Il devient impraticable de définir au cas par cas des projections pour toutes les combinaisons possibles ; la nécessité apparaît de définir un modèle plus général, permettant d'unifier les structures de chaque modèle particulier. Une projection de chaque modèle sur le modèle général suffit alors pour faire interagir deux modèles particuliers quelconques.

2.3.3 Incompatibilités entre mises en œuvre d'un modèle

Deux mises en œuvre du même modèle de répartition peuvent également être incompatibles. Même si les interactions possibles entre les objets applicatifs sont fixées par le modèle, les messages utilisés pour les représenter peuvent être différents.

Les extensions au modèle de répartition (telles que celles présentées dans la section 2.2.6) peuvent également être incompatibles.

Si l'utilisation d'un modèle de répartition lève certaines des limitations qui pèsent sur les combinaisons possibles entre composants logiciels (en permettant l'interaction de composants s'exécutant sur des ordinateurs hétérogènes), elle introduit de nouvelles contraintes : non seulement les relations entre les composants doivent être conformes au modèle de répartition, mais de plus tous les nœuds de l'application doivent utiliser la même mise en œuvre de ce modèle.

2.3.4 Synthèse

L'interopérabilité des systèmes répartis apparaît nécessaire pour réutiliser efficacement des composants existants et faciliter leur intégration dans des applications variées. La résolution de cette problématique suppose l'existence d'outils permettant la mise en relation de modèles de répartition hétérogènes. Le déploiement de services mutualisés entre différentes applications fait également naître le besoin de composants répartis supportant simultanément plusieurs modèles de répartition.

Ces outils peuvent être conçus pour permettre l'interaction de deux modèles de répartition fixés. Cependant, cette approche ne permet pas de faire face à la combinatoire induite par l'ajout de modèles supplémentaires. Des solutions plus générales, tendant à unifier les modèles de répartition, doivent être recherchées.

2.4 Intergiciels de répartition

La mise en œuvre d'un système réparti suppose l'existence de logiciels qui fournissent aux composants applicatifs les abstractions d'un modèle de répartition.

Définition 7 (intergiciel) *Un intergiciel (*middleware*) est un logiciel présent sur un nœud d'un système réparti, qui offre aux composants applicatifs présents sur ce nœud les abstractions d'un modèle de répartition [8].*

Pour chacune des classes de systèmes répartis identifiées dans la section 2.2, nous avons identifié les besoins qui doivent être satisfaits par ce logiciel. Dans la section 2.3, nous avons complété ces besoins par ceux liés à l'interopérabilité entre modèles de répartition.

Dans cette section, nous montrons comment les intergiciels peuvent satisfaire ces besoins en remplissant un certain nombre de fonctions. Nous identifions également les limites à l'interopérabilité introduites par l'utilisation d'intergiciels.

2.4.1 Rôle de l'intergiciel

Pour le développeur d'une application utilisant un modèle de répartition de haut niveau (appel de sous-programmes à distance, objets répartis ou mémoire partagée, par exemple), la répartition est introduite de façon « transparente », au sens où elle lui est dissimulée. Les opérations de communication sous-jacentes sont réalisées de façon implicite ; le développeur utilise les mêmes procédés de découpage d'une application en composants indépendants que dans le cas non réparti. Il définit les limites des sous-programmes, les interfaces et les relations des objets de la même façon. La répartition consiste alors à projeter les entités ainsi délimitées sur les différents calculateurs participant à l'application répartie.

Les échanges de messages sous-tendus par le modèle de répartition sont effectués par l'intergiciel. Celui-ci réalise ainsi les abstractions du modèle de répartition. De plus, il réduit la dépendance des composants vis-à-vis de l'environnement d'exécution, en masquant les aspects d'interfaçage de bas niveau avec les primitives de communication.

La figure 2.3 illustre l'organisation générale d'une application répartie utilisant un intergiciel : les composants de l'application utilisent les fonctions de l'intergiciel pour communiquer entre eux de façon transparente ; c'est l'intergiciel qui utilise les services de base du système d'exploitation et l'environnement matériel sous-jacents pour fournir ces fonctions de communication de haut niveau.

Chaque nœud possède donc une instance de l'intergiciel. Ces instances communiquent entre elles au moyen de primitives de bas niveau, et jouent le rôle d'intermédiaires entre les composants de l'application qui résident sur le même nœud et ceux qui résident sur d'autres nœuds. Les interactions avec ces derniers sont réalisées par échange de messages entre les instances d'intergiciel locale et distante.

Adressage

L'intergiciel doit avant tout définir un moyen de nommer les diverses entités qui participent à une application répartie : il leur attribue des identifiants susceptibles d'être échangés et compris par chacun des nœuds de l'application. Ces

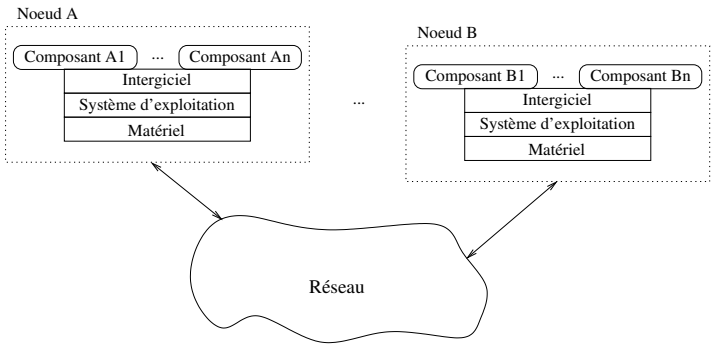


FIG. 2.3 – Organisation d'une application répartie utilisant un intergiciel

identifiants permettent à un composant d'en désigner un autre, et ainsi d'interagir avec lui. L'intergiciel remplit une fonction *d'adressage* des composants de l'application.

Cette fonction est rendue accessible aux composants applicatifs à travers des constructions de programmation qui permettent d'associer certains objets concrets du programme utilisateur à des abstractions d'un modèle de répartition. Le choix d'un modèle particulier conditionne donc la forme de ces objets de l'application.

Transport

À partir des identifiants d'objets, une instance d'intergiciel établit des canaux de communication avec d'autres. Ces canaux sont utilisés pour échanger les messages représentant les interactions entre composants applicatifs. À cet égard, l'intergiciel assure le *transport* de messages entre les nœuds, en utilisant le réseau de communication sous-jacent.

Protocole et représentation

L'intergiciel doit mettre en œuvre un protocole. Il se charge également de mettre les données à échanger sous une forme transmissible, et compréhensible par les nœuds distants.

Protocole et représentation définissent une syntaxe et une sémantique des messages reflétant les abstractions d'un modèle de répartition. En outre, le protocole peut être spécifié en fonction du seul modèle de répartition, indépendemment de sa projection sur un langage de programmation ou une architecture matérielle particuliers.

L'utilisation d'un intergiciel permet donc de faire interagir, au sein d'une même application répartie, des composants s'exécutant sur des architectures matérielles et dans des environnements logiciels différents. En contrepartie, le choix d'un protocole et d'une représentation limite les interactions possibles aux composants dont l'intergiciel supporte ce protocole.

Liaisons

Un objet qui dispose d'une référence désignant un autre objet, à distance, ne peut directement en appeler les méthodes. Il est nécessaire, au préalable, que l'intergiciel utilise les informations contenues dans la référence pour établir une connexion de transport et une pile de protocole de nature à permettre l'échange de messages avec l'intergiciel supportant l'objet distant. Les ressources nécessaires doivent également être mobilisées.

La fonction de liaison consiste à mettre en place ces composants et ressources, et à les associer à la référence d'objet, afin de rendre possible l'appel de méthodes sur l'objet distant. Elle détermine les associations possibles entre protocoles et mécanismes de transport disponibles.

L'interopération directe entre deux nœuds n'est possible que si leurs intergiciels respectifs sont capables d'utiliser le même protocole sur le même réseau de transport.

Activation

Lorsqu'un message est adressé à un intergiciel et comporte l'identifiant d'un composant applicatif associé à cet intergiciel, ce dernier doit faire en sorte que la structure concrète qui réalise ce composant (et qui est projetée sur une entité abstraite du modèle de répartition) existe bien, et soit correctement associé à cet identifiant. Nous désignons cette fonction de contrôle du cycle de vie des associations entre identifiants et composants applications par le terme *d'activation*.

Exécution

L'intergiciel doit enfin diriger les messages vers le contexte et les ressources *d'exécution* (tâches, composant applicatif concret) auxquels ils sont destinés, et synchroniser les échanges et les traitements.

Ainsi, un composant qui demande à un autre d'exécuter un service peut attendre la fin de l'exécution de ce service (matérialisée par la réception d'un message). Il se comporte alors de manière *synchrone*. Il peut également poursuivre son propre traitement pendant que le second composant traite sa demande. Il s'agit alors d'un échange *asynchrone*.

2.4.2 Limites introduites par l'intergiciel

L'utilisation d'un intergiciel dans une application répartie, par opposition à l'utilisation directe des primitives de communication fournies par le système d'exploitation, permet l'utilisation d'un modèle de répartition sophistiqué. Ce gain en abstraction facilite également la mise en place d'applications hétérogènes. Toutefois, les apports des intergiciels s'accompagnent de plusieurs contreparties.

Coûts relatifs des différents modèles de répartition

À chaque modèle de répartition correspond un coût de mise en œuvre. Plus les fonctionnalités offertes sont importantes, plus l'intergiciel est volumineux, et plus les ressources nécessaires sont considérables.

Un intergiciel offrant le modèle « passage de messages » peut n'introduire qu'un surcoût minime par rapport à l'utilisation directe des primitives de communication du système d'exploitation sous-jacent. En effet, il rend un service d'un faible niveau d'abstraction, dont la réalisation transparente ne nécessite pas d'opérations complexes.

Lorsqu'un modèle plus évolué (l'appel de sous-programmes à distance ou les objets répartis) est mis en œuvre, en revanche, l'intergiciel effectue des traitements complexes. Il occupe donc plus d'espace en mémoire. Les fonctions avancées sont également susceptibles d'introduire des délais de traitement supplémentaires pour tous les messages échangés, et d'augmenter le volume de données à transmettre entre les nœuds.

Adaptation aux besoins des applications

D'une application à une autre, les besoins peuvent être très divers. Une application de télévision, par exemple, nécessite la transmission de flux multimedia en temps réel vers des terminaux embarqués. Elle nécessite donc une transmission de messages avec une faible variance de délai. En revanche, elle peut souffrir la non-remise de quelques messages, par exemple au prix d'une baisse transitoire de qualité d'image. Une application réalisant des transactions bancaires, au contraire, pourra supporter des délais plus importants. En revanche, elle imposera une fiabilité importante dans la remise des messages.

Par ailleurs, au sein d'une même application, les fonctionnalités requises et les ressources disponibles (puissance de calcul, mémoire, capacité des liaisons de communication) peuvent varier d'un nœud à l'autre dans des proportions importantes. Un nœud pourra être une station de travail puissante, alors qu'un autre sera un système embarqué, un robot mobile autonome, ou une carte à microcircuit.

Les besoins globaux de l'application, ainsi que les caractéristiques particulières de chaque nœud, nécessitent un intergiciel adapté à chaque cas. Or, les intergiciels traditionnels sont développés en tant que produits d'usage général, afin d'être utilisés dans un grand nombre d'applications. Ils sont destinés à éviter le redéveloppement de modules de communication pour chaque projet. L'utilisation du même intergiciel sur tous les nœuds d'une application représente donc un compromis, et ne peut satisfaire idéalement l'ensemble des contraintes de tous les composants répartis.

Paradoxe de l'intergiciel

Les intergiciels traditionnels sont liés au choix d'un modèle de répartition en particulier. Deux mises en œuvre du même modèle ne sont pas non plus nécessairement compatibles. Cette contrainte tendant à imposer le choix d'un seul modèle de répartition, si ce n'est un seul intergiciel, pour l'ensemble d'une application, s'oppose à la nécessité d'adapter l'intergiciel à chaque nœud.

Comme nous l'avons indiqué dans la section 2.3, si l'utilisation d'un intergiciel permet au développeur d'application de s'affranchir de la dépendance vis-à-vis d'une architecture matérielle et d'un système d'exploitation particuliers, le choix d'une *personnalité d'intergiciel* donnée est la source de nouvelles incompatibilités.

Définition 8 (personnalité) *Une personnalité d'intergiciel est un ensemble de vues sur les fonctions assurées par l'intergiciel. Elle associe :*

- *un modèle de répartition ;*
- *une interface exposée aux composants d'application ;*
- *une interface exposée aux autres intergiciels.*

L'interface exposée aux composants d'application est une interface de programmation permettant de manipuler les abstractions du modèle de répartition.

L'interface exposée aux autres intergiciels est constituée de protocoles permettant de mettre en œuvre ces abstractions.

L'introduction de nouvelles contraintes sur les interactions entre composants consécutive à l'utilisation d'intergiciels constitue le *paradoxe de l'intergiciel* identifié dans [80].

Ces nouvelles contraintes posent un véritable problème de génie logiciel. Elles font naître le besoin d'interconnecter les intergiciels par des *passerelles* réalisant l'adaptation entre modèles de répartition. Celles-ci fondent la notion de communication entre intergiciels, ou « M2M » *(Middleware to Middleware)* [6].

Une telle « passerelle » peut être statique, liée à un couple de modèles fixé *à l'avance*, une fois pour toutes. Une telle approche est cependant limitée par l'explosion combinatoire qu'elle associe à l'augmentation du nombre de modèles supportés.

Les passerelles entre modèles peuvent également être dynamiques, réalisant l'adaptation en fonction des besoins, au moyen d'une bibliothèque permettant la mise en œuvre de différents modèles de répartition. Cette approche est plus complexe que la première, dans la mesure où elle exige des fonctionnalités particulières dédiées à l'interopérabilité au sein de la mise en œuvre des modèles de répartition.

En revanche, elle permet de faciliter l'interaction de composants basés sur des modèles différents, en limitant la lourdeur liée au nombre de modèles impliqués.

2.4.3 Architectures génériques d'intergiciel

Un unique intergiciel traditionnel, au comportement et aux fonctionnalités fixés une fois pour toutes, ne peut répondre à la diversité des besoins des applications, et à la diversité des ressources disponibles sur les différents nœuds. Cependant, il est possible de définir des intergiciels capables d'adapter leur comportement et leurs fonctionnalités à des besoins spécifiques.

Intergiciels flexibles

L'adaptation de l'intergiciel aux besoins peut être réalisée selon deux axes.

En premier lieu, elle peut être *paramétrable* : le dimensionnement de ses structures internes, par exemple, peut alors être ajusté en fonction de la taille de l'application, et des capacités de chaque nœud.

D'autre part, une conception modulaire et configurable permet de n'instancier sur un nœud que celles de ses fonctionnalités qui y sont nécessaires. Les fonctionnalités offertes (et donc le coût de la répartition) sont ainsi adaptés en fonction des ressources disponibles sur chaque nœud et des exigences de l'application.

Cette idée d'assembler les systèmes logiciels comme les ordinateurs, à partir de composants préexistants et réutilisables, date des origines du génie logiciel [63]. Son application au domaine des intergiciels a fait l'objet de plusieurs études. En particulier, les auteurs de Quarterware la systématisent, et proposent une analogie entre des intergiciels hautement modulaires et les architectures matérielles RISC, où les instructions machine offertes par le matériel sont des unités de fonctionnalité minimales, que l'utilisateur agence au plus juste pour répondre à ses besoins [108].

Dans le cas des intergiciels, une telle flexibilité rend possible la réutilisation d'un même intergiciel dans des contextes variés. La configuration permet la prise en compte des contraintes spécifiques de l'application, et de n'intégrer que les modules essentiels. Dans les cas où les ressources disponibles sont réduites, seules les fonctionnalités de base seront instanciées, à l'exclusion d'autres plus avancées. De tels *profils de fonctionnalité réduits* permettent la mise en place d'application réparties intégrant des systèmes embarqués.

Définition 9 (profil de fonctionnalités) *Un profil de fonctionnalités d'un intergiciel est un sous-ensemble des fonctionnalités qui peut être instancié sur un nœud, afin de limiter la consommation de ressources en excluant celles qui n'y sont pas nécessaires.*

Réutilisation de code au sein de l'intergiciel

Le développement d'intergiciels flexibles, supportant de multiples protocoles et des interfaces de programmation variées pour pouvoir être associés à des composants préexistants, pourrait être réalisé en agrégeant, au sein d'un même logiciel, des intergiciels complets préexistants.

Une telle approche n'est cependant pas souhaitable. En effet, elle conduit à de nombreuses duplications de fonctionnalités, qui alourdissent le code et en rendent la maintenance difficile. De plus, elle peut introduire des « collisions » de fonctionnalité, lorsque deux modules séparés sont conçus l'un et l'autre pour prendre en charge un même aspect du comportement de l'intergiciel de façon exclusive.

La mise en œuvre rationnelle de tels intergiciels nécessite donc la définition d'une architecture basée sur des modules réutilisés entre les différentes personnalités supportées. Ces architectures dites *génériques* ont déjà été étudiées, par exemple dans les projets Jonathan [24] et Quarterware [108], que nous présentons plus en détail dans la section 3.1.

Les intergiciels génériques sont fondés sur l'identification de fonctionnalités récurrentes, qui apparaissent dans plusieurs personnalités. Ces fonctionnalités sont alors factorisées sous forme de modules génériques (c'est-à-dire indépendant d'un modèle de répartition ou d'un protocole particulier), qui sont réutilisés.

La factorisation des modules génériques permet de limiter l'explosion de la taille de l'intergiciel au fur et à mesure de l'ajout de nouvelles personnalités. Cependant, il ne s'agit pas seulement d'une optimisation quantitative : la mise en commun de sous-systèmes de l'intergiciel est aussi un facteur favorisant l'interopérabilité entre ces différentes personnalités.

2.4.4 Synthèse

Tout intergiciel satisfait les besoins des applications réparties en mettant en œuvre un certain nombre de fonctions élémentaires : adressage, liaisons, transport, protocole, représentation, activation et exécution.

L'utilisation d'intergiciels introduit des limitations pour les applications réparties. Ces limitations sont de trois ordres :

- les divers modèles de répartition ont des coûts relatifs, qui sont fonction de la richesse des fonctionnalités offertes ;
- les intergiciels sont des produits d'usage général, qui ne sont pas nécessairement adaptés aux besoins précis d'une application particulière.

Ces limitations rendent nécessaire le choix d'un intergiciel adapté à chaque situation.

Cependant, l'utilisation d'intergiciels met en lumière un véritable problème de génie logiciel : le « paradoxe de l'intergiciel ». Si elle permet de s'affranchir de l'hétérogénéité des architectures matérielles et des systèmes d'exploitation, les incompatibilités entre modèles de répartition et entre intergiciels font apparaître de nouvelles contraintes sur les possibilités d'interactions.

Il apparaît donc nécessaire de favoriser l'interopération, au sein d'une même application, entre des nœuds utilisant des modèles de répartition et des intergiciels différents. Cette propriété d'interopérabilité ne peut être réalisée au moyen d'intergiciels traditionnels sans souffrir d'une explosion combinatoire rédhibitoire.

Les intergiciels génériques et configurables permettent d'adapter le comportement des intergiciels à des besoins variés. Ils apportent donc des éléments de solution pour faire correspondre les fonctionnalités offertes et les exigences de l'intergiciel vis-à-vis des besoins et des capacités d'un nœud particulier d'une application.

Les propriétés de généricité et de configurabilité permettent d'identifier les fonctions communes à plusieurs personnalités d'intergiciels, et de les mettre en œuvre d'une façon mutualisée. Leur exploitation rend donc possible la réalisation efficace de l'interopérabilité, et contribue à lever l'obstacle de la combinatoire.

En définissant une architecture efficace d'intergiciel interopérable tirant parti d'une conception générique et configurable, nous entendons contribuer à la résolution des problèmes auxquels fait face l'industrie des applications réparties lorsqu'elle utilise des intergiciels, notamment ceux liés à l'interopérabilité entre modèles de répartition et entre mises en œuvre différentes d'un même modèle.

2.5 Intergiciels pour les objets répartis

Dans cette section, nous nous intéressons en particulier aux intergiciels mettant en œuvre le modèle « objets répartis ». Nous identifions les besoins spécifiques de ce modèle et les réponses qui y sont apportées. Nous montrons également qu'un intergiciel répondant aux besoins du modèle « objets répartis » est capable de répondre également à ceux des modèles moins évolués : passage de messages et appel de sous-programmes à distance.

2.5.1 Fonctions spécifiques aux objets répartis

Références d'objets distants

Dans ce modèle, une fois l'application répartie analysée en termes d'objets, ces derniers sont projetés sur les nœuds. Chaque nœud exécute un programme qui fait exister les objets qui lui sont assignés, ainsi qu'une instance d'intergiciel qui permet à ces objets d'être la cible d'appels de méthodes à distance. L'intergiciel crée des *références* désignant les objets qu'il héberge. Celles-ci sont des informations communicables à tous les nœuds de l'application, et permettant de nommer sans ambiguïté chaque objet.

Tout composant distant qui acquiert une référence désignant un objet peut procéder à un appel de méthode sur cet objet. Pour cela, la référence doit pouvoir être convertie en un identifiant de l'objet au sein de son nœud, associé à une adresse au sein du réseau de communication. Cette adresse permet à l'intergiciel réalisant l'appel de méthode pour le compte de l'objet appelant de contacter l'intergiciel du nœud sur lequel réside l'objet appelé.

Diffusion des références d'objets

Comme nous l'avons indiqué dans la section 2.2.3, dans une application utilisant des objets répartis, l'intergiciel doit fournir un mécanisme permettant l'échange initial de références d'objets.

Ce mécanisme peut se limiter à l'existence d'une représentation externe des références, qui permette à l'utilisateur de l'application de transporter celles-ci d'un nœud à l'autre. Dans ce cas, l'intergiciel ne distribue pas lui-même les références initiales : c'est l'utilisateur qui met directement en relation les divers objets de l'application, en intervenant manuellement pour les désigner les uns aux autres.

Un service de dénomination prévu à cet effet peut également être offert par l'intergiciel. Dans ce cas, les objets de l'application sont associés à leur référence dans un répertoire global accessible depuis l'ensemble des nœuds. La mise en place de cet accès suppose cependant elle-même la diffusion d'une référence initiale (désignant le répertoire partagé), et nécessite *in fine* une intervention de l'utilisateur.

La diffusion manuelle des références d'objets est simple à mettre en œuvre si l'on dispose d'une représentation externe des références d'objets, mais elle nécessite une intervention de l'utilisateur pour chaque objet qui doit être initialement connu de tous les nœuds de l'application.

L'utilisation d'un service de dénomination auquel les nœuds accèdent (au moyen d'appels de sous-programmes à distance ou d'appels de méthodes sur un objet dont la référence est distribuée manuellement) ajoute un niveau d'indirection, et rend plus complexe le déploiement de l'application.

Elle limite en revanche l'intervention de l'utilisateur à la mise en place d'une référence vers un seul objet, et permet la reconfiguration de l'application par déplacement d'objets d'un nœud vers un autre. Il suffit, dans ce cas, de mettre à jour la référence de l'objet en question dans le seul répertoire centralisé. Elle est donc utile pour des applications complexes et de grande taille, faisant intervenir de nombreux objets.

2.5.2 Relations avec les autres modèles de répartition

Dans la suite de notre travail, nous nous intéressons spécifiquement aux intergiciels de répartition réalisant le modèle « objets répartis ». Dans cette section, nous détaillons les raisons de ce positionnement.

Expressivité du modèle « objets répartis »

Les abstractions qu'il définit sont plus évoluées que celles des modèles « passage de messages » et « appel de sous-programmes à distance ». Ces deux modèles peuvent être interprétés comme des cas particuliers d'objets répartis.

Dans le cas de l'envoi de messages, on peut considérer chaque composant participant à l'application comme un objet de type « boîte aux lettres » dont l'interface comporte une seule méthode appelable à distance : « recevoir un message ». Pour l'appel de sous-programmes à distance, on peut considérer que chaque nœud correspond à exactement un objet, dont la durée de vie est égale à celle du nœud, et dont la signature des méthodes correspond aux sous-programmes exportés par le nœud.

Le modèle « objets répartis » capture l'ensemble des abstractions des modèles « passage de messages » et « appel de sous-programmes à distance ». Il est donc possible de projeter les abstractions de ces modèles sur les objets répartis. Un intergiciel pour objets répartis muni d'une telle projection est donc en mesure d'interagir avec des intergiciels les mettant en œuvre. Il facilite ainsi l'interaction entre modèles. Cette faculté est obtenue au prix de l'introduction de la complexité et des indirections du modèle « objets répartis » dans les modèles de moindre abstraction.

Étendue des services de répartition

Pour chacune des classes de systèmes répartis identifiées dans la section 2.2, nous avons mis en évidence un certain nombre de besoins qui doivent être satisfaits par les intergiciels qui les mettent en œuvre. Nous avons résumé ces besoins dans le tableau 2.1.

Nous constatons que le modèle « objets répartis », qui propose les abstractions les plus évoluées, est aussi celui qui nécessite le plus de services de la part de l'intergiciel. Plus précisément, il apparaît que lorsque les besoins du modèle « objets répartis » sont satisfaits, alors ceux du modèle « appel de sous-programmes à distance » le sont également.

L'intergiciel peut également intégrer les extensions du modèle « objets répartis » permettant la prise en compte d'événements asynchrones dans les interactions entre objets, telles que l'avortement des appels de méthodes ou les appels unidirectionnels, que nous avons décrits dans la section 2.2.6. Dans ce cas, il devient à même de répondre aux besoins de flexibilité liés au modèle « passage de messages ».

L'introduction des fonctionnalités avancées nécessaires au support du modèle « objets répartis » s'accompagne d'un coût important. L'utilisation efficace d'un intergiciel mettant en œuvre ce modèle pour réaliser des échanges de messages ou des appels de sous-programmes à distance impose donc de pouvoir configurer et optimiser le choix des fonctionnalités présentes, afin de l'adapter aux besoins du modèle souhaité.

2.5.3 Synthèse

Les intergiciels mettant en œuvre le modèle « objets répartis » doivent répondre aux besoins liés à ce modèle, et réaliser des services spécifiques. Ce faisant, ils remplissent potentiellement les besoins d'autres modèles de répartition.

Il est donc possible, à partir d'un intergiciel pour objets répartis, de dériver des mises en œuvre d'autres modèles de répartition. Le coût élevé du modèle « objets répartis » suggère de rendre configurable l'intergiciel, afin de ne pas reporter la pénalité de ces fonctionnalités avancées sur les autres modèles.

Une architecture basée sur les objets répartis et suffisamment configurable pour que son utilisation pour d'autres modèles de répartition ne soit pas rédhibitoire permet donc l'utilisation du même intergiciel par des composants applicatifs utilisant des modèles de répartition différents. Par conséquent, la définition d'un tel intergiciel favorise l'interopérabilité entre modèles de répartition.

2.6 Synthèse et axes de travail

Les intergiciels offrent au développeur d'applications des « boîtes à outils » facilitant la répartition des composants sur des nœuds de traitement, en masquant les problèmes particuliers introduits par la répartition (par rapport à une application non répartie). Ils répondent ainsi aux besoins issus des modèles de répartition.

Ils introduisent cependant des problèmes de génie logiciel spécifiques, en particulier le « paradoxe de l'intergiciel » : les incompatibilités entre systèmes d'exploitation et entre architectures matérielles sont levées, mais de nouvelles incompatibilités, entre modèles de répartition et entre intergiciels, sont introduites.

Ce problème est d'autant plus important que les contraintes qui pèsent sur le choix d'un intergiciel sont fortement liées aux caractéristiques particulières de chaque application.

Il apparaît donc nécessaire de rendre les intergiciels adaptables et configurables, et surtout de favoriser l'interaction entre des composants répartis hétérogènes, c'est-à-dire définis dans des modèles de répartition différents, et mis en œuvre en utilisant des intergiciels différents.

Nous souhaitons définir une architecture d'intergiciel satisfaisant ces objectifs, prenant ainsi en compte les problèmes de génie logiciel introduits par l'utilisation d'intergiciels traditionnels.

Elle doit favoriser la réutilisation du code :

des composants applicatifs en permettant leur utilisation dans des situations diverses, chacune faisant intervenir des intergiciels adaptés ;

de l'intergiciel en autorisant son adaptation au plus juste à des besoins variés et aux fonctionnalités offertes par le système d'exploitation et l'architecture matérielle de chaque nœud ;

à l'intérieur de l'intergiciel en factorisant autant que faire se peut les fonctionnalités génériques utilisées par plusieurs modules.

En définissant cette architecture sur la base des besoins du modèle « objets répartis », nous nous proposons de satisfaire également ceux des modèles

« passage de messages » et « appel de sous-programmes à distance ». La configurabilité doit permettre que l'instanciation de l'architecture pour ces modèles limite le coût lié à l'utilisation du modèle le plus général.

Nous proposons donc d'utiliser les trois propriétés suivantes comme lignes directrices pour définir une architecture d'intergiciel pour objets répartis :

généricité : l'intergiciel devra être composé de modules réalisant de façon cloisonnée d'une part les interfaces présentées par l'intergiciel aux autres composants de l'application répartie, et d'autre part les fonctionnalités internes ; celles-ci seront mises en œuvre, autant que faire se peut, indépendamment des variations possibles sur les interfaces ;

configurabilité : les paramètres de l'intergiciel devront pouvoir être adaptés aux besoins de l'application ; l'intergiciel devra pouvoir fournir un profil fonctionnel réduit, n'exigeant pas de services avancés de la part du système d'exploitation et de l'environnement, notamment afin de pouvoir être utilisé dans des applications embarquées et temps-réel ;

interopérabilité : l'intergiciel devra permettre à des objets définis au moyen d'environnements de répartition et d'intergiciels incompatibles de coopérer, en supportant de multiples personnalités de répartition.

Ces trois propriétés sont liées, et doivent être envisagées simultanément.

L'interopérabilité entre modèles de répartition est nécessaire pour intégrer des composants divers dans de nouvelles applications. Elle est aussi la clé de l'évolution des applications existantes, car elle permet de les faire coopérer ou de leur ajouter de nouveaux composants.

La généricité est le moyen de réaliser l'interopérabilité de façon efficace. Elle évite l'explosion combinatoire liée au support de multiples personnalités.

La configurabilité permet d'adapter l'intergiciel aux besoins spécifiques des applications ; elle autorise l'adoption d'une architecture d'une grande richesse fonctionnelle, en permettant d'optimiser l'étendue des fonctionnalités effectivement instanciées sur un nœud d'une application en fonction des besoins réels.

CHAPITRE

3 État de l'art des intergiciels pour objets répartis

Sommaire

3.1 Intergiciels génériques 54
3.1.1 Généricité par liaisons flexibles : Jonathan 54
3.1.2 Généricité par décomposition : Quarterware 56
3.1.3 Discussion 58
3.2 Intergiciels configurables 59
3.2.1 Configuration des propriétés globales de l'application : GLADE 59
3.2.2 Configuration par patrons de conception : TAO . . . 64
3.2.3 Configuration par politiques d'adaptation d'objets : le POA de CORBA 65
3.2.4 Discussion 66
3.3 Intergiciels interopérables 67
3.3.1 Interopérabilité par partage de protocoles : LSA, SoapRMI 67
3.3.2 Interopérabilité par protocoles adaptables : TAO . . 70
3.3.3 Interopérabilité par génération de passerelles : CIAO 74
3.3.4 Trois perspectives sur l'interopérabilité 78
3.3.5 Discussion 80
3.4 Synthèse 81
3.4.1 Généricité 81

POUR ÊTRE en mesure de répondre aux besoins d'applications réparties avancées, faisant intervenir plusieurs modèles de répartition, les intergiciels doivent présenter trois aspects essentiels, que nous avons identifiés dans le chapitre 2.

Nous illustrons ici ces trois aspects (généricité, configurabilité et interopérabilité) par des projets existants. Nous examinons comment ceux-ci réalisent ces aspects, et nous discutons les choix de mise en œuvre adoptés. Nous observons enfin qu'aucun intergiciel existant ne paraît les réunir tous à la fois. Nous concluons que, dans la mesure où la réalisation *efficace* de l'interopérabilité rend

nécessaires la généricité et la configurabilité, la réalisation des objectifs que nous avons identifiés nécessite la définition d'une architecture nouvelle d'intergiciel qui incorpore ces trois aspects.

3.1 Intergiciels génériques

Définition 10 (intergiciel générique) *Un intergiciel générique est un intergiciel dans lequel certains modules sont rendus indépendants du choix d'une personnalité de répartition particulière, et réalisés sous une forme réutilisable.*

Une personnalité donnée est alors obtenue en instanciant ces modules génériques avec des composants de personnalisation.

3.1.1 Généricité par liaisons flexibles : Jonathan

Jonathan est un environnement de répartition pour le langage Java qui propose un contrôle fin de l'application sur les liaisons entre objets [24].

Jonathan offre un noyau de répartition générique réduit, qui consiste essentiellement en un ensemble d'interfaces abstraites qui doivent être concrétisées par chaque personnalité. Il contient également plusieurs composants de liaison, notamment une mise en œuvre du protocole GIOP de CORBA [71], et un composant basé sur un protocole spécifique, utilisant le transport par diffusion de groupe (*multicast*).

Plusieurs personnalités sont offertes. Celles-ci concrétisent les interfaces abstraites du noyau générique, et définissent les interfaces permettant aux objets de l'application de s'enregistrer auprès de l'intergiciel :

- David offre les fonctionnalités de base d'un intergiciel CORBA (un ORB, *Object Request Broker*) ;
- Jeremie met en œuvre la bibliothèque normalisée de répartition du langage Java : Java RMI *(Remote Method Invocation)* [115].

L'organisation de Jonathan est centrée sur la notion de *liaison* (*binding*). Cette notion, proposée par des projets antérieurs (notamment SOR [106]), représente à la fois le processus par lequel des objets sont associés suivant un mode de communication défini (qui comprend en particulier la mobilisation de ressources de communications) et le résultat de ce processus (l'ensemble des ressources affectées à une association entre objets). Elle est issue du modèle de référence *ODP* (*Open Distributed Processing*) [51].

La bibliothèque de contrôle des liaisons offre des outils de gestion d'identificateurs : il définit des *contextes d'identification* (*naming contexts*) au sein desquels des *noms* peuvent être attribués, c'est-à-dire des éléments d'information désignant une entité sans ambiguïté parmi toutes celles nommées dans le même contexte.

Elle définit également des *usines de liaisons* entre objets, qui permettent d'obtenir, à partir d'un nom désignant un objet, les structures nécessaires à un échange. L'interface des usines de liaisons permet à l'utilisateur de contrôler les propriétés des liaisons entre objets. Il peut ainsi en définir les paramètres de qualité de service. Il peut également utiliser ces interfaces pour créer des liaisons plus générales que la simple relation point à point entre un client et un serveur. Par exemple, il peut mettre en place une liaison point vers multipoint, utilisant un protocole de diffusion de groupe.

Dans le cas d'un objet local, une usine de liaison triviale peut renvoyer directement l'emplacement réel en mémoire de l'objet visé ; dans le cas d'un objet distant, en revanche, l'usine de liaison devra mettre en place une connexion de transport de données avec un intergiciel distant, et débuter une session de protocole.

Définition 11 (objet de liaison) *L'ensemble des structures mises en place sur des nœuds d'un système réparti et au sein d'un réseau de communication qui les relie, en vue de permettre des interactions entre les objets qu'ils hébergent, constitue un* objet de liaison.

Définition 12 (continuation) *La structure locale associée à un nom par une bibliothèque de liaison est la* continuation *de ce nom. Pour un nom désignant un objet local, sa continuation est l'objet lui-même ; pour un nom désignant un objet distant, sa continuation est un objet de liaison subrogé.*

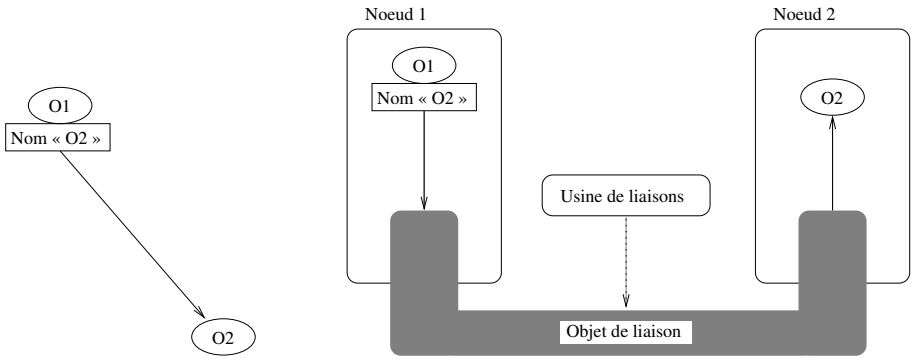
La notion de continuation est issue notamment du projet SOR [106].

Définition 13 (subrogé) *Un subrogé est un objet qui en représente un autre auprès de ceux qui résident sur le même nœud que lui : il interagit avec les objets locaux au nom et pour le compte de l'objet auquel il est rattaché.*

La figure 3.1 illustre les deux types les plus simples d'objets de liaison :

liaison locale entre une référence et l'objet associé, lorsque celui-ci réside sur le même nœud (3.1(a)) ;

liaison point à point entre une référence située sur un nœud et un objet situé sur un deuxième nœud directement accessible (3.1(b)).



(a) Liaison locale (b) Liaison à distance

FIG. 3.1 – Liaisons entre objets

Dans les deux cas, une fois la liaison établie, l'utilisateur peut effectuer des appels de méthodes sur la référence, qui (après transmission par l'objet de liaison) sont exécutés par l'objet réel désigné par la référence. Celle-ci se comporte comme un *subrogé* de l'objet réel : elle le représente, et reçoit des appels de méthode « en son nom ».

Les systèmes d'appels de sous-programmes ou de méthodes à distance sont habituellement basés sur des souches et squelettes (section 2.2.2). Lorsqu'une

souche est associée à un squelette au moyen d'un réseau de communication, elle présente à l'utilisateur la même interface que le sous-programme réel correspondant. On peut donc interpréter la souche cliente dans une telle association comme un cas particulier de subrogé du sous-programme réel.

3.1.2 Généricité par décomposition : Quarterware

Quarterware est à la fois une méthodologie pour construire des intergiciels, et une boîte à outils de composants réutilisables [108]. Dans cette section, nous décrivons en premier lieu la méthodologie utilisée pour identifier les éléments de décomposition pertinents, puis nous présentons les entités élémentaires définies par Quarterware.

Méthodologie *Software RISC*

L'approche méthodologique de Quarterware s'inspire des architectures de microprocesseurs *RISC* (*Reduced Instruction Set Computer*) [78]. Celles-ci ont pour principe de remplacer des jeux instructions complexes mais coûteuses à décoder et à exécuter par des jeux d'instructions élémentaires, simples et efficaces, qui peuvent être assemblées par l'utilisateur pour réaliser les mêmes opérations. Les instructions complexes, effectuant d'un bloc un ensemble d'opérations élémentaires, sont peu réutilisables, au sens où chacune s'applique à un problème macroscopique particulier.

En revanche, lorsque des opérations élémentaires de calcul et de contrôle sont mises à la disposition de l'utilisateur, celles-ci sont constamment réutilisées : au lieu d'utiliser une vaste collection d'opérations coûteuses s'appliquant chacune à un cas particulier restreint, l'utilisateur agence lui-même un petit nombre d'opérations de base d'application universelle. Ces agencements n'étant pas figés dans le jeu d'instructions, mais contrôlés par l'utilisateur en tant que de besoin, ils peuvent être optimisés en fonction des nécessités de l'application.

De la même façon, Quarterware propose une stratégie de décomposition des intergiciels visant à obtenir des systèmes flexibles et efficaces. Cet objectif est atteint par l'application de plusieurs principes généraux.

Utiliser les objets qui apparaissent « naturellement ». Les termes utilisés pour donner d'un système une description de haut niveau, en langage naturel, permettent l'identification de composants pertinents. Par exemple, les *noms* utilisés se traduisent par des *objets* dans la décomposition, tandis que les *verbes* définissent les *méthodes* de ces objets.

Emprunter des éléments d'autres décompositions. Pour un même problème, des décompositions différentes peuvent avoir été proposées. Le fait qu'un élément de décomposition soit récurrent dans les diverses solutions existantes, donne une indication de la pertinence de cet élément.

Rendre explicites les situations d'alternative. Dans chaque situation où plusieurs solutions sont possibles pour résoudre un même problème, une définition abstraite de ce problème doit apparaître dans la décomposition, et chaque terme de l'alternative doit être une concrétisation de cette définition. Il est alors possible de configurer dynamiquement le système en remplaçant une variante par une autre, sans qu'il soit nécessaire de remettre l'architecture en question.

Isoler les dépendances vis-à-vis de composants tiers. Toute interaction avec l'environnement (telles que les interfaces avec le système d'exploitation) doit être isolée dans un composant, afin de rendre la décomposition portable et de minimiser les dépendances cachées. Les modules internes de l'intergiciel ne doivent pas faire directement appel à des composants externes, mais utiliser systématiquement les modules d'isolement prévus à cet effet.

Représenter l'agencement des composants par un composant. Les relations entre composants sont représentées de manière explicite par des composants connecteurs. Il devient alors possible de contrôler ces relations, de les configurer, et d'effectuer des raisonnements et des optimisations sur la façon dont les composants sont agencés.

Les flots de données qui constituent les interactions entre composants doivent également être explicités, pour permettre leur manipulation par les composants connecteurs.

Composition récursive par couches

La réification des éléments d'interaction et d'agencement des composants (c'est-à-dire le fait de mettre ces éléments sous forme d'entités explicitement manipulables par l'intergiciel) est essentielle pour permettre la réutilisation et le remplacement des composants. En effet, elle rend possible des réorganisations à l'intérieur de l'intergiciel par remplacement de connecteurs et de composants.

Cette architecture, à l'instar de celle d'Adapt [11], montre que l'utilisation de composants réflexifs permet la construction d'intergiciels adaptables. Par ce moyen, elle permet l'optimisation du niveau d'abstraction présenté à l'utilisateur.

En effet, chaque niveau supplémentaire est obtenu par composition de composants de plus bas niveau. Cette opération peut être appliquée récursivement pour construire des fonctionnalités de plus en plus élaborées. Le niveau d'abstraction offert aux utilisateurs peut donc être ajusté par le choix des composants agencés. Les abstractions les plus coûteuses sont ainsi éliminées lorsque les besoins de l'application ne justifient pas leur utilisation.

Les hypothèses habituellement implicites sur la nature des composants et des connecteurs, sur la structure globale du système et sur le processus de construction utilisé pour le déployer sont exposées et configurables. La construction d'un système par composition de ces composants est ainsi facilitée. En effet, ces hypothèses cachées sont les principaux obstacles rencontrés habituellement lors de la composition [34].

L'architecture en couches successives de l'intergiciel générique ACT [28] répond également au besoin d'ajustement du niveau d'abstraction utilisé par une application. Cet intergiciel est organisé à partir d'une bibliothèque de communication de bas niveau, au-dessus de laquelle sont successivement construites des abstractions de plus en plus évoluées :

1. flux de données et datagrammes ;
2. protocoles d'appel de méthodes ;
3. adaptation d'objets.

L'utilisateur peut accéder à la couche qui fournit les abstractions qui conviennent le mieux à son application.

De plus, des interfaces uniformes entre couches sont définies, qui permettent à différentes variantes d'une même couche d'être interchangeables. Une telle organisation offre une structure flexible et efficace pour la mise en œuvre de protocoles. Elle a été notamment proposée par le noyau *x*-kernel [46]. L'objectif de *x*-kernel est de fournir un cadre générique pour le développement et l'expérimentation dans le domaine des protocoles réseau. Ce terme est ici entendu de façon générale comme toute abstraction au sein de laquelle un ensemble de participants échangent des messages. Ce cadre générique est obtenu en définissant des protocoles qui peuvent être empilés pour utiliser les services de protocoles de plus bas niveau, et fournir des services à des protocoles de plus haut niveau.

Un tel cadre générique facilite le développement de nouveaux protocoles en permettant facilement leur intégration dans une infrastructure existante. Une telle structure a ainsi été utilisée dans la pile réseau du système FreeBSD [26]. Le coût de ces organisations réside dans la nécessité d'éliminer les dépendances entre couches, et dans la difficulté d'optimiser une couche sans avoir connaissance des propriétés particulières à chaque variante possible des autres couches.

3.1.3 Discussion

Jonathan offre un bon exemple d'intergiciel générique, dans la mesure où il présente une dissociation entre les aspects d'interface entre l'intergiciel et les objets d'application et les aspects internes de l'intergiciel. De multiples personnalités sont proposées : les aspects d'interface de l'intergiciel (par opposition aux aspects internes de mise en œuvre) peuvent être instanciés selon différentes variantes.

Le mécanisme de liaisons flexibles de Jonathan, issu de projets antérieurs (notamment SOR [106]), et reprise également dans l'intergiciel réflexif Adapt, permet d'introduire la notion d'objets subrogés, c'est-à-dire d'objets représentant d'autres objets résidant sur un nœud différent. La relation entre subrogés et objets réels généralise l'organisation traditionnelle des systèmes d'appel de sous-programmes à distance en souches et squelettes. Un subrogé possède en effet toutes les qualités d'un objet réel. On peut donc définir des « subrogés de subrogés », faisant intervenir des objets de liaison complexes chaînés à travers plusieurs nœuds.

Les liaisons flexibles autorisent également un plus grand contrôle de l'utilisateur sur les modalités de mise en relation des objets. Des modalités de communication différentes des liaisons point à point peuvent ainsi être réalisées.

La personnalisation de Jonathan est effectuée en fournissant des mises en œuvre concrètes des interfaces abstraites définies par la partie générique. Celle-ci est relativement réduite (moins de 10 % du code), et ne fournit que peu de services. Elle a pour fonctions principales :

- d'organiser l'articulation entre les composants de personnalités de l'intergiciel ;
- d'offrir des services de base de gestion de ressources (tâches, mémoire, connexions de transport).

Les composants de personnalités, qui réalisent les fonctions essentielles de l'intergiciel, doivent être entièrement récrits pour chaque personnalité.

Dans ACT et Quarterware, en revanche, une décomposition des fonctionnalités de l'intergiciel en services élémentaires est proposée. Dans ACT, cette décomposition est strictement organisée en couches successives. Dans Quarterware, un

modèle plus général de composants interagissants est adopté, qui permet une analyse plus fine des relations entre les fonctions de l'intergiciel.

Celles de ces fonctions qui sont indépendantes de la personnalité peuvent être mise en œuvre une fois pour toutes, et être réutilisés dans toutes les instances personnalisées de l'intergiciel. Cette approche favorise une plus grande mutualisation de code entre personnalités.

Par ailleurs, Quarterware distingue des services de communication fondamentaux, et d'autre services avancés. Cette distinction est une condition nécessaire pour pouvoir ensuite définir des profils de services rendus par l'intergiciel aux composants d'application. Lorsque ceux-ci n'ont pas besoin de certaines fonctionnalités avancées, une architecture en composants modulaires doit permettre d'éliminer ces fonctionnalités, pour obtenir un intergiciel plus compact et plus efficace.

3.2 Intergiciels configurables

Définition 14 (intergiciel configurable) *Un intergiciel est dit* configurable *lorsque les composants qu'il intègre peuvent être choisis et leur comportement adapté en fonction des besoins de l'application et des fonctionnalités offertes par l'environnement.*

3.2.1 Configuration des propriétés globales de l'application : GLADE

GLADE [83] est la mise en œuvre de l'annexe des systèmes répartis (DSA, *Distributed Systems Annex*) du langage Ada 95 pour l'environnement de compilation libre GNAT.

GLADE a été principalement réalisé par le groupe du département Informatique et réseaux de l'ENST au sein duquel cette thèse a été préparée. Notre équipe est toujours actuellement responsable de la maintenance de ce projet.

Ada 95 et l'annexe des systèmes répartis

Le langage Ada, depuis ses origines, et en application de son cahier des charges, a comme propriétés principales :

Typage fort : le fait que deux types de données aient la même représentation interne n'entraîne pas leur compatibilité ; il appartient au programmeur de rendre explicites les cas où la conversion d'un type en un autre est souhaitée.

Abstraction et encapsulation : un programme Ada est organisé en paquetages et en sous-programmes. Chaque paquetage contient une partie visible représentant l'interface accessible aux utilisateurs de ses services, une partie privée contenant les détails des types utilisés dans la partie publique et un corps abritant la mise en œuvre des services du paquetage.

Conformité : tout programme prétendant être un compilateur Ada doit, au préalable, être validé par un institut indépendant. Cette validation est organisée autour d'une vaste suite de tests normalisés vérifiant la conformité à la norme.

En 1995, après un long processus de révision commencé en 1988, une nouvelle norme ISO correspondant au langage Ada a été adoptée [49], et introduit un grand nombre de nouveaux concepts tout en respectant une compatibilité ascendante [48].

Nous retenons en particulier deux nouvelles caractéristiques :

- Programmation orientée objets : Ada 95 supporte l'héritage simple le polymorphisme et l'aiguillage dynamique d'appels de méthodes (équivalent au mécanisme des méthodes virtuelles de Java ou C++). Ces notions se traduisent dans le langage par l'emploi de types dits *étiquetés* (*tagged*). L'héritage multiple peut être représenté par des combinaisons de mécanismes du langage.
- Annexes optionnelles : ces annexes, qui ne sont pas obligatoirement mises en œuvre dans tout compilateur Ada conforme à la norme, doivent, si elles sont fournies, respecter la norme. Des annexes existent, entre autres, pour le temps réel et la répartition. GLADE fait de GNAT le seul compilateur conforme à la norme sur l'ensemble des annexes.

L'annexe optionnelle des systèmes répartis (annexe E ou « DSA ») de la norme Ada 95 [49] étend le mécanisme d'abstraction du langage (l'utilisation de paquetages) au cas d'applications réparties.

La déclaration d'un paquetage contient des déclarations de types de données et des signatures de sous-programmes, tandis que le corps du paquetage contient le corps de ces sous-programmes, ainsi que les structures internes dont ils ont besoin, mais qui ne doivent pas être exposées aux utilisateurs. La déclaration d'un paquetage peut également contenir la définition de types étiquetés et de leurs opérations primitives, c'est-à-dire de classes d'objets.

L'annexe des systèmes répartis étend naturellement ce modèle, en permettant au programmeur d'indiquer que l'interface définie par une déclaration de paquetage doit être considérée comme « distante », au moyen de directives de compilation prévues à cet effet. Il appartient alors au compilateur de générer des appels à un sous-système de communication, afin de réaliser les échanges de messages nécessaires.

Les différentes unités constituant une application répartie sont finalement regroupés en *partitions*.

Définition 15 (partition) *Dans une application répartie, une* partition *est le programme exécuté par un nœud.*

Une mise en œuvre de l'annexe des systèmes répartis est donc constituée de trois éléments :

un compilateur qui génère les souches et squelettes nécessaires à la répartition d'unités sur différents nœuds (qui sont appelés *partitions* dans la norme) ;

une bibliothèque d'exécution ou *PCS* (*Partition Communication Subsystem*), qui fournit les services de communication nécessaires au fonctionnement des souches et des squelettes ;

un outil de déploiement qui permet l'affectation des diverses unités constituant une application répartie DSA sur les *partitions* définies par l'utilisateur. Cet outil est laissé entièrement à la charge du fournisseur de l'annexe : il n'est pas défini par la norme.

GLADE est basé sur l'environnement libre GNAT [104], qui fait partie de la famille de compilateurs GNU GCC. La partie « compilation » de l'annexe est di-

rectement intégrée dans le compilateur. Elle a fait l'objet de la thèse de doctorat de Samuel Tardieu à l'ENST [116]. La bibliothèque d'exécution GARLIC (*Generic Ada Reusable Library for Interpartition Communication*) constitue, elle, un module séparé qui peut être construit indépendamment [55]. L'outil de déploiement `GNATDIST` offre un langage permettant de spécifier le découpage d'une application en partitions, et de configurer certaines propriétés l'exécutif [54].

La norme définit l'interface entre les composants d'une application répartie et l'exécutif de répartition. Elle définit les constructions du langage qui donnent lieu à un fonctionnement réparti, et le prolongement de la sémantique des applications monolithiques dans le cas où des unités sont déployées sur des partitions différentes. Elle définit également l'interface entre le code produit par le compilateur pour les souches et les squelettes et la bibliothèque du sous-système de communication.

En revanche, elle ne définit pas l'interface entre différentes instances du sous-système de communication, ni l'outil de partitionnement. Le fournisseur d'une mise en œuvre de l'annexe a donc toute latitude pour choisir les moyens de communication, les protocoles et les représentations des données utilisés. Il fournit également à l'utilisateur une procédure permettant d'assembler des unités de bibliothèque pour constituer une partition.

Dans le cas de GLADE, une attention particulière a été portée à la configurabilité des aspects de communication et de parallélisme, ainsi qu'au contrôle du cycle de vie des partitions et de l'application dans son ensemble.

Mécanismes de transport

Une interface abstraite pour les modules de communication est définie par GLADE. Chaque tel module représente un support d'échange entre partitions offrant un service de transport de données fiable et ordonné [32]. Plusieurs mises en œuvre concrètes de cette interface ont été réalisées. De plus, deux partitions participant à la même application peuvent utiliser des modules différents : le support de communication utilisé peut être configuré indépendamment pour chaque couple de partitions.

Ainsi, le protocole par défaut transmet simplement un flot d'octets sur une connexion TCP. Dans le cas où l'on souhaite déployer une partition sur un système embarqué disposant de peu de ressources (mémoire et capacité de calcul), le coût de mise en œuvre de TCP peut être rédhibitoire. Il est alors possible d'utiliser un autre protocole utilisant une simple liaison série. Celle-ci peut être réalisée au moyen d'une connexion physique, ou bien par un lien radio, comme c'est le cas pour la plate-forme robotique SPIF [25].

Par ailleurs, si l'on désire garantir les échanges entre partitions contre des attaques hostiles (espionnage, interposition ou rejeu), il est possible d'utiliser un troisième module de communication dans lequel les données sont transmises dans une connexion chiffrée et authentifiée utilisant le protocole SSL (*Secure Socket Layer*) [30].

Filtres

GLADE autorise l'introduction de « filtres » dans la chaîne de communication [84]. Un filtre est une entité qui effectue une transformation arbitraire sur les données avant transmission, et la transformation réciproque après réception.

Des filtres successifs peuvent être empilés, pour composer plusieurs transformations (figure 3.2).

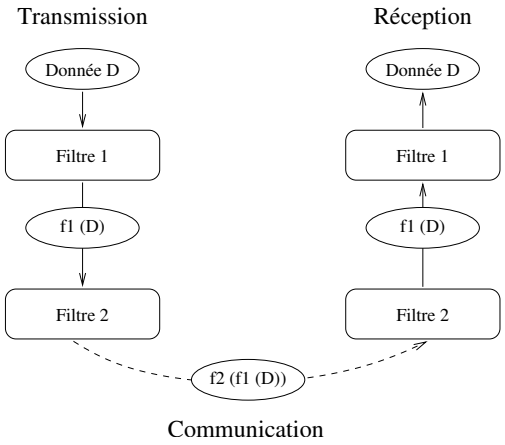


FIG. 3.2 – Filtres de communication

Plusieurs filtres utiles ont été mis en œuvre pour GLADE. Pour les liens de faible débit, un filtre de compression utilisant la bibliothèque Zlib a été développé. Pour la communication sur des liaisons non sûres, un filtre réalisant l'authentification des partitions et le chiffrement des données échangées au moyen de l'algorithme RSA a également été proposé [56].

Exécutif léger

La configuration normale de GLADE offre, sur chaque partition, la possibilité d'exécuter en parallèle les appels de sous-programme reçus, ainsi que des tâches de l'utilisateur. La contrepartie de cette fonctionnalité est une dépendance vis-à-vis de la bibliothèque de parallélisme de GNAT, dont l'encombrement et la consommation de puissance de calcul peuvent être considérables dans certaines applications.

Lorsqu'il n'est pas nécessaire qu'une partition effectue des traitements parallèles, la possibilité est donc offerte de configurer GLADE pour ne pas introduire cette dépendance. Cette configuration d'exécutif léger permet un fonctionnement entièrement monotâche ; le traitement des appels reçus est alors strictement sérialisé.

Les propriétés d'utilisation d'exécutif de parallélisme léger, de configurabilité explicite du parallélisme pour le traitement des requêtes et de passage de priorité sont fondamentales pour la réalisation d'applications réparties temps réel. Elles constituent des perspectives pour une future révision de la norme où le temps réel et la répartition seraient traités de manière intégrée [38].

Paramétrage du parallélisme

Dans les cas où l'exécutif complet est utilisé, il est possible de paramétrer la politique de création et de destruction des tâches qui concourent au traitement des requêtes entrantes [82].

Un « pool » de tâches est maintenu par l'exécutif. Chacune de ces tâches peut être occupée à traiter une requête, ou bloquée en attente d'un travail à réaliser. Les requêtes entrantes peuvent ainsi être prises en charge immédiatement, sans avoir à attendre en général le délai lié à la création d'une tâche.

L'exécutif tend à assurer que le nombre de tâches libres soit relativement stable, tout en fixant un nombre maximal de tâches dans le nœud. Les différents paramètres de cette politique sont ajustables par l'utilisateur en fonction du comportement particulier de chaque application (débit et variance de l'arrivée des requêtes) : nombre minimal de tâches libres, nombre maximal de tâches libres, nombre maximal de tâches total.

GLADE étend également la sémantique normalisée en autorisant la traduction et le passage, d'une partition à l'autre, de la priorité de la tâche appelante.

Outil de partitionnement et de configuration

L'utilisation de l'exécutif léger dans GLADE est automatique pour les partitions qui se contentent d'émettre des appels de sous-programmes (ou de méthodes) à distance, mais n'en reçoivent jamais, et dont les composants applicatifs n'utilisent pas de tâches.

La configuration des aspects de communication (choix des mécanismes de transport et des filtres) est effectuée en même temps que le découpage de l'application en partitions. Cette opération est réalisée au moyen du langage de configuration de l'outil de partitionnement `GNATDIST`, qui génère le code nécessaire à l'initialisation de GLADE pour chaque partition.

Le langage de `GNATDIST` permet également la configuration de propriétés globales de l'application, concernant le cycle de vie des partitions. Ainsi, plusieurs politiques de reconnexion des partitions peuvent être sélectionnées, qui permettent d'autoriser ou non le redémarrage d'une partition et son retour au sein de l'application lorsqu'elle s'est terminée de façon fautive. De même, un algorithme de terminaison globale est proposé. Celui-ci permet de déclencher l'arrêt de toutes les partitions lorsque plus aucune évolution de l'application répartie ne peut avoir lieu. Cet algorithme est configurable, pour permettre à des partitions qui se contentent de répondre à des appels de continuer à exister (en attente d'éventuels nouveaux clients) même lorsque tous leurs utilisateurs ont disparu.

Limitations

Les applications programmées avec l'annexe sont limitées par l'utilisation exclusive du langage Ada 95. De plus, la norme n'impose aucune contrainte à l'interface entre partitions : chaque mise en œuvre de l'annexe des systèmes répartis peut choisir son propre protocole. Cette limitation a fait l'objet de propositions pour une future révision de la norme [116].

La présente thèse a été encadrée par Laurent Pautet, l'un des principaux participants du projet GLADE. Elle s'inscrit dans le prolongement de ce projet ; elle vise en particulier à lever cette limitation, en ouvrant les applications Ada 95 vers d'autres modèles de répartition.

3.2.2 Configuration par patrons de conception : TAO

TAO (The ACE ORB) est un intergiciel destiné aux applications temps réel nécessitant des garanties de qualité de service de bout en bout [98].

Ce projet vise à réaliser un intergiciel portable, configurable, facile à étendre et à maintenir. En effet, TAO doit être adapté à la diversité des besoins de qualité de service qui peuvent être exprimés dans les applications réparties. Pour chaque aspect du comportement de l'intergiciel, l'utilisateur doit donc pouvoir choisir une stratégie adaptée parmi l'ensemble des options disponibles.

Dans TAO, ces objectifs sont atteints notamment par l'utilisation extensive de *patrons de conception* (*design patterns*). Un patron est un élément de conception de haut niveau : « Une description d'objets communicants et de classes qui sont particularisés pour résoudre un problème général dans un contexte particulier » [31].

L'architecture de TAO utilise de nombreux patrons, décrits en détail par [94]. En particulier, plusieurs patrons contribuent à la configurabilité de l'intergiciel.

Patron *Strategy*

De nombreux comportements de l'intergiciel peuvent être réalisés en utilisant un algorithme choisi au sein d'une famille de solutions. Ce choix ne doit pas être fixé à la compilation, mais doit pouvoir être choisi et modifié sur un intergiciel en cours d'exécution.

Le patron *Strategy* consiste à associer une classe abstraite à une telle famille de solutions, et une classe fille concrète à chaque membre de la famille ; une instance d'une classe fille encapsule ainsi un algorithme et son état, et permet à tout composant utilisateur ne connaissant que l'interface abstraite de faire usage de la solution qu'il réalise.

Ce patron de conception autorise le choix par l'utilisateur de variantes de mise en œuvre de services au sein de l'intergiciel. L'identification des cas où il peut être utilisé dans l'architecture apporte une information importante : elle signale des composants de l'intergiciel dont l'interface peut être fixée, et la mise en œuvre remplacée. Les occurrences du patron *Strategy* peuvent donc être vues comme autant d'axes de configuration de l'intergiciel.

Par exemple, plusieurs stratégies d'affectation des requêtes reçues à des tâches sont proposées [100, 102, 101]. Une tâche peut être créée pour chaque requête entrante, ou bien créée lors de l'établissement d'une session de protocole et utilisée pour traiter toutes les requêtes reçues par le truchement de cette session. La troisième stratégie proposée consiste à réutiliser un ensemble de tâches persistantes.

Patron *Abstract factory*

Les occurrences du patron *Strategy* représentent les points de l'intergiciel où une configuration est possible. Il s'agit d'un trait de conception de bas niveau : une stratégie est un plus petit élément de configuration. On peut ainsi rapprocher l'identification des stratégies qui interviennent dans un intergiciel de la décomposition en composants élémentaires proposée par la méthodologie *Software RISC* de Quarterware, exposée dans la section 3.1.2.

Le patron *Abstract factory* consiste à offrir un mécanisme uniforme pour la création d'un ensemble de stratégies, ayant pour fonction d'instancier un jeu de

choix compatibles qui se combinent de manière cohérente : une *Abstract Factory* est une stratégie pour l'instanciation de stratégies.

Chaque occurrence du patron *Abstract factory* correspond donc à un ensemble de stratégies liées par des relations de dépendance ou de compatibilité ; deux occurrences de ce patron correspondent à deux groupes d'axes de configuration qui peuvent être contrôlés par l'utilisateur de façon indépendante.

Patron *Service configurator*

La possibilité de créer de multiples stratégies pour chaque aspect du comportement de l'intergiciel peut conduire à embarquer dans celui-ci une grande quantité de code qui ne sera jamais exécuté, en fonction des options choisies par l'utilisateur.

Le patron *Service configurator* consiste à charger dynamiquement les seules mises en œuvre de stratégies souhaitées à l'exécution, au moyen de fonctions d'édition de liens dynamiques du système d'exploitation.

Les fonctionnalités configurées dans une instance d'intergiciel peuvent ainsi être modifiées aisément, sans qu'il soit nécessaire de le recompiler ou de refaire l'édition de liens. L'élimination complète des modules superflus pour une application donnée est également facilitée.

Le patron *Service configurator* permet donc de tirer profit de l'architecture modulaire et configurable de l'intergiciel pour en optimiser l'empreinte mémoire.

3.2.3 Configuration par politiques d'adaptation d'objets : le POA de CORBA

La spécification CORBA définit une architecture d'adaptateur d'objets, le *POA* (*Portable Object Adapter*) [71]. L'adaptateur d'objets est le composant d'un intergiciel CORBA qui contrôle l'association entre les références d'objets et les structures concrètes qui mettent en œuvre ces objets : les *servants*.

Définition 16 (servant) *Un servant est une structure concrète (souvent une instance d'objet d'un langage de programmation) qui met en œuvre un (ou plusieurs) objet d'une application à objets répartis.*

Le POA est mis en œuvre par de nombreux intergiciels tels que TAO ou AdaBroker, un intergiciel CORBA complet développé au sein de notre équipe à l'ENST, et distribué sous forme de logiciel libre.

Contrairement aux interfaces d'adaptateur d'objets qui existaient dans les premières versions de CORBA, le POA propose un espace d'attribution de références arborescent. L'utilisateur peut définir librement une hiérarchie entre différents espaces d'adressage. Il peut également configurer certains aspects du comportement de l'adaptateur d'objets, en contrôlant indépendamment chaque nœud de l'arborescence.

Ce paramétrage est réalisé en associant à chacun de ces nœuds des *politiques*, une politique étant une instance particulière d'un comportement configurable. Les axes de paramétrage ainsi exposés par la spécification CORBA peuvent être considérés comme des instances du patron *Strategy*.

Ils permettent d'agir sur de nombreuses facettes de l'intergiciel. En effet, dans le modèle de répartition de CORBA, l'adaptateur d'objets est un acteur central du traitement des requêtes reçues. En tant que composant responsable

de l'affectation d'une requête à un objet concret, il contrôle le déroulement des fonctions d'activation et d'exécution. Il intervient également dans la fonction d'adressage, car il contrôle l'espace local des identificateurs d'objets d'un nœud.

Ainsi, trois politiques d'activation sont normalisées, régissant l'association entre références d'objets et servants :

- Les servants peuvent s'enregistrer auprès de l'intergiciel et incarner chacun un objet. Dans ce cas, une référence d'objet correspond exactement à un servant, et l'intergiciel maintient une table associant chaque référence attribuée à l'objet de mise en œuvre correspondant ;
- Toutes les références correspondant à un même nœud de l'arborescence du POA peuvent également être associée à un même servant, qui est alors le *servant par défaut* de ce nœud. Ce servant par défaut traite alors les requêtes pour toutes les références créées à ce point de l'arborescence.
- Une usine de servants peut être utilisée pour créer ou retrouver dynamiquement le servant qui doit traiter une requête donnée.

Les politiques du POA de CORBA permettent d'exprimer une grande variété de comportements de l'intergiciel côté serveur. En revanche, elles ne permettent pas de paramétrer le fonctionnement du rôle client.

3.2.4 Discussion

Dans CORBA, l'utilisation du POA permet à l'utilisateur de contrôler les aspects du comportement de l'intergiciel qui sont liés à la création des références d'objets sur les nœuds serveurs, et à l'association des requêtes reçues avec les diverses ressources nécessaires à leur exécution.

Le mécanisme de création de références d'objets peut être utilisé indépendamment de l'instanciation d'un objet concret local, contrairement à la situation dans DSA. L'utilisateur peut ainsi mettre en œuvre un modèle de données généralisé, où la correspondance entre références d'objets et servants est plus générale qu'une simple bijection.

TAO met en œuvre le POA de CORBA. Il offre ainsi la possibilité de configurer les fonctions d'adressage, d'activation et d'exécution de l'intergiciel côté serveur.

TAO et GLADE offrent également d'autres aspects de paramétrage de leurs composants internes, qui autorisent la configuration des rôles clients aussi bien que des rôles serveurs. Les axes de configuration de TAO sont particulièrement orientés sur les propriétés temps réel des applications réparties. GLADE permet la configuration de certaines propriétés globales du cycle de vie de l'application répartie (déconnexion et reconnexion de partitions, terminaison globale).

TAO utilise les patrons de conception de façon extensive, en particulier pour favoriser la configurabilité. Les patrons de conception *Strategy* et *Abstract factory* interviennent explicitement dans la documentation de TAO, mais sont également utilisés dans GLADE.

Certaines fonctionnalités du sous-système de communication peuvent être étendues au moyen d'une organisation en couches, similaire à celles que nous avons décrites dans la section 3.1.2. Des piles de filtres peuvent être organisées, qui opèrent sur les flots de données échangés avec les autres intergiciels. Par ce moyen, l'utilisateur peut contrôler les composants responsables de la communication d'une façon globale. Ce contrôle s'ajoute à la faculté normalisée offerte par DSA qui autorise l'utilisateur à définir lui-même la représentation

des données transmises entre partitions : grâce à des directives normalisées, il peut fournir des sous-programmes chargés de transformer une valeur d'un type donné en une suite d'octets et réciproquement.

GLADE et TAO mettent en avant la nécessité de proposer un contrôle fin de l'allocation des ressources d'exécution, afin d'introduire la répartition dans des applications embarquées et temps réel. On retrouve notamment l'idée de recycler des tâches persistantes pour traiter les requêtes, afin d'économiser le coût de création de tâches au fur et à mesure de l'arrivée des demandes.

La possibilité, dans GLADE, d'obtenir un intergiciel léger, n'exigeant que des fonctionnalités réduites du système d'exploitation, est fondamentale dans de telles applications. Elle est réalisée au moyen d'un outil de partitionnement qui opère en amont de l'édition de liens des partitions. Pour certains aspects de configuration, TAO va au-delà de cette procédure allégement, en autorisant la reconfiguration dynamique (à l'exécution) des fonctionnalités instanciées.

3.3 Intergiciels interopérables

Dans cette section, nous présentons trois approches issues de travaux existants proposant des intergiciels interopérables.

Définition 17 (intergiciel interopérable) *Un intergiciel est* interopérable *lorsqu'il offre aux composants applicatifs qu'il héberge la faculté d'interagir avec des composants tiers utilisant des personnalités différentes.*

3.3.1 Interopérabilité par partage de protocoles : LSA, SoapRMI

Le Linear System Analyzer (LSA) est une application scientifique comprenant des modules de calcul parallèle écrits en Fortran et en HPC++, et des modules d'interface graphique programmés en Java [13]. Ce projet pose le problème de l'intégration entre ces différents composants au sein d'une architecture répartie. En effet, le mécanisme de répartition choisi pour les composants Java est RMI, tandis que les objets de calcul utilisent le modèle d'objets distants offert par HPC++. De plus, les composants graphiques et de calcul doivent s'exécuter sur des machines différentes. Il est donc nécessaire de faire interopérer les modèles de répartition de RMI et HPC++.

Les auteurs de LSA décomposent le problème d'interopérabilité entre les personnalités RMI et HPC++ en deux aspects :

1. la définition d'un sous-ensemble du modèle objets de RMI qui puisse être projeté sur celui de HPC++ ;
2. la réalisation d'un sous-système de communication capable d'offrir à l'utilisateur ces deux modèles de répartition.

Projection de RMI sur HPC++

Dans LSA, les appels de méthodes RMI sont projetés sur des appels de méthodes HPC++. Certains points de cette projection sont difficiles à mettre en œuvre, car la sémantique fine des appels n'est pas exactement identique entre les deux systèmes.

Le traitement des exceptions est l'un de ces cas problématiques. En effet, les exceptions susceptibles d'être levées dans la partie HPC++ de l'application doivent être connues au moment de la compilation, tandis qu'en Java il est possible de créer et lever un nouveau type d'exception à l'exécution. Un cas particulier doit donc être prévu dans la partie HPC++ pour lever une exception par défaut lorsqu'un message est reçu qui signale l'occurrence d'une exception Java inconnue.

De la même façon, lorsqu'un type objet Java est passé par valeur en paramètre d'une méthode d'un objet HPC++, il n'est pas possible de passer une instance d'une classe Java *dérivée* de la classe nominale du paramètre. En effet, rien ne permettrait de garantir que la projection de cette classe dérivée existe dans la partie HPC++. LSA impose donc que toute classe (Java) utilisée comme paramètre d'une méthode d'un objet HPC++ soit une classe « finale », c'est-à-dire une classe dont on interdit explicitement la dérivation.

La transmission de structures complexes ou d'objets multiples est également problématique, car la relation de référencement entre objets n'est pas conservée. Ainsi, lorsqu'une méthode HPC++ reçoit en argument deux références d'objets, LSA ne garantit pas en général qu'un appel depuis Java où deux références du même objet sont utilisées se traduira par un appel HPC++ portant également sur deux références du même objet ; une copie intempestive peut avoir été réalisée.

La transmission de structures de données basées sur les références entre objets, telles que les listes chaînées est également interdite. Ces restrictions sont dues à l'impossibilité de traduire les références Java en références HPC++ dans le cas général : la projection des objets HPC++ vers les objets Java n'est pas réalisée de manière symétrique et réversible.

LSA : utilisation d'un sous-système de communication mutualisé

Une fois définie la projection entre les personnalités Java RMI et HPC++, chaque personnalité est réalisée au moyen de la bibliothèque de communication Nexus. Une mise en œuvre de Java RMI est ainsi obtenue (Nexus/RMI). Nexus constitue également le sous-système de communication princeps de la mise en œuvre des objets répartis HPC++.

Ces deux mises en œuvre de personnalités de répartition utilisent le même sous-système de communication, et un protocole commun. Elles deviennent ainsi interopérables : les objets de l'un et l'autre modèle sont intégrés et peuvent interagir, par l'intermédiaire de Nexus (figure 3.3). Les composants d'application de part et d'autre sont réalisés d'une façon *ad hoc* pour que leurs interfaces soient isomorphes au sens de la projection précédemment définie.

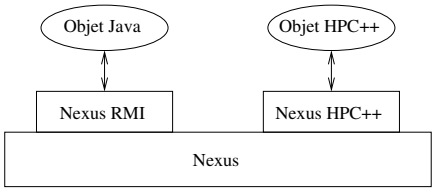


FIG. 3.3 – Interopérabilité dans LSA

L'utilisation d'un sous-système de communication *ad hoc* définissant son propre protocole et sa représentation des données permet à Nexus/RMI d'offrir des performances comparables à la mise en œuvre habituelle de Java RMI, tout en autorisant l'intégration de clients et de serveurs RMI dans une application HPC++. LSA démontre ainsi la faisabilité de l'interopérabilité entre des personnalités distinctes réalisant le modèle de répartition par objets.

En revanche, aucune interopérabilité avec d'autres modèles de répartition n'est possible. L'introduction d'objets n'utilisant ni Java RMI ni HPC++ n'est pas possible, à moins de redévelopper à chaque fois un intergiciel complet basé sur Nexus.

SoapRMI : utilisation d'un protocole commun

SoapRMI C++/Java [109] propose également une architecture permettant de faire interagir des objets Java et C++ en utilisant le modèle de répartition de RMI. Dans SoapRMI, une mise en œuvre de RMI différente est utilisée pour chaque langage, et l'interopérabilité est assurée par l'utilisation, dans les deux cas, du protocole normalisé SOAP [12].

Sans offrir l'intégration des objets dans des environnements utilisant des protocoles arbitraires, l'utilisation de SOAP, protocole public et normalisé, facilite l'interaction des applications qui l'utilisent avec des objets tiers, si ceux-ci sont hébergés par des intergiciels supportant SOAP.

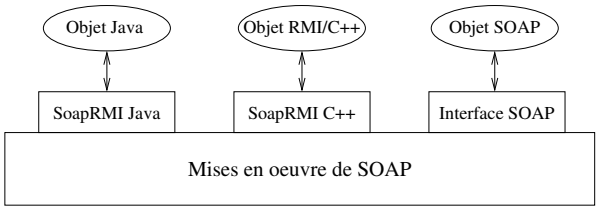


FIG. 3.4 – Interopérabilité dans SoapRMI

Comme l'illustrent les figures 3.3 et 3.4, LSA et SoapRMI permettent à des objets définis au moyen de modèles de répartition différents (des objets Java utilisant Java RMI d'une part, des objets C++ d'autre part) de coopérer, en utilisant des intergiciels possédant un protocole en commun (le protocole propriétaire de Nexus pour LSA, le protocole normalisé SOAP pour SoapRMI).

Initialisation

SoapRMI met en évidence la nécessité d'utiliser un serveur de noms commun aux deux personnalités lors de l'initialisation de l'application, comme nous l'avons évoqué dans la section 2.2.3. Ce serveur permet aux objets créés dans une personnalité de publier leur identité et une référence permettant de les atteindre. Ces références doivent contenir l'identification d'un point d'accès du protocole choisi pour réaliser l'interconnexion.

Plusieurs approches sont proposées pour l'enregistrement et la diffusion des références d'objets. Une mise en œuvre du registre Java RMI pour Java et C++ est proposée, mais également un mécanisme permettant d'utiliser divers services d'annuaire de façon interchangeable pour diffuser l'adresse des objets

de l'application, au moyen de l'interface normalisée JNDI *Java Naming and Directory Interface* [113].

Mandataires dynamiques

SoapRMI tire partie de la possibilité introduite dans Java 1.3 de créer des *mandataires dynamiques* (*dynamic proxies*). Il s'agit d'une construction réflexive qui réifie le mécanisme d'appel de méthode, et permet la construction d'objets qui interceptent ces appels. De cette façon, les souches clientes responsables d'émettre les requêtes d'appel de méthodes à destination d'objets distants peuvent être réalisées de façon générique.

Au lieu de générer une souche pour chaque objet, un mandataire dynamique est utilisé, qui reçoit l'appel à transmettre sous forme de valeurs indiquant un objet cible, l'identité de l'une de ses méthodes et une liste de paramètres. Ces valeurs permettent la construction de la requête correspondante de façon générique, indépendamment de la classe et de la méthode particulières concernées par un appel donné.

Le mécanisme de mandataire dynamique consiste à créer un objet possédant la même interface que l'objet d'origine, mais qui transforme les appels de méthodes en communications par le biais de l'intergiciel. Il recoupe la notion d'objet subrogé que nous avons évoquée dans la section 3.1.1.

Interfaces d'appel dynamique

Les mandataires de SoapRMI transforment chaque appel de méthodes dont ils font l'objet en une structure de données *décrivant* cet appel (y compris les valeurs de ses paramètres). Cette structure, similaire aux objets spécifiés par l'interface d'invocation dynamique (DII) de CORBA, réifie le mécanisme d'appel de méthodes : elle offre à l'intergiciel une interface réflexive pour manipuler ces appels. Contrairement au cas de LSA, il n'est pas nécessaire ici que chaque nœud contienne des composants générés spécifiquement pour intégrer avec l'intergiciel un objet applicatif particulier.

On retrouve une utilisation similaire de la DII dans CorbaWeb, un outil permettant d'accéder aux services d'objets CORBA quelconques à partir d'une page WWW [65]. Dans chacune de ces situations, on constate que les interfaces d'appel dynamique permettent d'écrire des composants liés à la répartition qui sont indépendants de l'interface d'un objet particulier. Elle facilite ainsi la création de modules génériques dans les intergiciels.

3.3.2 Interopérabilité par protocoles adaptables : TAO

TAO a été étendu pour permettre l'adaptation de multiples protocoles et de multiples services de transport dans l'intergiciel [76]. Cette fonctionnalité a été développée pour répondre aux besoins d'applications temps réel. En effet, le protocole GIOP normalisé par CORBA n'est pas adapté pour de telles applications : la taille des messages, la latence et la variance de délai associée doivent être réduites. L'utilisation de protocoles spécialisés permet un gain considérable, au prix de contraintes de localité (telles que la nécessité d'avoir une connexion par réseau local haut débit entre les nœuds) [60].

De plus, l'utilisateur souhaite pouvoir utiliser des mécanismes de transport de données offrant une qualité de service de bout en bout. Lorsqu'un tel service est disponible, il doit également pouvoir en ajuster les paramètres, afin de spécifier le niveau de qualité de service souhaité, à l'instar du mécanisme de liaisons de Jonathan que nous avons décrit dans la section 3.1.1.

Objectifs

Les protocoles adaptables de TAO ont pour objectif de s'intégrer naturellement dans l'intergiciel. Pour cela, il est nécessaire qu'une interface soit définie, à l'intérieur de l'intergiciel, qui permette l'enregistrement des protocoles disponibles, et leur mise en correspondance avec les composants de transport adéquats. Cette interface doit être distincte de celle utilisée par les composants d'application pour s'enregistrer auprès de l'intergiciel. En effet, elle n'intéresse que le développeur de protocoles, et non le développeur de composants applicatifs.

L'interface utilisée par ce dernier doit demeurer inchangée : il se contente d'utiliser les services offerts par l'intergiciel (dont font partie les divers protocoles), sans avoir à se soucier des détails de leur réalisation. Cependant, une interface de configuration doit permettre l'expression des besoins de l'utilisateur, tels que le niveau de qualité de service souhaité.

Dans la mesure où certaines optimisations sont liées à des contraintes de localité, et aussi pour permettre l'intégration de composants tiers utilisant chacun un protocole fixé à l'avance, plusieurs protocoles doivent pouvoir coexister au sein du même intergiciel. Par ailleurs, pour chaque protocole, un moyen de nommer des points d'accès doit être offert. Le nom d'un point d'accès d'un protocole consistera généralement en l'association d'un nom de point d'accès d'un service de transport avec la désignation du protocole à utiliser lorsqu'une session est établie avec celui-ci.

La normalisation de CORBA prévoit le support de multiples espaces de désignation des points d'accès : pour chaque protocole et chaque type de service de transport associé, un *profil d'adressage* est défini. Le support de multiples types de profils permet d'intégrer un nombre quelconque de points d'accès et de protocoles dans les références d'objets.

Enfin, le mécanisme d'adaptation de protocoles doit permettre le raccordement direct entre deux protocoles, et ainsi l'établissement dynamique de passerelles. En effet, chaque protocole adaptable dans l'infrastructure est capable de transformer une représentation uniforme d'une requête en un message, et d'effectuer l'opération inverse. Il est donc possible de configurer un nœud pour recevoir des requêtes avec un protocole, et les réémettre avec un autre.

Une telle fonctionnalité autorise la mise en relation d'intergiciels utilisant des protocoles distincts. Elle évite les synchronisations entre objets et les copies de données qui seraient occasionnées par la remontée de chaque requête à un composant applicatif. Elle évite aussi l'explosion combinatoire qui affecterait l'utilisation de passerelles *ad hoc*.

Contreparties

La souplesse fonctionnelle d'une architecture de protocoles adaptables a deux contreparties :

complexité : l'adaptation de protocoles nécessite l'introduction d'une représentation des requêtes indépendante d'un protocole particulier. Un niveau d'indirection supplémentaire est introduit dans l'architecture de l'intergiciel.

absence d'optimisation : l'intégration de multiples protocoles interdit certaines optimisations précoces pour l'un d'eux en particulier. Le niveau d'indirection complémentaire et la nécessité de manipuler les requêtes sous une forme neutre sont également des sources possibles de performances limitées.

Architecture d'adaptation de protocoles

Les objectifs fixés pour le mécanisme de protocoles adaptables sont atteints par une organisation du sous-système de communication en trois composants (figure 3.5), que nous détaillons ci-après.

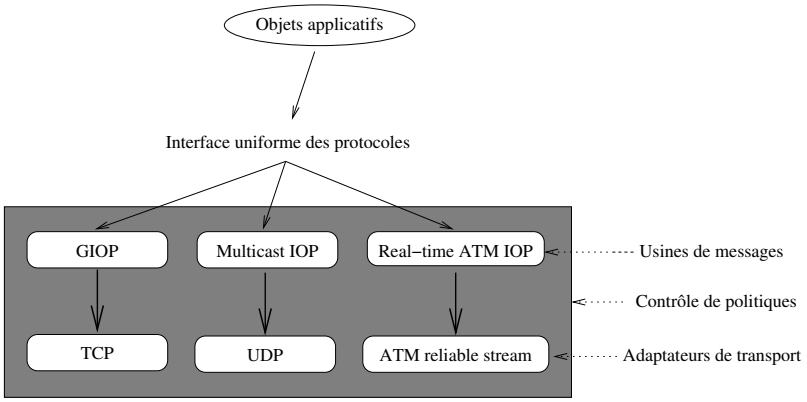


FIG. 3.5 – Composants de TAO pour l'adaptation de protocoles

Ceux-ci définissent des interfaces abstraites, qui doivent être concrétisées pour chaque protocole, et pour chaque projection d'un protocole sur un mécanisme de transport. L'utilisation de classes abstraites qui sont dérivées pour réaliser les fonctions de protocoles d'appel de méthodes a également été utilisée par Spring [42].

Par ailleurs, des services génériques, communs aux différents protocoles, sont proposés pour faciliter la réalisation des protocoles concrets.

Composant de création et de traitement des messages

À partir d'une représentation uniforme d'une requête, ce composant crée un message transportable au moyen d'une *usine de messages* (*message factory*), conformément à un protocole. Il y insère une représentation des données à transmettre (par exemple un flux CDR). Il peut également appliquer des politiques d'allocation de ressources telles que la préallocation d'espace mémoire pour les données à transmettre, qui permet de garantir constamment une faible latence des appels (en supprimant l'allocation dynamique de ressources).

Les messages reçus sont symétriquement transformés en représentation uniforme par un objet de traitement de messages.

Certains modules peuvent être partagés par plusieurs usines de messages. Ainsi, une mise en œuvre optimisée de CDR a été réalisée ; elle est réutilisée par de nombreux protocoles, et peut être adaptée pour utiliser des stratégies d'allocation et de traduction adaptées.

Composant d'adaptation de transport

Ce composant projette un protocole d'échange de requêtes, par exemple GIOP, sur un mécanisme de transport de données particulier, tel qu'une connexion TCP. Il permet la réalisation d'opérations et d'optimisation spécifiques d'un moyen de transport particulier. Il est responsable notamment de la réservation éventuelle de ressources réseau. Il met également en œuvre les techniques de transmission adaptées pour une interface particulière (par exemple la transmission sans copie ou l'utilisation de segments de mémoire partagés entre l'intergiciel et la bibliothèque de transport).

Composant de contrôle des politiques

Ce composant expose à l'utilisateur le contrôle des paramètres de qualité de services associés à chaque connexion. Pour cela, il met en œuvre les interfaces normalisées *CORBA Messaging* [103] et *Real-time CORBA*.

La première offre à l'utilisateur le contrôle des modalités d'échange de messages (asynchronisme, communication unidirectionnelle) et de la qualité de service liée à la remise des messages (remise au plus une fois, exactement une fois). Ces aspects sont configurés au moyen de politiques, comme pour la configuration du POA (section 3.2.3).

La seconde l'autorise à indiquer à l'intergiciel une préférence pour l'utilisation d'un protocole plutôt qu'un autre, lorsque plusieurs sont utilisables pour contacter un objet donné, et qu'ils n'offrent pas les mêmes garanties de temps réel. Le comportement en cas de violation d'une contrainte de qualité de service, telle qu'un dépassement de délai, peut également être configuré : la violation peut être signalée par exemple au moyen d'une exception.

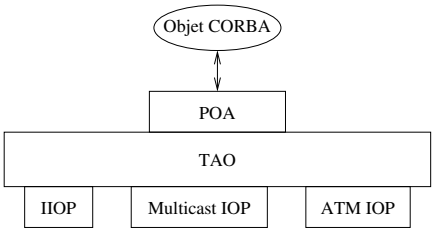


FIG. 3.6 – Interopérabilité avec les protocoles adaptables de TAO

Comme l'illustre la figure 3.6, les protocoles adaptables de TAO permettent à un objet défini dans le modèle de répartition CORBA d'interopérer avec des objets utilisant des protocoles différents : IIOP (GIOP sur TCP), Multicast IOP, ATM IOP.

L'existence d'une interface uniforme pour les protocoles permet également d'envisager la création de passerelles dynamiques (figure 3.7). En composant deux protocoles directement (sans faire intervenir de composant applicatif), on obtient un dispositif susceptible de recevoir une requête par un protocole, et

de la réémettre au moyen d'un second. Le nœud intermédiaire sur lequel cette composition est réalisée peut jouer le rôle de passerelle entre des composants applicatifs situés sur des nœuds tiers. Il permet à ces composants applicatifs d'interagir sans que leurs intergiciels respectifs partagent aucun protocole.

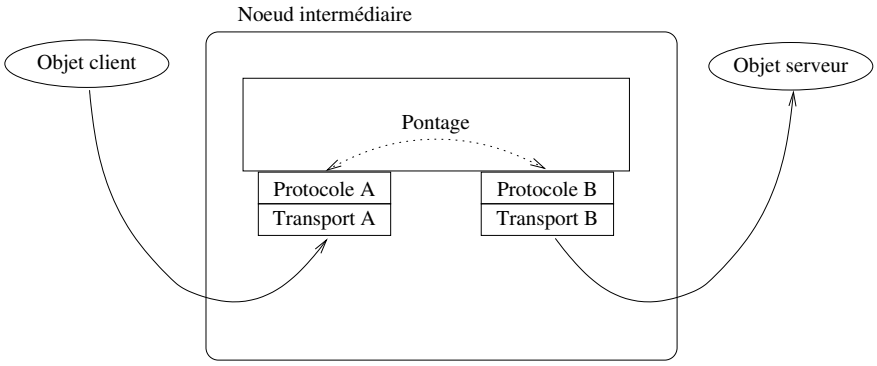


FIG. 3.7 – Pontage par composition de protocoles

Le modèle architectural de Jonathan permet d'éclairer l'interprétation de cette construction, en l'analysant comme un *chaînage d'objets de liaison*. Le chaînage d'objets de liaison consiste à étendre à un nombre quelconque de niveaux d'indirection la relation entre une référence et l'objet qu'elle désigne. Il a été proposé notamment par le projet SOR [107].

3.3.3 Interopérabilité par génération de passerelles : CIAO

CIAO *(CORBA Interface for Ada 95 distributed Objects)* est un générateur automatique de passerelles permettant à des clients CORBA d'accéder à des services définis en utilisant l'annexe des systèmes répartis d'Ada 95 (DSA). Contrairement aux passerelles dynamiques entre protocoles mentionnées dans la section 3.2.2, celles de CIAO sont réalisées comme des composants applicatifs. Nous avons réalisé cet outil à l'ENST, lors de travaux préparatoires à cette thèse [88, 89].

CIAO s'appuie sur plusieurs projets antérieurs et contemporains réalisés par notre équipe :

- GLADE, mise en œuvre de l'annexe des systèmes répartis d'Ada 95, décrite plus en détail dans la section 3.2.1 ;
- AdaBroker, un intergiciel CORBA pour Ada 95, décrit notamment dans [5].

Pour effectuer des requêtes sur un objet de l'annexe, l'utilisateur CORBA a besoin d'une description en OMG IDL du service offert. Ce contrat IDL peut être généré automatiquement si l'on dispose d'un schéma formel de traduction d'une spécification de paquetage de l'annexe vers OMG IDL.

Lors de l'exécution d'une application utilisant CIAO, les requêtes émises par les clients CORBA sont traitées par une passerelle, c'est-à-dire un composant logiciel se comportant à la fois comme un serveur CORBA, et simultanément comme un client dans l'annexe des systèmes répartis d'Ada 95. La passerelle est une mise en œuvre particulière du contrat IDL, dans laquelle le comportement associé à chaque méthode consiste à émettre la requête correspondante sur l'objet Ada 95.

Cette approche est similaire à celle déjà proposée par le projet ADEPT/JxA pour permettre l'interopérabilité entre des clients Java RMI et des serveurs DSA [33]. Comme LSA, elle comporte deux volets majeurs :

traduction des spécifications de services de DSA vers IDL ;

relayage des appels par les passerelles.

Dans CIAO, la génération du contrat IDL, basée sur un schéma de traduction formel, et la génération des composants passerelles sont automatisées.

Traduction des spécifications de services

De façon similaire à la première phase du projet LSA, décrite dans la section 3.3.1, nous avons donc défini une application de l'ensemble des spécifications de services de l'annexe (c'est-à-dire l'ensemble des arbres abstraits Ada 95 correspondant à la déclaration d'une unité de bibliothèque ayant l'une des catégories Pure, Remote_Types ou Remote_Call_Interface [49]) dans l'ensemble des contrats OMG IDL (l'ensemble des arbres syntaxiques OMG IDL) [88].

Cette application est définie dans le même esprit que la projection canonique d'OMG IDL vers Ada :

- Les constructions « orientées objets » d'Ada sont traduites en leurs équivalents IDL : un type objet réparti et le pointeur distant associé sont représentés par une *interface*, construction IDL correspondant à une déclaration de classe ; les primitives du type objet sont représentés par les opérations de cette interface.
- Un paquetage définissant des sous-programmes appelables à distance est traduits comme une interface ; chacun des sous-programmes du paquetage devient une opération de cette interface.
- Un pointeur distant sur sous-programmes est représenté par une interface ayant une seule opération « `Call` » permettant d'appeler le sous-programme désigné.

La traduction doit également conserver la structure générale du service : à chaque paquetage Ada correspond un module IDL ; un paquetage de bibliothèque est traduit par exactement un fichier source IDL ; si des sous-paquetages sont présents, ils sont traduits par des modules imbriqués.

Nous avons choisi, autant que faire se peut, de rester proches de la projection canonique d'OMG IDL vers Ada. Nous en avons dévié chaque fois que cela semblait nécessaire ou souhaitable, en considérant que le schéma de traduction devait permettre la mise en œuvre automatisée d'une passerelle entre des clients CORBA et le service de l'annexe, et que l'IDL généré doit être aisément lisible par un programmeur souhaitant écrire un client CORBA.

En exhibant un schéma de traduction formel des spécifications de services DSA vers OMG IDL, nous montrons que le modèle de répartition DSA peut être projeté dans celui de CORBA. Ce résultat prolonge les travaux du projet ADEPT, qui propose d'utiliser une variante du langage OMG IDL comme représentation intermédiaire lors de la compilation d'applications réparties DSA [110].

Contrairement au projet LSA, nous souhaitions appliquer notre schéma de traduction à toute application utilisant les systèmes répartis Ada 95. C'est pourquoi, après avoir défini un modèle de traduction des spécifications de service de l'annexe vers OMG IDL, nous avons réalisé un traducteur automatique conforme à ce modèle (cf. [89]).

Pour cela, il était nécessaire de disposer, sous une forme utilisable par un programme, de l'arbre sémantique d'une déclaration de paquetage Ada. Un tel mécanisme a été normalisé par l'ISO : ASIS (*Ada Semantic Interface Specification*) [52]. Il s'agit d'une interface de programmation permettant d'exploiter les représentations internes d'un compilateur Ada, sans avoir à redévelopper un analyseur syntaxique et sémantique.

Le traducteur CIAO génère un arbre abstrait IDL, à partir duquel le générateur de relais produit des paquetages convertissant les types de données CORBA projetés en types natifs Ada, ainsi que des paquetages de mise en œuvre d'interfaces CORBA. Le flot des données à travers l'outil CIAO est illustré par la figure 3.8.

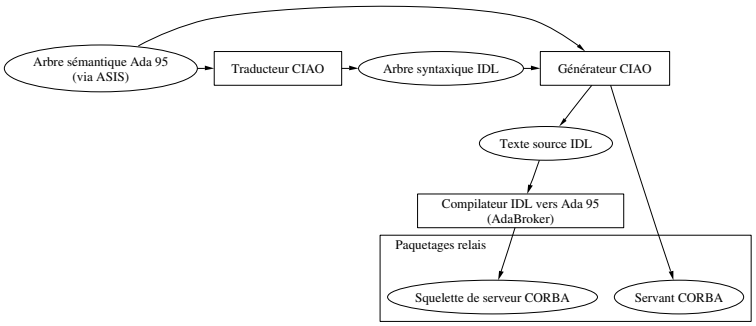


FIG. 3.8 – Flot des données dans l'outil CIAO

Fonctionnement de la passerelle

Pour obtenir une application Ada répartie interopérable avec CORBA, il reste à agencer les paquetages générés et à les enregistrer auprès d'un intergiciel CORBA, afin de constituer une *partition relais*, c'est-à-dire une partition au sens de l'annexe qui soit également un serveur contenant des objets CORBA.

Chaque paquetage-relais correspondant à un paquetage de l'annexe comporte une procédure d'initialisation du service CORBA. La procédure principale de la partition relais commence par initialiser l'intergiciel, puis elle appelle la procédure d'initialisation des différents relais.

Les paquetages Remote_Call_Interface définissent des sous-programmes appelables à distance depuis n'importe quelle partition. La procédure d'initialisation pour ces unités :

- instancie un unique objet concret ;
- demande à l'intergiciel d'activer cette instance et de lui attribuer une référence ;
- associe cette référence à un nom basé sur celui du paquetage, au sein du service de dénomination CORBA *COS Naming*.

Tout client CORBA peut alors obtenir la référence d'objet correspondant à l'interface qui traduit un paquetage Remote_Call_Interface.

De telles références sont essentielles pour l'initialisation de l'application. En effet, dans un système réparti, chaque participant doit utiliser un service du système pour prendre connaissance d'une « première » référence d'objet, qui lui permette ensuite d'interroger un service de dénomination, un annuaire

ou autre service fournisseur de références d'objets. Dans l'annexe, ce sont les paquetages Remote_Call_Interface qui jouent ce rôle, car un paquetage Remote_Call_Interface est instancié exactement une fois, sur une partition bien déterminée, dans une application Ada répartie ; à l'exécution, les paquetages Remote_Call_Interface s'enregistrent auprès du sous-système de communication au moment de leur élaboration, afin que chaque partition soit en mesure de les localiser. Ils peuvent alors donner aux autres partitions le moyen de publier et d'acquérir des références d'objets.

Ce mécanisme d'initialisation, illustré par la figure 3.9, est une solution au problème d'initialisation que nous avons évoqué dans la section 2.2.3.

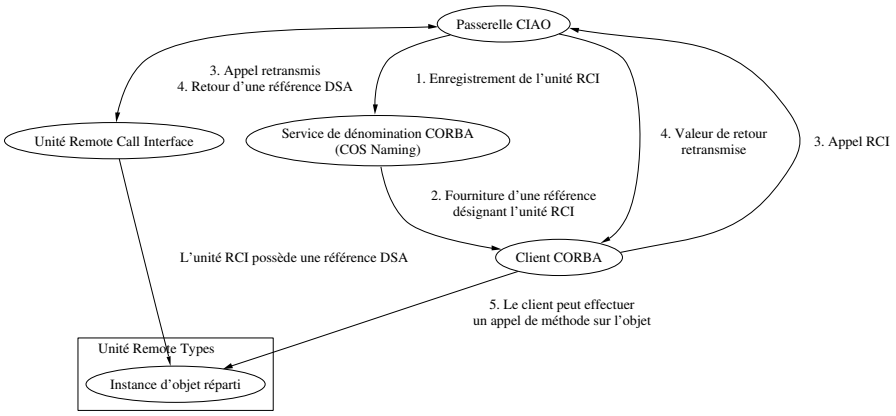


FIG. 3.9 – Initialisation d'une application CIAO

Ainsi, dans une application utilisation CIAO, une référence CORBA est associée à tout paquetage Remote_Call_Interface. Un client CORBA peut alors interroger le service de dénomination CORBA pour obtenir cette référence, puis effectuer des appels sur ce paquetage pour obtenir des références d'objets répartis.

Pour un paquetage définissant un type objet réparti Ada, le paquetage relais contient, là encore, une mise en œuvre d'objet CORBA. Un seul servant existe pour tous les objets Ada d'un même type. Nous enregistrons ce servant dans un nœud de l'arborescence des POA créé spécialement, mettant en œuvre la politique d'activation avec servant par défaut (décrite dans la section 3.2.3).

Ainsi, tout appel de méthode sur un objet de cette classe est traité comme un appel sur une méthode du servant. Lors de la conversion des arguments CORBA vers les types Ada correspondants, l'exécutif de CIAO extrait un identificateur d'objet. La spécification du POA permet d'utiliser toute séquence d'octets comme identificateur. Nous y indiquons donc tout simplement la référence d'objet DSA, sous sa forme emballée par la bibliothèque d'exécution de GLADE. Le relais est donc sans état (il n'a pas besoin de maintenir une liste des associations entre références CORBA et DSA) : il peut être arrêté et redémarré à volonté, et traiter un nombre quelconque d'objets sans augmentation de son empreinte mémoire.

Limitations de CIAO

Le prototype fonctionnel CIAO démontre la faisabilité d'une projection des objets DSA dans le modèle de répartition de CORBA. Cependant, il présente un certain nombre de limitations, que nous détaillons ci-après.

Types de données opacifiés Le schéma de traduction proposé traduit certaines constructions du langage Ada 95, en particulier les types objets (non répartis), ou *types étiquetés* (*tagged types*), comme des flots d'octets opaques pour le client CORBA. Une partie de la sémantique des services DSA est donc inaccessible pour ces clients. De même, dans le cadre de DSA, l'utilisateur peut toujours fournir lui-même les fonctions qui transforment un type de données en une représentation transportable par le réseau. Cette indication est donnée au compilateur sous la forme de *clauses de représentation*. Lorsqu'un type Ada possède une telle clause de représentation, il est également traduit comme un flot d'octets opaque.

Passerelle unidirectionnelle La passerelle offerte par CIAO entre les modèles de répartition CORBA et DSA est unidirectionnelle. Elle permet seulement à des clients CORBA d'effectuer des appels de méthodes sur des services DSA ; elle ne permet pas à des objets DSA de faire appel à des services offerts par des objets CORBA.

Étranglement du circuit des requêtes Le composant passerelle constitue un point de passage obligé pour toute requête émise par un objet CORBA à destination d'un objet DSA. Ce composant est généré automatiquement, et dépendant de l'application. Il introduit un coût supplémentaire lors de chaque appel de méthode, par rapport à un appel à distance entre deux objets du même modèle de répartition. Ce coût correspond à la réception d'une requête, à la conversion de ses arguments des types CORBA vers les types natifs Ada 95, et à sa réémission vers l'objet destinataire réel.

De plus, la passerelle est un point de faiblesse de l'architecture : la défaillance de ce seul composant entraîne une indisponibilité de l'application. La mise en place de passerelles redondantes pour améliorer la disponibilité et la tenue en charge poserait par ailleurs le problème de la cohérence des références CORBA attribuées aux objets DSA. En effet, ces références sont spécifiques d'une passerelle particulière. Un même objet, vu à travers deux passerelles différentes, se verrait donc attribuer deux références distinctes et non interchangeables.

3.3.4 Trois perspectives sur l'interopérabilité

Comme le montrent les travaux existants que nous avons présenté, l'interopérabilité peut être considérée sous trois perspectives différentes dans leurs moyens et leurs objectifs.

Multiples interfaces pour les objets applicatifs

Le projet LSA offre un premier exemple d'interopérabilité. Il confirme qu'il est possible de définir une projection *ad hoc* des interfaces applicatives de deux personnalités de répartition (Java RMI et HPC++) sur le modèle d'échanges d'un même protocole de communication entre intergiciels : le sous-système de communication Nexus. Ce sous-système est *mutualisé* entre la mise en œuvre

Nexus RMI de Java RMI et une mise en œuvre du modèle de répartition de HPC++.

Dans SoapRMI, le sous-système de communication n'est pas mutualisé, mais deux mises en œuvre du même protocole d'appel de méthode à distance, SOAP, sont réalisées. La situation est donc similaire à celle de LSA.

Nous observons, dans les deux cas, que la mise en commun de composants de communication (un trait architectural des intergiciels génériques) ou, pour SoapRMI, l'utilisation de composants de communication compatibles, contribue à l'interopérabilité. En effet, des objets applicatifs développés en utilisant des modèles de répartition différents peuvent interagir par le truchement du sous-système de communication mis en commun : la mutualisation entraîne un découplage favorable à l'interopérabilité. Les figures 3.3 et 3.4 illustrent cette possibilité d'interopérabilité entre modèles de répartition.

Dans CIAO, nous réalisons également l'interopérabilité entre des interfaces applicatives de personnalités différentes, en permettant à des clients CORBA d'accéder à des objets DSA. Pour ce faire, nous proposons l'utilisation d'un générateur de composants passerelles. Ces composants se comportent eux-mêmes comme des objets applicatifs, et réutilisent des mises en œuvre existantes des modèles de répartition considérés.

Ces projets sont destinés à permettre l'interopérabilité entre des composants existants, utilisant chacun un modèle de répartition et une personnalité d'intergiciel donnés. Ils nécessitent le déploiement d'intergiciels spécifiquement modifiés pour autoriser l'interopérabilité sur certains nœuds de l'application répartie, ou bien l'introduction de nœuds passerelles.

Multiples personnalités protocolaires

L'architecture de TAO pose les fondations nécessaires pour le support de multiples protocoles dans un même intergiciel. Ce projet démontre qu'un même modèle de répartition, celui de CORBA, peut être mis en œuvre par un intergiciel qui l'associe à divers protocoles, sans que les objets applicatifs aient à être adaptés spécifiquement. De plus, ces protocoles peuvent exister simultanément au sein d'un même intergiciel.

Lorsque celui-ci héberge un objet serveur, cet objet devient accessible à travers plusieurs protocoles simultanément. Il peut servir des clients qui utilisent chacun le protocole et le moyen de transport de données le plus adapté à ses besoins.

Lorsqu'il héberge un objet client, il permet à cet objet de contacter des serveurs utilisant des protocoles différents, et ce à faible coût, car tous les services communs sur lesquels sont construits les protocoles sont mutualisés. Lorsqu'un serveur supporte également plusieurs protocoles, l'utilisateur peut contrôler le choix de celui qui sera utilisé.

Cette architecture de protocoles adaptables permet donc à un objet CORBA d'interopérer avec des objets tiers accessibles à travers des protocoles différents, et ainsi de s'intégrer à des applications réparties hétérogènes, *a priori* incompatibles, comme l'illustre la figure 3.6.

Modèle unificateur pour les intergiciels interopérables

Dans les deux sections précédentes, nous avons évoqué la norme CORBA. Dans CIAO, nous montrons que son modèle de répartition peut être mis en correspondance avec celui de DSA. Dans LSA et SoapRMI, divers modèles sont projetés sur celui de Java RMI, qui lui-même peut être projeté sur CORBA [72]. Dans TAO, le modèle CORBA sous-tend l'ensemble de l'architecture de protocoles adaptables.

Le modèle de répartition et les protocoles de CORBA font l'objet de spécifications détaillées et publiques, librement accessibles. Ce modèle est non seulement adapté aux applications traditionnelles client/serveur, mais peut également être étendu aux applications temps réel et à un large spectre d'applications nécessitant des garanties de qualité de service variées, comme le montre en particulier TAO.

Le modèle de répartition de CORBA apparaît comme le *paradigme de choix* pour unifier les modèles de répartition et permettre leur interopérabilité [13, 61]. La convergence du modèle de répartition de Java RMI vers celui de CORBA confirme la pertinence de cette perspective. En effet, l'OMG a normalisé une projection de Java RMI vers l'IDL de CORBA [72].

CORBA 2.3 introduit la notion de *type par valeur* (*valuetype*). De tels types de données correspondent à des objets complexes, susceptibles d'être liés par des relations de référencement et d'héritage et de recevoir des appels de méthode, mais qui sont toujours transmis par valeur lors d'appels à distance. Les types par valeur constituent une solution élégante aux problèmes de passage d'objets soulevés par LSA et aux limitations de CIAO relatives aux types étiquetés. De plus, la norme CORBA prévoit que la représentation des types par valeur puisse être définie par l'utilisateur. Ils permettent donc de traduire les clauses de représentation problématiques également évoquées dans la section 3.3.3.

3.3.5 Discussion

LSA, SoapRMI et CIAO permettent l'interopération entre des objets applicatifs conçus au moyen de personnalités et de modèles de répartition différents, soit par l'utilisation de protocoles communs, soit par l'établissement de passerelles *ad hoc*.

L'adaptation de protocoles dans TAO permet à des objets applicatifs d'interagir avec d'autres objets utilisant des protocoles divers.

Un intergiciel destiné à favoriser l'interopérabilité entre modèles de répartition doit à la fois :

- être susceptible d'héberger des objets créés au moyen de modèles de répartition divers ;
- permettre à ces objets d'utiliser des protocoles différents pour interagir avec d'autres entités.

Les intergiciels interopérables que nous avons évoqués ne satisfont pas simultanément à ces conditions. En particulier, les intergiciels offrant des protocoles multiples (tels que TAO) sont spécifiques d'un modèle de répartition donné. Pour étendre le champs de l'interopérabilité aux interactions entre objets issus de modèles de répartition différents, il est nécessaire d'intégrer ces deux aspects au sein d'une même architecture unifiée.

Cette architecture doit elle-même proposer son propre modèle de répartition

généralisé, afin qu'il soit possible d'y projeter les modèles et protocoles existants de manière efficace. Nous nous proposons de définir ce modèle unifié à partir de celui de CORBA, paradigme de choix pour l'interopérabilité.

3.4 Synthèse

Dans le chapitre 2, nous avons établi la nécessité de définir une architecture d'intergiciel interopérable. Nous avons dégagé trois classes de propriétés que cette architecture doit posséder pour contribuer à la résolution efficace des questions d'interopérabilité entre intergiciels : généricité, configurabilité et interopérabilité proprement dite.

Nous avons illustré ces trois aspects par des projets existants, et pour chaque aspect nous avons discuté les choix de réalisation qu'ils proposent. Le tableau 3.1 donne une indication synthétique sur la réalisation de ces propriétés dans les intergiciels que nous avons présentés.

Propriété \ Projet	Jonathan	Quarterware	GLADE	TAO	LSA, SoapRMI	CIAO
Généricité	+	+				
Configurabilité		+	+	+	+	
Interopérabilité				+	+	+

TAB. 3.1 – Propriétés remarquables des intergiciels existants

3.4.1 Généricité

Plusieurs projets (Jonathan, Quarterware) mettent en évidence la faisabilité d'un *intergiciel générique* présentant plusieurs personnalités. Chacune d'entre elles offre des interfaces au moyen desquelles les objets de l'application et les intergiciels tiers utilisent les services des composants sous-jacents. Un même ensemble de fonctionnalités sous-jacentes permet effectivement d'offrir les abstractions constitutives des différentes personnalités telles que les objets répartis de CORBA et Java RMI, ainsi que le passage de messages de MPI.

L'identification à grain plus fin des fonctions élémentaires de l'intergiciel (notamment proposée dans le cadre de la notion d'intergiciel RISC de Quarterware) donne la base d'une modularisation de l'intergiciel. Pour chaque fonction, une interface abstraite doit être définie, avant de proposer une ou plusieurs mises en œuvre interchangeables.

En particulier, les protocoles de communication entre intergiciels doivent être modularisés. Ils doivent offrir une interface uniforme, afin de pouvoir être adaptés dans une architecture permettant leur coexistence ; plusieurs protocoles peuvent ainsi utilisés simultanément au sein de la même instance d'intergiciel.

De plus, l'utilisation d'une interface uniforme similaire à celle de l'invocation dynamique de CORBA (DII) permet une conception générique des mises en

œuvre de protocole. Comme l'illustre le projet CorbaWeb, elle permet de réaliser le traitement d'appels de méthodes de façon générique par rapport à l'interface des objets applicatifs, sans qu'il soit nécessaire de générer du code spécifique.

Configurabilité

La notion de liaisons flexibles entre objets mise en œuvre dans Jonathan permet également d'offrir à l'utilisateur le contrôle de certains paramètres des objets de liaisons employés pour mettre en relation deux objets. Elle formalise le rôle de l'objet de liaison comme *subrogé* de l'objet cible. Elle autorise des modalités de relations entre objets plus évoluées que la communication point à point directe : des liaisons point vers multipoint peuvent être établies, ainsi que des communications indirectes à travers des composants relais.

Les applications temps réel et embarquées suscitent le besoin *d'intergiciels configurables* (Quarterware, GLADE, TAO). L'utilisation de composants modulaires composables à volonté et interchangeables est une technique importante pour favoriser la configurabilité d'un intergiciel. Elle peut tirer parti des décompositions des fonctionnalités issues des intergiciels génériques.

Nous observons que, parmi les intergiciel configurables, deux projets ressortent et offrent des fonctionnalités qui préfigurent les traits nécessaires à l'interopérabilité entre modèles de répartition :

Quarterware est générique : il peut être instancié pour fournir différentes personnalités ;

TAO est interopérable : il peut supporter simultanément plusieurs protocoles.

L'utilisation de patrons de conception et de constructions réflexives à l'intérieur de ces intergiciels permettent de réaliser efficacement des aspects de personnalité variés par composition de modules élémentaires.

Interopérabilité

Enfin, nous avons présentés des *intergiciels interopérables* existants (TAO, LSA, SoapRMI, CIAO). Ceux-ci autorisent des interactions de deux ordres :

- entre objets utilisant des modèles de répartition différents mais un protocole commun(TAO, CIAO) ;
- entre objets utilisant le même modèle de répartition, mais des protocoles différents (LSA, SoapRMI).

Pour réaliser l'interopérabilité entre modèles de répartition, il est nécessaire d'intégrer ces deux aspects comme des traits fondamentaux d'une architecture d'intergiciel. À défaut, les solutions proposées sont sujettes à une combinatoire rédhibitoire.

La personnalisation de l'interface entre intergiciel et objets de l'application peut avantageusement être considérée comme un cas particulier de configuration. De la même façon, le choix d'un protocole et d'un moyen d'échange de messages avec un intergiciel tiers peut être considéré comme une opération de personnalisation des fonctions de communication. Dans une architecture d'intergiciel générique, ces aspects peuvent être modulés par l'utilisateur : l'architecture peut être instanciée suivant différentes personnalités.

En alliant généricité et configurabilité, Quarterware propose une stratégie de décomposition avancée, qui permet d'identifier quelles fonctionnalités d'un

intergiciel peuvent faire l'objet de plusieurs mises en œuvre interchangeables, correspondant à des modèles de répartition différents. Cette stratégie permet d'offrir aux composants applicatifs de multiples interfaces pour accéder aux fonctionnalités de répartition, et ce de manière efficace.

L'architecture de TAO, configurable et interopérable par support de protocoles multiples, offre la possibilité aux composants applicatifs d'interagir avec des objets utilisant divers protocoles. Là encore, les modules communs aux divers protocoles sont factorisés, et de plus la coexistence entre personnalités différentes au sein d'un même intergiciel est introduite.

La résolution des problèmes liés à l'interopérabilité entre modèles de répartition nécessite l'intégration des deux aspects d'interopérabilité (modèles de répartition et protocoles), à travers un modèle unificateur. Nous concluons, à la lumière des travaux existants, que l'intégration efficace de ces aspects nécessite une nouvelle architecture d'intergiciel, qui soit simultanément :

interopérable, conçue pour permettre la coexistence de multiples modèles de répartition au sein de la même instance d'intergiciel ;

générique pour factoriser le code indépendant des personnalités et prévenir l'explosion combinatoire en découplant les différents aspects de personnalité ;

configurable pour adapter et optimiser les composants de l'intergiciel par rapport aux contraintes des applications, et limiter la lourdeur de mise en œuvre liée aux aspects d'interopérabilité et de généricité.

CHAPITRE

4 Conception d'un intergiciel schizophrène

Sommaire

4.1 Intergiciel schizophrène 86
4.1.1 Motivation 86
4.1.2 Approches pour l'interopérabilité 87
4.1.3 Définition de la notion d'intergiciel schizophrène . . 89
4.2 Application témoin 90
4.2.1 Configuration homogène 91
4.2.2 Configuration d'interopérabilité avec protocole unique 92
4.2.3 Configuration d'interopérabilité avec protocoles multiples 92
4.2.4 Configuration d'interopérabilité indirecte 93
4.3 Traits architecturaux d'un intergiciel schizophrène 95
4.3.1 Découplage entre les aspects protocolaires et applicatifs 95
4.3.2 Introduction d'une couche neutre 96
4.3.3 Architecture générale 97
4.4 Une architecture d'intergiciel schizophrène 98
4.4.1 Vue d'ensemble 98
4.4.2 Composants de support 99
4.4.3 Couche neutre 100
4.4.4 Personnalités applicatives 101
4.4.5 Personnalités protocolaire 102
4.5 Projection des fonctions sur l'architecture 104
4.5.1 Adressage 104
4.5.2 Transport 105
4.5.3 Liaison 106
4.5.4 Représentation 107
4.5.5 Protocole 109
4.5.6 Activation 110
4.5.7 Exécution 111
4.6 Processus de personnalisation 111

4.7 Patrons de conception 113
4.7.1 Patrons comportementaux 113
4.7.2 Patrons de parallélisme 114
4.7.3 Patrons de polymorphisme 116
4.7.4 Synthèse 118
4.8 Synthèse 119

NOUS INTRODUISONS dans ce chapitre la notion d'intergiciel *schizophrène*. Nous appuyons la nécessité de cette notion par un cas d'application témoin, qui met en évidence l'apport de cette notion pour la réalisation de systèmes répartis interopérables.

4.1 Intergiciel schizophrène

Dans cette section, nous rappelons les motivations qui conduisent à rechercher les moyens de rendre interopérables les systèmes répartis. Nous proposons deux approches pour répondre à ce besoin. Nous dégageons en particulier une approche introduisant l'interopérabilité au cœur de la mise en œuvre des modèles de répartition : l'architecture d'intergiciel schizophrène.

4.1.1 Motivation

Ces dernières années, le développement des applications réparties a fait naître un large spectre de besoins et de contraintes. Les intergiciels, qui constituent l'infrastructure de ces applications, doivent évoluer au rythme de ces exigences et offrir une palette de services de plus en plus étendue.

Ainsi, le secteur des télécommunications intègre progressivement la technologie des applications réparties. Dans les réseaux intelligents, des calculateurs permettent à l'utilisateur de contrôler finement le processus de traitement d'appel du réseau. La mobilité des terminaux suppose également une coordination importante entre équipements de réseau. Ce type de service, utilisant des intergiciels pour relier les entités participantes, exige des garanties fortes de traitement en temps réel. De nombreux travaux visent à intégrer ces garanties dans les intergiciels [77].

La nécessité d'intégrer aux applications des systèmes embarqués tels que des terminaux mobiles ou des robots autonomes crée, par ailleurs, une demande pour des intergiciels fonctionnant avec des ressources restreintes. Un encombrement en mémoire minimal, et la possibilité de fonctionner avec un système d'exploitation présentant un profil de fonctionnalités réduit, sont essentiels pour ces applications [25, 37].

La croissance des services en ligne, en particulier sur Internet, suscite également une évolution des contraintes qui s'imposent aux entreprises. Le nomadisme des clients impose la mise en place de procédures d'accès à distance sécurisées, pour parer aux intrusions. La relation entre interlocuteurs emprunte des routes généralement placées sous le contrôle de plusieurs entités indépendantes (par exemple entreprises, fournisseurs d'accès, fournisseurs de transit). Les paramètres des liaisons (débit, latence, pertes, symétrie) ne peuvent donc pas, en général, être garantis de bout en bout. De plus, l'ouverture des services à un très large public les rend vulnérables à des attaques par déni de service.

Les entreprises tendent à s'organiser comme des fédérations d'entités spécialisées. Chaque entité utilise des logiciels adaptés à son métier et à ses spécificités ; des intergiciels adaptés permettent leur interconnexion.

Des architectures applicatives de haut niveau sont créées créées en combinant des composants autonomes et en utilisant les fonctionnalités des intergiciels pour leur faire échanger des messages.

Des modalités de communications variées sont offertes, au-delà de la relation traditionnelle client/serveur. Le mode de communication par *publication/abonnement* (*publish/subscribe*) formalise une relation entre des composants faiblement couplés, qui peuvent être assemblés de manière flexible. Ainsi, la plateforme de contrôle flexible OCP [120] et les JavaBeans de Sun [20] permettent le développement et la reconfiguration d'applications par composition de composants au moyen de canaux d'événements.

À travers ces exemples, nous observons que la diversité des contraintes et des situations ne permet pas le choix d'un modèle de répartition et d'un intergiciel uniques, idéalement adaptés dans toutes les situations. Chaque application a un modèle « de choix » qui répond spécifiquement à des exigences particulières.

Par conséquent, l'intégration de composants d'origines diverses au sein d'une même application entraîne la nécessité de faire interagir des entités définies dans des modèles de répartition différentes.

4.1.2 Approches pour l'interopérabilité

Dans la section 2.3, nous avons défini les questions qui se posent au développeur d'une application répartie lorsqu'il souhaite intégrer dans celle-ci des composants utilisant des modèles de répartition différents ou des intergiciels incompatibles.

Nous proposons ici deux classes de réponses possibles à ces questions, et nous mettons en évidence la nécessité de retenir une solution permettant la mise en relation dynamique de modèles de répartition hétérogènes.

Approche basée sur les passerelles statiques

La mise en place de passerelles *ad hoc* entre modèles de répartition a été envisagée comme réponse possible à ces questions. Le travail préparatoire sur CIAO, décrit dans la section 3.3.3, a porté sur la génération automatique de passerelles entre l'annexe des systèmes répartis du langage Ada 95 et les objets répartis CORBA. Nous avons évoqué dans la section 3.3.3 les limitations liées à CIAO en particulier. Cependant, il existe également des problèmes plus généraux, inhérents à l'utilisation de passerelles statiques entre modèles de répartition.

En premier lieu, un nouveau générateur de passerelles doit être développé pour chaque couple de modèles de répartition pour lequel l'interopérabilité est souhaitée. Le coût de cette approche, croît donc en fonction quadratique du nombre d'environnements supportés. Cette explosion combinatoire limite l'intérêt des passerelles comme solution générique pour l'interaction avec des composants hétérogènes.

Par ailleurs, aucune mise en commun de code n'est effectuée entre les intergiciels correspondant à chaque modèle supporté, ni entre les différentes passerelles. À l'exécution, l'empreinte d'utilisation de ressources de chaque passerelle

est donc très importante. Un point de passage unique entre modèles de répartition est introduit, qui est à la fois un facteur limitant des performances de l'application, et un facteur de vulnérabilité aux défaillances.

L'approche basée sur les passerelles statiques, si elle constitue un premier élément de réponse au problème de l'interopérabilité, n'est donc pas une solution pleinement satisfaisante. Il est nécessaire d'explorer d'autres voies pour obtenir des systèmes répartis interopérables performants, flexibles et sûrs.

Intergiciels à personnalités multiples

Pour lever les problèmes fondamentaux de l'approche précédente, nous proposons de mettre en place directement au sein de l'intergiciel les mécanismes nécessaires à l'intégration de multiples modèles de répartition.

Une solution naïve consiste à constituer un intergiciel en agrégeant plusieurs intergiciels existants, correspondant chacun à une personnalité. Une telle construction soulève plusieurs difficultés :

- Elle suppose que le code source des divers intergiciels utilisés est disponible, et que les langages, les outils et les bibliothèques qu'ils mettent en jeu sont compatibles.
- Les différences d'architectures et de structures de données limitent les possibilités de mutualisation de code.
- Les intergiciels conçus pour ne proposer qu'une seule personnalité sont peu adaptés à la mise en commun de structures internes, car ils sont conçus sur l'hypothèse que celles-ci ne seront manipulées que par une seule personnalité.

Les intergiciels personnalisables offrent un premier élément de résolution de ces problèmes. Ils identifient des composants génériques, indépendants d'un modèle de répartition. Ceux-ci sont mis en œuvre sous forme d'une « boîte à outils » dont les services sont utilisés par des composants spécifiques d'une personnalité. Ces derniers réalisent un modèle de répartition fixé. Un noyau neutre, constitué par les composants génériques, est ainsi délimité, autour duquel sont configurés les composants de personnalisation.

Cependant, si ces architectures réduisent l'ampleur des problèmes posés par l'agrégation de multiples intergiciels, ceux-ci demeurent présents. Nous avons observé, dans la section 3.1.3, que dans les mises en œuvre existantes d'intergiciels personnalisables, le partage de code entre personnalités est limité.

La réalisation d'une nouvelle personnalité demeure un travail considérable, qui se traduit pratiquement par le redéveloppement d'un intergiciel complet. De plus, la question de l'utilisation coordonnée de ressources internes par plusieurs personnalités demeure entière. Un couplage important entre tous les composants spécifiques d'une personnalité est encore observé, ainsi qu'une faible possibilité d'interaction entre personnalités différentes.

Découplage des aspects de personnalité

Le rôle de l'ensemble des intergiciels qui participent à une application répartie consiste à mettre en relation les objets qui la constituent. À cet effet, les intergiciels établissent entre eux des échanges de messages représentant les requêtes et les réponses échangées par les objets.

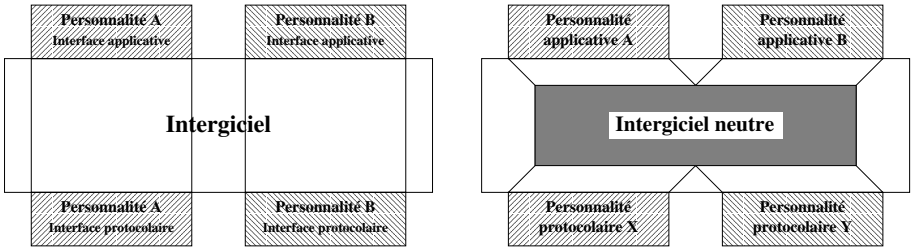
Pour satisfaire d'une manière générale les besoins d'interopérabilité entre modèles de répartition, il est nécessaire de lever les obstacles liés à l'agrégation d'intergiciels ou à l'agrégation d'instances d'un intergiciel personnalisable. En particulier, il est nécessaire de briser le couplage entre les deux aspects d'interface de chaque intergiciel :

- l'interface *applicative* présentée aux objets qu'il héberge localement et rend accessibles à distance ;
- l'interface *protocolaire* qu'il présente aux intergiciels distants.

La figure 4.1 illustre la différence entre les architectures qui agrègent des intergiciels et les architectures qui en découplent les composants. Dans l'approche par agrégation (4.1(a)), les composants de chaque intergiciel agrégé sont fortement couplés entre eux. L'interaction entre personnalités est limitée par l'architecture des intergiciels agrégés.

Dans l'approche par composants découplés (4.1(b)), au contraire, les composants de personnalisation des deux aspects d'interface sont découplés :

- Les composants réalisant l'interface entre les objets d'application et l'intergiciel n'utilisent que des services génériques de l'intergiciel. Ils n'effectuent aucun traitement, ne manipulent aucune représentation, qui soit spécifique d'un protocole particulier.
- De même, les composants réalisant un protocole ne font aucune hypothèse particulière sur le choix d'une interface entre objets de l'application et intergiciel.



(a) Couplage fort (b) Couplage faible

FIG. 4.1 – Intergiciels à personnalités multiples

4.1.3 Définition de la notion d'intergiciel schizophrène

Nous avons identifié dans la section 4.1.1 les conditions que doit remplir l'architecture d'un intergiciel pour permettre la coopération aisée et dynamique entre modèles de répartition :

- le découplage des aspects applicatifs et protocolaires des éléments de personnalité ;
- la cohabitation de personnalités différentes au sein de l'architecture ;
- le partage des services et des données dans un noyau indépendant de toute personnalité.

Le découplage des aspects applicatifs et protocolaires des intergiciels permet d'introduire deux notions complémentaires :

Définition 18 (personnalité applicative) *Une personnalité applicative est la spécification d'une interface entre des objets applicatifs et un intergiciel en vue d'en utiliser les services pour interagir avec d'autres objets applicatifs.*

Définition 19 (personnalité protocolaire) *Une personnalité protocolaire est la spécification d'une interface entre deux intergiciels destinée à l'échange de messages représentant les interactions entre les objets hébergés par ces intergiciels.*

Nous synthétisons ces conditions (découplage des personnalités applicatives et protocolaires, cohabitation de personnalités différentes, mutualisation de fonctionnalités) dans cette définition [79] :

Définition 20 (intergiciel schizophrène) *Nous qualifions de* schizophrène *un intergiciel capable de disposer, simultanément, de plusieurs personnalités applicatives et protocolaires et de les faire interagir efficacement.*

Dans la suite de ce chapitre, nous présentons une application témoin qui met en évidence les apports d'un intergiciel schizophrène pour l'interopérabilité entre objets répartis, puis nous présentons plus en détail l'architecture schizophrène que nous avons réalisée.

4.2 Application témoin

Dans cette section, nous décrivons une application utilisant des objets répartis, qui met en évidence les problèmes d'interopérabilité que nous avons évoqué dans le chapitre 2. Cette application nous servira à montrer que l'architecture d'intergiciel schizophrène que nous proposons permet de répondre aux besoins de cette application.

Considérons une application d'échange et de distribution de messages courts entre utilisateurs. Cette application comporte deux types d'objets :

- un *tableau d'affichage (Bulletin Board)* sur lequel chaque utilisateur peut déposer un message public destiné à l'ensemble des utilisateurs ;
- des *correspondants (Penpals)*, objets représentant les utilisateurs. Les correspondants émettent des messages (à destination de tout le monde ou bien d'un utilisateur en particulier) et en reçoivent.

La figure 4.2 présente les relations entre ces objets sous la forme d'un diagramme de classes UML.

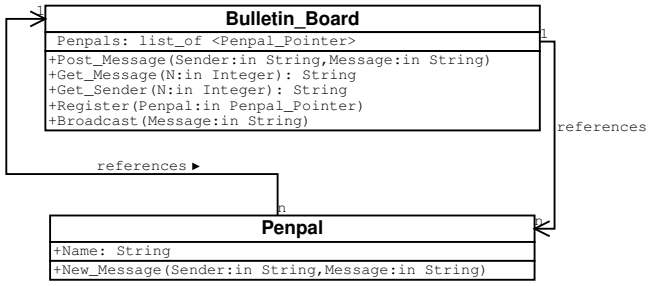


FIG. 4.2 – Objets de l'application témoin

Nous allons proposer trois configurations de cette application mettant en évidence la nécessité de l'interopérabilité.

4.2.1 Configuration homogène

Supposons en premier lieu que les deux objets ont été mise en œuvre au moyen d'une personnalité applicative A_0, en utilisant un protocole P pour la communication entre les partitions (par exemple avec $A_0 = \text{DSA}$ et $P = \text{SOAP}$. La configuration est alors celle de la figure 4.3.

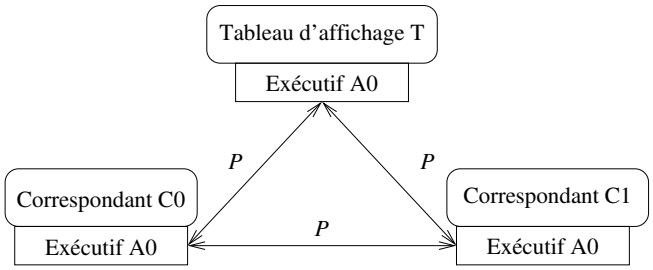


FIG. 4.3 – Configuration homogène

Dans cette configuration, l'architecture schizophrène réalise les même fonctions qu'un intergiciel traditionnel non schizophrène. Un appel de méthode se déroule suivant la chronologie suivante :

1. L'objet serveur étant enregistré auprès d'une personnalité applicative, la fonction d'*adressage* lui associe une référence contenant des profils (structures de données associant la désignation d'un point d'accès de transport et d'une personnalité protocolaire de l'intergiciel serveur, ainsi que l'identificateur localement assigné à l'objet).
2. L'objet client ayant pris connaissance de cette référence, il fait appel à la fonction de *liaison* pour obtenir un subrogé de l'objet cible au moyen de cette référence. Une connexion de transport est établie, et une session de protocole débutée sur cette connexion.
3. La requête à exécuter ayant été mise sous forme neutre par la personnalité applicative A_0 du client, la fonction de *représentation* P traduit ses paramètres en un flot de données transportable.
4. La fonction de *protocole* P construit un message signalant la requête et contenant la représentation de ses paramètres.
5. La fonction de *transport* est utilisée pour faire parvenir ce message à l'intergiciel serveur.
6. Les fonctions de transport, protocole et représentation interviennent symétriquement côté serveur pour reconstituer la représentation neutre de la requête, comportant l'identité de l'objet destinataire et de la méthode à exécuter, ainsi que des paramètres.
7. La fonction d'*activation* de l'intergiciel serveur détermine (et crée au besoin) l'objet physique (le *servant*) destiné à incarner l'objet cible de l'appel.
8. La fonction d'*exécution*, enfin, affecte le traitement de l'appel de méthode à une tâche ;
9. Une fois l'appel exécuté par le servant et le résultat connu, celui-ci est retourné à l'appelant par le circuit inverse.

4.2.2 Configuration d'interopérabilité avec protocole unique

Dans le scénario homogène, la personnalité applicative d'un nœud peut être remplacée par une autre ; si l'intergiciel utilise toujours le même protocole P, le nœud concerné demeure en mesure d'interagir avec tous les autres.

On aboutit ainsi dans la configuration de la figure 4.4, dans laquelle les correspondants C_0 et C_1, utilisant respectivement les personnalités applicatives A_0 et A_1, interagissent au moyen du protocole commun P.

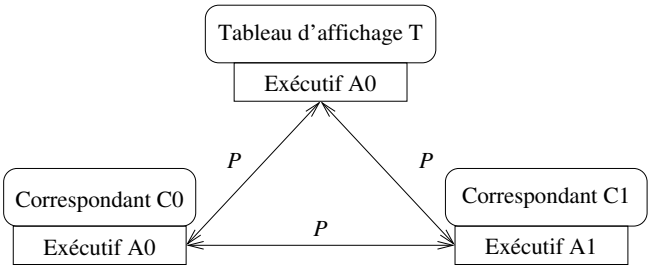


FIG. 4.4 – Configuration à objets interopérables avec protocole commun

4.2.3 Configuration d'interopérabilité avec protocoles multiples

Pour factoriser le coût de développement et de maintenance des objets applicatifs et profiter du travail déjà réalisé pour leur conception et leur réalisation, on se propose maintenant de faire participer à l'application des objets d'une autre personnalité applicative A_1 (par exemple CORBA) dans le rôle « Correspondant ».

Ces objets sont réalisés au moyen d'un intergiciel tiers utilisant un protocole P' (par exemple GIOP), comme illustré par la figure 4.5. Contrairement au cas précédent, on ne peut imposer à ces objets préexistants l'utilisation du protocole P.

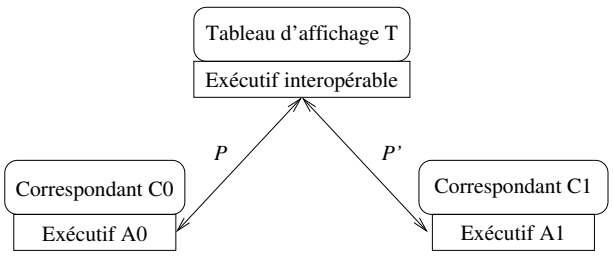


FIG. 4.5 – Interopérabilité par coexistence de protocoles multiples

À cet effet, on se propose d'ajouter une première forme d'interopérabilité à l'objet « Tableau d'affichage », en l'associant à un intergiciel supportant plusieurs protocoles simultanément. Dans ce cas, des points d'accès de transport distincts sont créés lors de l'initialisation de chacune de ces personnalités. Chaque point d'accès est associé à une usine d'objets de liaison qui instancie la pile

de protocole convenable lorsqu'une nouvelle connexion est acceptée. Le service d'adressage indique l'existence de ces points d'accès et du protocole correspondant en créant, pour chaque objet hébergé sur ce nœud, une référence contenant plusieurs profils.

Le traitement d'un appel reçu par un serveur possédant plusieurs personnalités protocolaires est en tous points identique au traitement précédent. Les correspondants ne supportant qu'un seul protocole ignorent les autres profils.

Ainsi, l'objet T peut communiquer indifféremment avec les correspondants précédents, utilisant le protocole P (et la personnalité applicative A_0), ainsi qu'avec le nouveau correspondant C_1 utilisant la personnalité applicative A_1 et la personnalité protocolaire P'.

La communication est possible dans les deux directions : le tableau d'affichage peut émettre aussi bien que recevoir des appels de méthodes avec les objets de C_2. Lorsque T se trouve dans le rôle client, la fonction de liaison sélectionne l'usine de liaisons appropriée au correspondant contacté. Une fois la liaison établie, la chronologie d'un appel de méthode est inchangée.

Dans cette configuration, le tableau d'affichage est réutilisable dans les deux environnements A_0 et A_1. En revanche, C_0 et C_1 ne peuvent plus communiquer directement, car ils ne possèdent pas de protocole commun.

4.2.4 Configuration d'interopérabilité indirecte

On considère alors la situation où, de nouveau, deux correspondants sont associés à des intergiciels supportant des protocoles distincts. Dans cette configuration, aucun des deux correspondants n'est muni d'un intergiciel interopérable. Tout appel direct de l'un à l'autre est donc impossible, faute d'un protocole commun (figure 4.6).

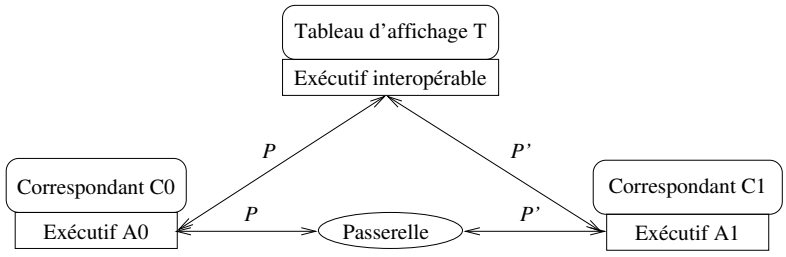


FIG. 4.6 – Configuration à objets interopérables par passerelle

Une telle situation se rencontre lorsque l'on se propose de construire une application à partir d'objets répartis déjà existants, fournis éventuellement par un développeur extérieur, et dont on ne peut pas contrôler la mise en œuvre. Dans ce cas, l'interopérabilité entre ces objets nécessite l'intervention d'un composant intermédiaire, qui joue le rôle de passerelle en traduisant les protocoles utilisés par les objets incompatibles. Cette traduction doit être symétrique pour permettre aux deux objets de coopérer normalement, comme s'ils faisaient partie du même environnement de répartition.

L'utilisation d'un intergiciel schizophrène permet la mise en place d'une telle passerelle de façon transparente et dynamique.

Préservation de l'intégrité référentielle

Nous cherchons à assurer, autant que faire se peut, que chaque référence d'objet qui existe dans l'application répartie vérifie la propriété :

Propriété 1 (Intégrité référentielle) *Si un nœud N détient une référence $R_{\langle N,O\rangle}$ désignant un objet O, alors cette référence contient un profil correspondant à une personnalité protocolaire de N.*

On remarque que les références créées initialement par la fonction d'adressage d'un intergiciel vérifient toujours trivialement cette propriété.

Cette propriété peut être perdue lorsqu'un nœud émetteur N_e transmet une référence R à un nœud récepteur N_r, et que cette référence ne contient pas de profils correspondant à une personnalité protocolaire supportée par N_r. Cela est possible par exemple lorsque N_e supporte plusieurs personnalités protocolaires et que N_r n'en supporte qu'une seule.

Pour préserver la propriété dans ce cas, nous introduisons un nouveau profil dans les références transmises dans une telle configuration.

Pour ce faire, supposons qu'avant la transmission d'une référence, la propriété est vérifiée sur le nœud N_t avec une personnalité protocolaire P.

Le module réalisant la fonction de représentation pour la transmission a connaissance du fait qu'il participe à une personnalité protocolaire P'. Cette personnalité est partagée par les nœuds N_t et N_r. Lorsqu'il est amené à représenter la référence $R_{\langle N_t,O\rangle}$, il peut donc vérifier si celle-ci contient un profil correspondant à la personnalité P'. Si c'est le cas, comme N_t supporte P', la référence peut être transmise en l'état, et la propriété est préservée.

Dans le cas contraire, N_t construit une nouvelle référence en associant un profil correspondant à l'un de ses propres points d'accès du service de transport (lié à la personnalité protocolaire P') et un identificateur d'objet encapsulant la valeur $R_{\langle N_t,O\rangle}$.

Cette nouvelle référence $R_{\langle N_r,O\rangle}$ désigne l'objet O indirectement à travers le nœud N_t qui servira de passerelle. La valeur transmise à N_r est alors $R_{\langle N_r,O\rangle}$, qui vérifie toujours la propriété.

Le mécanisme que nous avons indiqué permet donc de garantir la propriété d'intégrité référentielle que nous avons énoncée, sous réserve que la nouvelle référence $R_{\langle N_r,O\rangle}$ permette aux objets du nœud N_r de contacter l'objet O, ce que nous montrons en effet dans la section suivante.

Mécanisme d'appel

Lors de l'établissement d'une liaison par l'objet client sur le nœud N_r, celui-ci va établir une liaison avec le nœud N_t au moyen du nouveau profil contenu dans la référence $R_{\langle N_r,O\rangle}$, au moyen du protocole P'.

Lorsqu'une requête est reçue sur cette session par N_t, ce dernier nœud effectue un appel à sa fonction de liaison en vue de déterminer l'objet désigné par la référence. S'il s'agissait d'un objet local, la fonction de liaison ferait à son tour appel à la fonction d'activation, à travers la couche neutre, afin de retourner le servant correspondant.

Dans le cas d'une passerelle, cependant, l'adaptateur d'objets de N_t détermine que l'identificateur d'objet local auquel la requête est destinée correspond à une référence encapsulée, et non à un objet local réel.

La fonction de liaison construit donc un subrogé de de l'objet cible réel. Ce subrogé n'est autre qu'un objet de liaison utilisant la personnalité protocolaire P, établi au moyen de la référence $R_{\langle N_t, O\rangle}$ originelle, dont toute l'information était contenue dans l'identificateur d'objet.

La requête reçue par la personnalité protocolaire P' est sous forme neutre. Elle peut donc être transmise à l'objet de liaison nouvellement créé pour être réémise par la personnalité protocolaire P. Au bout d'un nombre fini de telles étapes, on atteint l'objet réel O qui peut traiter la requête. L'établissement dynamique d'objets de liaison permet donc de créer au vol les passerelles nécessaires à l'interopérabilité.

Notons que la reconstitution de la représentation neutre d'une requête par une passerelle nécessite que celle-ci possède une description de la signature de la méthode appelée. Faute de cette information, en effet, le nœud intermédiaire ne peut déballer correctement les paramètres de l'appel. Il est donc nécessaire que la description des interfaces des objets applicatifs soit accessibles aux passerelles.

4.3 Traits architecturaux d'un intergiciel schizophrène

Dans cette section, nous présentons les traits essentiels qui caractérisent une architecture d'intergiciel schizophrène. Ces traits sont issus notamment de l'état de l'art que nous avons étudié dans le chapitre 3, mais ils n'ont pas jusqu'à présent été intégrés au sein d'une même architecture.

Cette réunion, condition nécessaire d'une interopérabilité effective et efficace dans une application répartie, est la caractéristique novatrice qui différencie les intergiciels schizophrènes par rapport aux intergiciels existants.

4.3.1 Découplage entre les aspects protocolaires et applicatifs

Le trait architectural essentiel introduit par notre définition d'un intergiciel schizophrène est le découplage entre les aspects protocolaires et applicatifs de la personnalisation.

Les intergiciels traditionnels couplent fortement personnalité applicative et personnalité protocolaire. De plus, ils ne supportent en général qu'une seule combinaison entre personnalité applicative et protocolaire. Pour rendre un objet le plus interopérable possible avec l'ensemble des composants d'une application répartie, il est nécessaire de lever ces deux contraintes :

- plusieurs personnalités doivent pouvoir coexister pour chacun des aspects protocolaire et applicatif ;
- chaque personnalité de l'un des aspects doit pouvoir s'articuler avec chaque personnalité de l'autre aspect.

Nous proposons donc de découpler personnalité applicative et personnalité protocolaire, en définissant une interface uniforme autorisant des combinaisons arbitraires. De plus, nous faisons de cette interface un point de convergence entre composants de personnalisation, afin de permettre la coexistence de plusieurs personnalités sur chaque aspect au sein d'une même instance d'intergiciel.

Par exemple, un objet créé en utilisant la personnalité applicative de l'annexe des systèmes répartis d'Ada 95 peut communiquer aussi bien avec des objets

CORBA utilisant le protocole GIOP qu'avec un serveur utilisant le protocole SOAP.

Réciproquement, un objet dont on souhaite qu'il soit accessible à travers GIOP pourra être développé selon le modèle de CORBA, en utilisant un contrat IDL et un générateur de souches et de squelettes, ou bien au moyen de l'annexe des systèmes répartis du langage Ada 95.

4.3.2 Introduction d'une couche neutre

Le découplage est réalisé par l'introduction d'une couche générique neutre, c'est-à-dire indépendante à la fois de la personnalité applicative et de la personnalité protocolaire.

Articulation des personnalités

Cette couche réalise l'intégration des différentes personnalités applicatives d'une part, protocolaires d'autre part. Elle offre les représentations et les services nécessaires pour effectuer la mise en correspondance transparente entre personnalités applicatives et protocolaires.

Ces relations se traduisent par l'échange de données représentant les interactions entre les objets de l'application. Ces représentations réflexives des échanges entre objets doivent avoir une forme générique, indépendante tant de la personnalité applicative de l'objet local que de la personnalité protocolaire utilisée pour contacter l'objet distant.

Par ailleurs, ces représentations doivent se suffire à elles-mêmes et contenir toute l'information relative aux interactions. En effet, l'architecture doit garantir une complète indépendance entre les composants des personnalités applicatives et protocolaires. L'échange direct d'informations complémentaires (spécifiques d'une personnalité) entre les couches applicative et protocolaire n'est donc pas envisageable.

Représentations basées sur l'exécution dynamique de requête

Les structures de données de la couche neutre doivent contenir une représentation abstraite et auto-descriptive des appels de méthode entre les objets de l'application : elles réifient les objets et les échanges prévus par les modèles de répartition. De plus, ces structures doivent être indépendantes du choix d'une personnalité particulière.

Les normes CORBA définissent deux tels systèmes de représentation réflexive : la DII (*Dynamic Invocation Interface*) et la DSI (*Dynamic Skeleton Interface*). La DII est utilisée par des clients CORBA pour construire un objet représentant une requête d'appel de méthode, puis effectuer l'appel de sous-programme distant correspondant. À l'inverse, la DSI permet à un objet serveur d'accéder explicitement aux informations qui décrivent un appel de méthode dont il est l'objet.

Ces interfaces permettent de représenter sous forme d'objets manipulables une requête d'appel de méthodes, ses paramètres, la valeur retournée et une occurrence d'exception, le cas échéant. Elles utilisent pour la représentation des données utilisateurs (paramètres, valeurs de retours et valuations associées aux exceptions) des conteneurs universels, les *Any*. Une variable conteneur universel

permet de stocker la valeur de n'importe quelle donnée utilisateur transmissible, accompagnée de la description de sa structure. Leur utilisation permet l'écriture d'opérations liées à la répartition de façon générique, indépendamment des types particuliers manipulés par une application.

Les interfaces réifiant le traitement des appels de méthodes rendent possible la construction de clients et de serveurs génériques capables d'utiliser et d'offrir des services arbitraires, dont la spécification détaillée n'est pas nécessairement connue avant l'exécution. Leur intérêt dans de telles situations est attesté, dans le cadre du modèle de répartition de CORBA, par le langage CorbaScript [67] et son application CorbaWeb [66], que nous avons évoqué dans la section 3.3.1.

La représentation d'une requête dans la couche neutre, très similaire aux interfaces *Request* de la DII et *ServerRequest* de la DSI, doit contenir :

– l'identité de l'objet cible ;
– l'identification de la méthode à invoquer ;
– les paramètres et le résultat de l'invocation.

Il est donc nécessaire que les personnalités applicatives d'une part et protocolaires d'autre part offrent une projection des espaces d'identification des méthodes qu'elles définissent vers celui de la représentation neutre et unifiée.

En particulier, si l'on suppose que cette représentation neutre ne distingue les méthodes que par leur nom, il est nécessaire que les personnalités supportant la surcharge des identificateurs prévoient le moyen de lever les ambiguïtés.

Le résultat de l'invocation n'est pas initialement renseigné ; cette variable d'état est complétée par la personnalité applicative à l'issue du traitement par l'objet servant, puis cette modification est reportée sur le nœud appelant par la personnalité protocolaire. Il en va de même pour une éventuelle occurrence d'exception.

4.3.3 Architecture générale

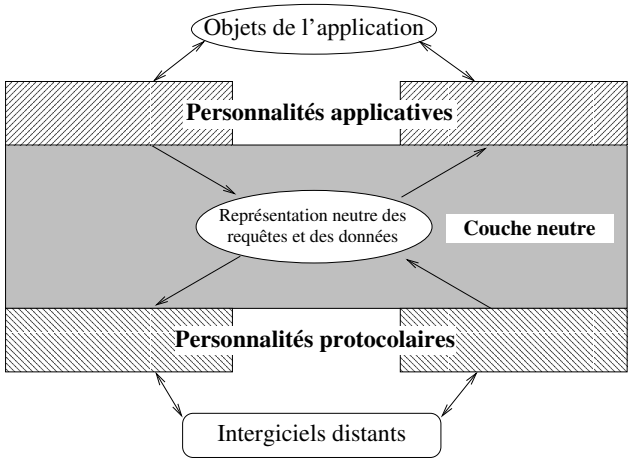


FIG. 4.7 – Traits architecturaux d'une architecture schizophrène

Nous avons présenté dans cette section les traits architecturaux à gros grain d'un intergiciel schizophrène. La figure 4.7 illustre ces traits :

- découplage et isolement entre les aspects applicatifs et protocolaires de l'intergiciel ;
- introduction d'une couche neutre ayant pour rôle d'articuler les personnalités applicatives et protocolaires en garantissant leur indépendance ;
- structures de données de la couche neutre permettant la représentation des interactions entre composants applicatifs indépendamment des modèles de répartition et des personnalités ;

4.4 Une architecture d'intergiciel schizophrène

Dans cette section, nous décrivons plus en détail l'architecture d'intergiciel schizophrène que nous avons définie. Nous introduisons un découpage de cette architecture en blocs fonctionnels, dont nous esquissons les fonctionnalités et les interactions. Ce découpage servira de guide à la description de la mise en œuvre.

4.4.1 Vue d'ensemble

Notre architecture schizophrène comprend quatre blocs fonctionnels, qui permettent la mise en œuvre des traits que nous avons définis à gros grain dans la section précédente.

Composants de support Ces composants utilitaires réalisent des fonctionnalités de base et des motifs de conception réutilisés par l'ensemble des modules de l'intergiciel. Ils constituent une boîte à outils d'éléments réutilisables dans des contextes variés.

Personnalités applicatives Ces modules permettant aux objets de l'application de s'intégrer à l'environnement réparti. Ils offrent une interface de programmation permettant aux objets de l'application de s'enregistrer auprès de l'intergiciel pour recevoir des requêtes provenant d'objets distants, et d'émettre des requêtes vers des objets distants.

Couche neutre elle-même subdivisée en

Composants applicatifs neutres Ces modules offrent des adaptateurs d'objets génériques, ainsi que les entités de base utilisées par les personnalités applicatives pour rendre les objets de l'application intégrables dans l'intergiciel. Ils sont génériques, indépendants d'une personnalité particulière, et sont utilisés par les mises en œuvre des différentes personnalités applicatives.

Noyau de répartition générique Ces éléments de l'intergiciel, indépendants de la personnalité, articulent l'ensemble des constituants et des activités de l'intergiciel. Ils offrent une bibliothèque de gestion de ressources, un ordonnanceur qui règle l'affectation des diverses activités de l'intergiciel aux tâches qui s'exécutent sur le nœud de répartition.

Le noyau générique joue également le rôle d'un registre de personnalités : il a connaissance des divers composants personnalisés qui existent au sein du nœud, et les connecte en tant que de besoin. Il reçoit des messages des couches applicatives et protocolaires, et les retransmet en fonction des liaisons établies.

Composants protocolaires neutres Ces composants fournissent l'infrastructure nécessaire à la réalisation de la communication entre intergiciels, tels que des composants génériques de transport et de filtrage. Ils sont indépendants d'un protocole particulier, et sont utilisés par les composants des personnalités protocolaires.

Personnalités protocolaires Les modules de personnalités protocolaires réalisent les protocoles d'appel de méthodes à distance. Des usines d'objets de liaison sont enregistrées auprès de l'intergiciel. Celles-ci permettent au cœur de répartition de demander l'établissement des communications nécessaires aux interactions entre les objets applicatifs.

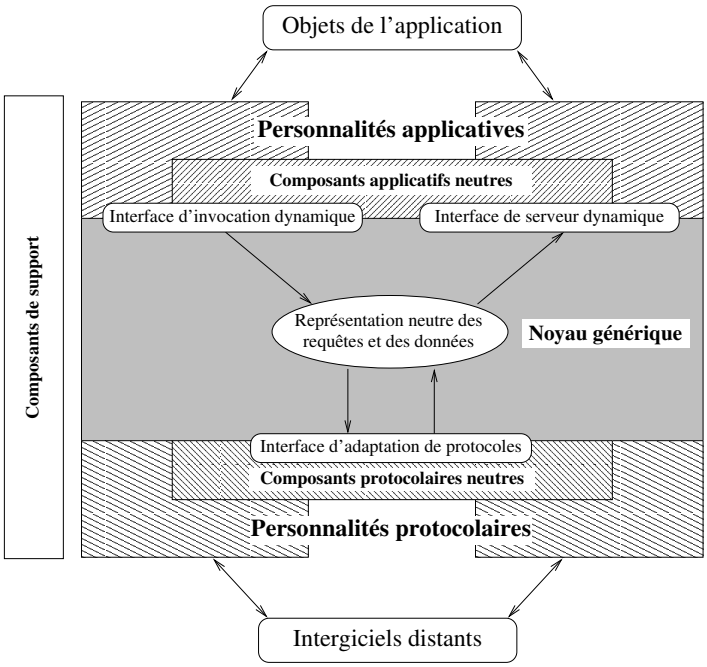


FIG. 4.8 – Architecture détaillée d'un intergiciel schizophrène

La figure 4.8 illustre le positionnement de ces composants dans l'architecture. Dans la suite de cette section, nous décrivons plus en détail leurs fonctionnalités.

4.4.2 Composants de support

Un ensemble de services de base, indépendants des personnalités, sont fournis par des composants génériques de l'intergiciel, et peuvent être utilisés par la couche neutre comme par les parties applicative et protocolaire.

Ainsi, la gestion des ressources telles que la mémoire ou les tâches est assurée par des bibliothèques génériques. Les personnalités utilisent ces composants neutres comme une boîte à outils spécialisée pour la réalisation de composants de répartition.

4.4.3 Couche neutre

La couche neutre d'un intergiciel schizophrène comprend des composants de support utiles à l'ensemble de l'intergiciel et un noyau générique responsable du découplage et de l'articulation entre les couches applicative et protocolaires.

Noyau générique

Le noyau générique constitue la charpente dans laquelle s'intègrent les composants des personnalités applicatives et protocolaires. Il permet ainsi la mise en relation des composants de l'application (qui utilisent les services d'une personnalité applicative) avec les personnalités protocolaires.

Il offre aux personnalités les services nécessaires pour enregistrer leurs mises en œuvre des interfaces abstraites correspondant aux fonctions qu'elles remplissent dans l'intergiciel.

Il n'existe aucune interface directe entre composants de personnalités applicatives et composants de personnalités protocolaires ; toutes les interactions entre couches traversent le noyau générique. Celui-ci est donc garant de l'indépendance et du découplage entre les deux aspects.

Composants applicatifs neutres

La partie applicative de l'intergiciel comprend également des composants neutres, non spécifiques de la personnalité. Ceux-ci constituent une boîte à outils pour la réalisation de personnalités applicatives.

En particulier, des adaptateurs d'objets génériques sont définis. Ces composants sont responsables de la gestion de l'espace de référencement des objets dans le contexte d'un nœud de répartition, ainsi que de la fonction d'activation des objets servants.

Composants protocolaires neutres

La couche neutre offre un certain nombre de fonctionnalités pouvant être réutilisées par plusieurs personnalités protocolaires.

Elle offre une bibliothèque de communication permettant l'utilisation de mécanismes de transport de données variés. Comme dans le système de communication de GLADE, que nous avons décrit dans la section 3.2.1, divers types de réseaux de communication ou de supports point à point peuvent être utilisés pour la communication entre nœud. Tous les sous-systèmes de transport de données mettent en œuvre la même interface abstraite. L'utilisateur peut ainsi choisir aisément le transport le mieux adapté à ses besoins.

Un mécanisme de filtres similaire à celui de GLADE (décrit dans la section 3.2.1) est également défini. Une interface générique des filtres est spécifiée, ainsi que des usines de filtres qui permettent d'instancier en une seule opération une pile de filtres complète.

Synthèse

La couche neutre, et plus précisément le noyau générique, est le composant central d'une architecture schizophrène : elle réalise le découplage entre les personnalités applicatives et protocolaires, ce qui permet l'adaptation efficace entre

des combinaisons arbitraires de personnalités applicatives et protocolaires, ou encore entre deux personnalités protocolaires.

Point de convergence entre les différentes personnalités, la couche neutre rend possible la mutualisation du code au sein de l'intergiciel :

- Les fonctions communes à plusieurs personnalités sont réalisées par la couche neutre ; chaque personnalité n'est qu'une façade sur les fonctionnalités de la couche neutre.
- Chaque personnalité applicative peut bénéficier de l'ensemble des personnalités protocolaires existantes : la mise en œuvre d'une nouvelle personnalité applicative n'implique aucune nécessité de développer en parallèle une personnalité protocolaire spécifique.
- Symétriquement, chaque personnalité protocolaire bénéficie de l'ensemble des personnalités applicatives existantes.

4.4.4 Personnalités applicatives

Chaque personnalité applicative de l'intergiciel contient des composants qui permettent aux objets de l'application d'accéder aux services de l'intergiciel, qui sont réalisés dans la couche neutre.

Les objets d'application n'utilisent donc pas directement les services de la couche neutre : une personnalité applicative est une instance du patron de conception *Wrapper façade* (décrit plus en détail dans la section 4.7.1) pour la fonctionnalité d'intergiciel de répartition.

Chaque personnalité est réalisée sous forme d'une bibliothèque qui fournit une interface de programmation permettant aux objets clients et serveurs de l'application d'interagir avec l'intergiciel et, à travers lui, avec d'autres objets applicatifs situés sur des nœuds distants.

Fonctionnalités offertes aux objets serveurs

Chaque personnalité applicative définit un moyen par lequel les objets serveurs de l'application s'enregistrent auprès de l'intergiciel afin de pouvoir recevoir des appels provenant d'objets distants. À cet effet, un composant adaptateur d'objets générique peut être utilisé, qui offre un mécanisme de gestion des associations entre identificateurs d'objets abstraits et objets concrets de l'application.

La notion d'adaptateur d'objets est issue du modèle de traitement de la norme CORBA. Cependant, elle peut être formalisée et généralisée, pour être réutilisée dans la mise en œuvre d'autres modèles de répartition.

Une interface de programmation spécifique à la personnalité est alors soit explicitement utilisée par le développeur de l'application pour enregistrer les objets, soit implicitement appelée par du code généré par un outil *ad hoc*, par exemple dans le cas des personnalités où la répartition est une caractéristique incluse dans un langage de programmation. La personnalité applicative présente donc une *façade* qui encapsule les fonctionnalités d'un adaptateur d'objets générique, et plus généralement de l'intergiciel.

Les références génériques associées aux objets de l'application lui sont retournées sous une forme spécifique de la personnalité. Ces objets peuvent alors se signaler auprès d'un service de dénomination, afin que leurs références deviennent accessibles pour des objets situés sur d'autres nœuds.

Lorsqu'une requête provenant d'un intergiciel distant est reçue par une personnalité protocolaire, la partie neutre de l'intergiciel consulte l'adaptateur d'objets pour déterminer quel objet concret doit la traiter, active cet objet concret, réserve les ressources nécessaires à l'exécution de la requête (en particulier une tâche), puis effectue un appel à l'objet concret, à travers une interface dépendant de la personnalité applicative choisie.

Une fois la requête exécutée, la fin de traitement est signalée à la couche neutre, afin que la réponse soit envoyée à l'intergiciel demandeur.

Fonctionnalités offertes aux objets clients

Supposons qu'un objet client a pris connaissance d'une référence désignant un objet serveur (cette opération peut être complexe, en particulier dans le cas où client et serveur n'utilisent pas la même personnalité applicative). Le client utilise alors les services de sa personnalité applicative pour émettre des appels de méthodes vers cet objet.

La personnalité applicative fournit une interface de programmation permettant l'émission d'une requête vers l'intergiciel distant hébergeant l'objet serveur. Cette interface de programmation peut :

- être appelée implicitement par des souches et des squelettes générés ;
- être utilisée explicitement par l'objet applicatif client. Dans ce cas, l'objet utilise une une façade qui dépend de la personnalité, pour construire dynamiquement une requête vers l'objet serveur.

La seconde option, proposée par exemple par la DII de CORBA, est particulièrement utile lorsqu'un client souhaite appeler des méthodes d'un serveur dont l'interface n'était pas connue au moment de la compilation du client.

Dans un cas comme dans l'autre, une représentation neutre de la requête à émettre est construite, puis confiée à l'intergiciel. La couche neutre se charge alors de faire établir une liaison avec l'intergiciel distant par la personnalité protocolaire appropriée. Cette dernière transmet ensuite la requête au nœud distant. Lorsque la réponse correspondante est reçue, la requête originale est mise à jour, puis la personnalité protocolaire signale la fin de traitement à la couche neutre, afin que le contrôle soit rendu à l'objet appelant.

4.4.5 Personnalités protocolaire

La partie protocolaire de l'intergiciel contient l'ensemble des composants dédiés à la communication entre nœuds de l'intergiciel. Elle permet l'établissement et le maintien de connexions de transport de données, ainsi que leur utilisation pour l'appel de méthodes à distance. Les personnalités protocolaires déterminent la façon dont les connexions de transport et les échanges de messages sous-jacents aux interactions entre objets sont mis en œuvre.

Les personnalités protocolaires sont constituées d'objets de liaison, qui réalisent les échanges avec les intergiciels tiers, et de modules permettant la création de ces objets de liaison.

Primitives des objets de liaison

Les objets de liaison sont constitués par assemblage de composants génériques de transport et de filtrage, auxquels on associe la mise en œuvre propre-

ment dite d'un protocole (équivalent des usines de messages de TAO, décrites dans la section 3.3.2).

Chaque objet de liaison est muni de deux ensembles de primitives :

L'interface exposée à la partie neutre :
ces primitives permettent l'émission d'une requête vers un objet distant, ou d'une réponse à destination d'un client. Elles sont appelées par la couche neutre pour effectuer les actions correspondant aux interactions demandées par les objets applicatifs.

L'interface exposée aux composants de transport :
ces primitives traitent les événements reçus d'intergiciels distants, représentant les messages échangés avec les autres nœuds de l'application (requêtes provenant de clients, réponses provenant de serveurs). Ceux-ci sont mis sous forme neutre, puis transmis à la couche neutre afin que celle-ci les oriente vers les objets appropriés de la couche applicative (et ultimement les vers les objets de l'application).

L'organisation interne d'un objet de liaison forme une pile de filtres. Chaque filtre participant expose une interface aux couches supérieures, permettant de transmettre des données vers les couches inférieures, et une interface aux couches inférieures, recevant les événements provenant de ces couches basses. Cette organisation est similaire aux interfaces des sessions de *x*-kernel [46].

Usines d'objets de liaison

La partie protocolaire contient également les usines d'objets de liaison nécessaires à l'établissement des connexions de transport et des sessions de protocoles sous-tendues par les interactions entre objets.

Dans le cas d'un client, cette opération est effectuée à partir d'une référence désignant un objet distant. Une telle référence contient un ensemble de profils. Chaque profil comporte une information désignant un point d'accès du service de transport adéquat, et une étiquette indiquant le protocole à utiliser sur ce point d'accès. L'établissement d'une liaison est donc réalisé en fonction de cette étiquette, par un composant usine spécialisé responsable de l'établissement du canal de transport, de l'instanciation de la pile de protocole et de leur connexion. Une étiquette de profil différente existe donc pour chaque combinaison d'un mécanisme de transport de données et d'un protocole.

Pour les serveurs, une usine de pile de protocole est associée à chaque point d'accès du service de transport. Lorsqu'une nouvelle connexion est signalée, un point de terminaison du service de transport est créé, et une pile de protocole est instanciée. L'intergiciel connecte alors ces deux nouvelles entités, afin que l'échange d'information puisse démarrer.

Composition de personnalités protocolaires

Comme nous l'avons évoqué dans la discussion relative aux représentations des requêtes dans la couche neutre, section 4.3.2, une représentation unifiée des requêtes est définie par la couche neutre. Cette représentation est la même que l'on soit dans le cas :

d'un client où elle est construite par une personnalité applicative et transmise à une personnalité protocolaire ;

d'un serveur pour lequel elle est construire par une personnalité protocolaire et remontée à une personnalité applicative.

Il en résulte en particulier que les primitives fondamentales d'une personnalités protocolaires (passage d'une requête sous forme neutre à un message de protocole et vice-versa) sont strictement réciproques l'une de l'autre.

Les mécanismes de signalisation de début et de fin de traitement d'une requête sont également symétriques entre clients et serveurs :

Sur un serveur, la fin du traitement d'une requête est matérialisée par la modification d'une variable d'état de celle-ci, effectuée par la personnalité applicative. La couche neutre signale alors à la personnalité protocolaire qu'elle peut émettre une réponse vers le client.

Sur un client, la personnalité protocolaire qui reçoit une réponse effectue la même modification de variable d'état sur la requête en attente, et la couche neutre signale à la personnalité applicative que l'objet applicatif client peut poursuivre son traitement.

De ce fait, deux personnalités protocolaires peuvent être composées directement, sans qu'il soit nécessaire que les requêtes transitent par une quelconque personnalité applicative. En particulier, un nœud de traitement sur lequel plusieurs personnalités protocolaires sont configurées peut tenir lieu de passerelle entre ses personnalités sans disposer d'aucune personnalité applicative.

L'existence de la représentation neutre unifiée entre les rôles clients et serveur et son utilisation pour la réalisation de personnalités protocolaires rendent ainsi possible la mise en place de passerelles dynamiques entre protocoles telles que celles évoquées dans la section 3.3.4, figure 3.7.

4.5 Projection des fonctions sur l'architecture

Dans la section 2.4.1, nous avons identifié un ensemble de fonctions fondamentales qu'un intergiciel pour objets répartis doit prendre en charge. Ces fonctions représentent les besoins *fonctionnels* des applications réparties vis-à-vis de l'intergiciel.

Par ailleurs, nous avons présenté dans la section 2.4.2 les besoins *non fonctionnels* des applications réparties ouvertes et hétérogènes. Pour satisfaire ces besoins, nous avons conclu qu'un intergiciel devait présenter des propriétés de configurabilité, de généricité et d'interopérabilité.

Nous avons déduit de ces besoins non fonctionnels une architecture d'intergiciel schizophrène, que nous avons présenté dans la section 4.4.

Nous montrons ici que cette architecture est adaptée pour répondre aux besoins fonctionnels d'un intergiciel pour objets répartis, en projetant chacune des fonctions de base que nous avons évoquées sur un ou plusieurs composants inscrits dans l'architecture. Ce faisant, nous mettons en pratique les principes de la méthodologie de décomposition *Software RISC* (section 3.1.2). En effet, nous identifions les composants qui émergent naturellement des fonctions que doit réaliser l'intergiciel.

4.5.1 Adressage

Fonction de la couche neutre. Cf. figure 4.9.

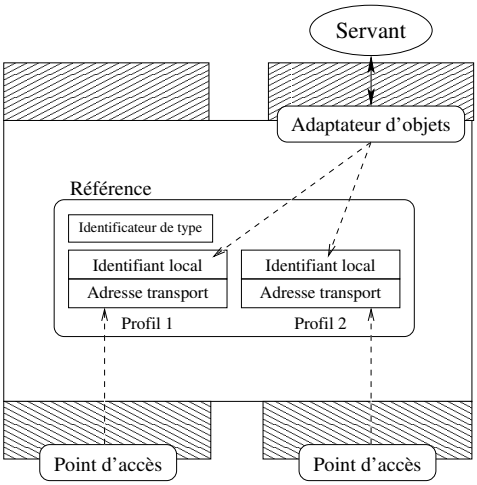


FIG. 4.9 – Fonction *adressage*

La création d'une référence pour un objet hébergé par l'intergiciel nécessite l'association :

- d'un identifiant local de l'objet ;
- de l'adresse d'un point d'accès du service de transport appartenant à cet intergiciel.

L'identifiant local de l'objet est fourni par la couche applicative, qui est responsable de la gestion de l'espace d'adressage local.

L'adresse d'un point de transport doit être fournie par une personnalité protocolaire ; elle est accompagnée de la désignation du protocole à utiliser sur ce point d'accès.

La réunion de l'identifiant local et de l'information de transport nécessite d'avoir visibilité à la fois sur les couches applicative et protocolaire. Elle doit donc être effectuée par un module situé dans la couche neutre.

Ce module crée une référence d'objet, qui associe une information de type et une collection de profils. Chaque profil agrège :

- un identificateur d'objet (suite d'octets opaque fournie par un adaptateur d'objets) ;
- l'adresse d'un point d'accès du service de transport enregistré par une personnalité protocolaire ;
- l'identification du protocole correspondant à cette personnalité.

Cette structure de données est utilisée comme une capsule opaque par la couche applicative : celle-ci ne prend pas connaissance des détails de sa structure interne et du contenu des profils. Elle n'utilise les références d'objets que pour les communiquer à d'autres nœuds en les passant en paramètres d'appels distants ou pour demander à l'intergiciel l'établissement d'une liaison avec un objet.

4.5.2 Transport

Fonction de la couche protocolaire. Cf. figure 4.10.

La fonction de transport consiste à transférer une information d'un point à un autre : elle correspond au niveau le plus bas du système de communication.

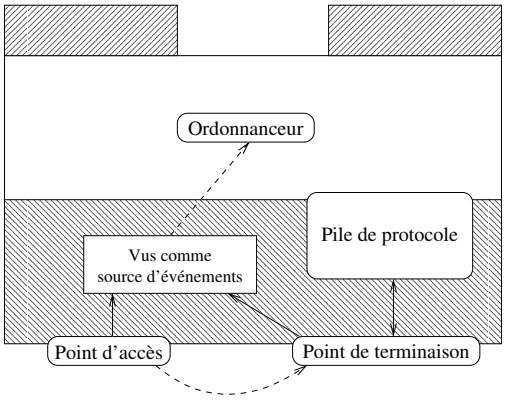


FIG. 4.10 – Fonction *transport*

Cette fonction offre deux abstractions principales : les *points d'accès du service de transport* (*transport service access points, TSAPs*) et les *points de terminaison du service de transport* (*transport endpoints, TEs*).

Les points d'accès du service de transport représentent les entités créées par le système d'exploitation à la demande de l'intergiciel afin de recevoir des connexions de la part de correspondants distants à travers un réseau de communication. Chaque point d'accès du service de transport possède une adresse dont le sens est défini dans le contexte d'un mécanisme de transport particulier. Cette adresse est utilisable par tout correspondant utilisant ce même mécanisme de communication : sa portée est un réseau donné.

Les points de terminaison du service de transport représentent les entités du système au moyen desquelles les données sont échangées. Pour les mécanismes de transport de données en mode connecté, un point de terminaison représente une extrémité d'une connexion établie. Lorsqu'une connexion est initiée par un client, un point de terminaison est créé à partir de l'adresse d'un point d'accès distant. Sur le serveur, le point d'accès accepte la connexion entrante et crée lui-même un point de terminaison. La connexion établie, les deux intergiciels peuvent dialoguer par échange de données au moyen des points de terminaison ainsi créés.

Points d'accès et points de terminaison du service de transport représentent des entités permettant à un intergiciel d'interagir avec le monde extérieur. Les instants où des événements se produisent sur ces entités et requièrent l'attention de l'intergiciel ne sont pas connus *a priori*. Ces objets constituent donc des *sources d'événements asynchrones* qui doivent être régulièrement scrutées par l'intergiciel afin que ces événements externes soient pris en compte.

4.5.3 Liaison

Fonction de la couche protocolaire. Cf. figure 4.11.

Étant donnée une référence désignant un objet, la fonction de *liaison* (*binding*) a pour rôle de créer une structure qui représente l'objet désigné et le fait qu'une liaison avec lui a été établie : un *objet de liaison* (*binding object*).

Un objet de liaison correspondant à un objet distant comporte :

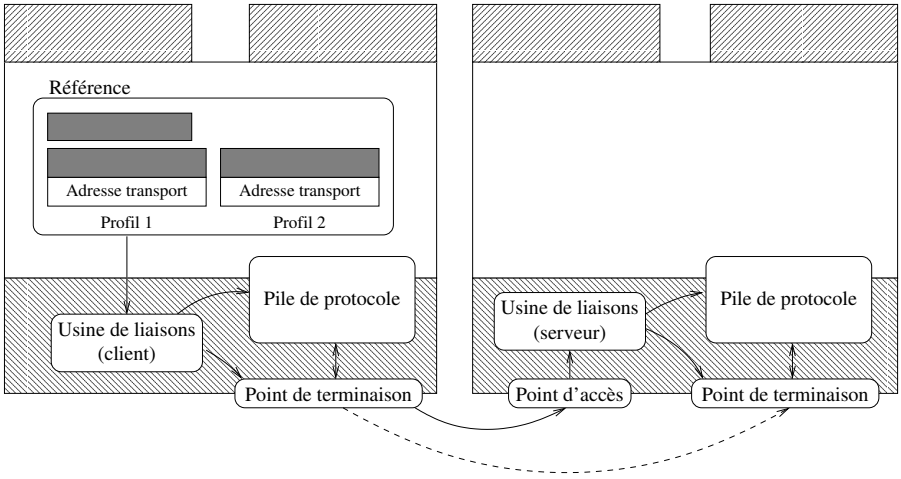


FIG. 4.11 – Fonction *liaisons*

côté client : un point de terminaison du service de transport et une pile de protocole créés à partir d'un profil de l'objet désigné ;

côté serveur : également un point de terminaison du service de transport et une pile de protocoles, créés à la suite de la réception d'une nouvelle connexion par un point d'accès du service de transport nommé par le profil utilisé.

Un objet de liaison permet de traiter des appels de méthode sur l'objet distant désigné, en les lui transmettant sous forme de messages. Cette structure constitue donc un *subrogé* de l'objet réel. Comme celui-ci, ce subrogé est susceptible d'être désigné par une référence, moyennant l'attribution d'un identifiant par la fonction d'adressage de l'intergiciel client. Une telle référence permet l'accès à l'objet initialement désigné, le nœud hébergeant le subrogé jouant le rôle de passerelle.

Le mécanisme d'établissement d'une liaison peut être :

implicite lorsque l'établissement d'une connexion de transport et son association avec une référence sont effectuées de façon transparente pour l'utilisateur, au cours d'un appel de méthode (c'est le cas par défaut dans CORBA) ;

explicite lorsque l'utilisateur souhaite contrôler certains aspects de l'établissement de la connexion, par exemple pour choisir des paramètres de qualité de service à négocier avec le réseau.

Lorsqu'une liaison est demandée pour une référence désignant un objet local, c'est le servant réalisant cet objet qui est retourné par la fonction de liaison (et non un subrogé de l'objet). Dans ce cas, la détermination du servant à partir de la référence est réalisée par un appel à la fonction d'activation, à travers la couche neutre.

4.5.4 Représentation

Fonction de la couche protocolaire. Cf. figure 4.12.

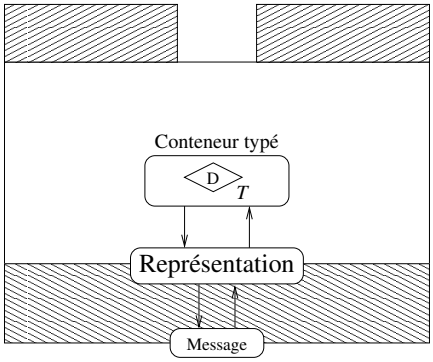


FIG. 4.12 – Fonction *représentation*

Les données manipulées reçues ou émises par un objet applicatif sont définies par leur type, au sein d'un modèle de types fixé par la personnalité applicative. Les modèles de types rencontrés dans les personnalités de répartition sont construits autour de types élémentaires (booléens, nombres, caractères) et de types composés formés par agrégation de types élémentaires ou d'autres types composés (tableaux, enregistrements).

La couche neutre est elle-même munie d'un modèle de types. Chaque personnalité applicative définit une projection de son modèle de types vers celui de la couche neutre. Cette projection permet la construction des structures matérialisant les requêtes confiées à l'intergiciel.

La fonction de représentation intervient pour la transmission des requêtes. Son rôle est de convertir une donnée D d'un type T (dans le modèle de types de la couche neutre) en message transmissible à un autre intergiciel. Le contenu de ce message doit être conforme à la représentation imposée par le protocole de communication utilisé.

La fonction de représentation définit, pour tout type du modèle de la couche neutre, une transformation injective projetant toute valeur du type sur une valeur d'un type du modèle de la personnalité protocolaire.

Elle réalise ainsi une transformation :

$$< D >_T \xmapsto{Representation} msg$$

Cette transformation peut s'effectuer avec une perte d'information : le message résultant msg doit contenir suffisamment d'information pour reconstituer la valeur D connaissant T et msg, mais pas nécessairement pour reconstituer $< D >_T$ en connaissant seulement msg. Par exemple, dans la représentation CDR de CORBA, une suite quelconque de quatre octets peut correspondre à un entier long ou bien à deux entiers courts.

La transformation réciproque que doit offrir le mécanisme de représentation doit donc prendre non seulement msg mais aussi T en entrée. En l'absence de T, elle ne pourrait pas « magiquement » reconstituer cette information perdue avant transmission.

Nous la définissons donc comme :

$$(msg, < \cdot >_T) \xmapsto{Representation} < D >_T$$

où $< \cdot >_T$ représente un conteneur vide mais possédant l'information de type T.

On impose ainsi à l'utilisateur de la fonction « représentation » de connaître *a priori* le type des données qu'il souhaite recevoir : on a éludé le problème de la reconstitution du typage en le lui transférant.

La fonction de représentation est réalisée en concrétisant une interface abstraite fournissant les deux transformations indiquées ci-dessus.

4.5.5 Protocole

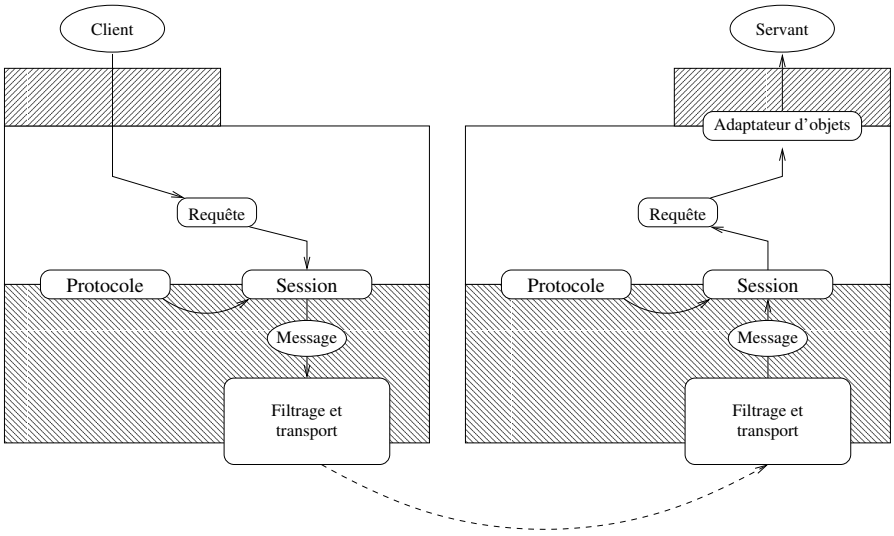


FIG. 4.13 – Fonction *protocole d'invocation*

Fonction de la couche protocolaire. Cf. figure 4.13.

La fonction « protocole d'invocation de méthodes » est complémentaire de la fonction de représentation : elle transforme des informations (en l'occurrence, la description des interactions entre les objets applicatifs) d'une représentation locale définie par la couche neutre vers une forme transmissible au sein d'un message, et réciproquement.

Par ailleurs, cette fonction orchestre le déroulement d'un appel de méthode : à partir d'une demande d'invocation locale, un message est préparé puis émis. L'intergiciel client est alors mis en attente d'une réponse ; lorsque celle-ci est reçue, elle est retraduite sous forme neutre et signalée à la couche neutre, qui la transmet finalement à la couche applicative.

Sur un serveur, à partir d'une demande d'invocation reçue d'un autre intergiciel, l'identité de l'objet cible et de la méthode appelée sont extraites, ainsi que les paramètres de l'appel. Une requête sous forme neutre est alors construite et confiée à la couche neutre, afin que celle-ci la fasse traiter soit par un servant local, soit en la retransmettant vers un objet de liaison lorsqu'une passerelle est établie. À l'issue du traitement de la requête, la couche neutre signale à la couche protocolaire que la réponse éventuelle peut être retournée à l'intergiciel client.

La mise en œuvre de la fonction « protocole » par une personnalité protocolaire contient donc les fonctionnalités suivantes :

- transformation d'une requête en message ;
- transformation d'un message en requête, en interrogeant l'Object adapter pour obtenir des informations sur la signature de la méthode appelée (et ainsi pouvoir fournir à la fonction de représentation le conteneur typé nécessaire au déballage des paramètres).

Ces fonctionnalités sont réalisées comme primitives d'une dérivation concrète de la classe *Session*. Une classe *Protocol* est associée à chaque classe *Session*, qui a pour fonction de créer des sessions dans les usines de liaisons.

4.5.6 Activation

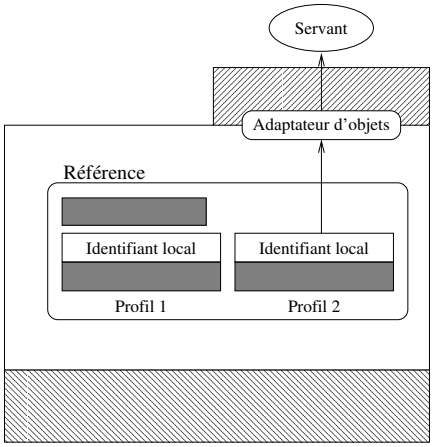


FIG. 4.14 – Fonction *activation*

Fonction de la couche applicative. Cf. figure 4.14.

Lorsqu'une requête reçue par la couche protocolaire est prête à être exécutée, il appartient à l'intergiciel d'associer l'objet logique désigné par la référence sur laquelle l'appel est effectué avec un objet concret local (un servant) qui exécute la méthode.

La détermination d'un objet servant local incarnant un objet logique est dévolue à l'*Adaptateur d'objets* (*Object Adapter*) (terme que nous empruntons à CORBA). L'objet réel peut être un objet déjà existant, qui a été préalablement enregistré de façon explicite auprès de l'intergiciel. Dans ce cas, celui-ci maintient une table associant les objets actifs à leurs identificateurs.

Il est également possible que l'objet réel soit incarné par une structure créée à la volée par une usine d'objets, et qui pourra, ou non, persister pour recevoir des appels ultérieurs. La référence peut également être utilisée comme une clé pour rechercher un objet existant parmi une collection, suivant un critère quelconque.

Enfin, un objet réel peut être désigné comme objet « par défaut » pour recevoir les appels de méthodes pour les références ne correspondant à aucun objet explicitement enregistré.

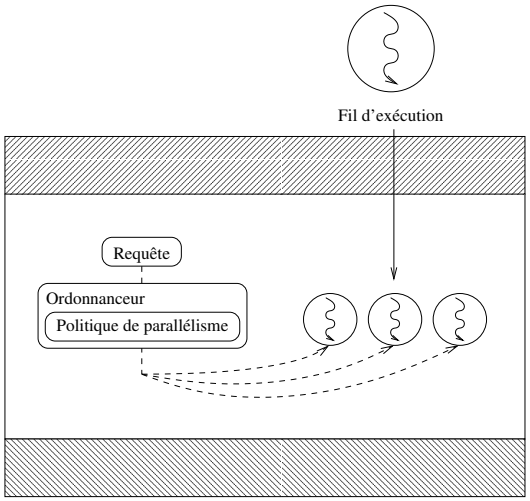


FIG. 4.15 – Fonction *exécution*

4.5.7 Exécution

Fonction de la couche applicative. Cf. figure 4.15.

Un appel de méthode correspondant à une requête reçue doit être affecté à une tâche, et être orienté vers le sous-programme applicatif approprié.

Celle-ci peut appartenir à l'application, et exécuter du code de l'intergiciel à la suite d'un appel à un sous-programme d'une personnalité applicative. Il peut également s'agir d'une tâche créée par l'intergiciel lui-même.

La tâche utilisée peut être une tâche prêtée temporairement à l'intergiciel par l'application, ou bien une tâche propre de l'intergiciel. Dans ce dernier cas, la tâche peut être banalisée, ou bien dédiée à l'exécution des requêtes liées à une entité particulière, par exemple une session donnée.

Lorsqu'une requête reçue est destinée à un objet local, son affectation à une tâche est réalisée par l'ordonnanceur. Les différentes variations du comportement de la fonction d'exécution sont mises en œuvre par délégation d'une partie des fonctionnalités de l'ordonnanceur à des objets de type *politique de parallélisme de l'intergiciel*.

La détermination du sous-programme applicatif correspondant est réalisée sous le contrôle de la personnalité applicative, en fonction de l'interface que lui présentent les objets applicatifs.

4.6 Processus de personnalisation

Notre architecture offre un mécanisme de mise en relation entre objets d'applications, ainsi qu'une infrastructure de transport de messages permettant la réalisation des échanges sous-tendus par les interactions entre les objets.

Deux facettes de l'intergiciel sont donc personnalisables :
- l'interface avec les objets d'application (« personnalité applicative ») ;
- l'interface avec d'autres intergiciels à travers l'architecture de transport (« personnalité protocolaire »).

La personnalisation de ces deux aspects peut être réalisée de manière indépendante. En effet, leur articulation est réalisée par une représentation des interactions entre objets générique et neutre vis-à-vis des personnalités

Personnalités applicatives

La réalisation d'une personnalité applicative consiste à mettre en œuvre une bibliothèque de programmation offrant :
- des structures de données représentant les références d'objets, construites à partir du type *Reference* générique défini par la couche neutre ;
- un mécanisme permettant d'effectuer des appels de méthodes sur ces objets, en construisant des objets *Request* génériques et en offrant une façade des primitives de ces objets ;
- des structures de données représentant les objets servants, qui doivent mettre en œuvre l'interface définie par la couche neutre, et qui de plus doivent pouvoir être dérivés par l'utilisateur pour réaliser ses objets applicatifs ;
- un mécanisme permettant à l'utilisateur d'enregistrer ses objets applicatifs, constituant une façade d'un adaptateur d'objets générique.

Ainsi, une personnalité applicative est essentiellement une façade sur les fonctionnalités des adaptateurs d'objets et sur les mécanismes d'appel de méthodes.

Personnalités protocolaires

La mise en œuvre d'une personnalité protocolaire est constituée de :
- la mise en œuvre d'une classe *Session*, regroupant les fonctions de création de messages à partir de requêtes et les fonctions réciproques de traitement des messages ;
- la mise en œuvre d'un procédé de représentation des données, définissant une méthode transformant un conteneur typé et la donnée qu'il contient en un flot d'octets, ainsi qu'une méthode réciproque reconstituant la donnée à partir d'un conteneur typé vide et du flot d'octets.
- la définition d'un ou plusieurs types de profils permettant de nommer les points d'accès de l'intergiciel et de les associer à la spécification d'un protocole ; ces profils doivent être associés à une étiquette qui identifie de façon unique une association entre protocole et mécanisme de transport ;
- la réalisation d'une usine de liaison capable d'instancier une session et la pile de transport et de filtrage sous-jacente à partir d'un profil ou d'une connexion reçue sur un point d'accès du service de transport.

La création d'une personnalité protocolaire consiste à réaliser les fonctions de protocole et de représentation correspondantes, ainsi que des usines de profils et de liaisons permettant leur utilisation par la couche neutre.

Intégration dans l'architecture

Une fois les composants personnalisés réalisés, comme nous l'avons décrit dans les sections précédentes, ils doivent être intégrés dans l'architecture d'intergiciel.

À cet effet, une opération de configuration de la couche neutre doit être effectuée. Cette opération consiste à déclarer les instances des composants des

personnalités souhaitées auprès du noyau générique. Ainsi enregistrés, ces composants peuvent jouer leur rôle dans l'intergiciel.

4.7 Patrons de conception

Un patron de conception (*design pattern*) réunit l'identification d'un problème récurrent dans la construction d'applications et d'une « bonne approche » pour le résoudre [31].

Les objets participants et leurs relations sont formalisés ; à cette description est associée une solution générale, qui peut être réutilisée dans chaque contexte particulier où le problème se présente. Un patron de conception traduit l'essence d'un problème qui se présente de façon récurrente, sous une multitude de formes particulières, au cours de la conception d'une application.

De nombreux patrons interviennent dans notre architecture. Ils autorisent la mutualisation de code générique, réutilisable par plusieurs sous-systèmes. Ils permettent également de faire apparaître clairement les structures récurrentes dans la mise en œuvre, et de restreindre le nombre de réalisations différentes d'une même structure. Ils contribuent ainsi à la lisibilité du code, facilitent sa maintenance et son évolution, garantissant ainsi sa pérennité.

4.7.1 Patrons comportementaux

Wrapper façade

Le développement d'un intergiciel nécessite l'utilisation de primitives de bas niveau du système d'exploitation, notamment pour accéder aux mécanismes de transport de données et aux fonctions de parallélisme. L'utilisation directe de ces fonctions rend l'intergiciel difficilement adaptable à de nouvelles interfaces, et compromet sa portabilité.

Le patron *Wrapper façade* [99, 93] consiste à encapsuler ces interfaces dans des classes uniformes. Par exemple, les fonctions de communication à travers un réseau sont réalisées, pour chaque variante de système d'exploitation et d'interface de programmation, comme une dérivation concrète d'une classe abstraite dont la spécification définit la fonctionnalité commune qu'elles offrent.

Cette encapsulation permet aux autres composants de l'intergiciel de s'affranchir des différences entre plates-formes, et d'accéder aux fonctions du système d'exploitation d'une manière homogène avec l'organisation de l'intergiciel : elle fait du système un composant comme un autre, et en offre une abstraction qui permet de compartimenter les dépendances.

Cette compartimentation est essentielle pour que le coût du portage de l'intergiciel d'un environnement vers un autre soit raisonnable. En effet, elle garantit que le portage de l'ensemble ne sera pas plus complexe que le portage de chacune des interfaces élémentaires : elle prévient l'explosion combinatoire de la difficulté du portage de l'intergiciel.

De même que dans TAO, une bibliothèque uniforme de transport de données est utilisée (ACE, ADAPTIVE Communication Environment), nous proposons une bibliothèque de transport pouvant encapsuler divers supports de communication et offrir aux utilisateurs une interface unifiée.

De même, les différents profils de parallélisme possibles (qui constituent les termes de l'alternative correspondant à une instance du patron *Strategy*) uti-

lisent les constructions de parallélisme et de synchronisation d'Ada 95 pour accéder aux ressources du système d'exploitation. Ainsi, les constructions du langage et sa bibliothèque d'exécution jouent également le rôle d'une façade par rapport aux primitives du système.

Acceptor-Connector

Dans un intergiciel utilisant un protocole basé sur des connexions, l'établissement d'une session est essentiellement asymétrique : il suppose qu'un intergiciel dans le rôle de serveur attende passivement une connexion entrante, tandis qu'un second initie activement la connexion. En revanche, une fois la connexion établie, le transport des données entre les deux extrémités est essentiellement symétrique.

Il est souhaitable de pouvoir choisir et modifier aisément le mécanisme d'établissement de connexions entre intergiciels. En particulier, il peut être nécessaire d'effectuer une réservation de ressources pour la connexion à établir, si des critères de qualité de service doivent être garantis. De plus, il est intéressant de permettre une dissociation entre la dissymétrie des rôles dans l'opération d'établissement des connexions et la symétrie de l'échange qui s'ensuit. Par exemple, une fois une connexion établie entre deux intergiciels CORBA utilisant le protocole GIOP, l'utilisateur peut souhaiter que chaque intergiciel puisse se comporter à la fois comme un client et comme un serveur, comme l'y autorise l'extension *Bidirectional GIOP*.

À cet effet, le patron *Acceptor-Connector* [96, 99] est introduit. Il consiste à dissocier les rôles d'établissement de sessions entre des composants *Connecteurs* actifs et des composants *Accepteurs* passifs, chacun se comportant comme des usines d'objets de liaison entre objets, ces derniers étant symétriques.

Dans notre architecture, ce patron est réalisé par les usines d'objets de liaison clients et serveurs de la couche protocolaire.

4.7.2 Patrons de parallélisme

Ces patrons régissent les interactions entre les fils d'exécution présents sur un nœud.

Half-sync/Half-async

Pour simplifier la programmation des objets applicatifs, il est souhaitable que ceux-ci puissent faire appel à des fonctionnalités de communications synchrones, qui bloquent l'appelant dans l'attente d'une donnée qui doit être reçue d'un nœud distant.

Cependant, la conception d'un moteur de communication efficace nécessite l'utilisation de primitives de communication non bloquantes au plus bas niveau, afin que les couches de communication ait un fonctionnement le plus proche possible des mécanismes d'interruption utilisés par le matériel. L'utilisation de primitives non bloquantes est également essentielle pour ne pas imposer l'utilisation de multiples tâches à l'intérieur de l'intergiciel.

Le patron *Half-sync/Half-async* [95, 99] est donc introduit pour découpler le sous-système de bas niveau, asynchrone, du sous-système de haut niveau, synchrone. Dans ce patron, les couches de communication de bas niveau utilisent

des primitives de communication asynchrones, et insèrent les messages reçus dans une file d'attente qui permet aux couches de traitement de haut niveau de les manipuler de façon synchrone.

Dans notre architecture, l'intégralité de la couche protocolaire est asynchrone. L'ensemble du traitement des messages est dirigé par les événements survenus sur les points de terminaison du service de transport, qui sont transmis aux piles de protocoles.

Les objets d'application utilisent des primitives synchrones offertes par la couche applicative ; l'ordonnanceur de la couche neutre permet le blocage des composants applicatifs jusqu'à ce que l'événement attendu arrive et soit signalé à la couche neutre par la couche protocolaire.

Reactor

L'intergiciel doit réagir à des événements provenant de l'extérieur, correspondant aux interactions avec des objets distants (établissement et rupture de connexions de transport, réception de données).

Le traitement à effectuer dépend de l'objet sur lequel il survient. Ainsi, un événement externe qui apparaît sur un point d'accès du service de transport doit être traité en acceptant une nouvelle connexion et en créant un nouveau point de terminaison. Un événement concernant un point de terminaison correspond à l'arrivée d'une donnée, et doit être traité en recevant cette donnée et en signalant sa présence à la pile de protocole associée.

Il est nécessaire que l'intergiciel puisse traiter de front des événements provenant de plusieurs sources distinctes. L'attente d'un événement sur une source ne doit pas bloquer le traitement d'événements survenant sur une autre. De plus, on souhaite ne pas imposer l'utilisation d'une tâche distincte pour chaque source possible d'événements asynchrones. En effet, les tâches peuvent avoir un coût élevé (encombrement en mémoire, changements de contexte). On souhaite également qu'un nœud puisse fonctionner avec un seul processus, le cas échéant.

On introduit donc le patron *Reactor* [92, 99]. Dans cette construction, un objet est responsable de l'attente d'un événement survenant sur un élément quelconque d'une collection de sources d'événements asynchrones. Le traitement d'une occurrence d'événement est délégué à un composant associé à la source qui l'a produite.

La boucle principale de l'intergiciel n'a alors plus à prendre en compte les différents mécanismes possibles d'attente d'événements, ni les différents traitements correspondant aux diverses classes de sources. Elle se contente de faire appel périodiquement à l'objet réacteur.

Active object

L'intergiciel doit pouvoir définir des politiques de parallélisme variées. Une requête reçue peut être traitée par une tâche « générique », appartenant à l'intergiciel. Elle peut également être assignée à une tâche spécialisée liée à l'un des éléments de structure de l'application répartie, par exemple un objet serveur, un objet client ou une session.

Le support de telles politiques nécessite l'existence d'une interface abstraite et uniforme pour l'affectation d'un travail à effectuer à une ressource d'exécution,

au-delà de la simple façade posée sur les primitives de parallélisme du système d'exploitation, selon le principe décrit dans la section 4.7.1.

Le patron de conception *Active object* [97, 99] dissocie la tâche qui reçoit une requête de celle qui la traite. La mise en œuvre des différentes stratégies d'affectation peut ainsi être réalisée indépendamment du fonctionnement des clients des objets actifs (qui se contentent de remettre un travail à effectuer à un relais offrant la même interface que l'objet réel) et de celui des objets réels qui exécutent ces travaux dans une tâche spécifique.

Nous utilisons ce patron pour la réalisation des différentes politiques de parallélisme de l'intergiciel. Ces politiques sont des stratégies pour l'opération consistant à faire exécuter une requête reçue par un objet applicatif local. Chaque politique définit un relais qui accepte les travaux à traiter, et les remet par l'intermédiaire de files d'attente à des objets chargés de leur exécution, qui possèdent leurs propres tâches.

4.7.3 Patrons de polymorphisme

Dans cette section, nous introduisons deux patrons inédits permettant à des objets constitutifs de l'intergiciel de coopérer en ayant une connaissance limitée de leurs pairs.

Component

Dans le modèle d'objets du langage Ada 95 [49], la dérivation des types correspond à une extension du comportement et des données du type ancêtre. Une instance d'un type dérivé *est* une instance du type ancêtre *avec* des champs de données supplémentaires, de nouvelles méthodes, et éventuellement des méthodes différentes. Les comportements du type ancêtre qui ne sont pas surchargés sont hérités à l'identique. La sémantique de la dérivation Ada 95 est donc essentiellement interne à la hiérarchie de types : elle représente la relation comportementale et informationnelle entre l'ancêtre et l'enfant. L'héritage multiple n'est pas supporté, car ce modèle de sémantique de la dérivation ne permet pas l'identification « naturelle » du comportement que doit adopter un fils héritant de plusieurs ancêtres simultanément.

Dans les modèles de types de CORBA et de Java, en revanche, l'existence de classes purement abstraites (les *interfaces*) autorise une dérivation de type dont la sémantique est *contractuelle* : lorsqu'un objet concret est dérivé d'une interface abstraite, le type dérivé *réalise*, *implémente* l'interface abstraite. Une interface abstraite est alors la spécification d'une vue externe d'un service qui peut être offert par un objet ; la sémantique de la dérivation est *externe* à la hiérarchie des objets, au sens où la relation de dérivation est alors une propriété qui caractérise les relations des objets ancêtres et dérivés avec leurs *clients*. L'héritage multiple prend ici tout son sens, puisqu'il signifie simplement qu'un objet offre plusieurs vues différentes.

Ce modèle de dérivation n'existe pas en tant que construction du langage Ada 95. De plus, nous souhaitons minimiser la visibilité qu'ont les clients d'un objet sur le type réel de cet objet (le même service, du point de vue du client, pouvant être rendu par des objets très différents et non liés par une relation d'héritage).

Nous proposons donc sa définition sous forme d'un patron de conception : *Component*. Dans ce patron, chaque composant a la propriété de pouvoir recevoir et traiter des messages ; des collections de classes de messages constituent les interfaces entre composants. Ces interfaces sont dissociées de la hiérarchie de types. Par cet aspect, notre patron *Component* se rapproche du patron *External polymorphism* [17].

Chaque composant est muni de ports qui lui permettent de communiquer avec d'autres. Certains composants particuliers se contentent de relayer les messages reçus vers d'autres ports : ils jouent le rôle de connecteurs, et peuvent réaliser des modalités de communication évoluées telles que la diffusion de groupe. Les autres composants reçoivent les messages, déterminent s'ils font partie des interfaces qu'ils supportent, et répondent le cas échéant après avoir effectué le traitement correspondant.

Des messages synchrones peuvent ainsi être échangés. Lorsqu'un composant reçoit un message, sa méthode *Handle_Message* est exécutée. Ce patron rend explicite l'équivalence entre un appel de méthode et un passage de message [47].

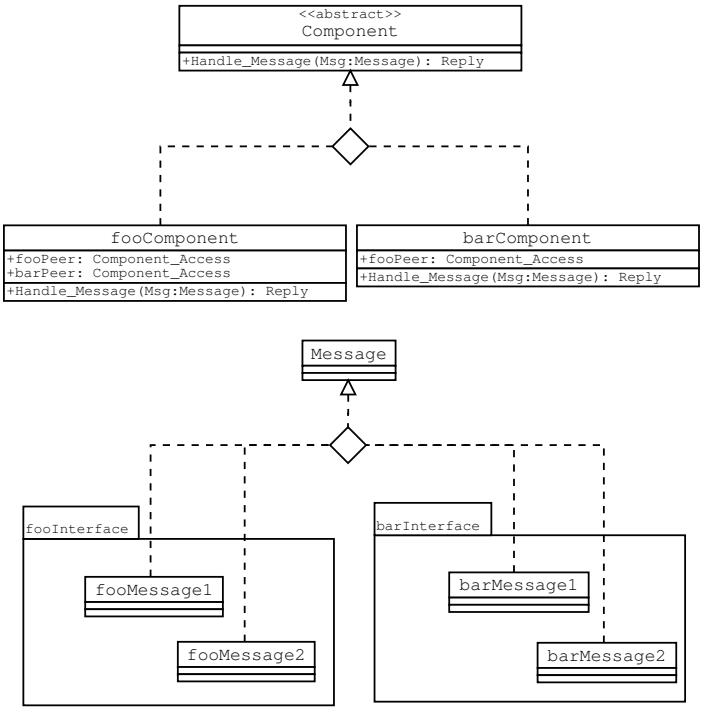


FIG. 4.16 – Patron « Component »

Dans la suite de ce mémoire, nous entendons toujours le terme « composant » au sens de ce patron.

Annotation

De la même façon que le patron *Component*, décrit dans la section précédente, vise à dissocier la relation entre une interface et les objets qui la mettent en œuvre de la relation de dérivation de type, il apparaît souhaitable de pouvoir

dissocier la relation entre un ensemble de données associées à un objet et le type de cet objet.

Plus spécifiquement, nous souhaitons qu'un objet puisse *contenir* des informations qui n'ont de sens que pour ses clients, sans que cet objet ait pour autant besoin de connaître les détails internes de ceux-ci.

En effet, si les clients devaient maintenir eux-mêmes une structure de données associant l'identité de chaque objet utilisé avec l'information pertinente, il serait nécessaire qu'il effectue constamment des recherches, potentiellement coûteuses, dans cette structure de données. De plus, il serait nécessaire de mettre en œuvre des mécanismes de coordination pour assurer la cohérence de la structure : lorsqu'un objet disparaît, il est nécessaire de supprimer les informations associées.

Ces considérations d'efficacité et de cohérence conduisent à introduire le patron *Annotation*. Pour permettre aux utilisateurs d'un objet de lui associer efficacement des données spécifiques, l'objet expose un attribut qui est une collection d'instances d'une classe-conteneur abstraite *Annotation*, étiquetées par l'identité des clients. Chaque client dérive la classe *Annotation* à sa convenance, et peut ajouter ou retirer une instance de son annotation concrète dans la collection.

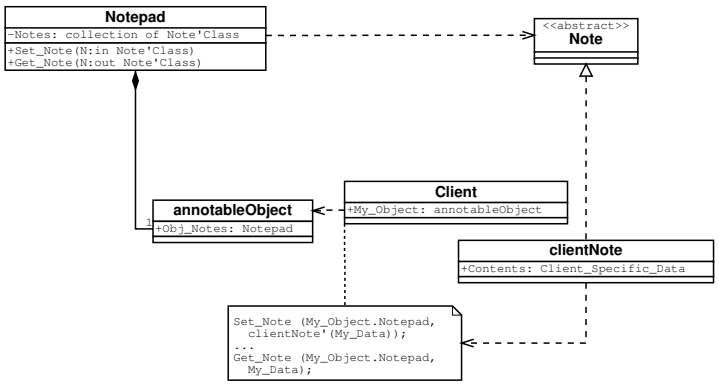


FIG. 4.17 – Patron « Annotation »

4.7.4 Synthèse

L'identification des patrons pertinents facilite la définition d'une architecture d'intergiciel modulaire et évolutive. L'utilisation des solutions canoniques de ces patrons garantit la portabilité et la pérennité de l'intergiciel, en rendant aisé son extension et sa maintenance.

Les patrons de polymorphisme (*Component*, *Annotation*) permettent la définition souple de contrats et de relations entre entités, sans pour autant imposer aux utilisateurs d'un objet plus de connaissance qu'il n'est strictement nécessaire concernant l'objet utilisé. Ces patrons favorisent le développement de fonctionnalités de façon compartimentée et indépendante.

Les patrons comportementaux *Strategy* et *Abstract Factory*, que nous avons décrits dans la section 3.2.2, sont utilisés pour rendre l'intergiciel configurable en fonction des besoins de l'application et des contraintes de l'environnement.

Le patron *Wrapper façade* permet, de même, de rendre l'intergiciel adaptable aux interfaces accessibles pour exploiter les services du système d'exploitation et du matériel.

Le patron comportemental *Acceptor-connector* décrit le découplage des différents rôles dans l'opération d'établissement des liaisons entre objets. Il permet aux sessions de protocole d'être indépendantes des mécanismes d'établissement des connexions de transport. Les rôles des objets d'application peuvent ainsi être rendus indépendants des moyens utilisés par les intergiciels pour les mettre en relation. Ce patron contribue donc au découplage entre les objets applicatifs et la couche protocolaire.

Les patrons de parallélisme qui sont identifiés (*Half-sync/Half-async*, *Active object*) mettent en évidence les synchronisations entre les composants ; les solutions proposées prennent en compte les problèmes inhérents à la réalisation de service temps-réel déterministes. La partie protocolaire est basée sur le traitement d'événements : sa structure générale est décrite par le patron *Reactor*.

4.8 Synthèse

Dans ce chapitre, nous avons introduit la notion d'intergiciel schizophrène. Un intergiciel schizophrène est caractérisé par le découplage des personnalités applicatives et protocolaires, et l'introduction d'une couche neutre à l'interface entre ces personnalités.

Nous avons présenté une application témoin qui met en évidence l'apport des intergiciels schizophrènes dans la problématique de l'interopérabilité des systèmes d'objets répartis.

Nous avons présenté notre architecture pour un intergiciel schizophrène. Nous avons montré que cette architecture permet de répondre aux *besoins fonctionnels* des applications utilisant les objets répartis, et simultanément de satisfaire les exigences non fonctionnelles tendant à résoudre le *paradoxe de l'intergiciel*.

La conception de cette architecture fait usage de nombreux patrons de conception, que nous avons détaillés, et qui contribuent à la rendre évolutive, modulaire et à en faciliter la maintenance.

Deuxième partie

Réalisation

CHAPITRE

5 Réalisation des services de base

Sommaire

5.1 Initialisation 123
5.1.1 Élaboration d'un programme Ada 95 124
5.1.2 Gestionnaire d'initialisation 125
5.2 Parallélisme et synchronisation 126
5.2.1 Configurabilité du parallélisme 126
5.2.2 Interface des objets primitifs 127
5.2.3 Profils de parallélisme 128
5.2.4 Objets de synchronisation composés 130
5.3 Structures de données de haut niveau 131
5.3.1 Pointeurs à comptage de références 131
5.3.2 Dictionnaires 134
5.4 Patrons de conception 136
5.4.1 Composants 136
5.4.2 Annotations 138
5.5 Synthèse 139

La construction de notre intergiciel utilise un ensemble de services de base, évoqués par la section 4.4.2. Ces services ne sont pas par eux-mêmes spécifiques aux questions de répartition ; ils constituent une « boîte à outils » qui peut être utilisée par l'ensemble des composants de l'intergiciel (personnalités applicatives, couche neutre et personnalités protocolaires).

5.1 Initialisation

Cette section présente le sous-système d'initialisation de PolyORB. Celui-ci permet aux différents modules constitutifs de l'intergiciel d'être initialisés lors du démarrage d'un nœud. Certains modules doivent impérativement avoir été initialisés avant que ceux qui utilisent leurs fonctionnalités puissent l'être. Il existe alors une *dépendance d'initialisation* des modules utilisateurs vis-à-vis des modules utilisés. Le sous-système d'initialisation doit tenir compte de ces dépendances lorsqu'il détermine l'ordre dans lequel les modules sont initialisés.

5.1.1 Élaboration d'un programme Ada 95

Définition 21 (élaboration) *Lorsqu'une partition Ada est exécutée, les déclarations des unités qui la constituent prennent effet selon un processus normalisé. Au cours de cette phase, les variables qui doivent être initialisées le sont, de la mémoire peut être réservée, des tâches créées. Ce processus constitue* l'élaboration *de la partition.*

Toutes les unités de compilation constituant une partition sont élaborées avant que sa procédure principale soit exécutée. L'ordre d'élaboration des unités est notamment déterminé par des contraintes imposées par la norme.

Celle-ci fixe en effet une relation d'ordre partiel de *dépendance d'élaboration* $\prec_{\text{elab}}$ sur l'ensemble des unités de compilation d'une partition. Cette relation est destinée à éviter que l'élaboration d'une unité fasse appel à des objets situés dans une autre unité qui n'aurait pas encore été élaborée.

On peut concevoir de manière équivalente la relation $\prec_{\text{elab}}$ comme un graphe acyclique orienté dont les nœuds sont les unités de compilation, et dont les arcs représentent les dépendances d'élaboration entre unités.

La relation de dépendance d'élaboration est une extension de la relation d'ordre partiel de dépendance sémantique $\prec_{\text{sem}}$: lorsqu'une unité de compilation Q (qui peut être la spécification ou le corps d'un paquetage, ou le corps d'un sous-programme isolé) rend visibles les déclarations d'une spécification de paquetage ou la déclaration d'un sous-programme isolé P (au moyen d'une clause `with P ;`), il est nécessaire que la spécification de P soit élaborée avant Q :

$$(P \prec_{\text{sem}} Q) \Rightarrow (P \prec_{\text{elab}} Q).$$

Cependant, cette seule règle de dépendance imposée par la norme ne suffit pas, en général, pour garantir qu'une unité ne sera élaborée que lorsque toutes les unités effectivement nécessaires pour ce faire l'auront été.

En effet, elle n'assure pas que le corps de P soit élaboré avant celui de Q. Un accès incorrect à des structures non encore élaborées demeure possible.

Ainsi, dans l'extrait de code 1, les dépendances sémantiques $P_{body} \prec_{\text{sem}} P_{spec}$ et $Q_{spec} \prec_{\text{sem}} P_{spec}$ ne suffisent pas à garantir que le corps du paquetage P est élaboré avant la déclaration du paquetage Q, alors que l'élaboration de cette dernière accède à la variable X.

Pour obtenir cette garantie, l'utilisateur ajoute au programme des directives de compilation qui ajoutent des arcs dans le graphe de dépendance d'élaboration, afin d'obtenir avec certitude un ordre correct, en fonction des appels particuliers nécessaires à l'élaboration de chaque unité.

Dans toute partition légale, $\prec_{\text{elab}}$ doit être un ordre partiel. Il admet alors une extension linéaire, qui est un ordre (total) d'élaboration garantissant que toute unité sera élaborée après toutes celles dont elle dépend.

Le compilateur GNAT peut détecter certaines situations risquant de poser des problèmes d'ordre d'élaboration, et suggérer l'ajout de telles directives. Si les directives suggérées sont mises en place, le compilateur peut déterminer statiquement un ordre d'élaboration qui garantit la correction de tous les appels.

Cependant, ces directives suggérées ne peuvent pas toujours être introduites en préservant la légalité du programme. En effet, elles sont susceptibles d'introduire des contraintes superflues, qui violeraient l'antisymétrie de l'ordre partiel $\prec_{\text{elab}}$ (en introduisant des dépendances circulaires d'élaboration). Dans un tel

Extrait de code 1 Élaboration et dépendance sémantique

```
package P is
   function F return Integer;
end P;
```

(a) P_{spec}

```
with P;
package Q is
   XX : Integer := P.F;
end Q;
```

(b) Q_{spec}

```
package body P is
   X : Integer;
   function F is
   begin
      return X;
   end F;
begin
   X := 123;
end P;
```

(c) P_{body}

cas, on doit renoncer à la sûreté offerte par la détermination statique de l'ordre d'élaboration (qui permet d'avoir, à la compilation, la garantie d'une élaboration correcte et déterministe), ou bien réorganiser le programme pour limiter les dépendances d'élaboration et briser les cycles.

5.1.2 Gestionnaire d'initialisation

Dans PolyORB, nous souhaitons que le processus d'élaboration soit le plus simple et le plus sûr possible. Nous utilisons donc le modèle statique d'élaboration de GNAT. Cependant, les divers composants d'un intergiciel entretiennent des relations complexes. Les contraintes conservatrices imposées par ce modèle d'élaboration sont trop restrictives pour rendre compte des dépendances entre modules de l'intergiciel au moment de l'initialisation.

Nous proposons donc un mécanisme d'initialisation distinct de l'élaboration Ada 95, dont nous donnons dans les sections suivantes la spécification et les grandes lignes de la réalisation.

Interface

La spécification du paquetage d'initialisation définit une structure de données destinée à contenir l'ensemble des informations associées à un module de l'intergiciel :

- son nom ;
- les noms des fonctionnalités qu'il fournit ;
- les noms des modules dont il dépend (qui doivent être initialisés avant lui) ;
- ceux des modules avec lesquels il est incompatible ;
- un pointeur sur sa procédure d'initialisation.

Chaque module, lors de son élaboration, enregistre une instance de cette structure de données auprès du sous-système d'initialisation. Une fois l'ensemble des unités élaborées, l'initialisation de l'intergiciel est déclenchée : la procédure d'initialisation globale vérifie l'absence de conflits et de dépendances circulaires entre les modules enregistrés, puis appelle tour à tour leurs procédures d'initialisation dans un ordre satisfaisant les dépendances indiquées à l'enregistrement.

Mise en œuvre

Lors de l'enregistrement d'un module, les informations le concernant sont stockées dans une simple liste de modules.

Une fois tous les modules enregistrés, la procédure d'initialisation globale parcourt cette liste. Pour chaque module, elle vérifie qu'aucun module incompatible n'est enregistré. Un dictionnaire associant chaque nom de module à un pointeur sur les informations correspondantes est établi, qui permet de construire le graphe des dépendances. Un tri topologique sur ce graphe est effectué. Lorsque chaque nœud est traversé, sa procédure d'initialisation est appelée. Si un cycle est détecté, l'initialisation se termine par une levée d'exception. Dans le cas contraire, tous les modules sont initialisés dans un ordre qui satisfait les contraintes de dépendance.

5.2 Parallélisme et synchronisation

Dans cette section, nous décrivons les diverses unités qui offrent aux composants de PolyORB l'accès aux primitives de parallélisme (création, acquisition, restitution et destruction de tâches) et de synchronisation entre tâches (sections critiques et variables conditionnelles).

5.2.1 Configurabilité du parallélisme

Dans certaines situations, on souhaite qu'un nœud d'une application répartie ne fasse pas usage de fonctionnalités évoluées de l'environnement, telles que le support du parallélisme et de la communication entre tâches concurrentes. En effet, ces fonctionnalités ont un coût important :

- elles nécessitent des bibliothèques et des fonctionnalités spécialisées de l'environnement ;
- à l'exécution, elles introduisent la nécessité d'actions de synchronisation.

De plus, le parallélisme peut introduire dans l'application des comportements non déterministes.

Il est nécessaire de définir des sous-ensembles des fonctionnalités de l'intergiciel qui réduisent l'utilisation de certaines constructions parallèles complexes. Ces *profils de fonctionnalités* correspondent à des compromis différents entre la richesse des fonctionnalités offertes par l'intergiciel d'une part, la consommation de ressources et d'autre part le niveau de fonctionnalité requis du système d'exploitation.

GLADE proposait une gestion configurable du parallélisme, en autorisant l'utilisation soit du profil de parallélisme complet, soit d'un profil sans tâches. Nous nous sommes inspiré de cette fonctionnalité pour spécifier une nouvelle bibliothèque de parallélisme configurable pour PolyORB [41], réalisée au cours de travaux de DEA à l'encadrement desquels nous avons participé.

Cette bibliothèque spécifie un ensemble de primitives de parallélisme, que nous présentons dans la section 5.2.2, et offre trois réalisations concrètes de ces primitives :

- une version sans tâches destinée aux nœuds n'utilisant qu'une seule tâche pour l'intergiciel et les objets applicatifs ;
- une version basée sur un profil restreint du langage Ada 95, le profil Ravenscar, qui interdit les constructions de parallélisme non déterministes ;

– une version autorisant l'utilisation sans restrictions de toutes les constructions parallèles du langage Ada 95.

5.2.2 Interface des objets primitifs

L'accès aux différentes variantes de la bibliothèque de parallélisme et de synchronisation se fait en utilisant des interfaces abstraites offrant trois classes d'objets de base : les tâches, les sections critiques et les variables conditionnelles. Le choix de ces objets et des opérations associées est inspiré par l'interface normalisée POSIX [50]. Les sections critiques (ou moniteurs) et les variables conditionnelles ont été proposées comme primitives pour la synchronisation entre tâches concurrentes par C. A. R. Hoare [44].

Chaque version (sans tâches, Ravenscar, parallélisme général) consiste en des dérivations concrètes des classes de base. Le passage par une interface de programmation uniforme plutôt que l'utilisation directe des fonctionnalités de parallélisme et de synchronisation du langage permet la configuration de ces aspects du fonctionnement de l'intergiciel. En particulier, nous indiquons dans la section suivante comment elle permet la mise en œuvre de divers profils fonctionnels.

Processus légers

Chaque tâche utilisable par l'intergiciel est représentée par un objet ; les méthodes de cet objet permettent d'affecter à la tâche un travail à exécuter. Chaque unité d'activité susceptible d'être affectée à une tâche est représentée par un objet possédant une unique primitive sans paramètres, correspondant à son exécution. Cette représentation est empruntée à la classe *Runnable* du langage Java.

La bibliothèque de parallélisme offre une usine de tâches, qui prend en paramètre un travail à exécuter et le confie à une tâche nouvellement créée (ou prise parmi un ensemble de tâches inoccupées).

Sections critiques

Les sections critiques de l'intergiciel sont les régions du code dans lesquelles une seule tâche est autorisée à se trouver à un instant donné. La mise en place de sections critiques permet de garantir que les accès aux données partagées par plusieurs tâches sont cohérents. Elles sont réalisées au moyen de verrous d'exclusion mutuelle, c'est-à-dire d'objets de synchronisation disposant de deux primitives Enter et Leave, correspondant respectivement à l'entrée en section critique et à la sortie (extrait de code 2).

La première est bloquante lorsqu'une tâche se trouve déjà dans la section critique concernée. Par conséquent, il est fautif pour une tâche d'appeler Enter sur le verrou d'une section critique qu'elle occupe déjà. La seconde signale qu'une tâche quitte la section critique, et permet à une autre tâche bloquée d'y pénétrer, le cas échéant.

Variables conditionnelles

Les variables conditionnelles permettent à une tâche en section critique d'attendre qu'un événement se produise [44]. La tâche est considérée comme ayant

Extrait de code 2 Interface des sections critiques

```
package PolyORB.Tasking.Mutexes is

   type Mutex_Type is abstract tagged limited private;
   --  Mutual exclusion locks (mutexes) prevent multiple threads
   --  from simultaneously executing critical sections of code
   --  which access shared data (that is, mutexes are used to seri-
   --  alize the execution of threads). While a thread is in the
   --  critical section protected by a mutex, it is designated as
   --  the owner of that mutex.

   procedure Enter (M : in out Mutex_Type)
      is abstract;
   --  A call to Enter locks mutex object M. If M is already locked,
   --  the caller is blocked until it gets unlocked. On exit from Enter,
   --  M is locked, and the caller is the owner.
   --  If the current owner of a mutex tries to enter it again, a
   --  deadlock occurs.

   procedure Leave (M : in out Mutex_Type)
      is abstract;
   --  Release M. M must be locked, and the caller must be the owner.
   --  The scheduling policy determines which blocked thread is waken
   --  up next and obtains the mutex.
   --  It is erroneous for any process other than the owner of a mutex
   --  to invoke Release.

end PolyORB.Tasking.Mutexes;
```

quitté la section critique pendant la durée de son attente, afin de garantir que toute occupation de section critique soit de durée bornée.

Les variables conditionnelles offrent un mécanisme d'attente d'un signal non persistant : lorsqu'une tâche est bloquée dans la primitive d'attente, elle sera réveillée lorsqu'une autre tâche appellera l'une des primitives de déblocage. Si une primitive de déblocage est appelée alors qu'aucune tâche n'est bloquée, ce déblocage est perdu.

L'extrait de code 4 donne un exemple simple, tiré de [44], d'utilisation de sections critiques et de variables conditionnelles : la gestion de l'accès exclusif à une ressource par des processus concurrents.

5.2.3 Profils de parallélisme

Profil sans tâches

Lorsque l'intergiciel est configuré pour ne pas introduire de tâches et que les objets applicatifs n'en utilisent pas eux-mêmes, une version concrète de cette interface est utilisée qui n'effectue aucune opération. L'entrée et la sortie d'une région critique sont des procédures vides. Le signalement d'une variable conditionnelle est également vide.

En revanche, l'attente sur une variable conditionnelle dans un contexte sans tâches bloquerait indéfiniment, car aucune tâche autre ne pourrait activer le signal de la variable. Elle témoigne donc d'une erreur, et se traduit par la levée d'une exception. De même, toute tentative de création de tâche lève une exception.

Dans la mesure où le profil sans tâches de PolyORB n'utilise aucune fonctionnalité de parallélisme du langage Ada 95 (ni tâches, ni objets protégés), les nœuds qui l'utilisent n'ont pas besoin d'incorporer l'exécutif de parallélisme, ni ne font appels aux fonctionnalités correspondantes du système d'exploitation. Il

Extrait de code 3 Interface des variables conditionnelles

```
package PolyORB.Tasking.Condition_Variables is

   type Condition_Type is abstract tagged limited private;
   --  Type for condition variables.

   procedure Wait
      (C : in out Condition_Type;
       M : access Mutex_Type'Class)
       is abstract;
   --  Wait for a notification on condition variable C.
   --  This procedure atomically:
   --    (1) leaves the critical section protected by M (which the
   --         caller must own);
   --    (2) blocks until a subsequent notification from another
   --        task (via the Signal or Broadcast operations below);
   --  On return, M is owned again.

   procedure Broadcast
      (C : in out Condition_Type)
       is abstract;
   --  Unblock all tasks blocked on C.

   procedure Signal
      (C : in out Condition_Type)
       is abstract;
   --  Unblock one task blocked on C.

end PolyORB.Tasking.Condition_Variables;
```

Extrait de code 4 Exemple d'utilisation de sections critiques et de variables conditionnelles

```
package body Resource is

   Monitor  : Mutex;
   Released : Condition_Variable;

   Busy : Boolean := False;
   --  Is the resource currently busy?

   procedure Acquire is
   begin
      Enter (Monitor);
      while Busy loop
         Wait (Released, Monitor);
         --  While this task is waiting for Condition_Variable to be
         --  signalled, Monitor is released; when the wait ends, Monitor
         --  is asserted again.
      end loop;
      Busy := True;
      Leave (Monitor);
   end Acquire;

   procedure Release is
   begin
      Enter (Monitor);
      Busy := False;
      Signal (Released);
      Leave (Monitor);
   end Release;

end Resource;
```

est donc possible de mettre en place une configuration d'intergiciel léger.

Profil Ravenscar (parallélisme déterministe)

Le profil Ravenscar est un sous-ensemble des fonctionnalités de parallélisme du langage Ada 95 [23]. Ce profil est destiné à la réalisation de systèmes embarqués, temps réel et sûrs. Il interdit certaines constructions, afin que le système résultant présente un comportement déterministe. La preuve statique de certaines propriétés de temps réel et de correction du code applicatif (absence d'interblocage, d'inversion de priorité, de famine) est alors facilitée.

L'un des aspects importants du profil Ravenscar est l'absence de création ou de destruction dynamiques de tâches. Lorsqu'un nœud est configuré avec ce profil, toutes ses tâches sont créées lors de l'élaboration, et leur exécution se poursuit en principe jusqu'à l'arrêt du nœud. Le mécanisme de création et de destruction de tâches de PolyORB est donc ici simulé en créant, lors de l'élaboration, un réservoir de tâches. Lorsque l'intergiciel demande la création d'une nouvelle tâche, une tâche libre est recherchée dans le réservoir, et le traitement à effectuer lui est confié. Lorsque celui-ci s'achève, la tâche retourne dans le réservoir de tâches libres en attente d'une nouvelle occupation.

Le profil Ravenscar interdit également que plusieurs tâches soient bloquées sur un même objet protégé. En d'autres termes, la gestion des files d'attente pour l'accès à des ressources partagées doit être géré explicitement par le programmeur, et non par l'exécutif restreint. Dans PolyORB, nous associons donc un objet de synchronisation à chaque tâche, qu'elle utilise lorsqu'elle doit être bloquée.

L'admission des tâches applicatives (en plus des tâches créées de façon interne par l'intergiciel) dans cette organisation nécessite qu'un tel verrou soit alloué non seulement pour les tâches internes, mais également pour les tâches applicatives. Un réservoir de verrous est donc également maintenu, qui permet de bloquer une tâche applicative lorsque celle-ci cherche à accéder à une section critique de l'intergiciel.

Profil avec parallélisme général

Le profil avec parallélisme général autorise l'emploi de toutes les constructions parallèles du langage Ada 95. La synchronisation entre les tâches est effectuée au moyen d'objets protégés. Dans ce profil, l'utilisateur peut créer dynamiquement des tâches.

Il est également possible de configurer l'intergiciel pour utiliser des politiques tirant partie de cette possibilité. En contrepartie, l'intergiciel devient dépendant de l'exécutif de parallélisme du langage. Un coût supplémentaire (encombrement mémoire, synchronisation) est donc supporté.

5.2.4 Objets de synchronisation composés

Les primitives de parallélisme de base (tâches, sections critiques et variables conditionnelles) sont suffisantes pour exprimer toutes les situations de synchronisation rencontrées dans l'intergiciel.

Cependant, elles ne sont pas nécessairement les abstractions de choix pour exprimer certaines situations de concurrence. Nous avons donc introduit des

objets de synchronisation composés à partir de ces primitives de base. Ceux-ci sont réalisés de manière générique, indépendamment du profil de parallélisme choisi : ils utilisent uniquement les primitives décrites dans la section 5.2.2.

Sections critiques pour appels récursifs

Certains traitements ne doivent pas être réalisés simultanément par deux tâches distinctes, mais doivent en revanche autoriser des appels récursifs par une même tâche. Ces traitements ne peuvent donc pas être simplement réalisés dans une section critique. En effet, les appels récursifs seraient alors interdits, car la tâche qui occupe la section critique ne pourrait elle-même y entrer de nouveau.

Par exemple, certaines fonctions nécessitant un accès à des structures internes de l'ordonnanceur doivent être effectuées en section critique pour garantir la cohérence des informations obtenues. Cependant, ces fonctions peuvent également être appelées au cours de traitements modifiant l'état de l'ordonnanceur, qui eux-mêmes sont effectués en section critique.

Ces traitements sont donc protégés par un verrou de section critique réentrant. Lorsqu'un tel verrou est détenu par une tâche, celle-ci peut à nouveau le prendre. Une autre tâche demandant à entrer dans la même section critique est bloquée jusqu'à ce que le verrou ait été libéré autant de fois qu'il a été pris.

Autres constructions classiques de synchronisation

Plusieurs objets de synchronisation classiques sont utiles pour la construction d'un intergiciel. Notre bibliothèque de parallélisme propose donc notamment des sémaphores et des verrous lecteurs/écrivains.

Les sémaphores [22] permettent de mettre un ensemble de tâches en attente d'occurrences d'un événement, et de réveiller exactement autant de tâches que d'occurrences survenues. Les sémaphores sont une construction de synchronisation plus générale, mais aussi plus coûteuse, que les sections critiques. Nous les réalisons au moyen d'une section critique et d'une variable conditionnelle.

Les verrous lecteurs/écrivains permettent de coordonner l'accès à une donnée partagée qui est :

- consultée par de nombreuses tâches « lecteurs » ;
- modifiée par un faible nombre de tâches « écrivains ».

Dans une telle situation, il est intéressant de permettre à plusieurs tâches de lire simultanément la donnée partagée. Il est en revanche nécessaire que les accès en écriture soient sérialisés, et qu'aucun accès en lecture ne se produise tant qu'un accès en écriture est en cours. Ainsi, un verrou lecteurs/écrivains offre deux couples de primitives de verrouillage et déverrouillage (l'un pour les lecteurs, l'autre pour les écrivains), comme illustré par l'extrait de code 5.

5.3 Structures de données de haut niveau

5.3.1 Pointeurs à comptage de références

Un intergiciel est susceptible de manipuler de nombreuses structures de données allouées dynamiquement et faisant référence les unes aux autres. L'utilisation de nombreux pointeurs est une source classique de problèmes :

Extrait de code 5 Interface des verrous lecteurs/écrivains

```
package PolyORB.Tasking.Rw_Locks is

   --  Several tasks can own the lock in read (shared) mode.
   --  Only one task can own the lock in write (exclusive) mode.
   --  No task can own the lock in exclusive mode while any other
   --  task owns it in read mode.

   --  The contention resolution policy gives priority to
   --  writers: as long as a task is blocked waiting for
   --  write access, requests for read access are declined.

   type Rw_Lock_Type is limited private;

   procedure Lock_W (L : access Rw_Lock_Type);
   --  Get lock in write mode.

   procedure Lock_R (L : access Rw_Lock_Type);
   --  Get lock in read mode.

   procedure Unlock_W (L : access Rw_Lock_Type);
   --  Release write mode lock.

   procedure Unlock_R (L : access Rw_Lock_Type);
   --  Release read mode lock.

end PolyORB.Tasking.Rw_Locks;
```

– nécessité de désallouer les structures quand elles ne sont plus utilisées ;
– nécessité de ne pas désallouer une structure tant qu'un composant y fait référence.

Ces problèmes sont particulièrement délicats lorsqu'ils concernent des données qui sont référencées par plusieurs composants différents de l'intergiciel. Il est alors impossible pratiquement d'effectuer manuellement le suivi de l'utilisation des structures dynamiques et leur désallocation au moment opportun ; il est nécessaire de déléguer ces fonctions à un sous-système automatique : un *ramasse-miettes* (*garbage collector*).

Dans PolyORB, nous avons réalisé un ramasse-miettes utilisant le comptage de références [121].

Notre mise en œuvre est inspirée de celle proposée par la projection normalisée d'OMG IDL vers Ada 95 [73]. Elle offre un type Reference que les composants de l'intergiciel utilisent en lieu et place des pointeurs. Ces références d'objets peuvent désigner n'importe quel objet dérivé de la classe Entity (extrait de code 6).

La projection normalisée prescrit l'utilisation de ce mécanisme pour les références d'objets ; nous l'avons étendue à l'ensemble des structures internes de l'intergiciel (aussi bien dans les personnalités que dans la couche neutre) pour lesquelles il était nécessaire d'effectuer un comptage de référence. Par exemple, nous utilisons ces pointeurs à comptage de références pour lier les références d'objets distants aux objets de liaison qui les continuent, de sorte qu'un objet de liaison soit détruit lorsque plus aucune référence d'objet détenue par le nœud ne l'utilise.

Les entités sont des types Ada 95 *limités*, c'est-à-dire des types dont le langage interdit la duplication. Cette restriction est destinée à prévenir la création de copies des entités qui ne soient pas prises en compte par le ramasse-miettes.

Les références, quant à elles, sont des types *contrôlés*, pour lesquels le langage permet la surcharge de l'opération de copie. Ainsi, lorsqu'une référence

Extrait de code 6 Interface des pointeurs à comptage de références

```
package PolyORB.Smart_Pointers is

   type Entity is abstract tagged limited private;
   procedure Finalize (X : in out Entity);
   type Entity_Ptr is access all Entity'Class;
   --  Entity is the base type of all objects that can be
   --  referenced. It contains a Counter, which is the number of
   --  references to this object, and is automatically destroyed when
   --  the counter reaches 0. Before the entity is destroyed,
   --  the Finalize operation is called. NOTE however that
   --  Entity is *not* a controlled type: Finalize is *only* called
   --  when an Entity is destroyed as a result of its reference
   --  counter dropping to 0.

   type Ref is new Ada.Finalization.Controlled with private;
   --  The base type of all references. This type is often derived
   --  but never extended. It contains one field, which designate
   --  the referenced object.

   procedure Set
     (The_Ref : in out Ref;
      The_Entity : Entity_Ptr);

   function Is_Nil (The_Ref : Ref) return Boolean;

   function Entity_Of (The_Ref : Ref) return Entity_Ptr;

private

   type Entity is abstract tagged limited record
      Counter : Integer := 0;
   end record;

   type Ref is new Ada.Finalization.Controlled with
      record
         A_Ref : Entity_Ptr := null;
      end record;

end PolyORB.Smart_Pointers;
```

est copiée, le compteur de l'objet désigné est automatiquement incrémenté. De même, lorsqu'une référence est détruite, le compteur est décrémenté.

Cette utilisation de types contrôlés pour réaliser des pointeurs « intelligents » à comptage de référence est un patron de conception fréquemment rencontré [39, 43].

Un ramasse-miettes efficace est ainsi obtenu. Il présente cependant deux limitations principales :

- comme dans tout système de comptage de références, les structures qui comportent des cycles ne peuvent être désallouées. Cette limitation est contournée en concevant nos structures de données de manière à ne jamais créer de cycles de références.
- l'utilisation d'objets contrôlés est coûteuse, car elle nécessite que le compilateur génère une quantité importante de code supplémentaire pour suivre le cycle de vie des objets contrôlés et appeler leur destructeur en tant que de besoin. De plus, ce mécanisme de génération de code interdit l'utilisation de types contrôlés dans les applications à très haute intégrité.

La première limitation est acceptable dans la mesure où nous nous assurons que les structures de données propres de l'intergiciel ne comportent pas de cycles.

En revanche, la seconde constitue une difficulté non résolue pour les applications de haute intégrité. Dans ces applications, un profil très restreint du langage est utilisé, afin de générer du code exempt de tout appel à l'exécutif Ada. Cette restriction a pour objectif de faciliter la certification du code généré dans le cadres de règles de qualité (par exemple les règles DO178 pour les systèmes embarqués aéronautiques [90]).

En particulier, ce profil restreint interdit l'utilisation d'objets contrôlés, afin de ne pas faire appel aux fonctions de gestion du cycle de vie de ces objets fournies par l'exécutif.

L'utilisation de notre intergiciel dans ces applications nécessitera donc la modification du procédé de comptage de référence ; il sera nécessaire d'établir de nouvelles règles de programmation pour garantir l'exactitude du comptage de références en l'absence des appels générés automatiquement par le compilateur.

5.3.2 Dictionnaires

Les dictionnaires sont des structures de données associant une valeur à des clés d'un type arbitraire. Les tableaux peuvent être vus comme des dictionnaires particuliers dans lesquels l'ensemble des clés est un intervalle d'entiers. Les dictionnaires généralisant la notion de tableau à des ensembles de clés quelconques, ils sont également désignés sous le terme de *tableaux associatifs*.

Fonctions de hachage dynamiques parfaites

Une mise en œuvre triviale d'un dictionnaire peut être réalisée au moyen d'une liste chaînée de couples (*clé*, *valeur*). Cependant, une telle mise solution présente deux inconvénients :

inefficacité : chaque recherche nécessite un parcours linéaire de la liste ;

temps d'exécution variable : pour un nombre n d'entrées, la complexité de la recherche est en $O(n)$. Pour des applications temps réel, on souhaite au contraire garantir autant que possible le déterminisme de l'intergiciel. Il

est donc souhaitable que les recherches puissent être effectuées en temps constant.

Dans PolyORB, nous avons donc adopté une mise en œuvre utilisant des tables de hachage basées sur des fonctions de hachage dynamiques parfaites. Les fonctions de hachage parfaites garantissent que deux chaînes différentes dans l'ensemble des clés donnent des valeurs de hachage différentes. Elles permettent donc l'extraction d'une entrée de table de hachage en temps constant (correspondant au calcul de la valeur de hachage et à la vérification de l'entrée associée). En contrepartie, l'insertion d'une entrée dans le dictionnaire entraîne un coût potentiellement important, car elle peut nécessiter la reconstruction complète des paramètres de la fonction de hachage. Ce compromis est intéressant car les insertions sont beaucoup moins fréquentes que les extractions.

L'algorithme que nous utilisons garantit une recherche en $O(1)$ dans le pire cas, et une insertion ou suppression en $O(1)$ dans la plupart des cas [21]. Il est basé sur un algorithme qui donne des fonctions de hachage parfaites dans le cas particulier des dictionnaires statiques (sur lesquels l'ensemble des clés est fixé une fois pour toutes, sans possibilité d'en ajouter ou d'en supprimer dynamiquement) [29].

Fonctions de hachage statiques minimales parfaites

Les fonctions de hachage parfaites, permettant l'identification d'un mot parmi un ensemble donné en temps constant, sont utiles dans de nombreux aspects du fonctionnement de l'intergiciel.

Définition 22 (fonction de hachage parfaite) *Une fonction de hachage est dite* parfaite *lorsqu'elle est exempte de collisions, c'est-à-dire lorsqu'elle est injective.*

Dans certains cas, les ensembles de clés sont fixés et ne varient pas au cours de l'exécution. Par exemple, dans le protocole HTTP, l'ensemble des en-têtes et des méthodes possibles est déterminé une fois pour toutes par la norme [27]. Il est alors intéressant de calculer statiquement une fonction de hachage parfaite pour reconnaître ces clés : ces fonctions sont utiles pour la réalisation de personnalités protocolaires. Nous avons utilisé cette approche dans notre mise en œuvre de HTTP, décrite dans la section 8.3.

Elles peuvent également être utilisées dans les personnalités applicatives, en particulier pour rechercher une méthode à partir de son nom lors de l'aiguillage d'un appel reçu. Dans TAO, cette option est offerte afin d'offrir un démultiplexage efficace et déterministe [36].

Lorsque les clés sont connues à l'avance, les fonctions de hachage calculées peuvent même être minimales.

Définition 23 *fonction de hachage parfaite minimale Une fonction de hachage parfaite sur un espace de clés $\{k_n\}_{n\in[\![0..N-1]\!]}$ est dite* minimale *lorsqu'elle prend ses valeurs sur $[\![0..N-1]\!]$. Les tables de hachage basées sur des fonctions de hachage minimales parfaites ne comportent pas de « trous ».*

Un générateur de fonctions de hachage minimales parfaites pour tables statiques a été développé pour PolyORB par Laurent Pautet, en utilisant un algorithme de Zbigniew Czech [18]. Cet algorithme est plus intéressant que celui de `gperf` [91], utilisé dans TAO, car il converge dans tous les cas.

Nous utilisons actuellement les fonctions de hachage minimales parfaites dans nos mises en œuvre des protocoles HTTP et SOAP.

5.4 Patrons de conception

Dans le chapitre 4, nous avons introduit deux patrons de conception inédits : *Component* et *Annotation*.

Dans cette section, nous présentons des mises en œuvre génériques et réutilisables de ces patrons de conception, que nous utilisons dans PolyORB pour limiter le couplage entre modules, et ainsi permettre la construction de l'intergiciel par assemblage flexible de composants.

5.4.1 Composants

Le patron *Component*, que nous avons présenté dans la section 4.7.3, est réalisé par la définition de deux classes abstraites (extrait de code 7) :

- Component est la classe de base de tous les composants susceptibles de prendre part au patron ;
- Message est la classe de base de tous les messages qui définissent les interfaces des composants.

Les messages sont des conteneurs de données. La classe Message est donc dérivée par ajout d'attributs, mais sans ajout de méthodes. Les traitements associés à un message sont effectués par les composants qui les reçoivent, dans la primitive Handle_Message.

Lorsqu'un composant doit être connecté à un autre, il déclare simplement un attribut de type « pointeur sur composant ». La fonction Emit est utilisée pour émettre un message de façon synchrone sur un port. Elle ne rend le contrôle à l'appelant que lorsque le composant destinataire a renvoyé un message en retour.

La mise en œuvre générique des composants n'offre pas de mécanisme de messages asynchrones. Cependant, un composant associé à une tâche peut recevoir un message, le stocker dans une file d'attente et rendre aussitôt la main à l'appelant en renvoyant un message vide, sans attendre d'avoir traité le message initial. Une communication asynchrone peut ainsi être réalisée.

Cette mise en œuvre soulève une difficulté : l'un des objectifs du patron *Component* est de ne pas contraindre des objets réalisant la même interface à être liés par une relation de dérivation de type. Or, tous les objets intervenant de ce patron doivent être descendants du type Component. Par conséquent, si un objet doit à la fois être un Component et dériver d'un autre type de données, par exemple Entity (un objet sur lequel est effectué un comptage de références au moyen du module décrit dans la section 5.3.1), il est nécessaire de définir un procédé particulier pour représenter en Ada cette situation d'héritage multiple, comme illustré par la figure 5.1(a).

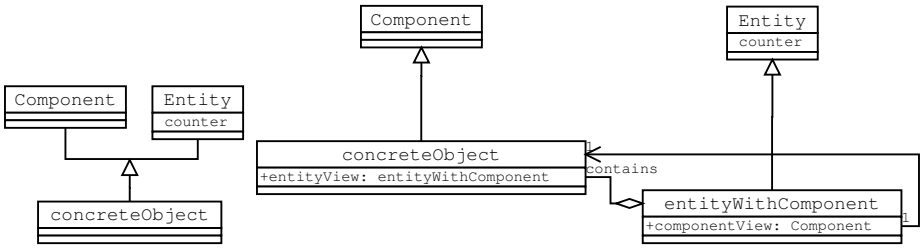
Nous levons cette difficulté en utilisant une construction de *vues multiples*. Cette construction idiomatique en Ada 95 permet de lier un objet à l'un de ses attributs de manière biunivoque, de manière à pouvoir passer de l'un à l'autre. Tout se passe, pour l'utilisateur, comme s'il pouvait « convertir » l'objet en son attribut, ou l'attribut en l'objet complet (figure 5.1(b)).

Une relation d'héritage multiple est ainsi obtenue (l'objet peut être vu comme héritant à la fois de son type parent nominal et du type de son at-

Extrait de code 7 Interface des composants

```
package PolyORB.Components is

   type Message is abstract tagged null record;
   --  The root type for all messages that can be exchanged
   --  between components.

   type Null_Message is new Message with private;

   type Component is
     abstract new Ada.Finalization.Limited_Controlled
     with private;
   type Component_Access is access all Component'Class;

   function Handle_Message
     (C : access Component;
      M : Message'Class)
     return Message'Class
      is abstract;
   --  Called internally when component C is to receive message M.
   --  Return a reply (possibly Null_Message if no specific contents
   --  are to be returned to the sender) if M has been handled.
   --  Otherwise, exception Unhandled_Message is raised.
   --  Each component type overloads this primitive, and
   --  thus defines its behaviour in terms of replies to
   --  a set of external stimuli (messages).
   --  This subprogram must not be called directly; Emit must be
   --  used instead.

   procedure Connect
     (Port   : out Component_Access;
      Target : Component_Access);
   --  Connect Port to Target: when a message M is emitted on Port,
   --  it is received by Target.

   function Emit
     (Port : Component_Access;
      Msg  : Message'Class)
     return Message'Class;
   --  Emit message Msg on Port. The reply is returned.

end PolyORB.Components;
```

tribut [15].



(a) Héritage multiple

(b) Réalisation par vues multiples

FIG. 5.1 – Héritage multiple en Ada 95

5.4.2 Annotations

Comme le patron *Component*, le patron *Annotation*, décrit dans la section 4.7.3 permet d'attribuer une propriété commune à des objets dont les types ne sont pas liés par une propriété d'héritage.

Pour rendre un objet capable de contenir des informations manipulées par ses clients, le patron *Annotation* définit deux types de données :

- une classe abstraite Note, similaire à la classe Message du patron *Component* : il s'agit d'un conteneur de données abstrait ;
- un type concret Notepad permettant de stocker un ensemble de Notes.

Un objet qui souhaite offrir à ses clients la possibilité de lui associer des données doit donc exposer un attribut de type Notepad. Les clients peuvent manipuler les annotations au moyen de l'interface illustrée par l'extrait de code 8.

Extrait de code 8 Interface des annotations

```
package PolyORB.Annotations is

   type Note is abstract tagged private;
   --  A note that can be attached to an object.

   type Notepad is private;
   --  A space for clients of an object to attach Notes into.
   --  Notepad_Access can be used by private types to selectively
   --  expose one Notepad component to their clients.

   procedure Set_Note (NP : in out Notepad; N : Note'Class);
   --  Add note N to notepad NP. If a note with the same tag as N
   --  exists, it is replaced by N.

   procedure Get_Note (NP : Notepad; N : out Note'Class);
   --  Retrieve a note of N's type from NP.

end PolyORB.Annotations;
```

Chaque Notepad constitue un dictionnaire de Notes dont les clés sont les types des notes. Dans PolyORB, aucun objet annotable ne reçoit plus d'une ou deux notes. Nous avons donc choisi d'utiliser une simple liste chaînée de notes plutôt qu'un dictionnaire basé sur une table de hachage (cf. section 5.3.2). Une

telle solution n'est pas moins efficace pour un très petit nombre d'entrées, et elle est beaucoup moins coûteuse.

De plus, le nombre de types dérivés de Note est connu au moment de la compilation. La longueur de ces listes est donc statiquement bornée, et leur utilisation n'introduit pas de non-déterminisme dans l'application.

5.5 Synthèse

Nous avons présenté la bibliothèque de services de base de PolyORB. Celle-ci offre des services opérationnels à l'ensemble de l'intergiciel ; elle peut être utilisée par l'ensemble des couches de l'intergiciel.

Les services offerts constituent le cadre dans lequel peuvent s'intégrer et coopérer les divers modules constitutifs de l'intergiciel. Les services d'initialisation rendent possible l'initialisation ordonnée de ces modules.

La bibliothèque de parallélisme et de synchronisation leur permet de bénéficier des capacités de traitement parallèle de l'environnement, lorsque celles-ci sont disponibles, et offre les outils nécessaires à leur synchronisation. Elle est conçue de façon configurable, pour permettre le déploiement de PolyORB dans des environnements sans parallélisme, ou avec un profil limité.

Des structures de données de haut niveau (pointeurs à comptage de références et dictionnaires basés sur des fonctions de hachage parfaites) sont réalisées.

Enfin, deux patrons de conceptions inédits sont proposés : *Component* et *Annotation*. Ces patrons découplent respectivement les propriétés comportementales et les données attachées à une classe de sa position dans la hiérarchie de types.

L'ensemble de ces services forme la fondation d'une architecture modulaire et flexible. Celle-ci permet la construction de l'intergiciel par assemblage de composants de façon adaptable aux besoins de l'application et de l'environnement.

CHAPITRE

6 Réalisation d'une couche neutre de répartition

Sommaire

6.1 Noyau générique 142
6.1.1 Modèle de répartition 143
6.1.2 Cycle de vie d'une requête 145
6.1.3 Références d'objets 147
6.1.4 Ordonnanceur-orienteur 150
6.1.5 Lieur 153
6.2 Composants applicatifs de la couche neutre 154
6.2.1 Interface abstraite des adaptateurs d'objets 155
6.2.2 Adaptateurs d'objets concrets 156
6.3 Composants protocolaires de la couche neutre . . 157
6.3.1 Tampons de communication 158
6.3.2 Mécanismes de transport 159
6.3.3 Objets de liaison 162
6.4 Synthèse 164

LA COUCHE NEUTRE de PolyORB correspond à l'ensemble des composants indépendants du choix d'une personnalité de répartition. Dans ce chapitre, nous détaillons la réalisation de ces composants.

La couche neutre a pour rôle d'isoler les personnalités applicatives des personnalités protocolaires. Elle manipule des données représentant les interactions entre les objets de l'application.

Nous présentons en premier lieu la représentation neutre des interactions utilisée par la couche neutre : nous définissions le modèle de type et le modèle d'objets qu'elle sous-tend.

Nous précisons ensuite comment les composants de la couche neutre remplissent les fonctions récurrentes des intergiciels, que nous avons identifiées dans le chapitre 2 :

le noyau générique, qui définit les représentations neutres des interactions entre objets applicatifs, réalise la fonction d'adressage, l'ordonnancement des traitements au sein de l'intergiciel, et la fonction de liaison ;

les composants applicatifs neutres remplissent les fonctions d'adaptateur d'objets : gestion d'un espace d'identification local et fonction d'activation ;

les composants protocolaires neutres responsables de la fonction de transport et offrent l'infrastructure nécessaire à la construction d'objets de liaison.

6.1 Noyau générique

Le noyau générique, dont nous avons présenté la conception dans la section 4.4.3, est la partie centrale de la couche neutre.

Il organise les interactions entre les différentes parties de l'intergiciel, et permet le dialogue entre la couche applicative et la couche protocolaire à travers des interfaces génériques.

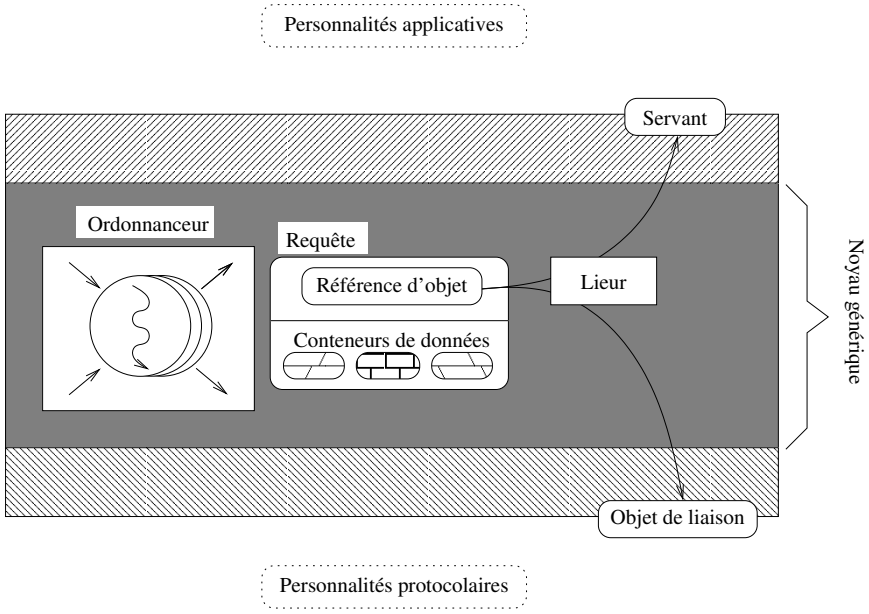


FIG. 6.1 – Composants génériques de la couche neutre

La figure 6.1 illustre les composants du noyau générique. Celui-ci définit des représentations des interactions entre objets indépendantes des personnalités :

- des requêtes, représentant le plus petit élément d'interaction entre des objets de l'application ;
- des données associées à ces requêtes ;
- des références permettant de désigner les objets de l'application.

Le noyau générique définit également les traitements appliqués à ces données. Les modules du noyau responsables de ces traitements sont :

- l'ordonnanceur-orienteur, qui détermine la séquence des traitements appliqués aux requêtes et achemine ces requêtes entre les différents composants de l'intergiciel ;

– le lieur, responsable de l'établissement des liaisons entre les références et les objets qu'elles désignent.

6.1.1 Modèle de répartition

Dans cette section, nous présentons le modèle de répartition générique choisi dans PolyORB. À partir des interactions entre objets prévues par ce modèle, nous introduisons les représentations génériques manipulées par la couche neutre.

Modèle d'objets répartis

Le modèle d'interactions entre objets utilisé par la couche neutre de PolyORB est basé sur l'architecture d'objets répartis définie par l'OMG : l'*Object Management Architecture* [70]. C'est le modèle sur lequel s'appuient les spécifications CORBA. Il définit les entités de base d'une application répartie :

Définition 24 (objet) *Une application répartie est considérée comme une collection d'objets : des entités identifiables, encapsulées (dont la structure interne est dissimulée à leurs utilisateurs), fournissant un ou plusieurs services à des clients.*

Définition 25 (requête) *Une requête est un événement par lequel un client demande à un objet de lui fournir un service à un instant donné. Une requête consiste en :*
– *l'identification d'une opération ;*
– *l'identification d'un objet destinataire ;*
– *des paramètres éventuels ;*
– *un contexte.*

Les paramètres d'une requête sont identifiés par leur position. Ils peuvent être transmis en entrée ou en sortie (ou les deux). À l'issue normale de l'exécution d'un service demandé par un client, les paramètres en sortie et un éventuel résultat sont retournés à celui-ci.

Une défaillance peut également être signalée au moyen de la levée d'une exception, soit par le système de répartition, soit par l'objet applicatif. L'occurrence d'une exception est associée à une identité (un type d'exception) et éventuellement à une information additionnelle (une valuation).

Les requêtes constituent la représentation centrale manipulée par la couche neutre. Sur chaque nœud manipulant une requête donnée (client, serveur ou passerelle), un objet est créé pour la représenter pendant toute la durée de son traitement. Certains attributs de cet objet sont initialisés dès sa création, d'autres sont renseignés au fur et à mesure de son traitement.

Le modèle de répartition OMA nous paraît suffisamment général pour représenter une très large classe d'applications. Il a été conçu, avec les spécifications CORBA, pour intégrer des applications existantes dans des organisations réparties. Le fait de retenir un modèle basé sur les objets répartis permet de bénéficier de l'expressivité de ce modèle : comme nous l'avons indiqué dans la section 2.5.2, les abstractions des objets répartis permettent également de modéliser les interactions dans les systèmes répartis utilisant l'appel de messages ou les appels de sous-programmes à distance. Par ailleurs, les fonctions présentes dans un intergiciel pour objets répartis constituent un sur-ensemble de celles nécessaires

à la réalisation des deux autres modèles. Un intergiciel offrant ces fonctions est donc également en mesure de supporter ces modèles.

Représentation des interactions

Dans PolyORB, l'interaction élémentaire entre deux objets applicatifs est donc représentée par une structure de type « requête », qui contient l'ensemble des informations relatives à un appel de méthode sur un objet, au sens de la définition 25. Cette structure contient :

- une référence neutre désignant l'objet destinataire ;
- le nom de la méthode à exécuter ;
- les paramètres à passer ;
- le résultat de l'exécution (une valeur retournée, ou bien une occurrence d'exception) ;
- des informations de contrôle nécessaires au suivi du traitement de la requête.

Une copie de cette structure existe sur chaque nœud qui intervient dans le traitement d'un appel de méthode : le client sur lequel elle trouve son origine, le serveur qui la traite, ainsi que tout nœud jouant le rôle de passerelle.

Modèle de types de données

Le modèle de types pour les données échangées par les objets de l'application est basé sur celui de CORBA. Ce modèle a l'avantage d'avoir été conçu pour la répartition. Il paraît suffisamment générique pour être considéré comme un modèle unificateur pour les applications réparties.

En effet, ce modèle est suffisamment expressif pour classifier les données transmissibles dans un système réparti (c'est-à-dire susceptibles de conserver leur sens globalement) et fixer une représentation adaptée pour chaque type de données.

D'autres modèles de types permettent d'exprimer des classifications plus fines des données. Ainsi, le typage fort du langage Ada permet de contraindre les valeurs d'une donnée de façon beaucoup plus restrictive que sur le simple critère de la compatibilité des représentations.

Cependant, la richesse supplémentaire apportée dans la description des types par un tel modèle ne modifie pas la représentation des données : elle modifie le sens attaché localement aux valeurs, et l'ensemble des opérations qu'il est possible d'effectuer sur une valeur donnée.

Le rôle de l'intergiciel se limite à transmettre les demandes d'interactions entre objets d'un nœud à l'autre. Les enrichissements sémantiques apportés par des descriptions fines des types, telles que des contraintes sur les intervalles de validité d'une valeur numérique, n'interviennent donc pas dans les fonctions qu'il remplit. En effet, il est possible de choisir un mode de transport des données d'un nœud à un autre indépendamment de ces contraintes.

Il n'est donc pas nécessaire de choisir pour modèle de types unificateur entre les systèmes répartis un modèle de type permettant d'exprimer les classifications les plus riches ; un modèle permettant de fixer la représentation des données transmises et d'en conserver le sens est suffisant. C'est pourquoi nous avons retenu le modèle de types IDL.

Représentation des données

Pour les besoins des interfaces d'invocation dynamique et de squelettes dynamiques, les normes CORBA proposent un mécanisme permettant d'encapsuler une donnée de l'utilisateur de n'importe quel type dans un conteneur universel. À chaque tel conteneur est associé un « code de type » (*TypeCode*). Les codes de types sont des descriptions de la structure d'une donnée ; ils permettent d'exprimer tout type qui peut être déclaré en OMG IDL.

Ensemble, codes de types et conteneurs universels permettent de spécifier le contenu d'un message échangé entre deux objets applicatifs de façon suffisamment complète pour que des intergiciels les transmettent de l'objet émetteur à l'objet destinataire.

Nous avons donc repris ces structures comme représentation neutre des données associées aux requêtes dans la couche neutre : valeurs des paramètres, valeurs de retour et occurrences d'exceptions. Elles remplissent le rôle des conteneurs typés que nous avons évoqué dans la section 4.5.4.

Nous définissions des fonctions de conversions entre les types primitifs du modèle de type et les conteneurs universels correspondants. Nous avons également réalisé les fonctions nécessaires à la construction et à la manipulation des codes de types et des conteneurs correspondant aux types de données complexes.

6.1.2 Cycle de vie d'une requête

Le cycle de vie d'une requête se décompose en trois phases indépendantes :

1. création ;
2. liaison ;
3. transmission.

Pour illustrer chacune de ces phases, nous allons considérer les traitements correspondants sur un client, puis sur un serveur, et enfin sur une passerelle réalisant l'adaptation entre deux objets hébergés par des intergiciels tiers, et ne disposant pas d'un protocole commun d'échange de requêtes (par exemple, l'un utilisant SOAP et l'autre GIOP).

Cas d'un client

La figure 6.2 illustre le cas d'un client. Une requête est créée par une personnalité applicative à la suite de l'exécution, par l'objet applicatif client, d'une construction dénotant un appel de méthode à distance (1).

Une fois la requête mise sous la forme d'un objet neutre de type « requête », la personnalité applicative demande à la couche neutre de l'exécuter. PolyORB utilise alors le lieur (que nous décrirons plus en détail dans la section 6.1.5) pour obtenir un objet de liaison subrogé de l'objet applicatif désigné comme destinataire de la requête (2).

Ayant obtenu le subrogé, la couche neutre lui transmet la requête et se met en attente, à moins que l'utilisateur n'ait demandé à reprendre le contrôle immédiatement (3). L'objet de liaison transforme la requête en message qu'il transmet à l'intergiciel distant.

Lorsqu'une réponse est reçue, l'intergiciel complète les attributs nécessaires de la requête (valeur de retour, paramètres transmis en sortie, occurrence d'exception) avant de signaler la fin du traitement à la tâche demandeuse.

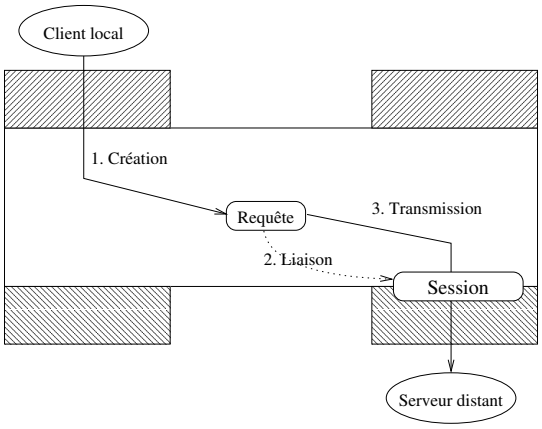


FIG. 6.2 – Cycle de vie d'une requête (rôle client)

La personnalité applicative peut alors récupérer les informations retournées pour les transmettre à l'objet applicatif, et rendre le contrôle à ce dernier.

L'utilisateur peut cependant demander à ce qu'un appel de méthode lui rende le contrôle immédiatement. Dans ce cas, lorsqu'une réponse est reçue, elle peut être ignorée, ou encore traitée de manière asynchrone.

Cette possibilité permet d'utiliser PolyORB pour réaliser un système de répartition basé sur le passage de messages [45].

Cas d'un serveur

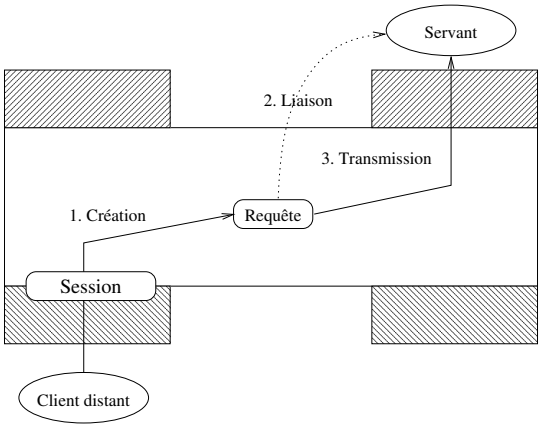


FIG. 6.3 – Cycle de vie d'une requête (rôle serveur)

Le cas d'un serveur est illustré par la figure 6.3. Une requête est construite par une personnalité protocolaire lorsqu'elle reçoit un message demandant un appel de méthode (1).

Le composant responsable du traitement de la requête est, comme sur un client, déterminé par le lieur (2). La référence étant cette fois locale, celui-ci fait

appel à l'adaptateur d'objets pour qu'il active le servant associé à l'identificateur d'objet. La requête est alors transmise à ce servant pour exécution (3).

Les paramètres ne sont pas extraits immédiatement : c'est le servant qui, par l'entremise d'une opération appropriée de sa personnalité applicative, indique la signature de la méthode, afin de permettre cette extraction.

En effet, pour assurer le découplage entre les personnalités applicative et protocolaire, il est nécessaire de ne pas répartir la connaissance des signatures de méthodes entre les couches. Seule la couche applicative a connaissance de ces signatures, et il appartient au servant de fournir les informations nécessaires en temps voulu à la couche neutre.

Une fois la requête traitée, les attributs de la requête sont complétés, et la fin de traitement signalée au composant demandeur. Dans le cas d'un serveur, ce composant est la session sur laquelle la requête a été reçue. Lorsqu'une session reçoit un signal de fin de traitement, elle émet la réponse vers l'intergiciel client.

Cas d'une passerelle

Les trois phases de traitement d'une requête sont indépendantes. Les variantes « client » et « serveur » de chacune de ces phases, évoquées ci-dessus, peuvent donc être composées de façon arbitraire. Ainsi, dans le cas d'un objet local, la création de la requête est faite par la personnalité applicative cliente, la liaison détermine que l'objet destinataire est un servant local et la transmission est faite directement à cet objet, sans intervention d'une personnalité protocolaire.

Dans le cas d'une passerelle, une requête est créée par réception sur une session : la phase de création est identique à celle du cas serveur.

L'identité de l'objet destinataire correspond à une référence de passerelle, c'est-à-dire une référence désignant un objet donné *qui doit être atteint par l'intermédiaire d'un autre nœud que le sien.* Le lieur associe cette référence à une nouvelle session, dans laquelle l'intergiciel local joue le rôle de client : l'objet prenant en charge la requête reçue n'est plus un servant (comme dans le cas d'un serveur ordinaire) mais un nouveau subrogé. La phase de transmission, enfin, est identique à celle du rôle client.

Le subrogé présent sur une passerelle ne dispose pas des informations complètes relatives aux signatures des méthodes de l'objet réel. Il doit donc déterminer de façon externe la signature à utiliser, en recherchant cette information dans une description externe de l'interface offerte par l'objet.

Il appartient aux personnalités applicatives de fournir le mécanisme nécessaire à cette détermination, sous forme d'une mise en œuvre concrète d'un composant *descripteur d'interface*, que nous détaillons dans la section 7.1.4. Un tel composant permet à la couche neutre de prendre connaissance de la signature d'une méthode sous forme d'une liste de typecodes. Cette information permet le déballage des paramètres en vue de construire une représentation neutre complète d'une requête, qui peut alors être retransmise à travers une personnalité protocolaire quelconque.

6.1.3 Références d'objets

Chaque objet d'une application répartie est une entité identifiable (définition 24). Une référence d'objet est une information communicable entre les

nœuds d'une application répartie qui désigne sans ambiguïté l'un des objets qui la constitue.

La couche neutre définit une représentation des références d'objets locaux et distants, indépendamment de la personnalité utilisée par chacun. Elle est responsable de l'attribution de références aux objets locaux : elle réalise la *fonction d'adressage* décrite dans la section 4.5.1.

Structure de données

PolyORB introduit un type de données permettant de manipuler, dans la couche neutre, des références désignant des objets distants utilisant des personnalités quelconques.

Comme nous l'avons indiqué dans la section 4.5.1, une référence est une structure de données regroupant un identificateur et une collection de profils. Chaque profil contient trois informations :

- un identificateur d'objet ;
- l'adresse d'un point d'accès du service de transport ;
- une étiquette identifiant le protocole correspondant.

L'identificateur d'objets est une suite d'octets fournie par la couche applicative. Pour le noyau générique et les personnalités protocolaires, il s'agit d'une information opaque qui n'est jamais interprétée. Elle est retransmise à la couche applicative lorsque la fonction d'activation est appelée.

Les informations de transport et de protocole sont utilisées pour établir un objet de liaison permettant l'envoi de messages à l'objet désigné par la référence. Le mécanisme de cet établissement est décrit plus en détail dans la section 6.1.5.

Construction de références pour les objets locaux

La construction d'une référence désignant un objet local nécessite la coopération des couches applicative et protocolaire. En effet, les profils d'une référence contiennent un identificateur d'objet fourni par la couche applicative, et des informations de transport issues de la couche protocolaire. La figure 6.4 illustre ce processus.

La coopération entre les couches est organisée par le noyau générique, qui garantit leur indépendance. Le noyau obtient d'abord un nom de type et un identificateur d'objet de la couche applicative. L'identificateur (local) est une suite d'octets opaque fournie par un adaptateur d'objets.

L'identificateur est alors transmis à des *usines de profils*, composants des personnalités protocolaires qui encapsulent un identificateur local dans un profil spécifique d'un protocole. Une usine de profils est associée à chaque point d'accès du service de transport enregistré auprès du noyau générique.

La référence d'objet est constituée à partir de la collection de profils ainsi obtenue. La réalisation des usines de profils est décrite plus en détail dans la section 8.1.5.

Construction de références de passerelles

Le cas des références de passerelles est particulier. Une telle référence désigne un objet distant. Cependant, au lieu de contenir des profils dénotant les points d'accès de l'intergiciel qui héberge cet objet, elle contient des profils d'un autre

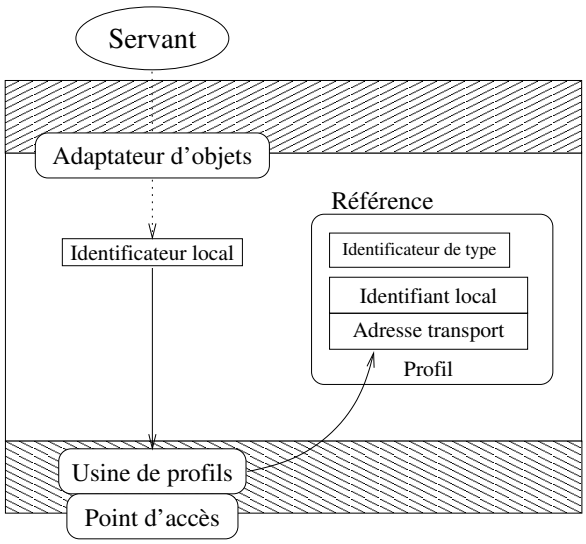


FIG. 6.4 – Création d'une référence d'objet local

intergiciel, amené à jouer le rôle de passerelle lorsque des requêtes doivent être envoyées à l'objet à l'aide de ces références.

La construction des références de passerelles est réalisée par le processus illustré par la figure 6.5.

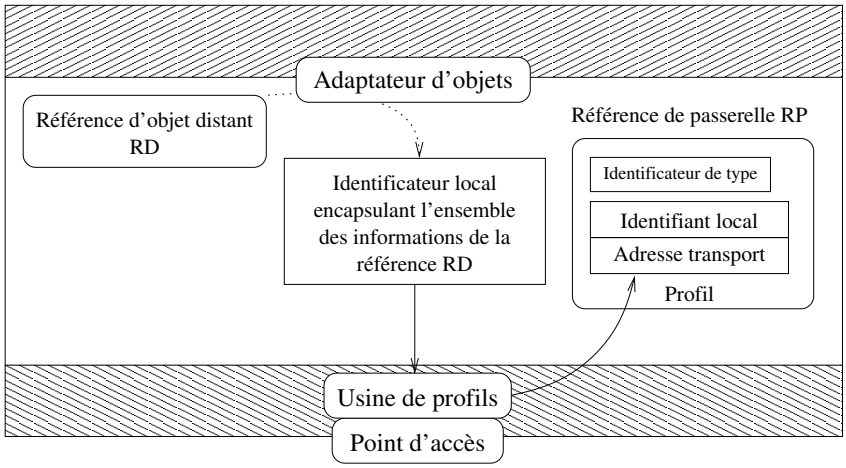


FIG. 6.5 – Création d'une référence de passerelle

Pour construire une référence de passerelle, la référence réelle de l'objet désigné est tout d'abord encapsulée sous forme d'une suite d'octets (au moyen d'une fonction de représentation). À partir de cette référence encapsulée, un adaptateur d'objet construit un identificateur local particulier.

Cet identificateur contient une indication de sa nature (c'est un identificateur de passerelle, et non un identificateur attribué à un servant local), et l'information de la référence encapsulée.

Cet identificateur est alors transmis à une ou plusieurs usines de profils, comme précédemment, pour construire une référence complète.

6.1.4 Ordonnanceur-orienteur

Le cœur de la couche neutre est l'ordonnanceur-orienteur. Cet objet possède une vue globale de l'intergiciel. Situé à l'interface entre les couches applicative et protocolaire, il permet le dialogue entre elles tout en garantissant l'isolement des connaissances. Il remplit cette fonction en acheminant un ensemble de messages internes entre les différentes entités qui composent chaque couche, et en contrôlant l'ordonnancement des activités de l'intergiciel.

Vue globale de l'intergiciel

L'ordonnanceur-orienteur reçoit des messages internes à ses interfaces avec la couche applicative (qui lui confie des requêtes à transmettre à des nœuds distants), et avec la couche protocolaire (qui lui signale les requêtes reçues). Il relaie ces informations en tant que de besoin aux composants appropriés de l'autre couche.

À cet effet, l'ordonnanceur-orienteur tient à jour :

une vue de la couche protocolaire :
- la liste de tous les points d'accès du service de transport ;
- la liste de tous les points de terminaison ;

une vue de la couche applicative :
- un pointeur vers l'adaptateur d'objets générique.

L'ordonnanceur-orienteur ajoute aux points d'accès et de terminaison du service de transport des données complémentaires, en utilisant le patron de conception *Annotation* (section 4.7.3).

Il peut ainsi accéder immédiatement, à partir de l'une de ces entités :
- à l'objet de liaison associé à un point de terminaison, afin de lui signaler les événements survenus sur le point de terminaison ;
- à l'usine d'objets de liaison associée à un point d'accès, afin d'instancier un objet de liaison lorsqu'une connexion est reçue ;
- à l'usine de profils associée à un point d'accès, afin de pouvoir utiliser ce point d'accès pour la création de références d'objets locaux.

Cette carte globale de l'organisation de l'intergiciel lui permet d'orienter les messages au cours des différentes phases du cycle de vie des requêtes.

Boucle principale

Les diverses opérations nécessaires au traitement des requêtes (les activités propres de l'intergiciel, ainsi que les appels au code des objets applicatifs serveurs) doivent être affectés à des tâches. En particulier, pour remplir sa part de la fonction d'exécution (décrite dans la section 4.5.7), l'ordonnanceur doit attribuer des ressources aux requêtes reçues afin qu'elles soient exécutées par les objets locaux.

Or, les activités de l'intergiciel sont liées à des événements dont l'intergiciel lui-même ne contrôle pas la séquence et la chronologie. En effet, elles sont déclenchées par des événements extérieurs (communications en provenance d'autres

nœuds de l'application répartie) ou par des activités des objets applicatifs locaux.

Les traitements de l'intergiciel sont donc représentés comme une suite de travaux élémentaires, et chaque tâche susceptible de participer à leur exécution entre dans une boucle consistant à attendre qu'un tel travail élémentaire se présente, à l'exécuter, et à poursuivre ce déroulement aussi longtemps qu'il est nécessaire.

La succession des états de ces tâches est illustrée par le réseau de Petri de la figure 6.6.

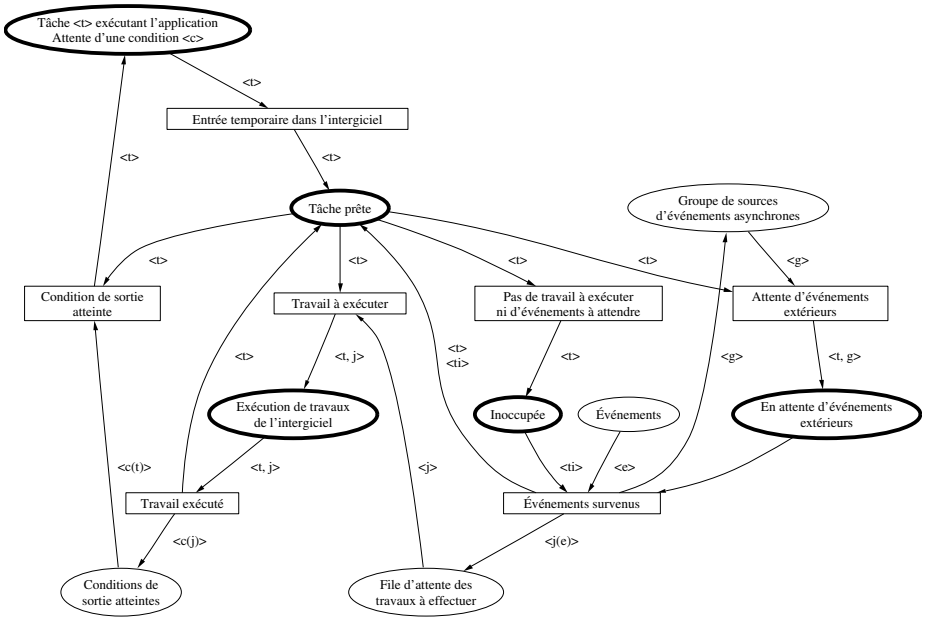


FIG. 6.6 – États des tâches de l'intergiciel

Les places en gras dénotent les états que peuvent occuper les tâches participant aux activités de l'intergiciel :

- prête à prendre en charge des travaux ;
- en train d'exécuter des travaux ;
- en attente d'un événement extérieur ;
- inoccupée.

Les tâches de l'application peuvent également être en train d'exécuter le code de celle-ci.

Lorsqu'une tâche est prête et qu'aucun travail à exécuter n'est en file d'attente, elle se bloque en scrutant un groupe de sources d'événements asynchrones. Lorsqu'un événement asynchrone survient (nouvelle connexion sur un point d'accès du service de transport, donnée reçue par un point de terminaison ou appel provenant d'un objet applicatif), elle met en file d'attente les travaux correspondant au traitement de cet événement.

À un instant donné, une seule tâche peut scruter un groupe de sources d'événements asynchrones. Les autres tâches qui seraient prêtes simultanément passent dans l'état « inoccupé ». Elles en ressortent lorsque de nouveaux travaux

sont placés dans la file d'attente.

Si l'intergiciel est configuré pour créer ses tâches propres à l'initialisation ou au cours de son exécution, celles-ci traversent indéfiniment les états prêt, en cours d'exécution, en attente d'événements et inoccupé.

Par ailleurs, lorsqu'une tâche de l'application demande l'exécution d'un appel de méthode sur un objet distant, cette tâche entre temporairement dans la boucle principale de l'intergiciel. Elle est associée à une condition de sortie correspondant à la fin du traitement de la requête concernée.

Cette organisation permet un fonctionnement uniforme de l'intergiciel. Il permet à un nœud ne possédant qu'une seule tâche d'exécuter alternativement l'intergiciel et le code applicatif : la tâche applicative est alors « prêtée » temporairement à l'intergiciel lorsqu'il est nécessaire de traiter une requête. Sur les nœuds où l'intergiciel est configuré pour créer plusieurs tâches, celles-ci peuvent exécuter la même boucle principale.

Politiques de parallélisme

Pour les nœuds utilisant un profil avec tâches, plusieurs politiques d'affectation des activités aux tâches peuvent être utilisées. L'intergiciel délègue un certain nombre de fonctions, correspondant au traitement d'événements particuliers, à un objet de type « politique de parallélisme » : la gestion de l'affectation des activités constitue une instance du patron *Strategy*, que nous avons décrit dans la section 3.2.2.

Les points de contrôle où s'exerce la délégation des fonctions de l'intergiciel au profit de la politique de parallélisme sont :

- l'établissement et la destruction d'une connexion de transport ;
- la réception d'une requête à exécuter ;
- la présence d'une tâche inoccupée.

Politique « pool de tâches » Dans la politique « pool de tâches », les activités de l'intergiciel sont exécutées par un ensemble de tâches créées à cet effet. Ces tâches peuvent être banalisées, exécutant la boucle principale de l'intergiciel, ou dédiées à une ressource particulière. Des tâches sont créées et détruites dynamiquement afin d'ajuster continuellement le nombre de tâches actives, et ainsi de maintenir systématiquement une réserve de quelques tâches inoccupées susceptibles de prendre en charge un événement extérieur dès qu'il survient.

Une telle politique a été proposée par plusieurs projets existants, en particulier GLADE [83].

Politique « une tâche par session » Une autre politique est offerte dans laquelle les requêtes provenant d'une même session sont exécutées par une tâche dédiée. Les comportements lors de la création et de la destruction d'une session sur un serveur sont spécialisés pour créer une tâche et la rattacher à la session au moyen d'une annotation.

Le comportement de traitement d'une requête est également modifié : les requêtes reçues sont transmises à la tâche créée à cet effet à travers une file d'attente tenant lieu de « boîte aux lettres » pour la communication entre les tâches génériques de l'intergiciel et la tâche de session.

Les requêtes reçues successivement par une même session sont donc sérialisées ; les différentes sessions actives sont traitées en parallèle. Cette

politique a été proposée en particulier dans TAO [101].

Politique « une tâche par requête » Enfin, une politique est offerte qui crée une tâche dédiée pour le traitement de chaque requête reçue. Cette politique a également été proposée dans TAO [100].

Dans ce cas, seul le traitement d'une requête est spécialisé par la politique. Dans cette politique, le parallélisme entre les requêtes n'est limité que par les objets applicatifs, et une requête est toujours certaine de disposer d'une tâche pour être traitée. En revanche, l'intergiciel doit supporter le coût de nombreuses créations et destructions de tâches.

6.1.5 Lieur

Le lieur est le module de PolyORB qui réalise la fonction de liaison, décrite dans la section 4.5.3. Il a donc pour rôle, à partir d'une référence, de déterminer quelle entité prend en charge les requêtes destinées à l'objet désigné. Il peut s'agir soit d'un servant local, soit d'un objet de liaison subrogé d'un servant distant.

Unification entre établissement des liaisons et activation

La liaison d'une référence désignant un objet distant a pour effet de créer un objet de liaison subrogé de l'objet désigné. L'objet de liaison est un composant qui peut prendre en charge une requête, au même titre que l'objet réel : du point de vue de la couche neutre, il n'y a pas de différence entre

- confier une requête à un servant (pour qu'il l'exécute et retourne un résultat) ;
- confier une requête à un objet de liaison (pour qu'il la transmette à un intergiciel tiers, au moyen d'un protocole adéquat, et retourne le résultat communiqué par ce dernier).

L'activation, quant à elle, a pour objectif de déterminer un servant à partir d'une référence désignant un objet local.

Ces deux opérations prennent en entrée une référence, et retournent le composant du nœud local auquel les requêtes ayant pour destinataire cette référence doivent être confiées. En d'autres termes, dans la terminologie proposée par les intergiciels basés sur les liaisons, elles visent toutes deux à déterminer la *continuation* (définition 12, section 3.1.1).

Nous unifions donc activation et liaison au sein d'un même module, le *lieur*. Étant donnée une référence d'objet, le lieur réalise :

pour les objets locaux : une liaison locale, en utilisant la fonction d'activation pour déterminer le servant correspondant à l'identificateur local d'objet ;

pour les objets distants : une liaison à distance, en choisissant un profil de la référence, et en faisant appel à l'usine de liaisons associée.

Le lieur intervient dans la phase de liaison du cycle de vie d'une requête, c'est-à-dire lorsque l'intergiciel doit déterminer la structure concrète extérieure (objet applicatif local ou session établie avec un intergiciel distant) vers laquelle une requête doit être acheminée.

Son rôle est analogue à la fonction de routage dans le protocole IP [86]. Celle-ci a en effet pour tâche de déterminer si un paquet est destiné au nœud local

(auquel cas il est transmis aux couches supérieures), et dans le cas contraire de déterminer la cible du prochain saut en fonction de l'adresse de destination. Le même algorithme est appliqué sur le nœud de départ du paquet, sur tous les nœuds intermédiaires et sur le nœud de destination.

De même, le lieur de PolyORB « route » une requête, éventuellement à travers des passerelles successives, du client qui l'a créée vers le servant qui doit la traiter.

L'utilisation du lieur pour réaliser la liaison des requêtes permet d'isoler cette phase du cycle de vie par rapport à la phase de création et à la phase de transmission. En mettant en œuvre la seule phase de liaison, de façon unifiée, le lieur autorise des combinaisons arbitraires entre les variantes client et serveur des phases de création et de transmission. Il autorise ainsi l'optimisation des appels locaux (qui n'utilisent pas les personnalités protocolaires), et permet la mise en place de passerelles dynamiques transparentes.

Liaison des objets locaux et des passerelles

Lorsque le lieur est appelé sur une référence dont les profils correspondent à des points d'accès de l'intergiciel local, l'identificateur d'objet est extrait de l'un des profils.

Cet identificateur a été attribué par l'adaptateur d'objets local. Celui-ci est donc interrogé pour déterminer le servant associé : la liaison des objets locaux est déléguée à la fonction d'activation.

L'identificateur correspond généralement à un servant local réel, auquel cas c'est ce servant qui est retourné par le lieur à l'ordonnanceur.

Nous avons indiqué dans la section 6.1.3 qu'un identificateur correspondant à une référence de passerelle encapsule une référence d'objet distant complète. Dans un tel cas, l'adaptateur d'objets indique au lieur que celui-ci doit s'exécuter récursivement pour déterminer l'objet subrogé correspondant. La référence distante est reconstituée à partir de l'identificateur, et une liaison distante est réalisée selon le procédé ci-dessous.

Liaison des objets distants

Lorsqu'une référence désigne un objet distant, le lieur choisit un profil de cette référence pour établir une liaison. Ce choix est effectué en fonction de préférences affectées par l'utilisateur à chaque type de profil au moyen d'un fichier de configuration. Le profil ayant la plus grande préférence est sélectionné.

Son usine de liaison établit une connexion de transport, et construit un objet de liaison à partir de composants génériques et personnalisés. Le procédé de création et d'assemblage de ces composants sera détaillé dans la section 6.3, relative aux composants protocolaires de la couche neutre.

L'objet de liaison ainsi obtenu constitue un subrogé de l'objet désigné par la référence : c'est cet objet que le lieur retourne au composant demandeur.

6.2 Composants applicatifs de la couche neutre

Les composants applicatifs de la couche neutre, dont le rôle a été évoqué dans la section 4.4.3, sont les adaptateurs d'objets génériques. Leur position dans l'architecture est illustrée par la figure 6.7.

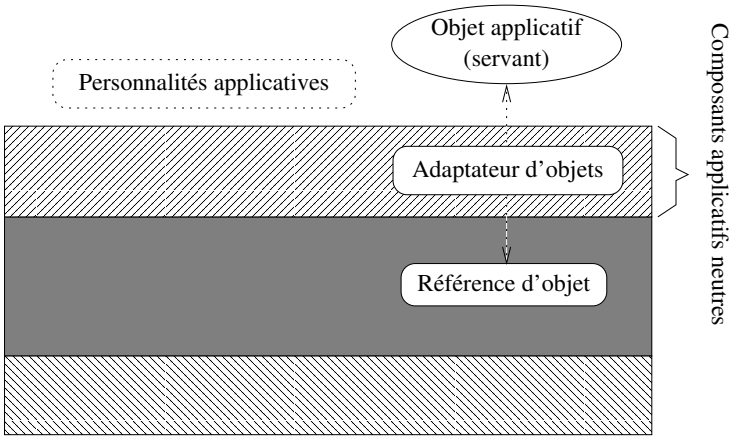


FIG. 6.7 – Composants applicatifs de la couche neutre

Le terme « adaptateur d'objets » est emprunté à l'architecture CORBA. Cependant, il correspond à une entité qui apparaît dans les autres personnalités applicatives. C'est pourquoi nous l'avons retenu dans notre architecture générique.

L'adaptateur d'objets joue deux rôles dans l'intergiciel :
- il contribue à la fonction d'adressage, décrite dans la section 4.5.1, en contrôlant l'espace des identificateurs local à un nœud ;
- il réalise la fonction d'activation, décrite dans la section 4.5.6, en déterminant l'objet concret local (servant) qui doit exécuter une requête, à partir de la référence de cet objet.

Nous avons défini une interface abstraite des adaptateurs d'objets, qui peut être concrétisée sous forme d'un adaptateur simplifié offrant à faible coût des fonctions de base, ou d'un adaptateur plus riche inspiré du POA de CORBA. Ces mises en œuvre concrètes sont génériques : elles ne dépendent pas du choix d'une personnalité particulière. Ainsi, le POA inspiré de CORBA peut être réutilisé dans d'autres personnalités applicatives pour ses fonctionnalités de découpage hiérarchique de l'espace d'identification local.

6.2.1 Interface abstraite des adaptateurs d'objets

Les principaux composants applicatifs neutres sont les adaptateurs d'objets. Un adaptateur est instancié sur chaque nœud susceptible de recevoir des appels de méthodes, soit parce qu'il héberge des objets applicatifs, soit parce qu'il est utilisé comme passerelle.

L'adaptateur d'objets est responsable de la gestion d'un espace d'identification local, permettant de nommer chaque objet applicatif à l'échelle d'un nœud. Il offre deux primitives principales aux personnalités applicatives :

Exporter un objet, c'est-à-dire affecter un nouvel identificateur local à un objet applicatif ;

Supprimer un identificateur pour libérer les ressources allouées par l'adaptateur d'objets pour maintenir son association avec un objet applicatif.

De plus, la personnalité applicative peut demander l'inclusion d'une information arbitraire qu'elle fournit au sein de l'identificateur d'objet, et récupérer ensuite cette information. Cette fonctionnalité est utile dans le cas où la personnalité applicative demande l'attribution de plusieurs identificateurs distincts pour un même servant. Elle lui permet alors de discriminer les appels selon l'identificateur utilisé.

L'adaptateur d'objets offre également deux primitives au noyau générique :

Résoudre un identificateur local en un servant :
cette opération permet au lieur d'associer une référence désignant un objet local à sa continuation. Elle correspond à la réalisation, par l'adaptateur d'objets, de la fonction d'activation.

Libérer un servant après utilisation :
au moyen de cette primitive, le noyau générique indique à l'adaptateur d'objets qu'il a terminé d'utiliser un servant préalablement obtenu par la primitive précédente.

Certaines mises en œuvre d'adaptateur d'objet peuvent instancier à la volée un servant au moment de la résolution d'un identificateur. Dans ce cas, la seconde primitive est utilisée pour détruire ce servant créé dynamiquement.

Plusieurs mises en œuvre de l'interface abstraite des adaptateurs d'objets peuvent être considérées, en fonction des besoins de l'application. Nous en avons réalisé deux. L'une offre des fonctionnalités minimales ; elle est destinée aux nœuds disposant de peu de ressources. L'autre offre à l'utilisateur un contrôle étendu sur son comportement par l'intermédiaire de politiques.

6.2.2 Adaptateurs d'objets concrets

Mise en œuvre minimale : l'adaptateur d'objets simplifié

L'adaptateur d'objets simplifié permet l'attribution d'identificateurs à des servants créés explicitement par l'utilisateur. L'attribution est bijective : un identificateur est attribué à exactement un servant.

L'adaptateur d'objets simplifié maintient un simple tableau de servants actifs, et l'identificateur de chaque servant est son indice dans le tableau. La mise en œuvre est donc très simple, en particulier lorsque l'ensemble des objets résidant sur un nœud est déterminé statiquement et ne change pas au cours du temps.

En effet, dans cet adaptateur, l'opération coûteuse est la création d'un objet. Elle nécessite la recherche d'un emplacement libre dans le tableau des servants actifs, et le cas échéant l'allocation d'un nouveau tableau (si tous les emplacements sont occupés). En revanche, la détermination du servant correspondant à un identificateur se fait en temps constant, indépendamment du nombre d'objets créés.

Mise en œuvre flexible : l'adaptateur d'objets portable

L'adaptateur d'objets simplifié n'est pas configurable, et ne permet pas de faire incarner plusieurs objets par le même servant. Afin de permettre la configuration des fonctions réalisées par l'adaptateur d'objets, en particulier la fonction d'adressage à l'échelle d'un nœud et la fonction d'activation, nous avons réalisé

un adaptateur d'objets évolué directement inspiré du POA (*Portable Object Adapter*) de CORBA, décrit dans la section 3.2.3.

Le POA de PolyORB est intégré à la couche neutre : il n'impose pas l'utilisation de la personnalité applicative CORBA. Ainsi, d'autres personnalités peuvent bénéficier des fonctionnalités avancées qu'il fournit. En particulier, les personnalités CORBA (décrite dans la section 7.2) et DSA (décrite dans la section 7.3) en tirent parti.

Il constitue une classe dérivée de Obj_Adapter, qui contrôle non plus un espace d'adressage « plat », mais une hiérarchie dans laquelle sont définis les identificateurs d'objets. Un POA « racine » est défini, qui correspond à l'adaptateur d'objets connu du noyau générique. Ce POA définit un espace d'adressage qui peut contenir non seulement des identificateurs d'objets, mais également d'autres POA « fils ». Une structure arborescente peut ainsi être mise en place, chaque POA étant repéré par les noms des POA successifs permettant de l'atteindre à partir de la racine.

Des fonctions supplémentaires, en plus de l'interface commune à tous les adaptateurs d'objets, sont offertes aux personnalités applicatives pour créer des identificateurs d'objets à un niveau particulier de la hiérarchie, et sans nécessairement les associer à un servant.

L'interface présentée au noyau de l'intergiciel, quant à elle, est strictement conforme à l'interface abstraite que nous avons présentée dans la section 6.2.1 : ni le noyau générique, ni les personnalités protocolaires, n'ont connaissance des détails de l'organisation de l'adaptateur d'objets.

Selon les principes posés par la spécification du POA de CORBA, le comportement du POA de PolyORB est configurable au moyen de politiques. Nos politiques sont basées sur celles de CORBA. De plus, nous les avons rendues configurables de façon indépendante.

La normalisation CORBA spécifie le comportement des méthodes du POA en fonction de la combinaison de politiques en vigueur. Dans notre mise en œuvre, le POA est programmé une fois pour toutes ; nous avons identifié les aspects de son comportement qui devaient être personnalisés pour chaque politique, et nous avons spécifié une interface abstraite pour chacun de ces aspects. Nous avons également proposé des réalisations concrètes de ces interfaces, correspondant aux différentes variantes possibles pour chaque politique.

Nous avons ainsi donné une description des politiques du POA de CORBA comme autant d'instances du patron de conception *Strategy*. Cette organisation permet de développer et de modifier chaque axe de configuration du POA de manière indépendante, sans remettre en cause son architecture globale.

6.3 Composants protocolaires de la couche neutre

Les composants protocolaires de la couche neutre, présentés dans la section 4.4.3, offrent des outils génériques (non liés à un protocole et une représentation particuliers) pour la réalisation des personnalités protocolaires. Leur position dans l'architecture est illustrée par la figure 6.8.

Ces composants fournissent :

des tampons de communication optimisés servant à l'échange des messages, ou fragments de messages, entre les différentes entités qui constituent la couche protocolaire ;

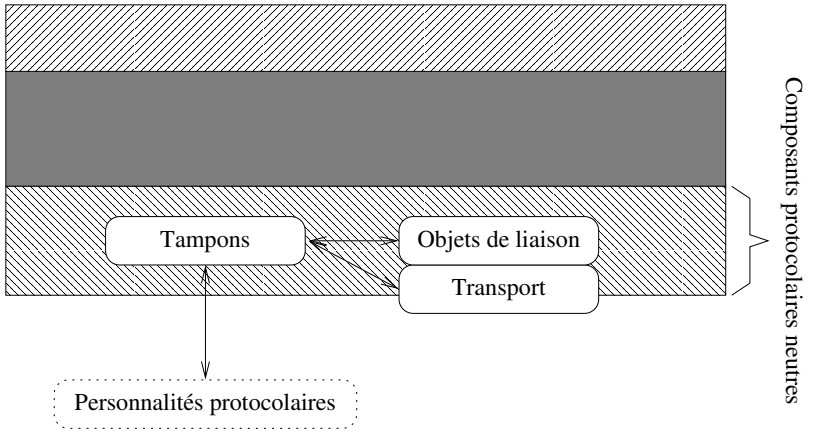


FIG. 6.8 – Composants protocolaires de la couche neutre

des mécanismes génériques de transport de données offrant une interface abstraite uniforme, encapsulant la fonction de transport présentée dans la section 4.5.1 ;

une architecture d'objets de liaison utilisée par le lieur pour construire les objets de liaisons à partir de modules réutilisables et de composants de personnalités.

6.3.1 Tampons de communication

Structure

Les messages reçus et transmis par les composants de communication sont stockés dans des tampons en mémoire. Chaque tampon représente une suite d'octets repérés par leur position relative à une position initiale fixée (cette position initiale pouvant être située à l'extérieur du tampon). La notion de position initiale associée à un tampon permet de préparer un message à transmettre, de traiter un message reçu, en plusieurs étapes indépendantes.

Chaque tampon est constitué d'une liste de blocs en mémoire ; chacun de ces blocs peut être rempli complètement ou partiellement de données. Chaque bloc intervenant dans les données d'un tampon est associé à un descripteur indiquant son adresse de départ, sa taille totale, et la longueur des données qu'il contient (qui peut être inférieure ou égale à la taille totale du bloc).

Chaque tampon contient également l'indication de la position relative de son premier octet, une indication de position courante et une indication de taille.

Utilisation

Un message à émettre est construit en ajoutant dans un tampon, initialement vide, une succession de blocs correspondant à des données contiguës.

Ces blocs peuvent être fournis par l'utilisateur, auquel cas seule leur adresse est enregistrée dans le tampon. Cette possibilité permet d'éviter les copies de données lorsqu'un fragment de message est déjà disponible en mémoire avec la représentation adéquate.

Ils peuvent également être alloués dynamiquement par la bibliothèque. Dans ce cas, l'allocation est réalisée par zones de taille fixe. Si l'utilisateur demande l'allocation de nombreux petits fragments successifs, la première demande déclenchera l'allocation d'un bloc de mémoire, dont seule une partie sera utilisée, et les demandes suivantes pourront être satisfaites sans nouvelle allocation dynamique, simplement en mettant à jour l'indicateur d'utilisation du tampon. Ce mécanisme a pour objectif d'optimiser la construction de messages formés d'une succession de valeurs de types élémentaires, tels que ceux rencontrés lors d'un appel de sous-programme à distance.

Une fois le message construit, la liste des blocs qui le constituent sont passés à un mécanisme de transport afin d'être transmis au nœud destinataire. Le mécanisme de transport est responsable de l'assemblage des différents blocs en un flot d'octets (l'opération de *scatter/gather*).

Un message reçu est stocké dans un bloc de mémoire contigu, et la position courante fixée au premier emplacement de ce bloc. Pour extraire les données reçues, une primitive est offerte qui retourne l'adresse d'une donnée de taille connue, et avance d'autant la position courante. Le décodage peut donc être réalisé sans copier les données hors du tampon.

6.3.2 Mécanismes de transport

Les mécanismes de transport sont les services offerts par le système d'exploitation qui permettent de transférer des messages d'un nœud à un autre d'un système réparti. Pour permettre l'utilisation de divers types de mécanismes de transport, en fonction des possibilités du système et des besoins de l'application, nous proposons une interface uniforme qui encapsule ces services, et permet à toute personnalité protocolaire de les utiliser.

Cette interface correspond à la fonction de transport que nous avons identifiée dans la section 4.5.2.

Elle est constituée de trois classes abstraites qui doivent être spécialisées pour chaque type de mécanisme de transport. Ces classes représentent respectivement les points d'accès et les points de terminaison du service de transport, ainsi que les sources d'événements asynchrones associées.

La séparation de ces trois classes a pour objectif d'exposer les différents aspects du sous-système de transport :

- synchronisation entre les événements extérieurs et l'intergiciel ;
- établissement d'une relation de transport entre deux intergiciels ;
- communication de données.

Elle permet ainsi l'intégration dans l'architecture de sous-systèmes de transport réalisant ces trois aspects de manières différentes.

Sources d'événements asynchrones

Les entités du service de transport (points d'accès et points de terminaison) doivent pouvoir déclencher des comportements de l'intergiciel en réponse à des événements extérieurs, provenant de nœuds distants. Elles sont donc vues par l'intergiciel comme des sources d'événements asynchrones.

Afin d'intégrer le traitement de ces événements dans l'ordonnancement des activités de l'intergiciel, nous associons chaque point d'accès et chaque point de terminaison à un objet le représentant en tant que source potentielle de tels

événements, et permettant à l'intergiciel d'être informé des occurrences de ces événements.

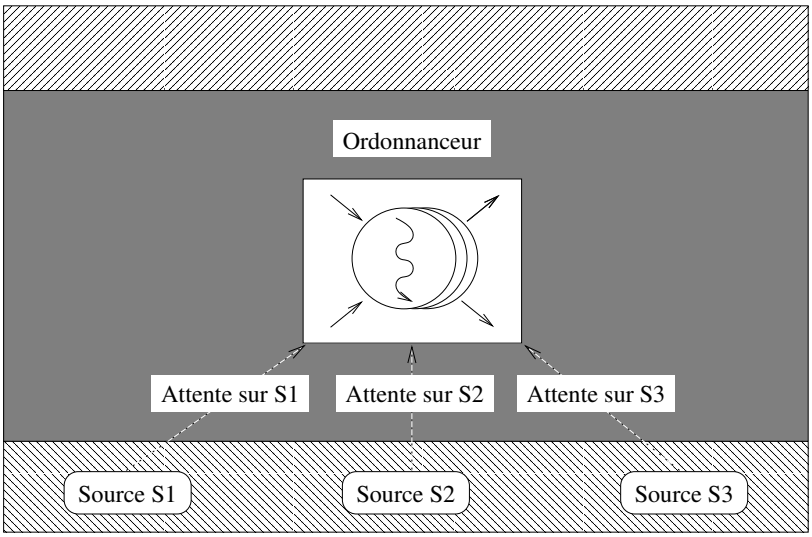


FIG. 6.9 – Sources d'événements asynchrones

Une source d'événement asynchrone est conceptuellement une entité offrant une primitive permettant d'attendre qu'un événement se produise (figure 6.9). Cependant, pour des raisons d'efficacité de mise en œuvre, il est intéressant de permettre à une tâche de se mettre en attente d'événements non sur une seule source d'événements asynchrones, mais sur une source quelconque parmi un ensemble de sources de même nature.

Nous introduisons donc une deuxième catégorie d'objets : les groupes de surveillance de sources d'événements asynchrones. Ces objets sont ceux auxquels l'intergiciel fait appel lorsqu'il souhaite attendre des événements asynchrones ou déterminer si des événements se sont produits, comme illustré par la figure 6.10.

Ainsi, lorsque le système d'exploitation permet de prendre connaissance d'événements asynchrones sur plusieurs entités de communication simultanément, il est possible de tirer parti de cette possibilité en définissant un seul groupe pour l'ensemble des sources.

Points d'accès du service de transport

Les points d'accès correspondent aux entités du système qui possèdent une adresse au sein d'un réseau de communication, et sont susceptibles de recevoir des connexions de nœuds distants ; les points de terminaison correspondent aux entités qui représentent une communication établie, et permettent l'échange de données. Dans le cas des modes de communication non connectés (par exemple le service de datagrammes UDP [85]), points d'accès et points de terminaison peuvent être deux vues différentes de la même entité du système d'exploitation.

La primitive essentielle des points d'accès est l'acceptation d'une nouvelle relation avec un intergiciel distant. Le point d'accès crée alors un nouveau point de terminaison, qui est ensuite intégré au sein d'un objet de liaison.

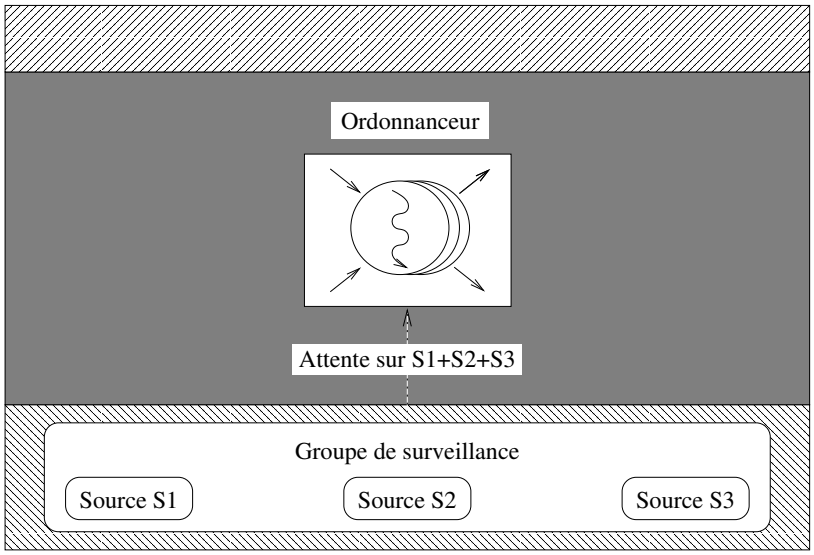


FIG. 6.10 – Groupes de surveillance de sources d'événements asynchrones

Points de terminaison du service de transport

Les points de terminaison du service de transport sont les entités réalisant la communication de données proprement dite, à l'intérieur de la fonction de transport. Un point de terminaison existe, dans l'intergiciel, pour chaque relation établie avec un intergiciel distant.

Les points de terminaison participent aux objets de liaison. Ils utilisent donc le patron *Component* ; leur interface est un sous-ensemble de celle des filtres, décrite plus en détail dans la section 6.3.3.

Les opérations primitives offertes par un point de terminaison consistent à :
- émettre le contenu d'un tampon vers l'intergiciel tiers auquel ce point de terminaison est associé ;
- recevoir des données provenant de cet intergiciel tiers.

La première est appelée par les couches supérieures de l'objet de liaison lorsque des données doivent être émises sur le réseau de communication.

La seconde est appelée par l'ordonnanceur lorsque la source d'événements asynchrones associée signale la présence de données reçues. Le point de terminaison copie alors ces données dans un tampon, puis signale aux autres composants de l'objet de liaison dont il fait partie qu'ils doivent les traiter.

Un exemple de transport : les *sockets* TCP

Nous avons réalisé une mise en œuvre concrète du sous-système de transport basée sur des connexions TCP. Cette mise en œuvre est portable : elle fonctionne dans tous les environnements compatibles avec GNAT, car elle utilise le paquetage Ada `GNAT.Sockets` [79], qui encapsule les différentes variantes de l'interface de programmation *sockets* [112].

Les sources d'événements asynchrones associées utilisent un moniteur basé sur l'appel système `select` : toutes les connexions basées sur les *sockets* peuvent

être surveillées par un unique groupe de surveillance.

L'uniformité de l'architecture de transport permet d'y intégrer d'autres mécanismes de communication tels que :

- *sockets* UNIX, pour la communication entre différents nœuds existant en fait sur une même machine physique ;
- liaisons point à point série, telles que la liaison radio pour robots autonomes que nous avons décrite dans la section 3.2.1.

Les objets de liaison n'utilisent que l'interface uniforme des mécanismes de transport, notre architecture permet de projeter un protocole donné sur n'importe lequel de ces mécanismes.

Le découplage entre les rôles de synchronisation, d'établissement de relation et d'échange de données permet la création de sous-systèmes de transports où ces aspects ne sont pas intrinsèquement liés, comme c'est le cas avec les *sockets*. Par exemple, un mécanisme de transport basé sur des segments de mémoire partagée et des mécanismes de synchronisation inter-processus pourrait être réalisé.

6.3.3 Objets de liaison

Organisation des échanges de messages

Comme nous l'avons évoqué dans la section 4.4.5, les objets de liaison créés par les personnalités protocolaires sont organisés comme des piles de filtres.

Au niveau le plus bas de chaque pile se trouve un point de terminaison du service de transport. Au niveau le plus haut est située une session, composant fourni par une personnalité protocolaire.

Entre les deux peuvent être insérés un nombre quelconque de filtres génériques, chargés de transformer et de réguler le flot de données entre leurs voisins. Cette organisation générale est inspirée de projets existants tels que *x*-kernel [46] ; elle est illustrée par la figure 6.11.

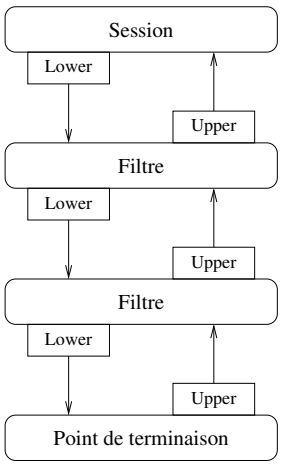


FIG. 6.11 – Organisation des piles de protocole

Chaque filtre est un composant muni de deux ports correspondant à ses voisins inférieur et supérieur.

Tous les services rendus par un objet de liaison sont réalisés par transfert de messages entre les couches successives. Chaque type de message ne traverse la pile de filtres que dans une direction fixée. Cependant, un message donné ne traverse pas nécessairement l'ensemble des couches : il peut être absorbé par un filtre intermédiaire au lieu d'être transmis au composant suivant.

Les messages essentiels échangés dans les piles de filtres sont :

- la demande d'émission de données, propagée des couches supérieures vers les couches inférieures ;
- l'indication de données reçues, propagée des couches inférieures vers les couches supérieures.

Un message complémentaire est utilisé par les couches supérieures pour signaler aux couches inférieures qu'elles sont prêtes à recevoir des données. Aucune indication de données reçues ne peut être émise alors que son destinataire n'y est pas prêt.

Aucun traitement de message par un composant d'un objet de liaison (point de terminaison, filtre ou session) n'est bloquant. L'ensemble des protocoles de PolyORB est piloté par événements.

Cette organisation réactive des objets de liaison autorise le découplage entre les objets de liaisons et les tâches. En particulier, elle permet la réalisation de serveurs n'utilisant qu'une seule tâche.

Chaque objet de liaison se comporte comme une machine à états qui réagit aux événements extérieurs provenant du sous-système de transport et du noyau générique. L'organisation des objets de liaison et du sous-système de transport piloté par événements est donc synchrone (un événement reçu est traité immédiatement par une suite d'opérations non bloquantes), tandis que l'ordonnanceur et les sources d'événements adoptent un fonctionnement asynchrone : notre architecture est similaire à celle du patron *Half-sync/Half-async*, notamment proposé par TAO [95], et que nous avons discuté dans la section 4.7.2.

Usines d'objets de liaison

Les objets de liaison, formés d'une pile de filtres liés à un point de terminaison du service de transport et à une session, sont créés au moyen d'usines d'objets de liaison. Ces structures sont des listes d'usines de filtres (la dernière étant en fait un protocole, c'est-à-dire une usine de sessions).

Lorsque la fonction de création d'une pile de filtres est appelée, les usines de filtres successives sont invoquées à tour de rôle. Chaque filtre est relié à ses voisins selon l'organisation indiquée par la figure 6.11. Un pointeur sur le filtre le plus bas de la pile est retourné à l'appelant, afin que celui-ci puisse connecter ce filtre au point de terminaison approprié.

Exemples de filtres

Dans PolyORB, des filtres sont utilisés en particulier pour effectuer la délimitation des messages qui sont traités par les entités réalisant les protocoles de communication.

En effet, une indication de données reçues provenant d'un point de terminaison du service de transport peut correspondre à un message complet (au sens du protocole), à un fragment de message, ou encore à la fin d'un message et au début d'un second.

Or, on souhaite qu'une session (c'est-à-dire le composant de plus haut niveau d'un objet de liaison, issu d'une personnalité protocolaire) reçoive exactement une telle indication lorsqu'un message (au sens du protocole qu'elle met en œuvre) a été reçu. En effet, cela permet de réaliser la session de façon naturelle, comme un automate dont les transitions sont déclenchées par les messages échangés.

On insère donc dans l'objet de liaison un filtre qui intercepte les indications de données reçues provenant des couches inférieures, et ne les retransmet à la session que lorsqu'un message complet a été reçu. Un tel filtre nécessite une connaissance *a priori* de la longueur des messages à recevoir. Nous l'utilisons notamment pour mettre en œuvre les objets de liaison IIOP, dans lesquels chaque message est précédé d'un en-tête de taille fixe indiquant sa longueur.

Le protocole HTTP est également réalisé sous forme d'un filtre. Il peut ainsi être utilisé comme support pour la mise en œuvre du protocole SOAP, aussi bien que pour l'utiliser en tant que protocole d'appel de méthodes à part entière.

Ces deux exemples montrent que l'architecture adoptée pour les objets de liaison de PolyORB permet de construire ceux-ci par assemblage de modules réalisant des traitements élémentaires de communication : cette architecture contribue à la factorisation de code entre différents objets de liaison, et à la réutilisation de ces modules élémentaires.

6.4 Synthèse

Nous avons présenté la réalisation de la couche neutre de PolyORB. Celle-ci remplit les fonctions d'adressage, de liaison et d'exécution de l'intergiciel, de façon indépendante du choix d'une personnalité.

Les modules de la couche neutre manipulent des représentations des interactions entre les objets définies par rapport à un modèle de répartition inspiré de celui de CORBA. Toute interaction entre objets applicatifs y est représentée par une « requête ».

Le noyau générique de la couche neutre comporte les outils nécessaires à la construction et à la manipulation des requêtes et des références d'objets. Il contient également deux composants centraux :

l'ordonnanceur-orienteur, responsable du contrôle du cycle de vie des requêtes et de leur acheminement entre les composants des personnalités ;

le lieur, responsable de la détermination de la continuation des références d'objets.

À ces composants s'ajoutent des modules génériques réutilisables par les personnalités applicatives : les adaptateurs d'objets génériques, capables de gérer l'espace d'identification local nécessaire à la fonction d'adressage, et réalisant la fonction d'activation.

Enfin, des outils pour la construction de personnalités protocolaires sont proposés :

- des tampons de communication optimisés pour limiter les copies de données ;
- un sous-système de communication réalisant la fonction de transport ;
- une architecture d'objets de liaison sous forme de piles de filtres.

CHAPITRE

7 Réalisation de personnalités applicatives

Sommaire

7.1 Anatomie d'une personnalité applicative 166
7.1.1 Interface pour les clients 167
7.1.2 Interface pour les serveurs 167
7.1.3 Génération de code 168
7.1.4 Descripteurs d'interface 169
7.2 CORBA 170
7.2.1 Compilateur IDL 170
7.2.2 Bibliothèque 171
7.2.3 Services 171
7.3 DSA 173
7.3.1 Génération de code 173
7.3.2 Exécutif 179
7.3.3 Services 179
7.4 MOMA 179
7.4.1 Intergiciels fondés sur le passage de messages 180
7.4.2 Bibliothèque 181
7.5 Synthèse — perspectives 182

DANS CE CHAPITRE, nous présentons la réalisation de personnalités applicatives pour PolyORB. En premier lieu, nous décrivons la structure générale d'une personnalité applicative, résultant de la mise en œuvre des principes énoncés dans la section 4.4.4 (réalisation d'une bibliothèque offrant aux objets clients et serveurs la possibilité d'interagir avec des objets situés sur des nœuds distants).

Nous présentons ensuite les personnalités que nous avons réalisées :

- une personnalité CORBA, qui offre le modèle « objets répartis » ;
- une personnalité réalisant l'annexe du langage Ada 95 pour les systèmes répartis (DSA, *Distributed Systems Annex*), offrant les modèles « objets répartis » et « appel de sous-programmes à distance » ;
- une personnalité réalisant le modèle « passage de messages », inspirée de l'architecture JMS.

7.1 Anatomie d'une personnalité applicative

Une personnalité applicative offre à des objets l'accès aux services d'un modèle de répartition. Son rôle consiste à lier certaines constructions de programmation dans le langage de l'application à des actions de répartition, telles que :
- appel de méthodes sur des objets distants ;
- mise à la disposition d'objets distants de services offerts par des objets locaux.

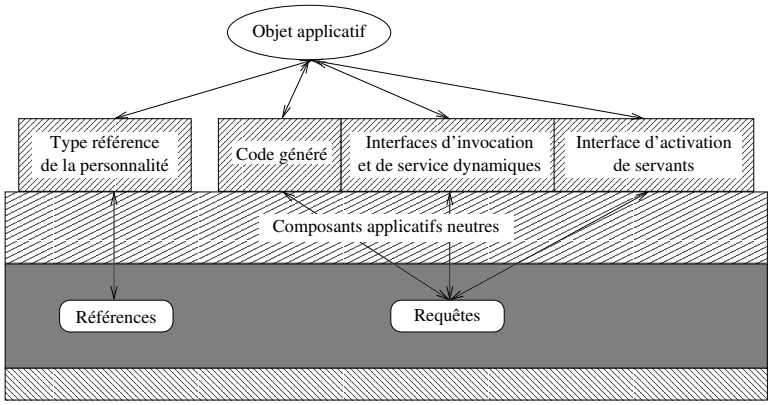


FIG. 7.1 – Anatomie d'une personnalité applicative

La figure 7.1 illustre le rôle de la personnalité applicative. Celle-ci fournit :
- une projection des références d'objets PolyORB vers un type de données du langage hôte ;
- un moyen d'initier et de recevoir des interactions (passage de messages, appels de sous programmes, appels de méthodes) avec les entités désignées par ces références :
  - soit au moyen de code généré spécifiquement pour l'application par un outil *ad hoc*
  - soit au moyen d'une bibliothèque d'invocation dynamique (côté client), d'une bibliothèque de service dynamique (côté serveur) ;
- un mécanisme permettant aux objets applicatifs de se se faire connaître de la couche neutre (un mécanisme d'activation des servants).

Elle offre ainsi :

aux objets applicatifs : des points d'entrée vers les fonctions de l'intergiciel ;

à la couche neutre : une interface lui permettant d'obtenir les servants correspondants aux objets applicatifs locaux (contribuant ainsi à la fonction d'activation, décrite dans la section 4.5.6) et de faire exécuter des requêtes par ces servants, dans le cadre de la fonction d'exécution de l'intergiciel (section 4.5.7).

Enfin, nous rattachons aux personnalités applicatives les services avancés qui peuvent être fournis aux applications pour faciliter leur déploiement. Ces services sont réalisés en utilisant les fonctions de base de l'intergiciel comme le font les objets applicatifs.

7.1.1 Interface pour les clients

Pour les clients, une personnalité applicative doit notamment offrir un moyen de représenter les références d'objets, et d'effectuer des appels de méthodes sur les objets qu'elles désignent. Ces facultés peuvent être offertes par deux moyens : un langage réparti ou une bibliothèque de répartition.

Langages répartis

Les références d'objets distants peuvent être représentées comme les références d'objets locaux (non répartis). C'est le cas des langages incluant directement des constructions réparties, telles que les *Network objects* de SRC Modula-3 [9]. De même, dans Java RMI ou DSA, un objet distant est désigné de la même façon qu'un objet local, respectivement à travers une référence d'objet et un pointeur.

Les appels de méthodes sur des objets distants sont alors programmés comme s'il s'agissait d'appels locaux. Il appartient au compilateur (ou à l'interpréteur) et à l'exécutif du langage de coopérer pour générer des appels au noyau générique.

Bibliothèques de répartition

Les références d'objets distants peuvent également être représentées par des types distincts des objets locaux. La répartition n'est alors pas aussi transparente que dans le cas des langages répartis : le programmeur doit rendre explicites les accès à des entités distantes.

Ainsi, dans CORBA, un compilateur IDL génère la déclaration d'un type « référence » dans le langage hôte, ainsi qu'une mise en œuvre des méthodes de ce type qui réalise les appels nécessaires à l'intergiciel. De même, dans le cas des bibliothèques de passage de messages, les actions de répartition apparaissent explicitement dans le code applicatif.

Une personnalité applicative dans laquelle les actions de répartition sont explicites peut également proposer une interface d'invocation dynamique. Une telle interface permet à l'application de faire appel à des méthodes dont la signature n'était pas connue au moment de la compilation. De plus, elle supprime la nécessité de générer du code spécifique à l'application.

7.1.2 Interface pour les serveurs

Pour les serveurs, le rôle de la personnalité applicative consiste, à partir d'objets concrets de l'application existant sur un nœud, à établir des servants qui les représentent auprès du noyau générique, et permettent à celui-ci de leur transmettre des appels de méthodes.

De la même façon que, pour les clients, les types définis par une personnalité applicative et représentant des références d'objets offrent une façade pour la fonctionnalité d'appel de méthodes à distance, la personnalité applicative côté serveur réalise une façade pour les appels de méthodes effectués par l'intergiciel pour le compte d'objets distants.

Cette façade est établie sur l'interface offerte par les objets de l'application à l'intergiciel : chaque objet de l'application est encapsulé par une instance de la classe abstraite des servants, afin de permettre à cet objet de traiter des

requêtes, c'est-à-dire de recevoir des appels distants. En fournissant un servant à partir d'une référence d'objet, la personnalité applicative contribue à la fonction d'activation (décrite dans la section 4.5.6).

L'interface des servants est la vue neutre des objets applicatifs que la personnalité applicative présente à la couche neutre. Cette interface comporte une unique primitive, correspondant à l'exécution d'une requête par l'objet applicatif. Cette primitive doit être mise en œuvre par chaque personnalité applicative ; elle correspond à la phase finale de la fonction d'exécution de l'intergiciel : l'aiguillage de la requête sur le sous-programme applicatif correspondant.

Elle réalise le démultiplexage des méthodes : à partir d'une requête neutre contenant le nom de la méthode à appeler, elle identifie le sous-programme applicatif approprié, prépare ses paramètres et effectue l'appel.

Pour qu'un intergiciel soit utilisable dans une application temps réel, son comportement doit être aussi déterministe que possible. On souhaite notamment que l'opération de démultiplexage soit effectuée en temps constant plutôt que par exploration linéaire ou dichotomique de la liste des méthodes. Il est possible d'utiliser à cet effet des fonctions de hachage parfaites, qui permettent de garantir un démultiplexage en un temps indépendant de la méthode appelée et du nombre de méthodes de l'objet.

Dans les cas, fréquents, où la liste des méthodes d'une interface est connue à l'avance et où les servants sont créés par un outil de génération de code, il est même possible d'utiliser des fonctions minimales parfaites (décrites dans la section 5.3.2). L'intérêt d'utiliser des fonctions de hachage parfaites pour le démultiplexage dans le cas d'applications temps réel a été mis en évidence dans TAO [36].

Toutefois, le choix d'une stratégie de démultiplexage entre fonctions de hachage et exploration d'une liste de méthodes résulte du compromis choisi par l'utilisateur entre le temps nécessaire pour démultiplexer un appel et la taille des données à stocker. En particulier, l'approche basée sur les fonctions de hachage n'est pas intéressante pour les objets comportant un très faible nombre de méthodes : elle impose alors un surcoût d'espace de stockage, sans apporter de gain de temps. Il est donc intéressant de proposer à l'utilisateur le choix de l'une ou de l'autre approche.

7.1.3 Génération de code

Comme nous l'avons indiqué dans les sections précédentes, la mise en œuvre d'une personnalité applicative peut nécessiter la production automatique de certains composants intermédiaires entre les objets applicatifs et l'intergiciel :

- côté client, pour créer des entités représentant les objets distants (ces entités déclenchant des actions de répartition lorsqu'elles sont utilisées par les objets applicatifs locaux) ;
- côté serveur, pour créer les servants encapsulant les objets applicatifs et permettant à la couche neutre d'y faire appel, dans le cadre de la fonction d'exécution.

Dans un intergiciel traditionnel, ces composants réalisent directement des opérations d'emballage de paramètres ou de communication. Dans PolyORB, ils réalisent l'adaptation entre les représentations de la personnalité applicative et celles de la couche neutre.

Le générateur de code d'une personnalité applicative se base sur une description des services offerts par un ensemble donné d'objets applicatifs, et génère le code correspondant pour les clients et pour les serveurs.

La génération de code peut être
- une étape distincte de la compilation du code applicatif (c'est le cas pour une personnalité basée sur une bibliothèque de répartition, telle que CORBA) ;
- intégrée de façon transparente dans le traitement du code de l'application (dans le cas d'un langage réparti).

7.1.4 Descripteurs d'interface

Dans la section 4.2.4, nous avons noté qu'une passerelle qui reçoit une requête sur une session et doit la réémettre sur une autre session doit disposer de la signature de la méthode appelée. Or, comme nous l'avons indiqué dans la section 6.1.2, c'est le servant qui fournit cette information lors du déballage des paramètres d'une requête.

Une passerelle ne dispose pas de l'objet applicatif local qui va *in fine* traiter la requête. Il est donc nécessaire de lui rendre disponibles les signatures de méthodes par un autre moyen, afin de permettre le déballage des paramètres dans le protocole originaire, et leur réemballage dans le protocole de l'objet destinataire. En l'absence de servant local, l'intergiciel jouant le rôle de passerelle doit disposer d'un mécanisme de description des interfaces.

Ce mécanisme est mis en œuvre lorsqu'il est nécessaire de transmettre une requête sur une session (pour laquelle le nœud local est client) alors que ses paramètres n'ont pas été déballés (depuis une autre session, pour laquelle il joue le rôle de serveur).

À cet effet, une interface abstraite uniforme est définie comportant les opérations nécessaires à la détermination de la signature d'une méthode connaissant la référence d'un objet et le nom de la méthode. Une mise en œuvre de cette interface est un *descripteur d'interface* (car elle permet d'obtenir la description d'une méthode).

Toute mise en œuvre de descripteur d'interface est liée à deux éléments de personnalité :
- un langage de description, c'est-à-dire un modèle formel de représentation des signatures de méthodes (dans le cas d'un langage réparti, c'est ce langage lui-même qui tient lieu de formalisme de description) ;
- un mécanisme permettant d'obtenir une description (dans le langage approprié), connaissant la référence d'un objet et le nom d'une méthode de cet objet.

À partir de ces éléments, un descripteur d'interface doit effectuer deux opérations successives :

1. obtenir la description de la méthode appelée, éventuellement en contactant un référentiel d'interfaces par le truchement d'un appel de sous-programme ou de méthode à distance ;
2. traduire cette description du langage de la personnalité vers le modèle neutre de PolyORB.

Nous rattachons le choix de ces éléments à une personnalité applicative. Il convient de noter cependant que le descripteur d'interfaces d'une personnalité

applicative donnée peut être instancié sur un nœud qui n'utilise pas lui-même cette personnalité (pour ses propres objets applicatifs).

La détermination du type de descripteur d'interface à utiliser pour le traitement d'une requête particulière demeure un problème ouvert, pour lequel existent de multiples propositions concurrentes :

- la norme CORBA prévoit que tout objet doit en principe offrir une méthode *get_ interface* retournant la référence de sa description dans le référentiel d'interfaces CORBA (*Interface Repository*) ;
- dans le cadre des applications basées sur le WWW, un langage de description basé sur une DTD XML a été proposé : WSDL [16]. Diverses méthodes d'accès peuvent être utilisées pour obtenir la description WSDL correspondant à un service donné, par exemple une requête HTTP sur un serveur approprié.

Plusieurs solutions peuvent être envisagées dans PolyORB pour déterminer le descripteur d'interfaces à utiliser pour un appel :

- l'essai successif de tous les descripteurs d'interface connus ;
- la détermination d'un descripteur en fonction de l'information de type contenue dans la référence d'objet ;
- la configuration par l'utilisateur de critères de décision externes.

Dans la mise en œuvre actuelle, un seul descripteur d'interface est disponible : il s'agit d'un client pour le référentiel d'interfaces CORBA.

7.2 CORBA

La personnalité applicative CORBA offre un compilateur IDL mettant en œuvre la projection normalisée pour Ada 95 [73], ainsi que les unités de bibliothèque définies par cette projection.

7.2.1 Compilateur IDL

Le compilateur IDL d'AdaBroker, `idlac`, a été adapté pour générer des appels à la couche neutre de PolyORB, au lieu de réaliser directement des échanges de messages GIOP.

Plusieurs unités sont produites à partir de chaque spécification IDL :

- une souche cliente, dont la déclaration est fixée par la projection normalisée d'IDL vers Ada 95 ;
- un squelette de serveur qui déclare un type de servant réalisant l'interface avec la couche neutre, décrite dans la section 7.1.2 ;
- une unité contenant les fonctions de conversion entre les types de l'utilisateur et les conteneurs Any, ainsi que les typecodes correspondants ;
- éventuellement, un canevas de mise en œuvre d'objet applicatif à compléter par l'utilisateur

Si l'utilisateur le demande, `idlac` peut également produire une unité réalisant l'enregistrement des services spécifiés par un contrat IDL dans un référentiel d'interfaces.

7.2.2 Bibliothèque

La personnalité CORBA de PolyORB fournit les unités définies par la projection normalisée d'IDL sur Ada 95.

Une partie de cette bibliothèque provient d'AdaBroker, l'intergiciel CORBA développé par notre équipe à l'ENST [5]. Elle a été adaptée pour utiliser et encapsuler les types de données et les services de base de PolyORB (décrits dans le chapitre 5) et les fonctionnalités de répartition de la couche neutre (chapitre 6).

Les types de base CORBA (en particulier les références d'objets et les Anys) encapsulent les types équivalents de la couche neutre. De même, les types de données de l'interface d'invocation dynamique (DII) et de l'interface de serveurs dynamiques (DSI) forment deux façades pour les mêmes objets sous-jacents, qui sont les requêtes de PolyORB.

La norme impose que le type de base de tous les objets applicatifs CORBA, CORBA.Impl.Object, fasse l'objet d'un comptage de références sur le principe des pointeurs intelligents que nous avons décrit dans la section 5.3.1. Nous devons donc le dériver du type Entity.

Cependant, l'interface de la couche neutre impose aussi que les objets applicatifs soient des Components. Comme indiqué dans la section 5.4.1, nous avons recouru à la construction de vues multiples pour permettre aux servants CORBA de posséder cette dualité.

Une mise en œuvre du POA CORBA est offerte. L'architecture du POA générique de PolyORB étant directement inspirée de celle du POA de CORBA, celui-ci est réalisé comme une façade très simple, qui n'apporte pas de fonctionnalités en plus de celles de la couche neutre, mais se contente d'offrir une interface de programmation appropriée.

7.2.3 Services

La norme CORBA définit un certain nombre de services communément utilisés pour la construction d'applications réparties : les *COS* (*Common Object Services*). Ces services sont définis comme des objets répartis, qui peuvent être réalisés une fois pour toutes et être utilisés dans des applications variées.

Nous détaillons ci-après la réalisation de quelques-uns de ces services pour PolyORB.

Service de dénomination

Le service de dénomination *COS Naming* permet d'associer un nom symbolique à une référence d'objet. Comme nous l'avons indiqué dans la section 2.5.1, un tel service joue un rôle important dans l'initialisation d'une application répartie. En effet, il permet à chaque nœud de l'application de prendre connaissance de références désignant les divers objets qui la composent.

Nous avons réalisé les objets élémentaires de ce service :

NamingContext représente un contexte de dénomination, au sein duquel des noms peuvent être définis, et associés à des références d'objets ;

BindingIterator permet de parcourir un NamingContext afin d'extraire la liste des associations entre noms et références (les *bindings*) qu'il contient.

Nous avons vérifié l'interopérabilité de notre mise en œuvre avec des clients utilisant des intergiciels tiers.

Référentiel d'interfaces

Le référentiel d'interfaces CORBA (*CORBA Interface Repository*) est un annuaire de descriptions d'interfaces. À partir d'une référence désignant un objet, ce service permet d'accéder à une description des services offerts par cet objet.

Ainsi, il est possible de déterminer à l'exécution quelle est l'interface offerte par un objet, quelles sont ses méthodes, et les signatures de ces méthodes. Cette description est elle-même fournie sous forme d'un objet dont les propriétés sont accessibles par appel de méthodes à distance.

Nous avons réalisé un serveur de référentiel d'interface complet. Ce serveur crée un objet initial, dont la référence est disséminée au moyen du service de dénomination. Les divers objets applicatifs ayant pris connaissance de cette référence peuvent enregistrer la description de leur interface auprès du référentiel. Cette description devient alors disponible pour tous les autres objets de l'application.

Le serveur référentiel d'interfaces peut alors être interrogé par tout nœud ayant besoin de connaître les propriétés d'un objet. En particulier, nous avons pu utiliser notre référentiel d'interfaces pour fournir, à l'exécution, la signature d'une méthode à un module descripteur d'interface, afin de permettre la création de passerelles dynamiques selon le procédé décrit dans la section 7.1.4.

Service d'événements

CORBA définit également un service de communication par événements, *COS Events*. Ce service permet à un certain nombre de consommateurs d'observer des événements produits de manière asynchrone par des objets producteurs.

Cette modalité d'interaction se rapproche de la diffusion de messages. Elle est utile en particulier pour assembler rapidement des applications à partir de composants modulaires. Ainsi, le service de communication par événements est à la base de la plateforme d'assemblage flexible de composants OCP [120].

Nous avons vérifié que notre mise en œuvre de la spécification *COS Events* sous forme d'objets CORBA standard, réalisée pour AdaBroker, était compatible avec la personnalité CORBA de PolyORB.

Bibliothèque

La personnalité applicative CORBA pourra être étendue pour fournir un environnement complet adapté à l'utilisation dans des applications embarquées et temps réel. Le noyau générique de PolyORB offre d'ores et déjà des modalités d'interaction autres que l'appel synchrone de méthode. En particulier, il est possible d'émettre des appels asynchrones, ou de mettre en place des mécanismes de rappel (*Callback*).

Ces fonctionnalités seront rendues disponibles pour les objets CORBA en mettant en œuvre l'interface CORBA Messaging (partie intégrante des spécifications CORBA de base). De même, l'interface Real-time CORBA [75] ofre à l'utilisteur le moyen de contrôler les personnalités protocolaire.

L'association de cette interface et de politiques de parallélisme adaptées au temps réel (déjà présentes dans PolyORB) peut être mise à profit dans des applications industrielles, où elle facilite la coordination de calculateurs indépendants embarqués [117].

7.3 DSA

La personnalité applicative DSA est intégrée dans l'environnement de compilation Ada 95 GNAT. Une partie de la mise en œuvre du processus de compilation existant dans GLADE a été reprise et modifiée. Le code généré réalise des appels à une bibliothèque *ad hoc* offrant une interface simplifiée aux fonctionnalités de la couche neutre.

7.3.1 Génération de code

Pour la production automatisée de code de répartition dans la personnalité DSA, deux options se présentaient :

- réutiliser le générateur de CIAO et le modifier pour produire des souches et des squelettes utilisant la couche neutre de PolyORB ;
- réutiliser le mécanisme de génération de code de GNAT pour GLADE, et l'adapter à PolyORB.

Dans l'optique de réaliser une mise en œuvre complète et conforme de l'annexe des systèmes répartis d'Ada 95, le projet GNAT avait d'ores et déjà envisagé l'approche consistant à générer du code source Ada pour les souches et les squelettes. Cependant, cette solution n'était pas suffisamment flexible pour permettre la génération aisée de code de répartition. La génération de code de répartition a donc été par la suite intégrée directement au compilateur.

De plus, l'architecture du code de répartition généré par la version existante de GNAT (souches et squelettes dédiés à l'exécutif de répartition GARLIC) était plus proche du résultat que nous souhaitions obtenir (souches et squelettes génériques liés à la couche neutre de PolyORB) que celle des passerelles produites par CIAO.

Enfin, la mise en œuvre de CIAO demeurait incomplète sur certains aspects du langage.

Nous avons donc retenu la solution consistant à intégrer notre génération de code au compilateur GNAT.

Phases de compilation dans GNAT

Nous rappelons ici brièvement l'organisation des diverses phases de traitement du compilateur GNAT, illustrées par la figure 7.2.

À partir du texte source d'une unité de compilation, les traitements suivants sont effectués :

Analyse lexicale : le texte source est découpé en symboles, qui sont classifiés en fonction des diverses catégories d'éléments terminaux de la grammaire du langage (mots-clés, littéraux, noms...).

Analyse syntaxique : Le compilateur vérifie que la séquence des terminaux est une instance bien formée de la grammaire du langage. Il construit une

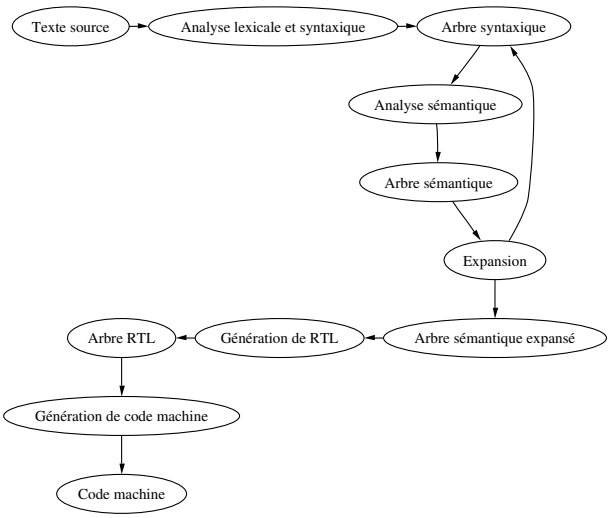


FIG. 7.2 – Phases de compilation dans GNAT

représentation de l'unité de compilation sous forme d'un arbre correspondant à sa structure grammaticale : l'arbre syntaxique.

Analyse sémantique : Les règles sémantiques du langage sont appliquées à l'arbre syntaxique :
- les noms sont reliés aux entités qu'ils désignent ; le type de chaque expression est déterminé, ce qui permet la levée des ambiguïtés résultant de la surcharge ;
- la légalité du programme est vérifiée ;
- les ambiguïtés de la grammaire sont levées en utilisant les informations sémantiques. Par exemple, une forme constituée d'un nom et d'une expression entre parenthèses peut correspondre à un appel de fonction ou bien à la désignation d'un composant d'un tableau. Cette ambiguïté de la grammaire ne peut être levée qu'une fois que la nature de l'entité désignée par le nom est connue.

L'arbre syntaxique est décoré d'informations supplémentaires ; la structure obtenue est l'arbre sémantique de l'unité de compilation.

Expansion : Les constructions complexes du langage sont récrites en termes de constructions plus simples, qui peuvent donner lieu à la génération de code machine. Lors de cette phase, en particulier, les constructions orientées objets du langage (types étiquetés et appels de sous-programmes donnant lieu à un aiguillage) sont transformées en appels indirects à travers une table de pointeurs sur sous-programmes.

Lors de la phase d'expansion, le compilateur remplace ou augmente certains nœuds de l'arbre sémantique avec des fragments d'arbre syntaxique. Ceux-ci subissent à leur tour, récursivement, les phases d'analyse sémantique et d'expansion.

Traduction : Chaque nœud de l'arbre expansé est finalement transformé en une représentation intermédiaire de bas niveau, indépendante à la fois du langage source et de l'architecture matérielle cible. Cette représentation

est l'arbre RTL (*Register Transfer Language*) de GCC.

Génération de code machine : Le dorsal de GCC traduit les fragments de RTL en code machine spécifique d'une architecture matérielle.

Au cours de l'analyse sémantique et de l'expansion, de nouveaux nœuds peuvent être introduits dans l'arbre, et donner lieu à des allers-retours entre les deux phases. Cela facilite l'écriture de code généré, car celui-ci peut être spécifié sous forme d'arbre syntaxique (proche du code source), et n'a pas besoin d'être écrit sous forme complètement analysée.

L'essentiel de la génération de code pour la répartition intervient au cours de la phase d'expansion, et utilise des informations rassemblées lors de la phase d'analyse sémantique.

Des souches clientes et des servants sont produits pour chaque type objet réparti. Pour chaque unité offrant des sous-programmes appelables à distance (Remote_Call_Interface), un objet est également défini, dont les méthodes sont les sous-programmes de l'unité.

Un objet est également défini pour chacun des sous-programmes appelables à distance, afin de permettre la mise en œuvre des types pointeurs sur sous-programmes distants.

Pour chaque sous-programme appelable à distance, une souche cliente est créée, qui remplit des fonctions comparables à celles de la personnalité CORBA :

1. stockage des paramètres dans des conteneurs universels Any ;
2. allocation d'un objet Request ;
3. appel de sa primitive Invoke.

La conversion des paramètres en Any est réalisée par des fonctions de la couche neutre pour les types de base. Pour les types complexes définis par l'utilisateur, ces fonctions d'assistance sont générées automatiquement en tant que de besoin. De même, des fonctions sont générées pour créer les typecodes qui décrivent ces types complexes.

Pour les types pointeurs distants, cependant, ces fonctions sont générées automatiquement au moment du traitement de leur déclaration. En effet, contrairement aux types « ordinaires », les types pointeurs à distance ont essentiellement vocation à être transmis entre des partitions lors d'appels de sous-programmes ou de méthodes. Il n'est donc pas intéressant de générer au vol leurs fonctions d'assistance.

Les différentes entités accessibles à distance mettent en jeu plusieurs mécanismes de génération de code, que nous détaillons ci-après.

Sous-programmes distants

Lorsque la directive de compilation Remote_Call_Interface, est appliquée à une unité, les sous-programmes qui y sont déclarés peuvent être appelés à distance. Comme illustré par la figure 7.3, deux versions compilées de chaque unité Remote_Call_Interface sont produites :

– l'une contient uniquement les souches clientes ;
– l'autre contient un squelette de serveur et les corps des sous-programmes.

La première version est générée à partir de la seule déclaration de l'unité Remote_Call_Interface.

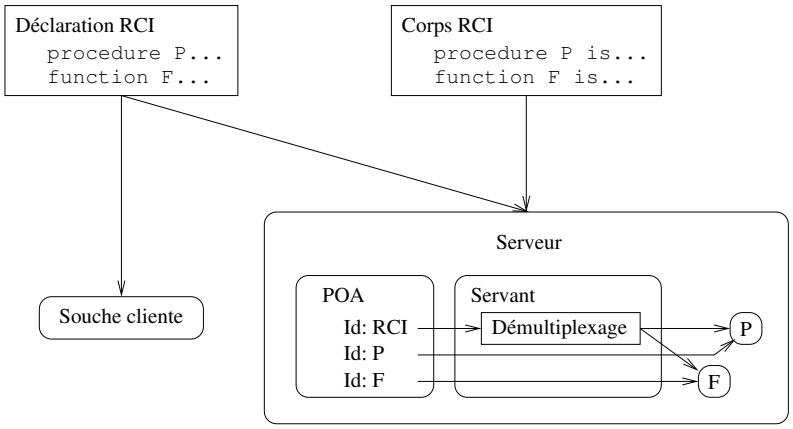


FIG. 7.3 – Code généré pour une unité RCI

La seconde contient, elle, la mise en œuvre des sous-programmes, ainsi que la déclaration d'un servant. Elle contient également les fonctions d'initialisation nécessaires à l'enregistrement de ce servant auprès de la couche neutre. Le servant est configuré comme servant par défaut d'un POA approprié, au sein duquel un identificateur d'objet est attribué pour l'unité dans son ensemble. Les appels directs aux sous-programmes de l'unité sont réalisés comme des appels de méthodes sur cet objet.

Un identificateur d'objet supplémentaire distinct est associé à chacun des sous-programmes, afin que ceux-ci puissent être désignés individuellement. Cet identificateur est utilisé pour la construction des pointeurs sur sous-programmes distants, qui permettent d'effectuer des appels de sous-programmes à distance liés dynamiquement.

Lorsque la directive de compilation All_Calls_Remote s'applique à une unité, tous les appels à cette unité doivent utiliser l'intergiciel même s'ils proviennent du même nœud. Dans ce cas, une copie des souches clientes est également produite pour la seconde version, avec des noms différents. Le compilateur récrit alors les appels locaux pour utiliser ces souches en lieu et place des sous-programmes réels.

Objets répartis

Lorsqu'un type pointeur sur objet distant (RACW, *Remote Access-to-Classwide*) est déclaré, un type objet « relais » compatible avec le type réel est synthétisé, comme illustré par la figure 7.4.

Ce type est construit pour avoir une table de méthodes identique à celle du type réel, de sorte qu'un pointeur (local) puisse désigner indifféremment un objet du type réel ou du type relais. Cela permet d'optimiser les appels locaux : lorsqu'un pointeur sur un objet potentiellement distant désigne un objet local, les appels de méthode sont effectués directement, à travers le mécanisme d'aiguillage local usuel, et sans pénalité liée à la répartition.

Le type relais contient une référence d'objet générique, et ses opérations primitives sont les souches clientes correspondant à celles du type réel.

Un servant est également généré pour tous les objets d'une même classe

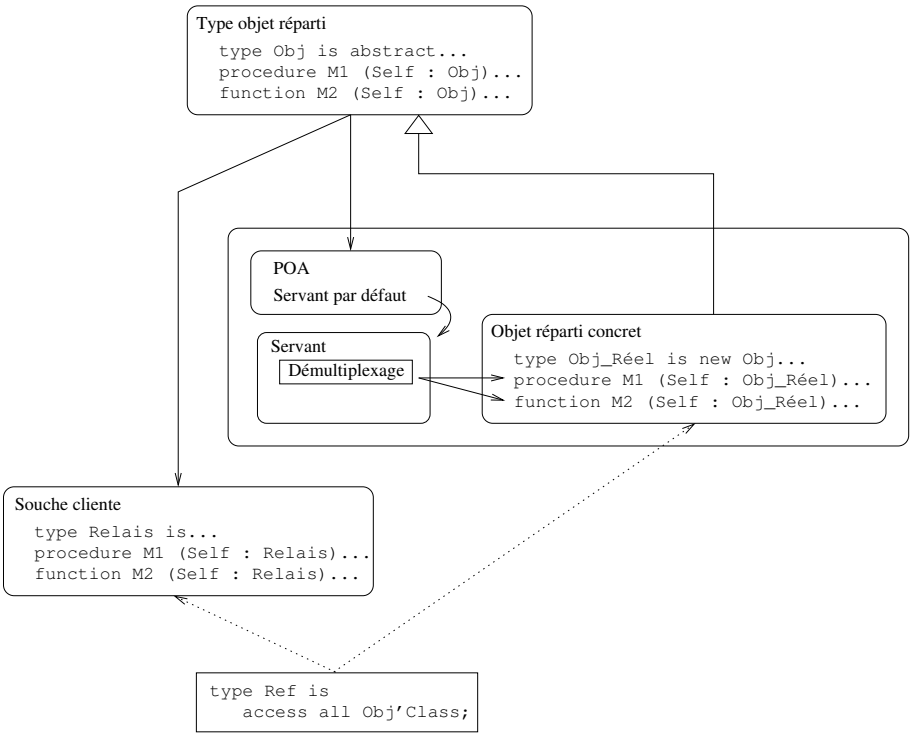


FIG. 7.4 – Code généré pour un type objet réparti

accessible à distance. Un POA est associé à chaque telle classe, avec le servant de la classe comme servant par défaut.

Dans DSA, un objet est rendu accessible à distance en convertissant un pointeur local sur cette objet en pointeur distant. Lors de cette conversion, le code généré par le compilateur demande au POA associé à sa classe l'attribution d'un identificateur d'objet. Une référence est ensuite construite, qui peut être transmise à d'autres partitions.

Pointeurs sur sous-programmes distants

Notre mise en œuvre des pointeurs sur sous-programmes distants (RAS, *Remote Access-to-Subprogram*) est basée sur celle des objets répartis : à chaque type pointeur sur objet réparti est associé un type objet réparti possédant une unique opération primitive Call et un type pointeur à distance correspondant. Cette évolution par rapport à la mise en œuvre était identifiée comme perspective de la version existante de GLADE [116]. Elle permet de limiter le coût des RAS désignant un sous-programme local. De plus, elle traduit, dans la réalisation de la personnalité, l'unification offerte par la couche neutre entre le modèle de répartition des objets répartis et celui de l'appel de sous-programmes à distance.

Lorsque l'adresse d'un sous-programme est prise et affectée à une variable d'un type RAS, cette opération est remplacée par l'appel d'une fonction déterminant la référence d'objet appropriée et créant un relais.

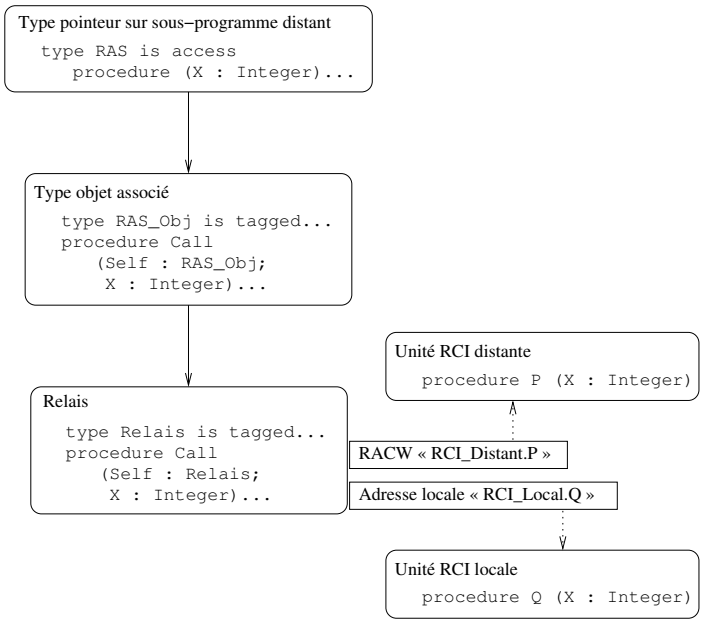


FIG. 7.5 – Code généré pour un type pointeur sur sous-programme distant

Lorsqu'une variable d'un type RAS est déréférencée, cette expression est réécrite par le compilateur. Elle est remplacée par l'appel d'une fonction réalisant soit un appel local (dans le cas d'un sous-programme d'une unité située sur la partition courante et à laquelle la directive All_Calls_Remote ne s'applique pas), soit un appel à la primitive Call du type relais sous-jacent (cette primitive jouant le rôle de souche cliente). Les appels à un sous-programme local à travers un RAS ne subissent pas le coût d'un passage par l'intergiciel.

Contrairement au cas des objets répartis, nous ajoutons ici un niveau d'indirection pour les appels locaux. En effet, pour les RACW, il est possible de tirer parti du mécanisme ordinaire d'aiguillage des objets Ada 95 pour discriminer les appels locaux et distants.

En revanche, il est possible pour une partition de détenir un pointeur sur sous-programme distant qui désigne un sous-programme pour lequel elle ne dispose pas de souche cliente. Par ailleurs, un même type RAS peut être utilisé pour désigner des sous-programmes différents. Le code généré pour déréférencer un tel pointeur doit donc recevoir en paramètre l'identité du sous-programme appelé.

La nécessité de ce paramètre additionnel interdit la discrimination entre appels locaux et distants au moyen du seul niveau d'indirection fourni par le pointeur lui-même à moins de passer des informations supplémentaires à la souche cliente dans le cas distant, en utilisant des variables spécifiques d'une tâche. Le gain minime obtenu en supprimant un niveau d'appel est alors perdu à cause de la complexité du passage de cette information.

7.3.2 Exécutif

La personnalité applicative DSA comprend une unique unité qui réalise l'interface entre les objets applicatifs et la couche neutre. Cette unité, System.PolyORB_Interface, n'est pas utilisée directement par les objets applicatifs, mais seulement par le code généré lors de la phase de compilation.

Son interface est spécialement adaptée pour pouvoir être utilisée sous les contraintes particulières qui s'appliquent au code synthétisé lors de la phase d'expansion par le compilateur :

- Chaque unité de bibliothèque utilisée par le code généré doit faire l'objet d'une déclaration spéciale. Il est donc intéressant de regrouper des déclarations de surnoms (*renaming declarations*) dans une seule unité.
- Les noms des entités de l'exécutif auxquelles le code généré fait référence ne sont pas résolus par le mécanisme usuel, mais par des fonctions simplifiées et optimisées. En particulier, elles ne peuvent pas être surchargées.

7.3.3 Services

À partir du service de dénomination CORBA COS Naming, que nous avons évoqué dans la section 7.2.3, nous avons défini un service de dénomination générique permettant de localiser les unités Remote_Call_Interface. Chaque unité est enregistrée dans un contexte global à l'application, selon le mécanisme que nous avons décrit pour CIAO dans la section 3.3.3.

Le mécanisme des *boot servers* de GLADE, décrit dans [116], était également basé sur l'utilisation d'un service de dénomination pour disséminer les coordonnées des unités Remote_Call_Interface d'une application DSA à l'ensemble des partitions.

Nous avons également réalisé un service réalisant l'attribution d'identificateurs uniques à chaque partition. La norme Ada 95 impose en effet l'existence de ce service. Nous l'avons réalisé à partir des fonctionnalités de répartition de base : il s'agit d'une unité Remote_Call_Interface, offrant des sous-programmes appelables à distance.

7.4 MOMA

Les personnalités applicatives CORBA et DSA réalisent le modèle « objets répartis ». La personnalité DSA offre également l'appel de sous-programmes à distance.

Dans cette section, nous décrivons la troisième personnalité que nous avons réalisée : MOMA (*Message-Oriented Middleware for Ada*). Celle-ci réalise le modèle de répartition « passage de messages ».

Nous décrivons en premier lieu l'architecture retenue dans PolyORB pour mettre en œuvre ce modèle. Elle est inspirée du service de passage de messages normalisé pour Java, JMS [114]. Le choix de cette spécification offre une base éprouvée pour la proposition d'une architecture d'intergiciel basé sur le passage de messages pour Ada 95. Par ailleurs, cette spécification est la première normalisation d'intergiciel pour le passage de messages (MOM, *Message-Oriented Middleware*) largement adoptée, les produits existants ayant auparavant retenu des interfaces spécifiques et incompatibles.

Nous détaillons ensuite l'interface offerte aux objets applicatifs et sa réalisation.

7.4.1 Intergiciels fondés sur le passage de messages

Architecture

Les intergiciels basés sur le passage de message offrent des fonctionnalités permettant de créer des messages, de les router à travers des nœuds intermédiaires jusqu'à leur destinataire, et de les remettre à celui-ci.

L'architecture de JMS définit plusieurs modalités de passage de messages :

échange simple dans le cas où les messages sont transmis directement d'un émetteur à un destinataire, sans transiter par un nœud intermédiaire de traitement de messages ;

files de messages lorsque les messages émis par un nœud sont stockés dans une file d'attente où le nœud destinataire peut les récupérer ;

bus à messages lorsque les messages émis par un nœud sont transmis à un nœud intermédiaire qui a la charge de les redistribuer à un nombre quelconque de destinataires « abonnés ».

La dernière politique de passage de messages est similaire à la modalité de communication par publication et abonnement usuellement rencontrée dans les systèmes de communication par événements tels que *COS Events*, décrit dans la section 7.2.3.

Un système réparti JMS se compose d'un ensemble de « fournisseurs » de services, qui communiquent à travers un réseau et permettent à des objets applicatifs « clients » d'envoyer des messages directement à d'autres clients (qui sont alors des « destinations »), ou bien à des entités intermédiaires jouant le rôle de « réservoirs de messages » (files et bus).

Cette architecture permet une grande flexibilité dans l'organisation du routage et de la remise des messages. Elle est illustrée par la figure 7.6.

Mise en œuvre à partir d'un intergiciel schizophrène

À partir de l'étude des architectures d'intergiciels existants pour le passage de messages synthétisée dans [45], nous proposons une architecture de MOM basée sur une infrastructure schizophrène.

Les entités qui communiquent dans une application utilisant un MOM sont les clients et les réservoirs de message du MOM. Ces entités échangent des messages, et ne doivent pas avoir à se préoccuper des détails de leur transfert. Nous les identifions donc aux objets applicatifs de PolyORB.

Les clients sont les objets applicatifs de l'utilisateur ; les réservoirs de messages se placent au même niveau d'abstraction, mais sont fournis par la mise en œuvre du MOM : ils font partie de la personnalité applicative MOMA. Nous définissons ainsi deux classes Queue et Topic, correspondant respectivement aux files de messages et aux bus à messages. Leurs primitives permettent aux clients d'envoyer des messages (opération Post) et d'en recevoir (opération Get). Les bus à messages offrent également des primitives d'abonnement (Subscribe) et de désabonnement (Unsubscribe).

Le passage de messages, interaction entre un client et un réservoir de messages (ou entre deux clients), peut donc être représenté, au sein de l'intergiciel,

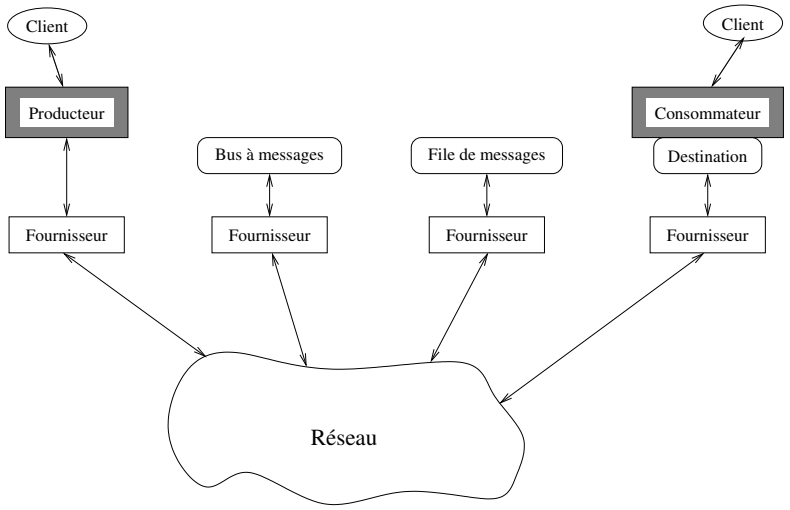


FIG. 7.6 – Architecture typique d'un système réparti utilisant le passage de messages

comme un appel de méthode de l'émetteur vers le destinataire. Nous retrouvons ainsi dans cette architecture l'isomorphisme entre les deux paradigmes que nous avons évoqué dans la section 2.2.3 : un appel de méthode sur un objet est équivalent au passage d'un message à cet objet.

Le traitement asynchrone des messages nécessite, de la part de l'intergiciel neutre, le support d'un modèle d'objets répartis généralisé, incluant les appels unidirectionnels (sans réponses). La réception des messages à l'initiative du réseau (et non du client destinataire) nécessite également que la personnalité applicative utilisée prévoie un mécanisme de rappel, permettant au client d'enregistrer un sous-programme destiné à être exécuté automatiquement lorsqu'un message se présente.

Un tel mécanisme s'intègre naturellement à l'architecture pilotée par les événements retenue pour le noyau de PolyORB. Par ailleurs, l'intégration de l'asynchronisme dans la couche neutre pour les besoins de MOMA a bénéficié aux autres personnalités : elle a permis la mise en œuvre des appels unidirectionnels CORBA et des sous-programmes asynchrones DSA.

7.4.2 Bibliothèque

Interface de programmation

L'interface offerte par la personnalité MOMA est une projection sur Ada 95 de celle de JMS.

L'interface de MOMA formalise la relation entre un client et son fournisseur. Cette relation est représentée par un objet Connection. À partir d'une Connection, le client peut accéder à des objets *producteurs* (Producers) et *consommateurs* (Consumers) de messages. Ces objets appartiennent à des groupes, les Sessions, qui permettent de contrôler de façon globale les ressources qui leur sont attribuées.

Les opérations d'émission et de réception de messages sont alors effectuées en

appelant des méthodes des producteurs et des consommateurs. Les producteurs et consommateurs concrets rendus accessibles par MOMA sont spécialisés en fonction du type d'entité auquel les messages sont envoyés : bus à messages (Topic) ou file de messages (Queue ou Destination).

Composants de personnalité minimale

La personnalité MOMA doit offrir un service de répartition à moindre coût. Elle est donc réalisée par assemblage de composants génériques de PolyORB, et ne fait intervenir qu'un minimum de composants de personnalité.

Les divers objets constitutifs de MOMA (producteurs, consommateurs, réservoirs, destinations) sont mis en œuvre comme des servants simplifiés. Ceux-ci ajoutent une interface simplifiée aux servants définis par la couche neutre, pour permettre leur intégration le plus directement possible dans l'architecture schizophrène.

L'utilisateur peut choisir d'utiliser le POA générique comme adaptateur d'objets s'il souhaite disposer des politiques flexibles de paramétrage des fonctions d'activation et d'exécution. Lorsqu'une telle souplesse n'est pas nécessaire, il peut également configurer MOMA pour utiliser l'adaptateur d'objets simplifié (SOA, *Simple Objet Adapter*). L'utilisation du SOA permet de limiter la consommation de ressources de l'intergiciel.

7.5 Synthèse — perspectives

Nous avons mis en œuvre trois personnalités applicatives sur le noyau générique PolyORB. Chacune de ces personnalités est issue d'un modèle de répartition différent :

CORBA offrant le modèle « objets répartis » ;

DSA offrant à la fois les modèles « objets répartis » et « appel de sous-programmes à distance » ;

MOMA offrant le modèle « passage de messages ».

Ces personnalités participent aux fonctions d'activation et d'exécution de l'intergiciel, en exposant à la couche neutre une encapsulation générique des objets applicatifs sous forme de servants. Elles offrent également aux objets applicatifs les points d'entrée nécessaires à l'utilisation de l'intergiciel.

Pour permettre l'adaptation entre les objets applicatifs et la couche neutre, les personnalités applicatives proposent des outils de génération de code.

Les outils actuellement développés pour les personnalités CORBA et DSA (respectivement `idlac` et GNAT) pourraient faire l'objet d'améliorations, telle que la génération de fonctions de hachage minimales parfaites pour réaliser un démultiplexage des méthodes en temps constant, possibilité évoquée et discutée dans la section 5.3.2.

Par ailleurs, il serait utile pour les utilisateurs de disposer d'un outil générant les unités nécessaires à l'élaboration et à l'initialisation d'un intergiciel complet dans une configuration donnée.

Notre mise en œuvre de l'annexe des systèmes répartis sur la couche neutre de PolyORB ne comporte pas actuellement d'outil de déploiement permettant la construction automatique des différentes partitions composant une application.

Un tel outil faciliterait la réalisation d'applications et permettrait aux développeurs de tirer efficacement parti de la configurabilité de PolyORB. GLADE offre une fonctionnalité similaire, sous forme de l'outil `gnatdist` ; il serait profitable d'adapter `gnatdist` à notre mise en œuvre de DSA, et de l'étendre aux autres personnalités de répartition.

Il serait également utile de disposer d'un outil de génération de code pour la personnalité MOMA. Actuellement, lorsque l'utilisateur souhaite échanger des données structurés, il doit construire lui-même les messages sous forme de conteneurs Any.

Cette opération est fastidieuse et potentiellement source d'erreurs. De plus, si des partitions écrites dans des langages différents doivent être amenées à communiquer *via* MOMA, cette construction doit être récrite.

On souhaiterait donc disposer d'un outil de génération de code similaire à ceux utilisés pour les systèmes d'appel de sous-programmes à distance et d'objets répartis, qui produirait des fonctions de formatage de messages à partir de descriptions abstraites de la structure des messages (par exemple des contrats IDL ou des spécifications WSDL).

La réalisation d'un tel outil pour la personnalité MOMA, en partant du compilateur `idlac` et de la partie liée à la répartition de GNAT, pourrait être une étape vers la définition d'un compilateur générique, permettant la génération du code nécessaire à la répartition pour des personnalités applicatives variées.

À l'instar du découplage entre les différents aspects de personnalité (applicatif et protocolaire) dans un intergiciel schizophrène, un tel compilateur séparerait les différentes phases de traitement nécessaires à la production de code de répartition (analyse d'une description de service réparti et génération de code d'adaptation pour une personnalité applicative donnée). Cette architecture de compilateur générique adapterait à la répartition la séparation entre frontal et dorsal, déjà largement acceptée pour la génération de code machine [3].

De nouvelles personnalités pourraient être ajoutées à celles déjà disponibles. En particulier, le projet AWS [74] définit une interface de programmation pour le développement rapide de services pour le WWW en Ada. Cette interface de programmation présente certaine similarités avec les services offerts aux objets d'une application par un intergiciel : elle permet à ces objets de s'enregistrer auprès d'un sous-système de communication pour recevoir des requêtes.

Nous pourrions donc la mettre en œuvre comme personnalité applicative. Les objets applicatifs pourraient alors être atteints non seulement à travers les protocoles usuels du WWW (HTTP et SOAP), au moyen des personnalités protocolaires correspondantes, mais également à travers d'autres protocoles (par exemple GIOP) et d'autres modèles de répartition (par exemple comme des objets CORBA).

Nous envisageons enfin l'ajout de nouveaux services à ceux existants :

- un service de dénomination résistant aux défaillances, permettant la poursuite de l'exécution de l'application en cas de dysfonctionnement du serveur de noms principal ;
- un service de gestion du cycle de vie des partitions et des unités, permettant la reprise sur terminaison d'une partition ;
- un service de consensus réparti permettant la détermination de la terminaison de l'application.

Ces services sont inspirés de fonctionnalités actuellement présentes dans GLADE, et que nous souhaiterions rendre accessibles dans des applications ré-

parties faisant intervenir plusieurs modèles de répartition.

De même, la mise en œuvre des unités Shared_Passive de DSA (offrant un service d'accès à des données partagées) pourrait être utile également pour les personnalités MOMA et CORBA.

CHAPITRE

8 Réalisation de personnalités protocolaires

Sommaire

8.1 Anatomie d'une personnalité protocolaire 185
8.1.1 Interface basse des sessions 187
8.1.2 Interface haute des sessions 187
8.1.3 Fonctions de représentation 187
8.1.4 Usines d'objets de liaison 188
8.1.5 Usines de profils 189
8.2 CORBA 189
8.2.1 Mise en œuvre de GIOP (fonction protocole) 189
8.2.2 Mise en œuvre de CDR (fonction représentation) . . 190
8.2.3 Usines de liaisons et de profils IIOP 191
8.3 SOAP 191
8.3.1 SOAP RPC (fonction protocole) 191
8.3.2 Représentation SOAP 193
8.3.3 Usine de profils SOAP 193
8.4 Synthèse — perspectives 195

La conception des personnalités protocolaires de PolyORB a été présentée dans la section 4.4.5. Dans ce chapitre, nous décrivons la structure générale de la mise en œuvre d'une personnalité protocolaire. Nous présentons ensuite deux personnalités que nous avons réalisées : GIOP et SOAP.

8.1 Anatomie d'une personnalité protocolaire

Une personnalité protocolaire est composée de deux modules principaux :
- une classe « session », qui réalise la fonction « protocole » de l'intergiciel, décrite dans la section 4.5.5 ;
- un module d'emballage et de déballage de données, qui réalise la fonction « représentation », décrite dans la section 4.5.4.

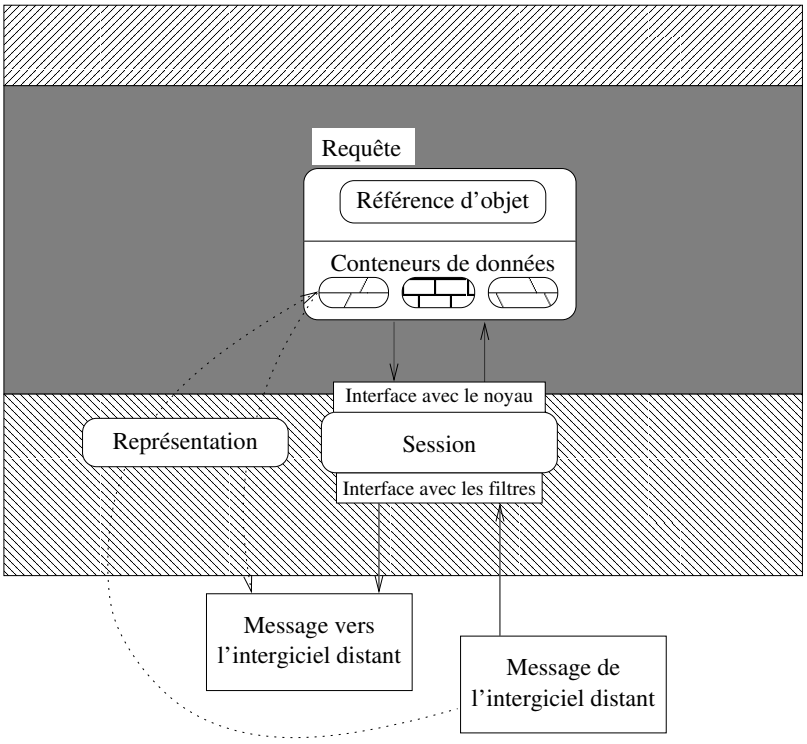


FIG. 8.1 – Anatomie d'une personnalité protocolaire

La position de ces composants dans l'architecture est illustrée par la figure 8.1.

Une session d'un protocole donné est instanciée pour chaque relation établie avec un intergiciel distant. À cet effet, chaque personnalité protocolaire définit une usine d'objets de liaison, qui instancie une session, ainsi que les filtres et points de terminaison de transport appropriés.

Pour permettre aux objets locaux de recevoir des appels, une personnalité protocolaire doit également fournir une usine de profils, utilisée par la couche neutre pour la construction de références d'objets.

Nous décrivons ici l'interface présentée par les sessions aux couches inférieures des objets de liaison, ainsi que celle qu'elles exposent au noyau générique, à travers laquelle elles prennent en charge le traitement des requêtes.

Nous présentons ensuite les interfaces des usines de liaisons et de profils.

8.1.1 Interface basse des sessions

Les sessions doivent également s'intégrer aux objets de liaisons : elles doivent être capables de traiter les messages en provenance des couches inférieures.

Chaque personnalité protocolaire doit donc fournir une mise en œuvre des opérations primitives correspondantes, qui sont appelées lorsque les événements suivants se produisent :

- indication d'établissement de connexion (côté serveur) ;
- confirmation d'établissement de connexion (côté client) ;
- indication de données reçues.

8.1.2 Interface haute des sessions

L'interface offerte par les sessions au noyau générique est identique à celle des servants, que nous avons présentée dans la section 7.1.2.

En effet, les sessions sont des subrogés d'objets distants. Vis-à-vis du noyau générique, elles doivent donc se comporter de la même façon que les objets :

côté client : elles acceptent des requêtes à traiter, et retournent des requêtes exécutées ;

côté serveur : elles créent des requêtes à exécuter, et acceptent des requêtes exécutées.

Les sessions possèdent donc deux opérations primitives, qui doivent être surchargées par chaque personnalité protocolaire :

- acceptation d'une requête à exécuter et émission du message correspondant à destination de l'intergiciel distant ;
- acceptation d'une requête après son exécution par un servant et émission de la réponse correspondante vers l'intergiciel demandeur.

8.1.3 Fonctions de représentation

Les fonctions de représentation sont utilisées par les sessions pour traduire en un flot d'octets les paramètres des requêtes, et réciproquement. Elles doivent réaliser les deux transformations que nous avons définies dans la section 4.5.4. L'interface abstraite des fonctions de représentation comporte donc les deux opérations primitives d'emballage et de déballage.

Emballage

Chaque fonction de représentation définit une procédure prenant en entrée un conteneur universel typé et un tampon destiné à recevoir les données emballées.

Cette procédure réalise la transformation

$$< D >_T \xmapsto{Representation} msg$$

en écrivant la représentation des données dans le tampon.

Déballage

La procédure réciproque de la précédente doit également être mise en œuvre. Elle prend en entrée un tampon (contenant les données à déballer), ainsi qu'un conteneur typé vide. Elle réalise la transformation :

$$(msg, < \cdot >_T) \xmapsto{Representation} < D >_T$$

en insérant une valeur dans le conteneur vide.

8.1.4 Usines d'objets de liaison

Dans la section 6.3.3, nous avons décrit le mécanisme qui permet d'instancier une pile de filtres correctement reliés à partir d'une liste d'usines de filtres.

Sur les clients, ces usines de piles de filtres sont appelées lorsque la fonction de liaison est appelée sur un profil désignant un objet distant (c'est-à-dire un profil ne correspondant à aucun point d'accès du service de transport de l'intergiciel local). Dans ce cas, la fonction de liaison fait appel à l'opération primitive Bind_Profile du profil choisi. Une version concrète de cette opération est définie pour chaque type de profil existant. Un type de profil dénote l'association entre un protocole et un mécanisme de transport. Chaque personnalité protocolaire définit donc un ou plusieurs types de profils, qui lui sont spécifiques.

L'opération Bind_Profile crée un point de terminaison du service de transport et établit une connexion. Elle crée ensuite la pile de filtres correspondante (y compris la session) et relie le point de terminaison au filtre le plus bas de la pile. Elle parcourt enfin la pile de filtres pour obtenir un pointeur sur la session, et retourne ce pointeur à l'appelant, pour servir de subrogé de l'objet distant.

Les profils concrets sont donc les usines de liaisons côté client : ils associent une usine de pile de filtres à la création d'un point de terminaison du service de transport par établissement d'une connexion vers un intergiciel distant.

Sur les serveurs, une usine de pile de filtres est associée à chaque point d'accès du service de transport. Lorsqu'une nouvelle connexion est reçue, l'intergiciel appelle la fonction Accept_Connection du point d'accès afin d'obtenir un point de terminaison. Elle fait également appel à l'usine de piles de filtres afin de créer l'objet de liaison approprié, et y relie le point de terminaison. Ce dernier est finalement ajouté à la liste des points de terminaison surveillés par l'intergiciel, afin que les événements qu'il produit soient traités.

Les structures liant chaque point d'accès du service de transport à une usine de pile de filtres constituent donc les usines d'objets de liaison côté serveur : elles associent une usine de pile filtres à un point de terminaison du service de transport créé par acceptation d'une connexion reçue d'un intergiciel distant.

8.1.5 Usines de profils

Comme nous l'avons évoqué dans la section 6.1.3, la création d'une référence pour un objet local (ou un objet distant pour lequel l'intergiciel local joue le rôle de passerelle) est traitée par le noyau générique. Celui-ci fait construire un identificateur par un adaptateur d'objets, puis transmet cet identificateur aux usines de profils associées aux points d'accès du service de transport enregistrés par les personnalités.

Chaque personnalité protocolaire doit donc fournir une usine de profils correspondant au type de profil concret qu'elle définit. L'interface des usines de profils est donnée par l'extrait de code 9

Extrait de code 9 Interface des usines de profils

```
package PolyORB.Binding_Data is

   type Profile_Factory is abstract tagged limited private;

   procedure Create_Factory
     (PF : out Profile_Factory;
      TAP : Transport.Transport_Access_Point_Access;
      ORB : Components.Component_Access)
      is abstract;
   --  Initialize PF to act as profile factory for transport
   --  access point TAP managed by ORB.

   function Create_Profile
     (PF  : access Profile_Factory;
      Oid : Objects.Object_Id)
     return Profile_Access
      is abstract;
   --  Create a profile of the type determined by PF, using
   --  Oid as the object specification.

end PolyORB.Binding_Data;
```

Une usine de profils est spécifiquement associée à une instance de point d'accès du service de transport. En effet, la construction d'un profil nécessite la connaissance de l'adresse du point d'accès considéré. Lorsqu'une personnalité protocolaire crée un point d'accès, elle est également responsable de la création d'une usine de profils appropriée. Elle enregistre le point d'accès et son usine associée auprès de la couche neutre.

8.2 CORBA

La personnalité protocolaire CORBA de PolyORB est une mise en œuvre du protocole normalisé GIOP et de sa projection sur TCP, IIOP.

Elle comprend :

- une mise en œuvre des échanges de messages du protocole GIOP lui-même ;
- une fonction de représentation conforme à la spécification CDR ;
- les usines de liaisons nécessaires à la projection de GIOP sur le transport TCP.

8.2.1 Mise en œuvre de GIOP (fonction protocole)

Le protocole GIOP normalise le format des messages nécessaires à la représentation des interactions entre objets, ainsi que les séquences d'échanges de ces messages.

La mise en œuvre de GIOP consiste en des versions concrètes des classes Session et Protocol. La classe GIOP_Protocol est une usine de GIOP_Session. Les GIOP_Session réalisent l'émission et la réception de requêtes conformément au protocole GIOP. Les versions 1.0, 1.1 et 1.2 du protocole sont réalisées ; chaque instance de GIOP_Session possède un attribut qui détermine quelle version utiliser pour cette session.

Délimitation des messages

Dans GIOP, la taille des données à recevoir de l'intergiciel distant est toujours connue à l'avance. En effet, les messages GIOP commencent par un en-tête dont la taille et le format sont fixés à l'avance.

Cet en-tête contient l'indication de la taille totale du message. Les sessions GIOP peuvent donc tirer parti du filtre Slicer, qui effectue le réassemblage des données reçues en utilisant cette information.

Entrelacement des requêtes

Au cours d'une session GIOP, un client peut émettre plusieurs requêtes en rafale, sans avoir reçu les réponses associées. Afin de permettre l'appariement entre réponses et requêtes, chaque requête en attente est identifiée par un numéro de séquence et enregistrée dans une liste.

Lors de la réception d'une réponse, la requête correspondante est déterminée en parcourant la liste des requêtes en attente, afin de décoder le résultat conformément au profil de la méthode appelée et de le retransmettre à l'objet applicatif client approprié.

8.2.2 Mise en œuvre de CDR (fonction représentation)

Notre mise en œuvre de CDR fournit des procédures permettant d'emballer et de déballer tous les types de base du modèle de types d'IDL. Ces procédures sont utilisées directement pour la construction des messages GIOP, dont le format est fixé par la norme.

L'emballage et le déballage des paramètres des appels de méthodes et des valeurs de retour est réalisé au moyen de deux procédures qui opèrent sur des conteneurs universels Any. Ces procédures permettent l'emballage de valeurs de types simples, ainsi que de tout type complexe défini par un typecode.

Le paquetage réalisant CDR introduit également la notion d'encapsulation. Il s'agit d'un tableau d'octets contenant un flot de données emballées. L'utilisation d'encapsulations permet de réaliser une fois pour toutes l'emballage de certaines informations complexes mais constantes, telles que des descripteurs de types de données (typecodes). Elle permet également le déballage partiel de structures de données : une encapsulation permet à un nœud de communiquer une information à un tiers sous forme d'un conteneur opaque, sans avoir à en décrire la structure interne. Le correspondant peut manipuler l'encapsulation sans la déballer, et la restituer ultérieurement au nœud qui l'a créée.

Les encapsulations sont utilisées en particulier pour obtenir des représentations transcriptibles des références d'objets : les IOR (*Interoperable Objet References*) de CORBA.

8.2.3 Usines de liaisons et de profils IIOP

Le protocole GIOP est indépendant d'un mécanisme de transport particulier. Il peut être utilisé avec tout support de communication offrant un service de transport fiable de flot d'octets en mode connecté. Ainsi, GIOP peut être utilisé avec le protocole de transport TCP [111]. La projection des échanges de messages GIOP sur des connexions TCP est normalisée ; elle constitue la spécification *IIOP* (*Internet Inter-ORB Protocol*).

La réalisation d'IIOP dans PolyORB consiste en la définition d'usines de liaisons permettant de construire des objets de liaison associant des connexions TCP à des sessions GIOP.

Les usines de liaisons pour les clients consistent en un type de profil concret, qui contient :
- une adresse IP et un numéro de port ;
- une identification de version d'IIOP ;
- une suite d'octets constituant un identificateur d'objet.

La primitive Bind_Profile réalise une connexion TCP à l'adresse indiquée, et relie le point de terminaison du service de transport ainsi obtenu à un filtre Slicer (réalisant la délimitation des données reçues, dont la taille est toujours connue à l'avance dans GIOP) et à un objet session GIOP. La version de GIOP utilisée par la session est déterminée par le numéro de version contenu dans le profil.

De même, pour les serveurs, un paquetage configurant un point d'accès IIOP est fourni. Celui-ci crée un point d'accès TCP, et enregistre ce point d'accès auprès du noyau générique. Lors de cet enregistrement, plusieurs informations sont associées avec le point d'accès :
- un pointeur sur une usine de pile de filtres responsable de la création des objets de liaison associés aux connexions reçues par ce point d'accès ;
- une usine de profils permettant la prise en compte de ce point d'accès pour la création de références associées à des objets locaux.

8.3 SOAP

Le protocole SOAP permet d'effectuer des appels de sous-programmes à distance sur des objets en représentant ces appels et les données associées sous formes d'entités XML.

8.3.1 SOAP RPC (fonction protocole)

La norme SOAP [12] définit :
- un format de message générique ;
- un mode d'encodage permettant de représenter des données utilisateur sous forme d'entités XML ;
- une convention de projection de l'appel de sous-programmes à distance sur des messages SOAP.

Messages SOAP

Le système d'appel de sous-programmes à distance sur SOAP (SOAP RPC, *SOAP Remote Procedure Call*) définit les composants des messages génériques

qui doivent être présents dans une requête et dans une réponse, et les moyens d'associer les unes aux autres.

La norme ne fixe pas le mécanisme utilisé pour transférer les entités XML d'un nœud à un autre. Un nœud utilisant SOAP doit réaliser ces transferts au moyen d'un protocole convenu avec ses interlocuteurs. Il s'agit généralement du protocole HTTP [27], pour lequel une projection normalisée est définie par la norme SOAP.

Les messages SOAP peuvent être utilisés pour réaliser diverses modalités de communication :

- messages unidirectionnels ;
- messages diffusés (*multicast*) ;
- messages échangés selon le canevas « requête/réponse » de l'appel de sous-programmes à distance.

L'utilisation de l'une ou de l'autre de ces modalités est indépendante du choix d'une projection de SOAP sur un protocole d'échange d'entités XML. Cependant, dans le cas de protocoles de transfert proposant une modalité de communication par requêtes et réponses, il est possible de transposer naturellement les requêtes et réponses d'un appel de sous-programme à distance au sens de SOAP vers leurs équivalents au sens du protocole de transfert.

Ainsi, dans le cas de l'utilisation de HTTP pour transporter des entités SOAP, il est possible d'effectuer un appel de sous-programme SOAP en transportant l'entité XML de la requête SOAP dans une requête HTTP, et en transmettant l'entité XML de la réponse SOAP dans la réponse HTTP associée.

Manipulation des arbres XML

Dans PolyORB, les entités XML sont manipulées sous forme d'arbres syntaxiques (et non sous forme de représentations textuelles) au moyen de la bibliothèque XML/Ada [14]. Cette bibliothèque offre plusieurs interfaces normalisées permettant la génération et le traitement de flots XML :

DOM (*Document Object Model*) définit une représentation et des fonctions de manipulation pour les arbres XML [58] ;

SAX (*Simple API for XML*) définit une interface basée sur des événements entre un analyseur syntaxique XML générique et des traitements spécifiques d'une application [64].

XML/Ada définit un composant Tree_Reader qui consomme des événements SAX en construisant un arbre DOM. Nous utilisons ce composant pour analyser la structure des entités XML reçues par PolyORB, et les projeter sur la représentation neutre des requêtes.

Cette mise en œuvre de la fonction protocole pour SOAP réutilise et étend certains composants du projet AWS (*Ada Web Server*) [74].

HTTP

Nous avons réalisé une mise en œuvre originale du protocole HTTP sous forme de filtre (élément de base des objets de liaison de PolyORB que nous avons décrit dans la section 6.3.3).

Ce filtre réalise la délimitation des messages HTTP (requêtes et réponses) et effectue l'analyse syntaxique des éléments du protocole :

– il analyse la structure des requêtes (nom de méthode HTTP, URI, paramètres) et des réponses (code de retour) ;
– il analyse les en-têtes associés aux requêtes et aux réponses ;
– il met en œuvre les mécanismes permettant le transfert du flot de données associé à chaque message HTTP.

Le décodage des noms de méthodes et des en-têtes utilise les fonctions de hachage statiques minimales parfaites que nous avons évoquées dans la section 5.3.2.

Nous avons choisi de réaliser HTTP sous forme de filtre pour permettre la réutilisation de ce protocole pour d'autres personnalités protocolaires que SOAP.

8.3.2 Représentation SOAP

Comme PolyORB, AWS définit un type conteneur universel permettant de stocker les valeurs des données de l'utilisateur associées à une requête, et offre deux fonctions de transformation permettant de passer de ces conteneurs à leur représentation (sous forme d'entités XML conformes à la DTD SOAP) et réciproquement.

Notre mise en œuvre de la représentation SOAP pour PolyORB a donc consisté en une adaptation de ces fonctions pour utiliser nos propres conteneurs universels au lieu de ceux originellement utilisés par SOAP.

Nous confirmons ainsi que le modèle de types et le formalisme de représentation des données que nous avons retenus pour la couche neutre, inspirés de CORBA, sont compatibles avec d'autres personnalités de répartition.

8.3.3 Usine de profils SOAP

Comme toute personnalité protocolaire dans PolyORB, la personnalité SOAP est associée à un type de profil. Le profil SOAP correspond à l'utilisation de ce protocole sur une connexion HTTP.

Référencement par URI

Dans SOAP, l'indication de l'objet destinataire d'un message doit être une URI [7]. Une URI est une forme normalisée et transcriptible (au sens que nous avons évoqué dans la section 8.2.2 à propos des IOR) d'identification d'une ressource de nature quelconque.

Une URI désignant un objet accessible par SOAP sur HTTP comporte un nom de machine (ou une adresse IP) et un numéro de port, puis un chemin relatif au nœud contenant l'objet, par exemple :

`http://machine.example.net:1234/x/y/z/Obj`.

Nous lui associons donc un profil associant une adresse IP et un numéro de port à une URI (extrait de code 10).

La réalisation d'une personnalité protocolaire SOAP suppose donc la fourniture de fonctions permettant :
– la traduction de références d'objets neutres (contenant un profil SOAP) en URI ;
– réciproquement, la traduction d'une URI en un profil SOAP permettant de créer une référence PolyORB.

Extrait de code 10 Profil SOAP

```
package PolyORB.Binding_Data.SOAP is

   type SOAP_Profile_Type is new Profile_Type with private;
   --  A profile that designates an object accessible through
   --  SOAP RPC over HTTP.

   ------------------------------------------------
   -- Supplementary operations of SOAP profiles --
   ------------------------------------------------

   function Get_URI_Path (Profile : SOAP_Profile_Type)
     return Types.String;
   --  Return the 'path' component of the object URI.

   function Create_Profile (URI : Types.String) return Profile_Access;
   --  Create a SOAP RPC/HTTP profile from an absolute URI.

   function To_URI (Prof : SOAP_Profile_Type) return String;
   --  Convert a SOAP profile to an equivalent URI.

   function Image (Prof : SOAP_Profile_Type) return String;
   --  Represent Prof as a string, for debugging purposes.

private

   type SOAP_Profile_Type is new Profile_Type with record
      Address  : Sockets.Sock_Addr_Type;
      URI_Path : Types.String;
   end record;

end PolyORB.Binding_Data.SOAP;
```

Ces fonctions sont réalisées comme des opérations supplémentaires spécifiques des profils SOAP : To_URI et Create_Profile.

Dans les profils SOAP, le composant « URI relative » du profil est une représentation de l'identificateur local attribué par l'intergiciel hébergeant l'objet. Sur un serveur PolyORB, il appartient donc à l'adaptateur d'objet de traduire les identificateurs qu'il attribue en URI relatives.

Cette traduction et la traduction réciproque sont réalisées au moyen de deux messages Oid_Translate et URI_Translate échangés entre le noyau générique et l'adaptateur d'objets, et documentés par l'extrait de code 11.

Dans le cas général, les identificateurs d'objets sont des suites d'octets quelconques, et la traduction par défaut consiste simplement à encoder ces suites d'octets suivant les conventions de représentation des URI (qui permettent de représenter toute valeur d'octet sous la forme `%xy`, où `x` et `y` sont des chiffres hexadécimaux).

Cas particulier de l'utilisation du POA

Dans le cas de nœuds utilisant le POA, il est possible de tirer parti de l'espace d'adressage hiérarchique offert par cet adaptateur d'objets pour produire des URI qui reflètent cette hiérarchie.

Pour ce faire, le POA surcharge le comportement par défaut de l'opération de traduction d'un identificateur d'objet en URI. Il détermine les noms des POA successifs qui constituent le chemin entre le POA racine et le POA contenant directement l'identificateur d'objet concerné, et génère une URI en utilisant la notation des chemins préconisée par la spécification de ces identificateurs.

Ainsi, un objet `01` enregistré dans le POA fils `mySubPOA` du POA racine se verra attribuer l'URI : `/mySubPOA/01` plutôt qu'une suite de chiffres hexadéci-

Extrait de code 11 Messages permettant la traduction entre identificateurs d'objets et URIs

```
package PolyORB.ORB.Interface is

   type Oid_Translate is new Components.Message with record
      Oid : Objects.Object_Id_Access;
   end record;
   --  When sent by middleware core to object adapter:
   --    Request that Oid be translated into a relative URI.
   --    The expected reply is URI_Translate.
   --  When replied by object adapter to middleware core:
   --    Returns relative URI translated to object id.

   type URI_Translate is new Components.Message with record
      Path : Types.String;
   end record;
   --  When sent by middleware core to object adapter:
   --    Request that Path (a relative URI) be translated into
   --    an object id.
   --    The expected reply is Oid_Translate.
   --  When replied by object adapter to middleware core:
   --    Returns object id translated to relative URI.

end PolyORB.ORB.Interface;
```

maux. Cette propriété permet aux applications de voir attribuer aux objets des identificateurs lisibles et compréhensibles lorsqu'elles utilisent des personnalités protocolaires manipulant des URI.

8.4 Synthèse — perspectives

Dans ce chapitre, nous avons présenté la structure générale d'une personnalité protocolaire. Chaque personnalité protocolaire fournit les fonctions :

protocole en fournissant une classe « session » ;

représentation en mettant en œuvre les fonctions d'emballage et de déballage opérant sur des conteneurs universels typés.

Les personnalités protocolaires contribuent également aux fonctions :

adressage en définissant des usines de profils ;

liaison en offrant des usines d'objets de liaison.

Nous avons décrit les personnalités protocolaires que nous avons réalisées : GIOP et SOAP.

Ces deux protocoles peuvent être projetés sur des mécanismes de transport variés. Nous avons réalisé les projections sur des connexions TCP ; il serait intéressant d'offrir de nouvelles projections. Ainsi, l'utilisation d'UDP permettrait la réalisation de liaisons utilisant la diffusion de groupe similaires à celles proposées par Jonathan. Le support de transports non connectés autoriserait également l'utilisation du courrier électronique pour transmettre des requêtes, autorisant ainsi le déploiement de systèmes d'information très faiblement couplés.

L'architecture de PolyORB permettrait l'ajout aisé de ces nouvelles projections : cette opération nécessiterait uniquement la réalisation des composants de transport et des usines d'objets de liaison appropriées ; les composants de protocole et de représentation existants pourraient être réutilisés sans modification.

De même, une personnalité protocolaire HTTP pourrait être réalisée à faible coût en réutilisant le filtre HTTP réalisé pour la personnalité SOAP. Dans une telle personnalité, chaque requête HTTP serait directement transmise, sous forme d'appel de méthode, à un composant applicatif. Tout objet réparti pourrait ainsi être accessible comme un serveur WWW ordinaire.

CHAPITRE

9

Validation et mesures

Sommaire

9.1 Fonctionnalités d'intergiciel 198
9.1.1 CORBA 198
9.1.2 DSA 200
9.1.3 MOMA 204
9.2 Contribution à l'interopérabilité 205
9.2.1 Interopérabilité entre mises en œuvre des objets répartis 205
9.2.2 Interopérabilité avec un intergiciel existant 207
9.2.3 Interopérabilité entre modèles de répartition 208
9.3 Évaluation des performances 209
9.3.1 Temps d'exécution 209
9.3.2 Empreinte mémoire 211
9.3.3 Leçons tirées des observations 212
9.4 Génie logiciel et coût de personnalisation 212
9.4.1 Répartition du code 212
9.4.2 Analyse sur des configurations complètes 214
9.5 Synthèse 214

Les tests et les mesures que nous avons conduits sur notre prototype mettent en évidence les fonctionnalités réalisées et notre contribution à l'interopérabilité entre applications réparties.

Notre architecture permet de construire des intergiciels réalisant les modèles de répartition « objets répartis », « appel de sous-programmes à distance » et « passage de messages ». Nous vérifions que notre mise en œuvre de ces modèles de répartitions est opérationnelle au moyen d'un ensemble de tests. Nous en présentons les résultats dans la section 9.1.

L'application témoin que nous avons présentée dans la section 4.2 fait intervenir plusieurs nœuds. Lorsque ceux-ci sont dotés d'intergiciels différents, elle nécessite la réalisation de l'interopérabilité entre ceux-ci. Nous montrons que PolyORB permet de répondre à ces besoins de façon efficace. Nous présentons

également une démonstration des mécanismes d'interopérabilité entre modèles de répartition, dans une application faisant interagir un objet réparti avec un système de passage de messages.

Nous évaluons les performances de PolyORB, et nous les comparons avec celles d'intergiciels existants. Nous discutons les résultats de ces mesures, et nous proposons des pistes qui permettront encore de les améliorer.

Nous analysons enfin la répartition du code de PolyORB entre les couches neutre, applicative et protocolaire. Nous montrons ainsi l'intérêt de concevoir l'intergiciel autour d'une couche neutre, et nous évaluons le coût marginal de la personnalisation.

9.1 Fonctionnalités d'intergiciel

L'architecture schizophrène de PolyORB répond aux besoins que nous avons définis. Elle permet de réaliser les fonctionnalités d'intergiciel pour des modèles de répartition et des personnalités différents. Nous illustrons ces fonctionnalités sur trois configurations :

- CORBA (objets répartis) ;
- DSA (objets répartis et appel de sous-programmes à distance) ;
- MOMA (passage de messages).

9.1.1 CORBA

Tests de transport de données

Nous vérifions la mise en œuvre des spécifications CORBA (personnalité applicative et protocole GIOP) en réalisant des appels de méthodes comportant des transferts de données des divers types prévus par le modèle de types d'OMG IDL.

Test	Statut	Test	Statut
test not null	PASS	test struct	PASS
test unsigned_long	PASS	test nested struct	PASS
test string	PASS	test self reference	PASS
test boolean	PASS	test object	PASS
test short	PASS	test object typedef	PASS
test long	PASS	test union	PASS
test unsigned_short	PASS	test union w/ enum switch	PASS
test float	PASS	test simple array	PASS
test double	PASS	test matrix	PASS
test char	PASS	test attribute	PASS
test octet	PASS	test read-only attribute	PASS
test enum	PASS	test user exception	PASS
test unbounded sequence	PASS	test unknown exception	PASS
test bounded sequence	PASS	test fixed point	FAIL

Tab. 9.1 – Tests fonctionnels CORBA : types de données

Le tableau 9.1 synthétise les résultats obtenus pour chaque type de données ;

un test est considéré comme satisfait (PASS) pour un type de données lorsqu'un appel de méthode prenant une donnée de ce type et retournant la même valeur a pu être effectué, et que la valeur retournée au client est bien identique à celle passée initialement.

Ces tests ont été réalisés en utilisant le protocole GIOP. Le test concernant les nombres décimaux à virgule fixe a délibérément été omis, car il nécessite le développement de routines de représentation complexes, sans apporter une contribution fondamentale à l'architecture que nous souhaitions mettre en œuvre.

Tests de modalités d'appel

Nous avons également vérifié le fonctionnement des divers modes de passage des paramètres, ainsi que les appels de méthode unidirectionnels (permettant une communication asynchrone entre objets).

Test	**Statut**
test void procedure	PASS
test in param procedure	PASS
test out param procedure	PASS
test in out param procedure	PASS
test in and out param procedure	PASS
test in and inout param procedure	PASS
test inout and out param procedure	PASS
test in and out and inout param procedure	PASS
test void function	PASS
test in param function	PASS
test out param function	PASS
test inout param function	PASS
test in and out param function	PASS
test in and inout param function	PASS
test out and inout param function	PASS
test in and out and inout param function	PASS
test void one way procedure	PASS
test in param one way procedure	PASS

Tab. 9.2 – Tests fonctionnels CORBA : modes de passage de paramètres et appel unidirectionnel

Le tableau 9.2 synthétise les résultats obtenus. Un test est considéré comme satisfait lorsqu'un appel de méthode comportant des paramètres des catégories indiquées a pu être réalisé avec succès. En outre, le test sur les appels unidirectionnels vérifie la création de parallélisme induite par cette construction (le client poursuit son exécution pendant que le serveur traite la requête).

9.1.2 DSA

Tests normalisés pour DSA

La seconde personnalité applicative réalisée est l'annexe des systèmes répartis d'Ada 95. L'évaluation de la conformité d'un processeur du langage Ada est encadrée par un processus normalisé par l'ISO [53] ; une suite de tests de référence est définie et tenue à jour par une autorité indépendante, l'ACAA (*Ada Confirmity Assessment Authority*). Nous vérifions notre mise en œuvre de DSA au moyen de cette suite de tests (ACATS, *Ada Conformity Assessment Test Suite*). Le tableau 9.3 récapitule la liste des tests relatifs à l'annexe E (DSA) que nous avons considérés.

Test	Description	Statut
CXE1001	'Partition_Id	PASS
CXE2001	Shared passive	PASS
CXE2002	sous-programmes de bibliothèque RCI	non implémenté dans GLADE
CXE4001	Exceptions	PASS
CXE4002	Passage de types de taille connue	PASS
CXE4003	Répartition et parallélisme	implémenté (sauf abort)
CXE4004	Passage de types de taille inconnue	convertisseurs non générés
CXE4005	RACW et types étiquetés	convertisseurs non générés
CXE4006	RCI et types étiquetés	convertisseurs non générés

Tab. 9.3 – Tests fonctionnels DSA : ACATS

Notre mise en œuvre de DSA est, comme GLADE, basée sur le compilateur GNAT. En Ada 95, les indications de répartition sont données au moyen de directives de compilation affectant une catégorie à des unités de compilation : Remote_Call_Interface pour les unités déclarant des sous-programmes appelables à distance, Remote_Types pour celles définissant des types de données transportables ou des objets répartis.

Les tests suivants sont satisfaits par notre version de DSA :

CXE1001 : nous fournissons le service d'identification des partitions prévu par la norme. Notre mise en œuvre de DSA utilise un service de dénomination spécifique : elle ne nécessite pas la détermination des identificateurs préalablement à l'utilisation d'appels de sous-programmes à distance. Nous avons donc pu réaliser ce service comme une unité Remote_Call_Interface. Le test CXE1001 vérifie la validité des identificateurs de partitions attribués (en particulier leur unicité). Dans notre mise en œuvre, il démontre aussi que nous fournissons le mécanisme d'appel de sous-programme à distance.

CXE2001 : les unités Shared_Passive sont actuellement réalisées au moyen d'un système de fichiers partagé sous-jacent. D'autres mises en œuvre pourraient être proposées, par exemple en utilisant l'algorithme de Li et Hudak [59].

CXE4001 : les exceptions levées par un sous-programme sont transmises correctement entre partitions, y compris lorsqu'une occurrence transite par

une partition n'ayant pas visibilité sur la déclaration de l'exception. Ce test met en évidence le fait que la couche neutre de PolyORB réalise un modèle de répartition plus général que celui de CORBA. En effet, dans ce dernier, toutes les exceptions potentiellement levées par une méthode doivent être connues à l'avance.

CXE4002 : ce test est similaire au test de transport de données de CORBA, décrit dans la section 9.1.1. Il consiste en la réalisation d'appels de sous-programmes à distance faisant intervenir des paramètres de taille fixe.

CXE4003 : La norme Ada 95 permet à un traitement en cours d'être avorté, par exemple à la suite de l'expiration d'un délai. Lorsque ce traitement comporte un appel de sous-programme à distance, un message doit être transmis au serveur afin que celui-ci interrompe l'exécution du servant.

Cette possibilité n'est pas actuellement réalisée dans la couche neutre. Cependant, le support nécessaire dans la couche protocolaire est réalisé, et l'architecture de la couche neutre permettra aisément l'intégration de cette possibilité.

Les tests faisant intervenir des types de données de tailles variables, en revanche, ne sont pas actuellement satisfaits :

CXE4004 : tableaux de tailles variables et enregistrements à variantes ;

CXE4005, CXE4006 : types étiquetés.

Les fonctions de conversion entre représentation native des données et conteneurs Any neutres pour ces types de données ne sont pas actuellement générées. Cependant, l'ajout de ces fonctions de conversion ne pose pas de difficulté théorique particulière. En effet, nous disposons d'ores et déjà d'une projection du modèle de types complet d'Ada 95 vers le modèle de données IDL, sur lequel est basé celui de PolyORB (il s'agit d'un résultat du projet CIAO [81]). La satisfaction complète des tests CXE4004, CXE4005 et CXE4006 ne nécessite donc que l'ajout des cas manquants dans le générateur de code de répartition, et ne remet pas en cause l'architecture de l'exécutif.

Chacun des tests ACATS comporte un grand nombre de vérifications. Ainsi, les seuls tests CXE4002 et CXE4004 vérifient le transport correct de l'ensemble des types de données. Ils ont donc à eux seuls une portée équivalente à l'ensemble des tests de transport présentés pour CORBA dans le tableau 9.1 de la section 9.1.1. Le spectre des fonctionnalités vérifiées par chacun de ces tests est donc fort étendu, puisqu'il détermine, par définition, la conformité d'un compilateur à la norme. La satisfaction de ces tests est donc une garantie importante sur la validité de la mise en œuvre de DSA.

Les tests concernant l'utilisation de l'unité System.RPC ont délibérément été omis. En effet, ces tests visent à vérifier l'utilisation d'une interface normalisée entre le code généré par le compilateur et l'exécutif de répartition. Cette contrainte normative avait pour objectif de permettre le remplacement de l'exécutif de répartition indépendamment du compilateur.

L'expérience de GLADE a montré que l'interface fixée par la norme est trop restrictive, et que la réalisation des fonctionnalités de l'annexe nécessite des extensions (autorisées) à cette interface [116].

La normalisation n'a donc pas atteint son objectif, et la suppression de cette règle a été proposée par Laurent Pautet pour une future révision du langage. Cet amendement a été adopté en juillet 2002 [2].

Application témoin

Dans la section 4.2, nous avons donné un exemple d'application basée sur les objets répartis : un ensemble de correspondants dialoguant par publication de messages déposés sur un tableau d'affichage commun et par envoi de messages directs d'un correspondant à un autre.

Nous utilisons notre mise en œuvre de l'annexe pour réaliser cette application. Dans un premier temps, nous vérifions que la configuration homogène, faisant intervenir une seule personnalité applicative et une seule personnalité protocolaire (section 4.2.1) est opérationnelle.

Nous avons testé cette configuration successivement avec la personnalité DSA et les deux personnalités protocolaires réalisées : SOAP et GIOP. Chacune de ces combinaisons a permis de faire fonctionner l'application. Il est à noter que le passage d'un protocole à l'autre est une simple opération de reconfiguration : il ne nécessite pas de recompilation.

Nous avons évalué les performances de la version GIOP par rapport à celles de la mise en œuvre existante de l'annexe, GLADE (que nous avons décrite dans la section 3.2.1).

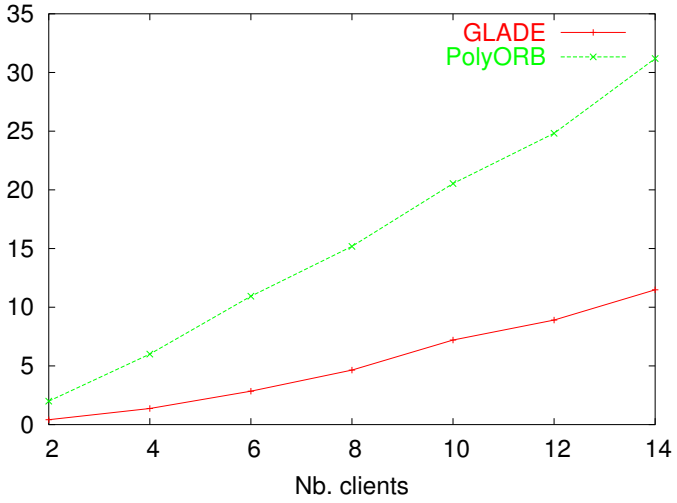
Pour cela, nous exécutons simultanément des clients réalisant 100 envois de messages directs vers chacun des autres clients (et vers lui-même), et nous mesurons le temps mis par chacun pour émettre et recevoir ces 100 messages. Nous effectuons cette mesure pour un nombre de client variant de 2 à 14. Le nombre de messages émis et reçus par chaque client varie donc de 200 à 2 800, et le nombre total de messages transmis varie de 200 à 19 600.

La figure 9.1 résume ces mesures. Nous observons, pour les deux mises en œuvre, que le temps moyen d'exécution augmente linéairement avec le nombre de clients. En effet, le réseau local utilisé pour ce test n'est pas saturé par les messages échangés. Les échanges entre des couples de nœuds distincts peuvent donc être considérés comme indépendants et exécutés en parallèle : le temps d'exécution est proportionnel au nombre de messages échangés par un nœud.

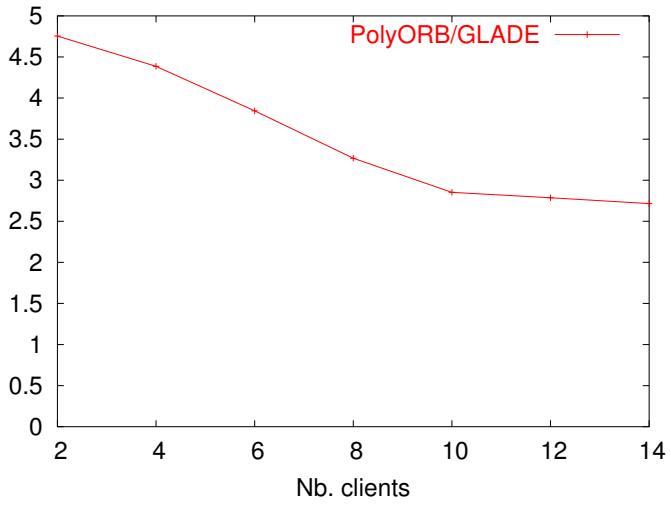
Nous concluons que PolyORB n'introduit pas, par rapport à GLADE, de surcoût d'échelle. Le rapport des temps entre les deux versions s'établit, avec un nombre suffisant de clients, à environ 2,7. Cet écart de performances s'explique notamment par l'optimisation avancée de GLADE, qui utilise une représentation des données et un protocole définis spécialement. Le compilateur produit directement du code pour générer et traiter les messages de ce protocole dans les souches et les squelettes. Au contraire, les partitions utilisant PolyORB supportent le coût des conversions de données entre la représentation applicative et la représentation neutre sous forme d'Anys.

Pour un faible nombre de clients, nous constatons un surcoût initial de PolyORB. Nous proposons plusieurs explications de ce surcoût :

- l'initialisation d'une application DSA utilisant PolyORB fait intervenir un service de dénomination basé sur le *COS Naming* de CORBA pour localiser les unités RCI. Ce service est plus complexe et plus coûteux que le mécanisme spécialisé existant dans GLADE. En effet, *COS Naming* permet la définition d'un espace de noms structuré, découpé en multiples contextes contrôlés par des nœuds différents, tandis que le serveur d'initialisation de GLADE ne gère qu'un seul espace plat associant les noms d'unités aux partitions qui les contiennent.
- Dans PolyORB, chaque enregistrement d'unité nécessite un appel de sous-



(a) Temps d'exécution



(b) Rapport $\frac{\text{PolyORB}}{\text{GLADE}}$

FIG. 9.1 – Performances de GLADE et de PolyORB dans l'application BBS

programme distant complet, alors que dans GLADE les fonctions d'initialisation sont assurées directement au moyen de primitives spécialisées du protocole.

Nous disposons d'ores et déjà de solutions pour certains aspects de ce surcoût. Ainsi, la définition d'un service spécialisé de gestion du cycle de vie des unités optimisera les performances de la phase d'initialisation des applications. En outre, ce service sera accessible depuis tous les modèles de répartition, et constituera ainsi un apport pour l'ensemble des personnalités.

Les optimisations réalisées sur GLADE depuis plusieurs années lui permettent aujourd'hui de dépasser les performances des intergiciels CORBA pour Ada disponibles dans l'industrie. De même, la phase d'industrialisation du prototype PolyORB permettra d'en rendre les performances compétitives.

9.1.3 MOMA

La personnalité applicative MOMA pour le passage de messages est inspirée de l'interface de programmation JMS. Elle est présentée en détails notamment dans [45]. Les composants applicatifs constituent des « clients », qui utilisent un service de passage de messages mis en œuvre par les « fournisseurs », composants de la personnalité applicative MOMA.

Nous avons défini deux tests pour cette personnalité, qui correspondent à deux modalités d'échange de messages :

- la communication point à point ;
- la communication par publication et abonnement.

Point à point

Le premier test (figure 9.2) fait intervenir un réservoir de messages de type « file de messages ». Celui-ci offre aux clients une communication point à point. Une fois l'application démarrée, un client envoie un message à destination de la file (1). Celle-ci le stocke jusqu'à ce qu'un second client le consomme (2).

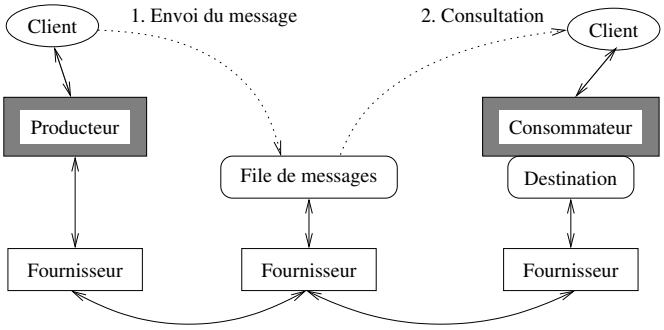


FIG. 9.2 – Envoi de message MOMA point à point

Publication et abonnement

Le test de publication/abonnement fait intervenir le second type de réservoir de messages proposé par MOMA : les « bus à messages » (figure 9.3).

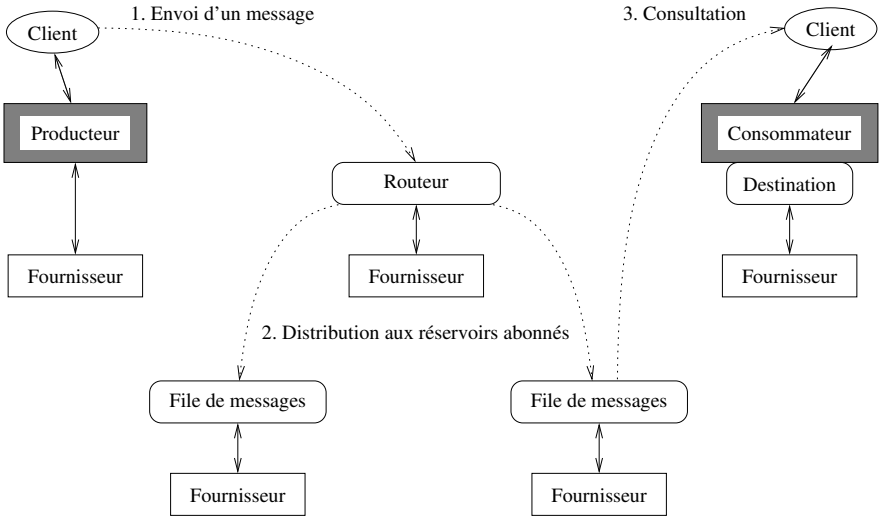


FIG. 9.3 – Envoi de message MOMA par publication et abonnement

Les bus à messages sont réalisés au moyen de composants *routeurs*, qui redistribuent les messages qu'ils reçoivent à un ensemble de réservoirs abonnés. Dans cet exemple, un client envoie un message au routeur (1). Celui-ci le redistribue aux files de messages qui lui ont été connectées (abonnées) (2). Un second client consulte finalement le message (3).

9.2 Contribution à l'interopérabilité

Dans cette section, nous présentons des situations dans lesquelles on souhaite rendre interopérables plusieurs intergiciels ou plusieurs modèles de répartition. Nous montrons que l'architecture schizophrène répond aux besoins exprimés dans ces cas d'application.

9.2.1 Interopérabilité entre mises en œuvre des objets répartis

Nous mettons en évidence l'apport de PolyORB pour rendre interopérables des applications basées sur les objets répartis en prenant à nouveau pour exemple l'application témoin de la section 4.2.

Nous avons décrit précédemment plusieurs situations où cette application doit être déployée en faisant varier le choix des personnalités applicatives et protocolaires présentes sur chaque nœud. Chacune de ces situations pose un problème d'interopérabilité lorsque des intergiciels traditionnels sont utilisés.

Dans chaque cas, nous présentons une configuration opérationnelle de PolyORB qui répond au besoin exposé. Ces configurations constituent le jeu de tests validant la contribution de PolyORB à l'interopérabilité.

Interopérabilité avec protocole unique

Nous souhaitons montrer que notre architecture permet l'interopérabilité entre différentes personnalités applicatives réalisant le modèle « objets répartis ». Pour cela, nous intégrons à l'application témoin des nœuds *Correspondant* utilisant la personnalité applicative CORBA, mais utilisant la même personnalité protocolaire que les nœuds DSA. Cette configuration correspond au cas d'interopérabilité avec protocole unique que nous avons présenté dans la section 4.2.2.

Nous vérifions que l'utilisation d'un protocole commun permet aux objets des deux personnalités applicatives d'échanger des appels de méthodes. Cette interopérabilité symétrique est, à elle seule, un résultat nouveau par rapport à l'interopérabilité unidirectionnelle offerte par la passerelle CIAO, que nous avons décrite dans la section 3.3.3.

Nous montrons également que les unités offrant des sous-programmes appelables à distance (Remote_Call_Interface) sont accessibles aux clients CORBA. À chacune de ces unités correspond un objet dont les méthodes sont les sous-programmes qu'elle offre.

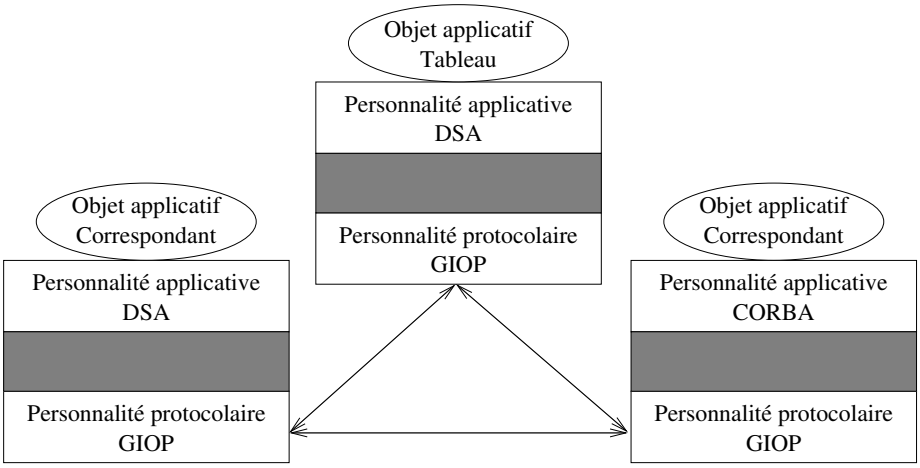


FIG. 9.4 – Interopérabilité entre personnalités applicatives

La figure 9.4 illustre cette première configuration d'interopérabilité. Les correspondants CORBA et DSA diffusent des messages en les envoyant au serveur. Deux correspondant peuvent également dialoguer par appel direct de méthodes (sans passer par le tableau d'affichage public).

Interopérabilité avec protocoles multiples

Dans la configuration précédente, l'ensemble des nœuds utilisent le protocole GIOP pour communiquer. Nous déployons maintenant l'application avec un serveur possédant à la fois les personnalités protocolaires GIOP et SOAP.

Cette configuration est utile notamment lorsque les correspondants DSA utilisent SOAP, alors qu'un correspondant CORBA basé sur un intergiciel tiers utilise GIOP. Cette situation correspond au cas proposé dans la section 4.2.3.

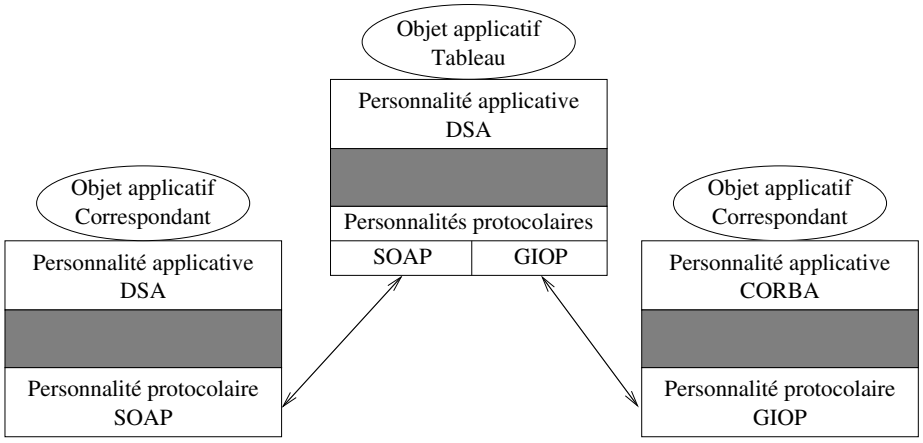


FIG. 9.5 – Interopérabilité entre personnalités applicatives

La figure 9.5 illustre cette configuration. L'utilisation de PolyORB sur le serveur permet à celui-ci de dialoguer indifféremment avec des clients SOAP et CORBA. Chacun de ces clients publie des messages, que le serveur propage à tous. En revanche, les deux correspondants ne disposent d'aucun protocole commun. Ils ne peuvent donc pas échanger d'appels directs.

Interopérabilité indirecte par passerelle dynamique

L'interaction entre des correspondants sans protocole commun est rendue possible grâce à la création d'une passerelle. Dans PolyORB, cette passerelle est un composant d'un objet de liaison qui connecte, de façon dynamique et transparente, deux personnalités protocolaires. Le principe d'une telle passerelle est décrit dans la section 4.2.4.

Dans l'application témoin, la présence des deux personnalités protocolaires SOAP et GIOP sur la partition « tableau d'affichage » lui permet de jouer le rôle de relais entre correspondants SOAP et correspondants GIOP.

La figure 9.6 illustre la situation d'interopérabilité par passerelle dynamique. Dans cette configuration, lorsque le correspondant SOAP demande au serveur la référence désignant le correspondant GIOP, le serveur crée dynamiquement une référence de passerelle. Cette référence correspond au point d'accès SOAP du serveur, et son identificateur d'objet contient suffisamment d'informations pour reconstituer la référence complète du correspondant GIOP.

Ainsi, lors d'un appel de méthode du correspondant SOAP au correspondant GIOP, le serveur reçoit l'appel sous forme SOAP, effectue une recherche dans le référentiel d'interfaces pour déterminer le profil de la méthode appelée, et retransmet l'appel au correspondant GIOP.

9.2.2 Interopérabilité avec un intergiciel existant

Dans cette section, nous proposons des tests faisant intervenir un intergiciel CORBA existant : omniORB. Cet intergiciel pour C++ a été développé originel-

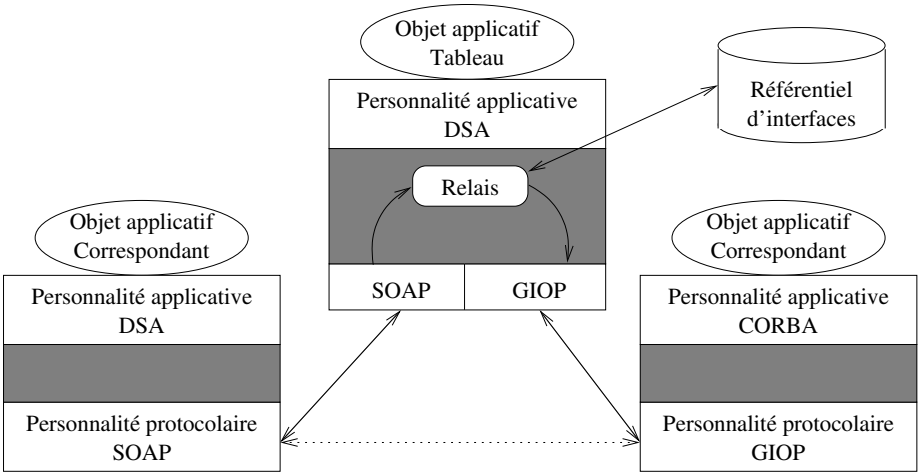


FIG. 9.6 – Interopérabilité par passerelle dynamique

lement par les laboratoires d'AT&T [40]. omniORB est particulièrement adapté pour servir de référence à nos tests :

- il a reçu la certification de l'OMG pour la conformité à la spécification CORBA 2.1 ;
- il est reconnu par la communauté comme l'une des mises en œuvre les plus performantes de CORBA.

Nous utilisons omniORB pour tester la compatibilité de notre mise en œuvre de GIOP avec un intergiciel tiers. Nous avons pu vérifier que l'ensemble des tests figurant dans le tableau 9.1 sont également satisfaits par un client PolyORB communiquant avec un serveur omniORB, et réciproquement par un client omniORB et un serveur PolyORB.

9.2.3 Interopérabilité entre modèles de répartition

Nous mettons en évidence l'interopérabilité entre modèles de répartition en faisant interagir un client utilisant la personnalité applicative MOMA avec un serveur CORBA.

L'interaction entre client MOM et objet réparti est fondée sur l'assimilation entre un envoi de message à un objet et l'appel d'une méthode de cet objet. Le message envoyé par le client MOM est formaté de façon à contenir les informations nécessaires à la réalisation d'un appel de méthode : il contient l'identité de la méthode à exécuter et les paramètres correspondants.

L'objet reçoit le message sous la même forme que n'importe quel appel qui proviendrait d'un autre objet réparti. Ayant traité la requête, il retourne une réponse au système de passage de messages.

Le scénario que nous présentons figure 9.7 fait intervenir un réservoir de messages, qui fait partie de la personnalité MOMA. Le client MOMA envoie un message à l'objet réparti au moyen d'un composant producteur, responsable de l'encodage de la requête correspondante. Le producteur transmet cette requête à l'objet cible. Si celui-ci retourne une réponse, elle est placée dans un réservoir

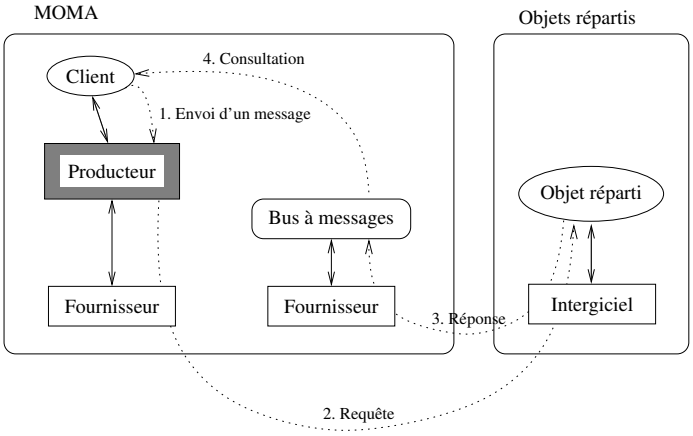


FIG. 9.7 – Interopérabilité entre modèles de répartition

de messages afin que le client puisse en prendre connaissance.

L'objet réparti est donc intégré à l'architecture de passage de messages au même titre que n'importe quel client MOM : notre architecture fournit les mécanismes de base nécessaires à l'interaction entre les modèles de répartition « objets répartis » et « passage de messages ».

9.3 Évaluation des performances

Dans cette section, nous proposons des mesures de performances de quelques aspects de PolyORB, afin d'évaluer en particulier le coût associé à l'architecture schizophrène. Pour cela, nous effectuons un ensemble d'appels de méthodes d'un objet par un client, en utilisant successivement omniORB puis PolyORB dans chacun des rôles client et serveur. Nous avons retenu omniORB comme intergiciel CORBA de référence car la communauté le reconnaît comme une mise en œuvre très performante de CORBA.

9.3.1 Temps d'exécution

Procédure de mesure

La première série de mesures correspond au temps mis par l'intergiciel pour traiter entièrement 10 000 appels de méthode. Le client et le serveur se trouvent sur le même calculateur ; l'appel effectué prend un entier long en paramètre et retourne la même valeur, sans aucun traitement.

Ce test est exécuté en utilisant ;

Côté client :

- un client PolyORB sans tâches ;
- un client omniORB.

Le client omniORB utilise l'interface d'invocation dynamique de CORBA. Cette comparaison nous paraît la plus pertinente, car PolyORB utilise des représentations comparables, du fait de son architecture. La comparaison

avec un client statique (utilisant du code généré par un compilateur IDL pour réaliser les appels de méthodes) est donc moins significative.

Côté serveur :

- PolyORB sans tâches ;
- PolyORB avec le profil de parallélisme restreint Ravenscar ;
- PolyORB avec le profil de parallélisme complet d'Ada 95 ;
- omniORB en utilisant l'interface de squelettes dynamiques (DSI).

Résultats et interprétation

Le tableau 9.4 résume les résultats des mesures de temps d'exécution.

Client \ Serveur	omniORB (DSI)	Jonathan	PolyORB monotâche	PolyORB parallèle
omniORB (DII)	1,16	3,71	2,69	6,61
Jonathan (statique)	4,17	7,09	5,91	10,50
PolyORB monotâche	2,30	5,14	3,63	7,83

TAB. 9.4 – Temps d'exécution de 10 000 appels de méthodes

Dans toutes les mesures effectuées, les performances de PolyORB sont supérieures à celles d'un serveur équivalent utilisant l'intergiciel générique Jonathan, bien que celui-ci utilise des souches et squelettes statiques : elles sont d'ores et déjà comparables à celles d'intergiciels industrialisés et déployés en production.

Les performances du client PolyORB demeurent proches de celles du client omniORB pour tous les tests face à un serveur PolyORB. Toutefois, nous observons que, face aux serveurs les plus performants (omniORB et PolyORB sans tâches), le client PolyORB est distancé de façon significative par le client omniORB. Nous expliquons cette disparité par le fait que le serveur PolyORB constitue le facteur limitant : le client PolyORB sature sa capacité de traitement, et le passage au client omniORB n'apporte pas de gain de temps d'exécution significatif. En revanche, le client PolyORB ne sature pas la capacité de traitement d'un serveur omniORB : le client omniORB en tire meilleur parti.

Les serveurs PolyORB avec tâches ont actuellement des performances inférieures à celles d'omniORB. Plusieurs facteurs contribuent à cette différence :

- Le POA, adaptateur d'objets évolué de PolyORB, n'est pas optimisé. Ses performances pourraient être améliorées, bénéficiant à toutes les configurations serveur présentées. Par ailleurs, pour des applications ne justifiant pas l'usage d'un POA complet, l'utilisateur pourrait utiliser l'adaptateur d'objets simplifié, moins riche fonctionnellement, mais plus efficace.
- Dans les configurations avec parallélisme, les tâches inoccupées de l'intergiciel sont fréquemment réveillées pour regagner aussitôt l'état inoccupé. Une gestion plus fine de la file des travaux à effectuer permettrait de limiter les synchronisations superflues.
- Lorsque le profil Ravenscar est utilisé, nous avons relevé des performances légèrement inférieures par rapport au profil parallèle général. Cette dif-

férence est due au fait que, dans le cas de Ravenscar, la bibliothèque de l'intergiciel doit gérer les files d'attente de tâches bloquées, alors que dans le profil avec parallélisme général, cette activité est prise en charge par l'exécutif Ada 95 et le système d'exploitation.

La différence notable de performances entre les profils avec et sans tâches montre qu'il est effectivement intéressant d'autoriser l'utilisateur à agir sur cet aspect de configuration.

Comme le POA, la bibliothèque de parallélisme de PolyORB a fait l'objet de développements successifs au cours de stages de DEA et de mémoires de fin d'études à l'ENST. Des revues d'architecture et de réalisation ont déjà permis d'améliorer substantiellement leurs performances. Des ajustements sur la granularité des objets de synchronisation utilisés par l'ordonnanceur-orienteur ont également apporté un gain significatif. D'autres améliorations seront apportées au cours de l'industrialisation de PolyORB, notamment en ce qui concerne le profil Ravenscar.

9.3.2 Empreinte mémoire

Procédure de mesure

Nous nous intéressons maintenant à l'encombrement en mémoire dû à l'intergiciel. Nous reprenons le scénario de test défini dans la section 9.3.1, et nous mesurons l'espace total en mémoire virtuelle occupé par le serveur au moment de son initialisation. Nous mesurons à nouveau cette empreinte mémoire après 10 000 et 20 000 appels de méthodes, afin de vérifier que notre prototype est exempt de fuites de mémoire.

Résultats et interprétation

Le tableau 9.5 résume les mesures d'encombrement mémoire que nous avons effectuées.

Mesure \ Serveur	omniORB	PolyORB sans tâches	PolyORB Ravenscar	PolyORB complet
Initiale	5 428	4 720	8 524	5 660
Après 10 000 RPC	5 576	4 720	8 524	5 660
Après 20 000 RPC	5 576	4 720	8 524	5 660

Tab. 9.5 – Encombrement mémoire d'un serveur

L'encombrement en mémoire que nous avons mesuré est plus important pour PolyORB que pour omniORB. Nous observons en particulier que la version Ravenscar est presque deux fois plus volumineuse que la version sans tâches : dans ce profil, les tâches, leurs structures de contrôle et de synchronisation doivent être allouées statiquement dès l'initialisation du programme.

Des mesures plus détaillées et un profilage de la consommation de ressources permettront de réduire ces différences. Par ailleurs, la configurabilité de l'inter-

giciel permet d'éliminer certaines fonctionnalités, pour les cas où celles-ci sont superflues au regard des besoins d'une application particulière. Le gain en mémoire lié à la configurabilité est mis en évidence par le faible encombrement de la version sans tâches.

9.3.3 Leçons tirées des observations

Les performances de PolyORB sont actuellement inférieures à celles des intergiciels classiques auxquels nous l'avons comparé : GLADE et omniORB. L'écart mesuré est surtout important côté serveur, notamment lorsque l'intergiciel est configuré pour utiliser plusieurs tâches.

Cet écart demeure néanmoins acceptable, surtout dans la mesure où de nombreuses optimisations pourraient être envisagées dans la bibliothèque de parallélisme, ainsi que dans l'adaptateur d'objets, afin de les rendre plus efficaces. Par ailleurs, les fonctionnalités apportées (interopérabilité entre intergiciels et entre modèles de répartition) représentent un avantage par rapport à l'ensemble des intergiciels existants.

L'encombrement mémoire de PolyORB avec parallélisme complet est proche de celui d'omniORB ; l'utilisation du profil sans tâches permet d'obtenir un intergiciel plus compact. Des fuites de mémoire résiduelles sont encore présentes dans le prototype. Toutefois, l'analyse de leur origine a été entamée, avec des résultats probants. La poursuite de ces travaux permettra de supprimer l'ensemble de ces fuites.

Grâce aux mesures effectuées, nous disposons d'un plan d'action pour optimiser les performances de PolyORB. Plusieurs entreprises ont manifesté leur intérêt pour notre prototype. Ces optimisations seront donc réalisées à l'occasion de la phase d'industrialisation du produit.

9.4 Génie logiciel et coût de personnalisation

Dans cette section, nous proposons une évaluation de la répartition du code de PolyORB entre la couche neutre et les personnalités. Cette répartition donne une indication du coût marginal lié à l'ajout d'une nouvelle personnalité applicative ou protocolaire. Nous comparons cette répartition avec celle rencontrée dans Jonathan, intergiciel générique dont les sources sont librement disponibles. Nous estimons les volumes de code au moyen de l'outil SLOCCount de David A. Wheeler [119].

Nous étudions ensuite quelques exemples concrets d'applications utilisant PolyORB, qui mettent en évidence l'importance de la factorisation de code au profit de la couche neutre.

9.4.1 Répartition du code

Répartition dans PolyORB

Le tableau 9.6 présente la répartition du code de PolyORB entre la couche neutre et les personnalités applicatives et protocolaires.

Nous observons que plus de 57 % du code de PolyORB est situé dans la couche neutre : celle-ci contient des fonctionnalités concrètes de l'intergiciel, qui sont effectivement mutualisées entre les différentes personnalités. Ce partage de

SLOC	Partie	SLOC-by-Language (Sorted)	
24458	couche neutre	ada=24321	
4243	p. a. MOMA	ada=4239	Personnalités applicatives
3802	p. a. CORBA	ada=3819	
1117	p. a. DSA	ada=1034	
3148	p. p. GIOP	ada=3102	Personnalités protocolaires
4138	p. p. SOAP	ada=4027	
42747	TOTAL		

Tab. 9.6 – Répartition du code dans PolyORB

code important contribue à réduire le coût de développement induit par l'ajout d'une personnalité, ainsi que l'impact de cet ajout sur la taille de l'intergiciel.

Répartition dans Jonathan

Jonathan, décrit dans la section 3.1.1, est un intergiciel en Java qui peut être instancié :

- sous forme d'un intergiciel CORBA, David ;
- sous forme d'un exécutif RMI, Jeremie.

SLOC	Partie	SLOC-by-Language (Sorted)
4177	jonathan	java=4177
39452	david	java=39452
4809	jeremie	java=4809
48438	TOTAL	

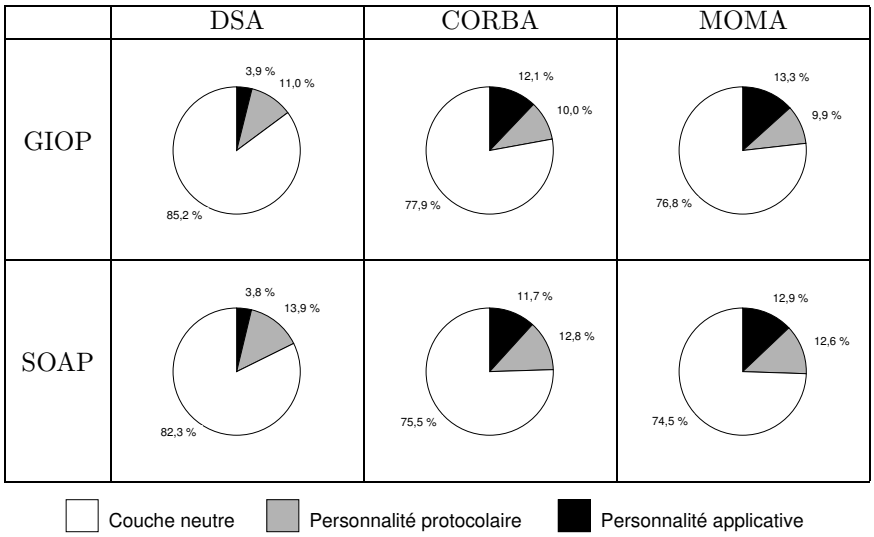
Tab. 9.7 – Répartition du code dans Jonathan

Le tableau 9.7 illustre la répartition du code entre la partie neutre de Jonathan et les personnalités David et Jeremie.

Nous observons que, sur ce projet de dimension comparable à PolyORB, la partie neutre, le noyau générique Jonathan proprement dit, correspond à 8,6 % du code. En effet, elle est constituée essentiellement d'interfaces abstraites, qui doivent être concrétisées par chaque personnalité. Cette organisation explique la part plus importante de la personnalité CORBA, David.

Jonathan ne distingue pas les personnalités applicative et protocolaire ; les deux personnalités David et Jeremie partagent un même protocole : GIOP. La mise en œuvre de GIOP fait partie de la personnalité David. Les classes de Jeremie dépendent donc de celles de David : les deux personnalités ne peuvent être instanciées de façon entièrement indépendante.

Dans PolyORB, la mutualisation de code entre les personnalités au sein de la couche neutre permet de réduire la taille de celles-ci, et ainsi de proposer cinq personnalités (trois applicatives, deux protocolaires, pouvant être combinées arbitraitement) dans un encombrement comparable à Jonathan, qui offre l'équivalent de deux personnalités applicatives et d'une seule personnalité protocolaire.



TAB. 9.8 – Répartition de code sur des configurations complètes de PolyORB

9.4.2 Analyse sur des configurations complètes

Après avoir proposé une mesure de la répartition du code entre couche neutre et personnalités dans l'ensemble du code de PolyORB (synthétisée dans le tableau 9.6), nous mesurons cette répartition sur quelques cas concrets d'applications, en fonction des personnalités configurées.

Le tableau 9.8 illustre la répartition du code de l'intergiciel pour des configurations typiques. Nous avons considéré les configurations possédant une seule personnalité applicative et une seule personnalité protocolaire.

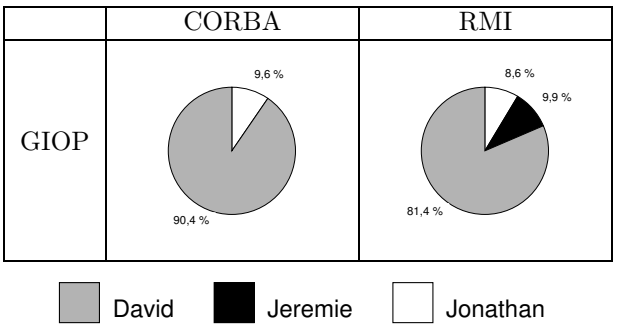
Dans chacune des combinaisons étudiées, la couche neutre approche ou dépasse 75% du code de l'intergiciel. Celle-ci réalise effectivement l'essentiel des fonctionnalités de l'intergiciel ; les personnalités constituant seulement des façades pour ces fonctionnalités. Le caractère générique de notre architecture est ainsi confirmé : nous avons identifié et isolé les fonctions récurrentes des intergiciel, et leur réalisation sous forme d'une couche neutre de répartition permet la mise en œuvre de différents modèles de répartition à faible coût (en développant uniquement les modules de personnalité correspondants).

Le tableau 9.9 illustre la répartition observée dans Jonathan.

La partie générique correspond à une quantité limitée de code. L'essentiel des fonctionnalités de répartition sont réalisées dans la personnalité CORBA David ; la personnalité Java RMI Jeremie constitue une surcouche ajoutée à David.

9.5 Synthèse

Nous avons vérifié que l'architecture schizophrène permet la réalisation de multiples personnalités d'intergiciels. En particulier, nous avons validé la personnalité applicative DSA au moyen des tests normalisés ACATS relatifs à cette annexe de la norme. Nous avons également vérifié le fonctionnement de la per-



TAB. 9.9 – Répartition de code sur des configurations complètes de Jonathan

sonnalité protocolaire GIOP par des tests d'interaction avec un intergiciel tiers de référence : omniORB.

Nous avons mis en place différents exemples mettant en évidence la contribution de PolyORB à la résolution de problèmes d'interopérabilité :

- entre personnalités applicatives, avec protocole fédérateur ;
- entre personnalités applicatives et protocolaires, avec protocoles multiples ;
- entre personnalités protocolaires, avec passerelle dynamique ;
- entre modèles de répartition.

Les mesures de performances préliminaires que nous avons conduites sur le prototype de PolyORB mettent en évidence le coût lié à l'architecture schizophrène. Celui-ci est essentiellement lié au passage par la représentation intermédiaire neutre des requêtes.

Malgré ce coût intrinsèque, les performances observées demeurent dans les mêmes ordres de grandeur que celles rencontrées avec des intergiciels existants, alors que cet aspect de PolyORB n'a pas jusqu'ici fait l'objet d'efforts de développement importants. Elles sont mêmes meilleures que celles observées avec d'autres intergiciels génériques déjà industrialisés tels que Jonathan.

Plusieurs aspects du prototype seront optimisés lors de travaux d'industrialisation. Nous obtiendrons ainsi des performances compétitives par rapport aux intergiciels traditionnels, tout en conservant l'apport fonctionnel que constitue l'interopérabilité.

CHAPITRE

10 Conclusion générale et perspectives

Sommaire

10.1 Interopérabilité : le paradoxe de l'intergiciel . . . 217
10.2 Architecture schizophrène 218
10.3 Réalisations 218
10.4 Perspectives 220

10.1 Interopérabilité : le paradoxe de l'intergiciel

Le développement des réseaux et des applications réparties entraîne un besoin croissant d'intégrer des composants logiciels d'origines diverses, ainsi que la nécessité de faire collaborer des applications existantes. Il est donc nécessaire de rendre les applications et les composants répartis *interopérables*.

L'utilisation d'intergiciels permet au développeur d'une application répartie de s'affranchir des incompatibilités entre environnements matériels et systèmes d'exploitation. Cependant, elle introduit de nouvelles incompatibilités, qui résultent de la variété des modèles de répartition disponibles et des mises en œuvre de ces modèles. Cette situation constitue le *paradoxe de l'intergiciel* [79].

La résolution de ce paradoxe passe par la définition d'outils autorisant la mise en relation de composants applicatifs basés sur des modèles de répartition différents. La seule utilisation de passerelles statiques d'un modèle à un autre n'est pas pleinement satisfaisante : elle multiplie les composants nécessaires dans une application répartie et limite la possibilité d'ajouter de nouveaux modèles par l'explosion combinatoire qu'elle induit.

Le choix d'un protocole commun pour l'ensemble des nœuds d'une application répartie n'est pas non plus une solution générale. Il ne peut pas convenir, en particulier, lorsque des composants préexistants basés sur des protocoles différents doivent coopérer au sein d'une même application.

10.2 Architecture schizophrène

Nous avons souhaité intégrer l'objectif d'interopérabilité dès la conception d'un intergiciel pour permettre à des composants applicatifs d'interagir avec des entités réparties issues de différents modèles de répartition.

Les architectures existantes d'intergiciels génériques introduisent la notion de *personnalité* et considèrent que chaque variante d'intergiciel est une vue différente des mêmes fonctions fondamentales.

Nous prolongeons cette démarche en permettant à différentes personnalités de *cohabiter* dans un même intergiciel, et d'utiliser *simultanément* une mise en œuvre générique mutualisée des fonctions de répartition.

Pour pouvoir répondre à la diversité des besoins de chaque application et des contraintes imposées par l'environnement de chaque nœud, l'intergiciel doit être configurable. Ainsi, lorsqu'un nœud d'une application répartie est un système embarqué temps réel, les fonctionnalités de l'intergiciel doivent être réduites à l'essentiel, et ne pas introduire le besoin de fonctionnalités telles que le parallélisme et les synchronisations entre tâches.

À partir des trois objectifs d'interopérabilité, de généricité et de configurabilité, nous avons présenté la conception d'un intergiciel *schizophrène*. Celle-ci découple les deux interfaces de l'intergiciel :

l'aspect applicatif exposé aux objets applicatifs locaux ;

l'aspect protocolaire exposé aux intergiciels tiers.

Nous avons ainsi étendu les architectures génériques existantes en introduisant la distinction entre les *personnalités applicatives* et les *personnalités protocolaires*. Celles-ci sont organisées autour d'une *couche neutre* mettant en œuvre les fonctions récurrentes des intergiciels de façon générique.

La cohabitation efficace de plusieurs personnalités partageant la même couche neutre est la clé de l'interopérabilité. Un objet basé sur une personnalité donnée peut alors utiliser un protocole quelconque pour interagir avec d'autres objets. Réciproquement, un même protocole peut être associé à des objets utilisant des personnalités applicatives différentes. Les comportements communs aux différentes personnalités applicatives et protocolaires ont été mutualisés, et sont réalisés de façon générique par la couche neutre.

10.3 Réalisations

À partir de l'architecture que nous avons définie, nous avons mis en œuvre un intergiciel schizophrène, PolyORB. Celui-ci offre une couche neutre réalisant de façon générique et configurable les fonctions récurrentes des intergiciels que nous avons identifiées :

- adressage ;
- transport ;
- liaison ;
- représentation ;
- protocole ;
- activation ;
- exécution.

La couche neutre permet la mise en place de personnalités applicatives multiples, qui sont autant de « façades » sur ces fonctions ; elle autorise leur cohabitation

dans une même instance d'intergiciel. Nous avons proposé deux gabarits de conception inédits, *Components* et *Annotation*, qui permettent un agencement flexible de modules indépendants.

Nous avons également réalisé des personnalités applicatives issues de divers modèles de répartition :
- une personnalité CORBA, basée sur le modèle « objets répartis » ;
- une personnalité mettant en œuvre l'annexe des systèmes répartis du langage Ada 95, offrant les modèles « objets répartis » et « appel de sous-programmes à distance » ;
- une personnalité d'intergiciel de passage de messages (MOM, *Message-Oriented Middleware*) inspiré de JMS.

Enfin, nous avons produit deux personnalités protocolaires :
- GIOP, protocole associé à la norme CORBA ;
- SOAP, protocole proposé pour les services WWW.

Ces réalisations nous ont permis de valider l'architecture schizophrène. Nous avons proposé une application témoin mettant en évidence les problèmes d'interopérabilité que nous souhaitions résoudre, et nous avons montré que notre architecture répond à ces besoins.

Nous avons ainsi confirmé que la coexistence de multiples personnalités permise par l'architecture schizophrène contribue à l'interopérabilité entre modèles de répartition, et entre différentes mises en œuvre du même modèle.

L'interopérabilité entre objets possédant un protocole commun est obtenue sans introduction de composants passerelles. Des objets tiers, ne possédant aucun protocole commun, peuvent également interagir grâce à l'intergiciel schizophrène, qui joue alors le rôle de passerelle dynamique et transparente.

Au cours de la réalisation des personnalités, nous avons été amené à ajouter des fonctionnalités à la couche neutre et à la rendre plus flexible. Par exemple, nous avons intégré le passage de messages unidirectionnels dans la couche neutre au cours du développement de la personnalité MOMA. Cette évolution nous a permis d'ajouter les appels asynchrones à la personnalité DSA et les méthodes unidirectionnelles à la personnalité CORBA. De même, nous avons réalisé un service de dénomination générique qui bénéficie simultanément aux personnalités CORBA et DSA.

Notre approche du développement d'intergiciels et de services pour la répartition par les fonctions génériques a non seulement rendu possible la mutualisation de fonctionnalités entre plusieurs personnalités, mais aussi permis l'enrichissement de chacune de ces personnalités par les fonctionnalités issues des autres.

Nous avons effectué des mesures préliminaires de performances sur notre prototype. Si celles-ci témoignent du coût lié à l'architecture schizophrène, leur ordre de grandeur est comparable à celui des intergiciels existants auxquels nous les avons comparées. L'optimisation du prototype les rapprochera de valeurs comparables à celles rencontrées avec des produits largement plus matures.

Nous avons publié PolyORB sous forme de logiciel libre. Son code source et l'ensemble des articles et documents relatifs au projet sont accessibles sur le WWW à l'adresse `http://libre.act-europe.fr/polyorb/`.

10.4 Perspectives

Le prototype que nous avons réalisé préfigure un intergiciel utilisable dans des applications réelles. Plusieurs entreprises ont manifesté leur intérêt pour ce produit, qui va entrer en phase d'industrialisation.

En outre, l'architecture schizophrène constitue la base de travaux futurs sur les outils génériques pour la répartition.

Optimisation des performances

La couche neutre de PolyORB devra faire l'objet d'un profilage détaillé, afin d'en améliorer les performances. En particulier, l'adaptateur d'objets portable générique devrait être revu pour le rendre plus efficace.

L'utilisation de certaines constructions évoluées du langage Ada 95 devra également être remise en question, non seulement dans le but d'améliorer les performances d'ensemble, mais encore afin de permettre l'utilisation de PolyORB sous un profil de restrictions pour les systèmes temps réel de haute intégrité.

Nouvelles personnalités — guide méthodologique

Grâce à l'architecture fortement modulaire de PolyORB, nous pourrons y intégrer à faible coût de nouvelles personnalités applicatives (notamment une personnalité pour objets WWW basée sur l'interface de programmation AWS, *Ada Web Server* [74]) et protocolaires (en particulier une personnalité HTTP).

Leur mise en œuvre sera l'occasion de formaliser un guide méthodologique de personnalisation.

Configurations temps réel

La configurabilité constitue un atout majeur de PolyORB, notamment pour son utilisation dans des systèmes embarqués ou temps réel, avec des fonctionnalités allégées. Son amélioration autoriserait même l'utilisation de certains composants de PolyORB dans des applications de haute intégrité.

Le profil Ravenscar en fait un intergiciel de choix pour les systèmes temps réel. La proposition de nouveaux services, tels qu'un service d'événements temps réel, et de gabarits de conception pour le temps réel, augmenteront encore son intérêt pour de telles applications.

Aide au déploiement d'applications réparties

La richesse fonctionnelle de notre intergiciel rend complexe le déploiement d'une configuration complète adaptée aux besoins d'une application. Nous envisageons d'adapter l'outil de partitionnement de GLADE, `gnatdist`, à notre mise en œuvre de l'annexe des systèmes répartis d'Ada 95, et de l'étendre pour tirer parti de l'ensemble des axes de configurabilité de PolyORB.

Ce travail préfigurera la spécification d'un outil générique (indépendant d'une personnalité applicative particulière) d'aide à la création et au déploiement d'applications réparties.

Compilateur générique

Lors de la réalisation des personnalités applicatives CORBA et DSA, nous avons réalisé deux outils de compilation : le compilateur IDL `idlac` et une version adaptée de GNAT. Si une partie du code généré est spécifique de chaque personnalité applicative, certaines structures rencontrées sont communes. De prochains travaux pourront donc étudier la mise en œuvre d'un outil de compilation générique pour la répartition.

La démarche que nous avons adoptée pour l'exécution des applications réparties, consistant à découpler les aspects liés à une personnalité des aspects récurrents, sera alors prolongée vers l'amont du processus de développement.

Services génériques

Enfin, nous souhaitons développer autour de PolyORB un ensemble de services génériques susceptibles de bénéficier à l'ensemble des personnalités, en particulier :

– la gestion du cycle de vie des partitions ;
– la gestion d'un espace de stockage partagé.

Le service d'espace de stockage partagé pourrait constituer une première étape vers l'intégration du modèle de répartition « mémoire partagée répartie » dans l'architecture schizophrène.

Les intergiciels schizophrènes permettent la coopération d'objets d'origines diverses à toutes les échelles des applications informatiques, des calculateurs embarqués aux systèmes d'information d'organisations étendues. Nous contribuons ainsi à l'évolution des systèmes répartis vers une intégration globale de l'ensemble des objets communicants.

Bibliographie

[1] M. Accetta, R. Baron, D. Golub, R. Rashid, A. Tevanian et M. Young. Mach: A New Kernel Foundation for UNIX Development. Dans *Proceedings of the Summer 1986 USENIX Technical Conference and Exhibition*. juin 1986.

[2] Ada Rapporteur Group. Use of PCS should not be normative. Ada Issue AI95-00273/02, ISO/IEC JTC1/SC22/WG9, 2002. URL `http://pebbles.ocsystems.com/cgi-bin/cvsweb.cgi/AIs/AI-00273.TXT?rev=1.7`, adopted amendment to the Ada 95 standard [49].

[3] Alfred V. Aho, Ravi Sethi et Jeffrey D. Ullman. *Compilers — Principles, Techniques and Tools*. Addison-Wesley, 1986.

[4] Christiana Amza, Alan L. Cox, Sndhya Dwarkadas, Pete Keleher, Honghui Lu, Ramakrishnan Rajamony, Weimin Yu et Willy Zwaenepoel. TreadMarks: Shared Memory Computing on Networks of Workstations. *IEEE Computer*, 29(2):18–28, février 1996.

[5] Fabien Azavant, Jean-Marie Cottin, Vincent Niebel, Laurent Pautet, Sébastien Ponce, Thomas Quinot et Samuel Tardieu. CORBA and CORBA Services for DSA. Dans *Proceedings of SigAda'99*. ACM, Redondo Beach, CA, USA, octobre 1999. URL `http://www.infres.enst.fr/~quinot/publis/SigAda99.ps`.

[6] Sean Baker. A2A, B2B — Now We Need M2M (Middleware to Middleware) Technology. Dans *Proceedings of the 3rd International Symposium on Distributed Objects and Applications (DOA'01)*. IONA Technologies, septembre 2001. Keynote speech.

[7] Tim Berners-Lee, Roy T. Fielding et Larry Masinter. Uniform Resource Identifiers (URI): Generic Syntax. Request for Comments 2396, Internet Engineering Task Force, août 1998.

[8] Philip A. Bernstein. Middleware: A Model for Distributed System Services. *Communications of the ACM*, 39(2):86–98, février 1996.

[9] Andrew Birrell, Greg Nelson, Susan Owicki et Edward Wobber. Network Objects. Rapport technique 115/94, Digital Systems Research Center, février 1994.

[10] Andrew D. Birrell et Bruce J. Nelson. Implementing Remote Procedure Calls. *ACM Transactions on Computer Systems*, 2(1):39–59, février 1984.

[11] Gordon S. Blair, Geoff Coulson, P. Robin et Michael Papathomas. An architecture for Next Generation Middleware. Dans *Proceedings of the 1st IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware'98)*. Springer Verlag, septembre 1998.

[12] Don Box, David Ehnebuske, Gopal Kakivaya, Andrew Layman, Noah Mendelsohn, Henrik Frystyk Nielsen, Satish Thatte et Dave Winer. Simple Object Access Protocol (SOAP) 1.1. W3C note, World Wide Web Consortium, mai 2000. URL `http://www.w3.org/TR/SOAP/`.

[13] Fabian Breg, Shridhar Diwan, Juan Villacis, Jayashree Balasubramanian, Esra Akman et Dennis Gannon. Java RMI Performance and Object Model Interoperability: Experiments with Java/HPC++ Distributed Components. Dans *Proceedings of the ACM 1998 Workshop on Java for High-Performance Network Computing*, pages 91–100. ACM, mai 1998. URL `http://www.extreme.indiana.edu/hpjava/nexusrmi/java98.ps`.

[14] Emmanuel Briot. *XML/Ada: a full XML suite*, 2001. URL `http://libre.act-europe.fr/xmlada/`.

[15] Martin C. Carlisle. Multiple inheritance using Ada 95. *Ada Letters*, 18(1):101–105, janvier 1998. URL `http://usafa.af.mil/dfcs/papers/mcc/aseet97_oo.html`.

[16] Erik Christensen, Francisco Curbera, Greg Meredith et Sanjiva Weerawarana. Web Services Description Language (WSDL) 1.1. W3C note, World Wide Web Consortium, mars 2001. URL `http://www.w3.org/TR/wsdl/`.

[17] Chris Cleeland, Douglas C. Schmidt et Timothy H. Harrison. External polymorphism — an Object Structural Pattern for Transparently Extending C++ Concrete Data Types. Dans Martin *et al.* [62]. URL `http://www.cs.wustl.edu/~schmidt/PLoP-96/External-Polymorphism.ps.gz`, based on papers presented at the Third Annual Conference of Pattern Languages Of Programming (PLoP'96) and EuroPLoP'96.

[18] Zbigniew J. Czech, George Havas et Bohdan S. Majewski. An optimal algorithm for generating minimal perfect hash functions. *Information Processing Letters*, 43(5):257–264, octobre 1992.

[19] Ole-Johan Dahl, Bjørn Myrhaug et Kristen Nygaard. Simula: Common Base Language. Rapport technique 743, Norsk Regnesentral, février 1984.

[20] Linda G. DeMichiel. *Enterprise JavaBeans Specification, version 2.1*. Sun Microsystems, Inc., juin 2002. URL `http://java.sun.com/products/ejb/`.

[21] Martin Dietzfelbinger, Anna Karlin, Kurt Mehlhorn, Friedhelm Meyer auf der Heide, Hans Rohnert et Robert E. Tarjan. Dynamic perfect hashing: upper and lower bounds. *SIAM Journal on Computing*, 23(4):738–761, août 1994.

[22] Edsger W. Dijkstra. The structure of the "THE"-multiprogramming system. *Communications of the ACM*, 11(5):341–346, mai 1968.

[23] Brian Dobbing et Alan Burns. The Ravenscar tasking profile for high integrity real-time programs. Dans *Proceedings of SigAda'98*. Washington, DC, USA, novembre 1998.

[24] Bruno Dumant, François Horn, Frédéric Dang Tran et Jean-Bernard Stéfani. Jonathan: an Open Distributed Processing Environment in Java. Dans *Proceedings of the IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware'98)*. Springer Verlag, septembre 1998.

[25] Bertrand Dupouy, Olivier Hainque, Laurent Pautet et Samuel Tardieu. The SPIF project. Dans *Proceedings of the 1997 Ada-Europe International Conference on Reliable Software Technologies*. Springer Verlag, juin 1997.

[26] Julian Elischer et Archie Cobbs. *The Netgraph Networking System*. Whistle Communications, Inc., 1996. URL `http://www.elischer.org/netgraph/`, see also `netgraph`(4) FreeBSD manual page.

[27] Roy T. Fielding, James Gettys, Jeffrey C. Mogul, Harry Frystyk Nielsen, Larry Masinter, Paul J. Leach et Tim Berners-Lee. Hypertext Transfer Protocol — HTTP/1.1. Request for Comments 2616, Internet Engineering Task Force, juin 1999.

[28] Cristian Francu et Ivan Marsic. An Advanced Communication Toolkit for Implementing the Broker Pattern. Dans *Proceedings of the 19th International Conference on Distributed Computing Systems (ICDCS'99)*. IEEE, IEEE, juin 1999. URL `http://www.caip.rutgers.edu/~francu/DISCIPLE/ACT/icdcs99.ps.gz`.

[29] Michael L. Fredman, János Komlós et Endre Szemerédi. Storing a sparse table with $O(1)$ worst case access time. *Journal of the ACM*, 31(3):538 – 544, juin 1984.

[30] Alan O. Freier, Philip Karlton et Paul C. Kocher. The SSL Protocol — Version 3.0. Internet Draft draft-freier-ssl-version3-02.txt, novembre 1998. *Work in progress*. The Secure Socket Layer has become a *de facto* standard for secure communication over the Internet.

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

[32] Anthony Gargaro, Yvon Kermarrec, Laurent Pautet et Samuel Tardieu. PARIS: Partitionned Ada for Remotely Invoked Services. Dans *Proceedings of AdaEurope'95*. Springer Verlag, mars 1995.

[33] Anthony Gargaro, Gary Smith, Ronald Theriault, Richard Volz et Raymond Waldrop. Towards Interoperation of Ada and Java Using ADEPT (Advanced Distributed Execution Partioning Toolset). Dans *Proceedings of the 9th Annual Software Technology Conference*. Salt Lake City, Utah, USA, avril 1997.

[34] David Garlan, Robert Allen et John Ockerbloom. Architectural Mismatch, or Why it's hard to build systems out of existing parts. Dans *Proceedings of the 17th International Conference on Software Engineering (ICSE'95)*. IEEE/ACM, ACM, avril 1995.

[35] Al Geist, Adam Beguelin, Jack Dongarra, Jiang Weicheng, Robert Mancheck et Vaidyalingam S. Sunderam. *PVM: Parallel Virtual Machine*. MIT Press, 1994.

[36] Aniruddha S. Gokhale et Douglas C. Schmidt. Evaluating the performance of demultiplexing strategies for Real-time CORBA. Dans *Proceedings of GLOBECOM'97*. Phoenix, Arizona, novembre 1997. URL `http://www.cs.wustl.edu/~schmidt/PDF/GLOBECOM-97.pdf`.

[37] Aniruddha S. Gokhale et Douglas C. Schmidt. Optimizing a CORBA Internet Inter-ORB Protocol (IIOP) Engine for Minimal Footprint Embedded Multimedia Systems. *IEEE Journal on Selected Areas in Communications*, 17(9):1673–1706, septembre 1999.

[38] Michael González Harbour et Scott Arthur Moody. Distributed Ada and real-time (session summary). Dans *Proceedings of the 9th international workshop on Real-time Ada (IRTAW'99)*, pages 15–18. ACM Press, 1999.

[39] Christoph Grein. Safe pointers. *Ada User Journal*, 20(2):162–168, juillet 1999.

[40] Duncan Grisby, Sai-Lai Lo et David Riddoch. *The omniORB version 4.0 User's Guide*, août 2002. URL `http://omniorb.sourceforge.net/omni40/omniORB/`.

[41] Jérôme Guitton. *Utilisation de profils de restrictions dans un intergiciel générique*. Mémoire de DEA, ENST, septembre 2002.

[42] Graham Hamilton, Michael L. Powell et James G. Mictchell. Subcontract: A Flexible base for Distributed Programming. Dans *Proceedings of the 14th ACM Symposium on Operating Systems Principles*, pages 69–79. Asheville, NC (USA), 1993.

[43] Matthew Heaney. Design patterns in Ada 95. Dans *Tutorials of SIGAda'99*. ACM, Redondo Beach, California, octobre 1999.

[44] C. A. R. Hoare. Monitors: An Operating System Structuring Concept. *Communications of the ACM*, 17(10):549–557, octobre 1974. ISSN 0001-0782. Erratum in *Communications of the ACM*, Vol. 18, No. 2 (February), p. 95, 1975. This paper contains one of the first solutions to the Dining Philosophers problem.

[45] Jérôme Hugues, Fabrice Kordon, Laurent Pautet et Thomas Quinot. A case study of Middleware to Middleware: MOM and ORB interoperability. Dans *Proceedings of the 4th International Symposium on Distributed Objects and Applications (DOA'02)*. University of California, Irvine, Irvine, CA, USA, octobre 2002. URL `http://www.infres.enst.fr/~quinot/publis/doa02.ps`.

[46] Norman C. Hutchinson et Larry L. Peterson. The x-Kernel: An Architecture for Implementing Network Protocols. *IEEE Transactions on Software Engineering*, 17(1):64–76, janvier 1991.

[47] Daniel H. H. Ingalls. The Smalltalk-76 Programming System Design and Implementation. Dans *Conference Record of the Fifth Annual ACM Symposium on Principles of Programming Languages, Tucson, Arizona*, pages 9–16. janvier 1978. URL `http://users.ipa.net/~dwighth/smalltalk/St76/Smalltalk76ProgrammingSystem.html`.

[48] Intermetrics. *Ada 95 Rationale: The Language, The Standard Libraries*. Intermetrics, janvier 1995. URL `http://www.adahome.com/Resources/refs/rat95.html`.

[49] ISO. *Information Technology – Programming Languages – Ada*. ISO, février 1995. ISO/IEC/ANSI 8652:1995.

[50] ISO. *Information technology – Portable Operating System Interface (POSIX) – Part 1: System Application Program Interface (API) [C Language]*. ISO, 1996. ISO/IEC 9945-1:1996, « POSIX.1 ».

[51] ISO. *Information technology – Open Distributed Processing – Reference model: Overview*. ISO, décembre 1998. ISO/IEC 10746-1:1998.

[52] ISO. *Information Technology – Programming Languages – Ada Semantic Interface Specification (ASIS)*. ISO, 1998. ISO/IEC 15291:1998.

[53] ISO. *Information technology – Programming languages – Ada: Conformity assessment of a language processor*. ISO, décembre 1999. ISO/IEC 18009:1999.

[54] Yvon Kermarrec, Laurent Nana et Laurent Pautet. GNATDIST: a configuration language for distributed Ada 95 applications. Dans *Proceedings of Tri-Ada'96*. Philadelphia, Pennsylvania, USA, 1996.

[55] Yvon Kermarrec, Laurent Pautet et Samuel Tardieu. Garlic: Generic Ada Reusable Library for Interpartition Communication. Dans *Proceedings of Tri-Ada'95*. ACM, Anaheim, California, USA, 1995.

[56] Jörg Kienzle, Thomas Wolf et Alfred Strohmeier. Secure Communication in Distributed Ada. Dans *Proceedings of AdaEurope'96*. Springer Verlag, juin 1996. URL `http://lglwww.epfl.ch/~wolf/docs/Cryptopaper.ps.gz`.

[57] Serge Lacourte et Michel Riveill. Don't bother the application designer with system issues. Dans *Proceedings of the Granularity of objects in distributed systems workshop*. Kaiserlautern, Germany, juin 1993.

[58] Arnaud Le Hors, Philippe Le Hégaret, Lauren Wood, Gavin Nicol, Jonathan Robie, Mike Champion et Steve Byrne. Document Object Model Level 2 Core. W3C recommendation, World Wide Web Consortium, novembre 2000. URL `http://www.w3.org/DOM/`.

[59] Kai Li et Paul Hudak. Memory Coherence in Shared Virtual Memory Systems. *ACM Transactions on Computer Systems*, 7(4):321–359, novembre 1989.

[60] Rajesh S. Madukkarumukumana, Hemal V. Shah et Carlton Pu. Harnessing User-Level Networking Architectures for Distributed Object Computing over High-speed Networks. Dans *Proceedings of the 2nd USENIX Windows NT Symposium*. Seattle, Washington, août 1998. URL `http://www.usenix.org/publications/library/proceedings/usenix-nt98/full_papers/madukkarum/madukkarum.pdf`.

[61] Silvano Maffeis et Douglas C. Schmidt. Constructing Reliable Distributed Communication Systems with CORBA. *IEEE Communications Magazine*, 14(2), février 1997. URL `http://www.cs.wustl.edu/~schmidt/PDF/ieeecomm.pdf`.

[62] Robert C. Martin, Dirk Riehle et Frank Buschmann, coordinateurs. *Pattern Languages of Program Design*, volume 3 de *Software Patterns Series*. Addison-Wesley, 1997. Based on papers presented at the Third Annual Conference of Pattern Languages Of Programming (PLoP'96) and EuroPLoP'96.

[63] M. Douglas McIlroy. Mass-Produced Software Components. Dans P. Naur et B. Randell, coordinateurs, *Software Engineering, Report on a conference sponsored by the NATO Science Committee*, pages 138–155. NATO Science Committee, octobre 1968.

[64] David Megginson *et al. Simple API for XML*, 2000. URL `http://www.saxproject.org/`.

[65] Philippe Merle, Christophe Gransart et Jean-Marc Geib. CorbaWeb: A WWW and Corba Worlds Integration. Dans *Proceedings of the Second USENIX Conference on Object-Oriented Technologies (COOTS'96), Workshop on Distributed Object Computing on the Internet*. The USENIX Association, juin 1996.

[66] Philippe Merle, Christophe Gransart et Jean-Marc Geib. Generic tools: a new way to use Corba. Dans *Proceedings of the European Conference on Object-Oriented Programming (ECOOP'97)*. Springer Verlag, juin 1997.

[67] Philippe Merle, Christophe Gransart et Jean-Marc Geib. Using and Implementing CORBA Objects with CorbaScript. Dans Takanobu Baba, Jean-Paul Bahsoun, Jean-Pierre Briot et Akinori Yonezawa, coordinateurs, *Object Oriented Parallel and Distributed Programming*. Hermès Science Publications, janvier 1999. Papers from the proceedings of the First and Second France-Japan Workshops on Object Based Parallel and Distributed Computing (OBPDC'95 and '97).

[68] Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard*, juin 1995. URL `http://www.mpi-forum.org/docs/mpi-11.ps.Z`, version 1.1.

[69] Sape J. Mullender, Guido van Rossum, Andrew S. Tanenbaum, Robbert van Renesse et Hans van Staveren. Amoeba: A distributed operating system. *IEEE Computer*, 23:44–53, mai 1990.

[70] Object Management Group. *Discussion of the Object Management Architecture*. Object Management Group, janvier 1997. OMG formal/00-06-41.

[71] Object Management Group. *The Common Object Request Broker: Architecture and Specification, revision 2.2*. Object Management Group, février 1998. OMG Technical Document formal/98-07-01.

[72] Object Management Group. *Java Language to IDL Mapping*. Object Management Group, janvier 2000. OMG ptc/00-01-06.

[73] Object Management Group. *Ada Language Mapping Specification, v1.2*. Object Management Group, octobre 2001. OMG Technical Document formal/2001-10-42.

[74] Pascal OBRY et Dmitriy ANISIMKOV. *AWS: a complete Web development framework*, 2001. URL `http://libre.act-europe.fr/aws/`.

[75] OMG. *Real-time CORBA v. 1.1*. OMG, 2002. OMG formal/2002-08-02.

[76] Carlos O'RYAN, Fred KUHNS, Douglas C. SCHMIDT, Ossama OTHMAN et Jeff PARSONS. The Design and Performance of a Pluggable Protocols Framework for Real-time Distributed Object Computing Middleware. Dans *Proceedings of the IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware'00)*. avril 2000. URL `http://www.cs.wustl.edu/~fredk/papers/pluggable_protocols.ps`.

[77] Carlos O'RYAN, Douglas C. SCHMIDT, Fred KUHNS, Marina SPIVAK, Jeff PARSONS, Irfan PYARALI et David L. LEVINE. Evaluating Policies and Mechanisms for Supporting Embedded, Real-Time Applications with CORBA. *Journal of Theory and Practice of Object Systems*, 13(2), février 2001. Special issue on Distributed Objects and Applications.

[78] David A. PATTERSON. Reduced instruction set computers. *Communications of the ACM*, 28(1):8–21, janvier 1985.

[79] Laurent PAUTET. Intergiciels schizophrènes : une solution à l'interopérabilité entre modèles de répartition. Mémoire d'habilitation à diriger des recherches, Université Paris VI — Pierre-et-Marie-Curie, décembre 2001.

[80] Laurent PAUTET. Intergiciels schizophrènes : une solution à l'interopérabilité entre modèles de répartition. Dans Isabelle DEMEURE et Élie NAJM, coordinateurs, *Avancées dans le domaine des intergiciels*. Hermès Science Publications, 2002.

[81] Laurent PAUTET, Thomas QUINOT et Samuel TARDIEU. CORBA & DSA: Divorce or Marriage? Dans *Proceedings of the 1999 Ada-Europe International Conference on Reliable Software Technologies*. Santander, Spain, juin 1999. URL `http://www.infres.enst.fr/~quinot/publis/ae99.ps`.

[82] Laurent PAUTET et Samuel TARDIEU. GLADE: a Framework for Building Large Object-Oriented Real-Time Distributed Systems. Dans *Proceedings of the 3rd IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC'00)*. Newport Beach, California, USA, juin 2000.

[83] Laurent PAUTET et Samuel TARDIEU. *GLADE User Guide*. Ada Core Technologies, 2000.

[84] Laurent PAUTET et Thomas WOLF. Transparent Filtering of Streams in GLADE. Dans *Proceedings of Tri-Ada'97*. ACM, Saint-Louis, Missouri, USA, 1997.

[85] Jonathan B. POSTEL. User Datagram Protocol. Request for Comments 768, Internet Engineering Task Force, août 1980.

[86] Jonathan B. POSTEL. Internet Protocol Specification. Request for Comments 791, Internet Engineering Task Force, septembre 1981.

[87] Thomas QUINOT. CIAO: Opening the Ada 95 Distributed Systems Annex to CORBA Clients. Dans *Ada France 1999*. Brest, France, septembre 1999. URL `http://www.infres.enst.fr/~quinot/publis/adafrance.pdf`.

[88] Thomas Quinot. Mapping the Ada 95 Distributed Systems Annex to OMG IDL — Mapping definition. Rapport technique, ENST Paris and University Paris VI — Pierre-et-Marie-Curie, mai 1999. URL `http://www.infres.enst.fr/~quinot/publis/ciao-mapping.ps.gz`.

[89] Thomas Quinot. *Mapping the Ada 95 Distributed Systems Annex to OMG IDL — Specification and implementation.* Mémoire de fin d'études, ENST Paris, août 1999. URL `http://www.infres.enst.fr/~quinot/publis/ciao-impl.ps.gz`.

[90] SC-167. Software considerations in airborne systems and equipment certification. Advisory circular DO-178B, Radio Technical Commission for Aeronautics, 1992.

[91] Douglas C. Schmidt. GPERF: a perfect hash function generator. Dans *Proceedings of the Second USENIX C++ Conference*, pages 87–102. San Francisco, California, avril 1990. URL `http://www.cs.wustl.edu/~schmidt/PDF/C++-USENIX-90.pdf`.

[92] Douglas C. Schmidt. Reactor — An Object Behavioral Pattern for Event Demultiplexing and Event Handler Dispatching. Dans James O. Coplien et Douglas C. Schmidt, coordinateurs, *Pattern Languages of Program Design*, volume 1 de *Software Patterns Series*. Addison-Wesley, 1995. URL `http://www.cs.wustl.edu/~schmidt/PDF/reactor-siemens.pdf`, based on papers presented at the First Annual Conference of Pattern Languages of Programming (PLoP'94).

[93] Douglas C. Schmidt. Wrapper Facade: A Structural Pattern for Encapsulating Functions within Classes. *SIGS C++ Report*, 11(2), février 1999. URL `http://www.cs.wustl.edu/~schmidt/PDF/wrapper-facade.pdf`.

[94] Douglas C. Schmidt et Chris Cleeland. Applying Patterns to Develop Extensible ORB Middleware. *IEEE Communications Magazine Special Issue on Design Patterns*, 1998. URL `http://www.cs.wustl.edu/~schmidt/PDF/ORB-patterns.pdf`.

[95] Douglas C. Schmidt et Charles D. Cranor. Half-Sync/Half-Async — an Architectural Pattern for Efficient and Well-structured Concurrent I/O. Dans Vlissides *et al.* [118]. URL `http://www.cs.wustl.edu/~schmidt/PDF/PLoP-95.pdf`, based on papers presented at the Second Annual Conference of Pattern Languages of Programming (PLoP'95).

[96] Douglas C. Schmidt et Charles D. Cranor. Acceptor-Connector — An Object Creational Pattern for Connecting and Initializing Communication Services. Dans Martin *et al.* [62]. URL `http://www.cs.wustl.edu/~schmidt/PDF/Acc-Con.pdf`, based on papers presented at the Third Annual Conference of Pattern Languages Of Programming (PLoP'96) and EuroPLoP'96.

[97] Douglas C. Schmidt et R. Greg Lavender. Active Object — An Object Behavioral Pattern for Concurrent Programming. Dans Vlissides *et al.* [118]. URL `http://www.cs.wustl.edu/~schmidt/PDF/Act-Obj.pdf`, based on papers presented at the Second Annual Conference of Pattern Languages of Programming (PLoP'95).

[98] Douglas C. Schmidt, David L. Levine et Sumedh Mungee. The design of the TAO real-time object request broker. *Computer Communications*, 21(4):294–324, avril 1998.

[99] Douglas C. Schmidt, Michael Stal, Hans Rohnert et Frank Buschmann. *Patterns for Concurrent and Networked Objects*, volume 2 de *Pattern-Oriented Software Architecture*. John Wiley & Sons, 1996.

[100] Douglas C. Schmidt et Steve Vinoski. Comparing Alternative Programming Techniques for Multi-threaded Servers – the Thread-per-Request Concurrency Model. *SIGS C++ Report*, 8(2), février 1996.

[101] Douglas C. Schmidt et Steve Vinoski. Comparing Alternative Programming Techniques for Multi-threaded Servers – the Thread-per-Session Concurrency Model. *SIGS C++ Report*, 8(7), juillet 1996.

[102] Douglas C. Schmidt et Steve Vinoski. Comparing Alternative Programming Techniques for Multi-threaded Servers – the Thread-Pool Concurrency Model. *SIGS C++ Report*, 8(4), avril 1996.

[103] Douglas C. Schmidt et Steve Vinoski. An introduction to CORBA Messaging. *SIGS C++ Report*, 10(10), novembre 1998.

[104] Edmond Schonberg et Bernard Banner. The GNAT Project: A GNU-Ada 9X Compiler. Dans *Proceedings of Tri-Ada'94*. ACM, Baltimore, Maryland, USA, 1994.

[105] Michael D. Schroeder. A State-of-the-Art Distributed System: Computing with BOB. Dans Sape J. Mullender, coordinateur, *Distributed Systems*. Addison-Wesley, deuxième édition, mars 1993.

[106] Marc Shapiro. A Binding Protocol for Distributed Shared Objects. Dans *Proceedings of the 14th International Conference on Distributed Computing Systems (ICDCS'94)*. IEEE, IEEE, juin 1994. URL `http://www.laas.research.ec.org/broadcast/trs/papers/56.ps`.

[107] Marc Shapiro, Peter Dickman et David Plainfossé. SSP Chains: Robust, distributed references supporting acyclic garbage collection. Rapport de recherche 1799, INRIA Rocquencourt, novembre 1992.

[108] Ashish Singhai. *Quarterware: A middleware toolkit of software RISC components*. Thèse de doctorat, University of Illinois at Urbana-Champaign, 1999.

[109] Aleksander Slominski, Madhusudhan Govindaraju, Dennis Gannon et Randall Bramley. SoapRMI C++/Java 1.1: Design and Implementation. Rapport technique 548, Indiana University, mai 2001. URL `http://www.cs.indiana.edu/ftp/techreports/TR548.html`.

[110] Gary W. Smith et Richard A. Volz. Distributed Programming with Intermediate IDL. Dans *Proceedings of the 9th international workshop on Real-time Ada (IRTAW'99)*. ACM Press, 1999.

[111] Richard W. Stevens. *TCP/IP Illustrated*, volume 1: The Protocols. Addison-Wesley, 1994.

[112] Richard W. Stevens. *UNIX Network Programming*, volume 1: Networking APIs: Sockets and XTI. Prentice-Hall, deuxième édition, 1998.

[113] Sun Microsystems, Inc. *Java Naming and Directory Interface*. Sun Microsystems, Inc., 1999.

[114] Sun Microsystems, Inc. *Java Message Service Specification*. Sun Microsystems, Inc., 2002. Version 1.1.

[115] Sun Microsystems, Inc. *Java Remote Method Invocation Specification*. Sun Microsystems, Inc., 2002. Version 1.4.

[116] Samuel Tardieu. *GLADE, une implémentation de l'annexe des systèmes répartis d'Ada 95*. Thèse de doctorat, École nationale supérieure des télécommunications, octobre 1999.

[117] Bruce Trask. Using CORBA Asynchronous Messaging, Pluggable Protocols and the Real-Time Event Service in a Real-Time Embedded System. Dans *Proceedings of the Object Management Group's First Workshop on Real-Time and embedded Distributed Object Computing*. OMG, Falls Chruch, VA, USA, juillet 2000.

[118] John M. Vlissides, James O. Coplien et Norman L. Kerth, coordinateurs. *Pattern Languages of Program Design*, volume 2 de *Software Patterns Series*. Addison-Wesley, 1996. Based on papers presented at the Second Annual Conference of Pattern Languages of Programming (PLoP'95).

[119] David A. Wheeler. SLOCCount, a set of tools for counting physical Source Lines of Code (SLOC). URL `http://www.dwheeler.com/sloccount/`.

[120] Linda Wills, Sam Sander, Suresh Kannan, Aaron Kahn, J. V. R. Prasad et Daniel Schrage. An Open Control Platform for reconfigurable, distributed, hierarchical control systems. Dans *Proceedings of the Digital Avionics Systems Conference*. Philadelphia, PA, USA, octobre 2000.

[121] Paul R. Wilson. Uniprocessor Garbage Collection Techniques. Dans *Proceedings of the 1992 International Workshop on Memory Management*. Saint Malo, France, septembre 1992.