

CorbaScript - CorbaWeb :  
Propositions pour l'accès à des  
objets et services distribués

Philippe Merle

Laboratoire d'Informatique Fondamentale de Lille - URA CNRS 369

Université des Sciences et Technologies de Lille

Bâtiment M3, 59655 Villeneuve d'Ascq Cedex - France

Tél.: (33) 20 43 47 21 - Fax: (33) 20 43 65 66

*E-mail: merle@lfl.fr*

*URL: <http://www.lfl.fr/~merle/>*

23 janvier 1997



# Remerciements

Je remercie les membres de mon jury :

- Monsieur **Vincent CORDONNIER**, professeur à l’Université de Lille I, pour m’avoir fait l’honneur de présider ce jury. Il a accepté très simplement et amicalement lors d’une de nos nombreuses discussions tardives et enfumées dans les couloirs du laboratoire.
- Monsieur **Guy BERNARD**, professeur à l’Institut National des Télécommunications d’Evry, et Monsieur **Michel RIVEILL**, professeur à l’Université de Savoie, d’avoir accepté de rapporter cette thèse dans des délais très courts.
- Monsieur **Alain DERYCKE**, professeur à l’Université de Lille I, et Madame **Valérie GAY**, maître de conférences à l’Université de Paris 6 au MASI, pour s’être intéressés à mon travail et avoir accepté d’examiner ce mémoire.
- Monsieur **Jean-Marc GEIB**, professeur à l’Université de Lille I, de m’avoir proposé cette thèse et accueilli au sein de son équipe. Il a su à la fois diriger et conseiller mon travail tout en me laissant une grande liberté dans mes investigations et choix de réalisation. Je tiens aussi à le remercier pour nos nombreuses discussions et ses précieux conseils lors de la rédaction de ce mémoire.
- Monsieur **Christophe GRANSART**, maître de conférences à l’Université de Lille I, pour avoir suivi et participé à mon travail ainsi qu’à la rédaction d’articles. Son amitié, son enthousiasme et son intérêt pour ce travail m’ont encouragé tout au long de ces derniers mois lorsque je doutais un peu de tout. Nos nombreuses discussions et ses multiples relectures m’ont permis d’achever ce manuscrit.

Je tiens également à remercier les autres membres de l’équipe : **Yves DENNEULIN** pour ses nombreux conseils d’anglais, **Cédric DUMOULIN** et **Chrystel GRENOT** avec qui j’ai mené mes premiers travaux en DEA, **Jean-François MEHAUT** pour m’avoir proposé en Maîtrise de participer aux travaux de l’équipe, **Raymond NAMYST** et **Jean-François ROOS** pour leur sympathie et amitié.

Je remercie aussi tous les membres du laboratoire LIFL pour l’agréable ambiance de travail aussi bien dans le cadre de la recherche que de l’enseignement.

De plus, j’ai eu beaucoup de plaisir à discuter et travailler avec **Eric DUFRESNE** du CERIM, **Emmanuel HORCKMANS** ex étudiant en DEA et **Jean-Jacques VANDEWALLE** de RD2P. J’espère fortement que nos collaborations pourront se poursuivre dans l’avenir.

Finalement, je remercie mes parents pour leurs aides et encouragements au cours de mes études. Je tiens à remercier ma tendre compagne **Valérie** pour m’avoir encouragé tout au long de ces années et d’avoir accepté mon humeur lors de ces longs derniers mois. Ce mémoire doit beaucoup à sa relecture attentive et sa chasse aux fautes de français. Sans eux, ce mémoire n’aurait jamais pu voir le jour : **je leur dédie ce manuscrit.**



# Table des matières

<b>Introduction</b>	<b>13</b>
Proposition . . . . .	14
Plan détaillé . . . . .	15
Contexte du travail . . . . .	16
<b>1 Les services à grande échelle</b>	<b>17</b>
1.1 Introduction à la notion de service à grande échelle . . . . .	17
1.2 Quelques exemples de services à grande échelle . . . . .	18
1.2.1 Les services de l'infrastructure réseau . . . . .	19
1.2.1.1 La communication entre individus . . . . .	20
1.2.1.2 Les sources d'informations . . . . .	21
1.2.1.3 Les sources d'applications . . . . .	23
1.3 Caractéristiques . . . . .	24
1.3.1 La variété des modes d'interactions . . . . .	26
1.3.2 La variété des intervenants . . . . .	26
1.3.3 La variété des ressources . . . . .	27
1.3.4 La variété des relais . . . . .	28
1.4 Supports pour la conception, le déploiement et l'utilisation . . . . .	29
1.4.1 Les objets répartis . . . . .	29
1.4.2 Architectures réparties ouvertes . . . . .	32
1.4.3 Interfaces Homme-Machine . . . . .	33
1.5 Résumé . . . . .	34
<b>2 Etude et critiques de l'infrastructure Corba</b>	<b>35</b>
2.1 Une vision globale . . . . .	35
2.1.1 La genèse de Corba . . . . .	35
2.1.2 L'architecture globale de l'OMG . . . . .	36
2.1.3 Le langage de description d'interface IDL . . . . .	37
2.1.4 Les projections vers les langages de programmation . . . . .	39
2.2 Le négociateur de requêtes à objets . . . . .	40
2.2.1 L'architecture . . . . .	40
2.2.2 Les référentiels . . . . .	41
2.2.3 L'interopérabilité . . . . .	42
2.2.4 Quelques exemples d'ORBs . . . . .	42
2.3 Un exemple de système bancaire . . . . .	43
2.3.1 L'architecture de ce service . . . . .	44
2.3.2 Le modèle objet . . . . .	45
2.3.3 La spécification en IDL . . . . .	46
2.3.4 Le processus de développement et de déploiement . . . . .	48

2.4	La réalisation des serveurs et des clients . . . . .	50
2.4.1	La projection des interfaces IDL . . . . .	50
2.4.2	L'implantation des classes d'objets . . . . .	52
2.4.3	La création et destruction des objets . . . . .	53
2.4.4	L'initialisation des serveurs . . . . .	54
2.4.5	L'invocation d'opérations sur des objets distants . . . . .	54
2.4.6	La navigation dans le graphe . . . . .	56
2.5	La liaison entre les serveurs et les clients . . . . .	57
2.5.1	La liaison directe . . . . .	57
2.5.2	Les services d'annuaire . . . . .	57
2.5.3	Le service de nommage . . . . .	58
2.5.4	Le scénario d'utilisation du service de nommage . . . . .	60
2.5.4.1	Référencer un contexte initial . . . . .	61
2.5.4.2	Enregistrer une référence . . . . .	61
2.5.4.3	Obtenir une référence . . . . .	62
2.5.4.4	Configurer et administrer . . . . .	62
2.6	Critiques et conclusion . . . . .	63
2.6.1	La complexité de Corba . . . . .	63
2.6.2	L'évolutivité des services . . . . .	63
2.6.3	Les mécanismes dynamiques de Corba . . . . .	64
2.6.4	Vers une approche générique et dynamique . . . . .	66
<b>3</b>	<b>CorbaScript : un langage de script pour Corba</b>	<b>69</b>
3.1	Pourquoi des scripts? . . . . .	69
3.1.1	Interactivité et interprétation . . . . .	70
3.1.2	Survol des environnements de script . . . . .	70
3.1.2.1	Les systèmes d'exploitation . . . . .	70
3.1.2.2	Les langages de programmation . . . . .	71
3.1.2.3	Les bases de données . . . . .	71
3.1.2.4	Les applications généralistes . . . . .	72
3.1.3	Des scripts pour les objets répartis Corba . . . . .	72
3.1.4	Liaisons entre les langages existants et les objets répartis . . . . .	74
3.1.4.1	Projection vers un langage de script . . . . .	75
3.1.4.2	Intégration de l'invocation dynamique dans un langage de script . . . . .	77
3.1.5	Pourquoi un nouveau langage? . . . . .	78
3.2	Survol du langage CorbaScript . . . . .	79
3.2.1	Les fonctionnalités de CorbaScript . . . . .	79
3.2.2	Les expressions CorbaScript . . . . .	81
3.2.2.1	Les valeurs et types de base . . . . .	81
3.2.2.2	Les variables . . . . .	82
3.2.2.3	Les opérateurs . . . . .	82
3.2.2.4	Les tableaux . . . . .	83
3.2.2.5	Les types IDL . . . . .	83
3.2.2.6	Les objets répartis Corba . . . . .	84
3.2.3	La structuration des scripts . . . . .	86
3.2.3.1	L'affichage . . . . .	86
3.2.3.2	L'alternative . . . . .	86
3.2.3.3	La boucle conditionnelle . . . . .	87

3.2.3.4	L'itération . . . . .	87
3.2.3.5	Les exceptions . . . . .	87
3.2.3.6	Les procédures . . . . .	88
3.2.3.7	Les modules . . . . .	89
3.2.4	Des classes et des instances . . . . .	91
3.2.4.1	Quelques précisions sur les concepts objet de CorbaScript . . . . .	91
3.2.4.2	Un exemple simple d'utilisation des classes . . . . .	92
3.2.4.3	L'héritage et la recherche des méthodes . . . . .	93
3.2.4.4	L'implantation d'objets Corba . . . . .	94
3.2.5	Quelques exemples . . . . .	95
3.2.5.1	Référencer le service de nommage . . . . .	95
3.2.5.2	Enregistrer un objet dans un contexte . . . . .	96
3.2.5.3	Obtenir un objet depuis un contexte . . . . .	96
3.2.5.4	Appliquer des traitements sur un compte . . . . .	97
3.2.5.5	Parcourir le graphe des objets du service bancaire . . . . .	97
3.2.5.6	Fédérer des services de nommage . . . . .	98
3.2.6	Résumé des fonctionnalités de CorbaScript . . . . .	100
3.3	Implantation . . . . .	101
3.3.1	L'architecture . . . . .	101
3.3.2	La hiérarchie des classes . . . . .	103
3.3.2.1	La machine virtuelle de l'interpréteur . . . . .	103
3.3.2.2	Les valeurs de base CorbaScript . . . . .	105
3.3.2.3	Les exceptions CorbaScript . . . . .	105
3.3.2.4	Les objets CorbaScript nommés . . . . .	105
3.3.3	Le fonctionnement de l'interpréteur . . . . .	106
3.3.4	La liaison avec Corba . . . . .	108
3.3.4.1	Le cache du référentiel des interfaces . . . . .	109
3.3.4.2	L'interface d'invocation dynamique ou DII . . . . .	109
3.3.4.3	L'interface de squelettes dynamiques ou DSI . . . . .	110
3.3.5	Synthèse et performance . . . . .	111
3.4	Conclusion . . . . .	112
<b>4</b>	<b>CorbaWeb : l'intégration des objets Corba dans le WWW</b>	<b>115</b>
4.1	Le World Wide Web . . . . .	115
4.1.1	Les principes . . . . .	115
4.1.2	Les ressources et leur désignation . . . . .	116
4.1.3	L'interface universelle d'accès . . . . .	118
4.1.4	Un serveur général de ressources . . . . .	119
4.1.5	Synthèse et critiques . . . . .	120
4.2	Des objets Corba dans le WWW . . . . .	122
4.2.1	L'intégration du WWW et de Corba . . . . .	122
4.2.2	Interopérabilité des protocoles HTTP et IIOP . . . . .	123
4.2.3	Des objets dans les serveurs . . . . .	124
4.2.3.1	Des ressources basées sur les technologies objet . . . . .	125
4.2.3.2	Des passerelles Corba . . . . .	125
4.2.3.3	Des documents dynamiques . . . . .	127
4.2.4	Des applets Corba dans le WWW . . . . .	129
4.2.5	Synthèse . . . . .	131
4.3	L'environnement CorbaWeb . . . . .	132

4.3.1	Les principes de CorbaWeb . . . . .	133
4.3.1.1	L'architecture CorbaWeb . . . . .	133
4.3.1.2	La correspondance entre le langage IDL et le langage HTML . . . . .	135
4.3.1.3	Les modules et variables utilitaires de CorbaWeb . . . . .	137
4.3.2	Des ressources WWW dynamiques . . . . .	141
4.3.2.1	Des documents HTML dynamiques . . . . .	141
4.3.2.2	Des scripts d'accès à des objets Corba . . . . .	143
4.3.2.3	Les limites de cette approche . . . . .	147
4.3.3	Des méta scripts . . . . .	147
4.3.3.1	Le méta script d'interface . . . . .	148
4.3.3.2	Le méta script d'exécution d'opération . . . . .	151
4.3.3.3	Le méta script d'exécution de script . . . . .	152
4.3.3.4	Le méta script de représentation . . . . .	155
4.3.4	L'illustration de la navigation dans le service bancaire . . . . .	160
4.3.4.1	La représentation d'un contexte de nommage . . . . .	160
4.3.4.2	La représentation d'une banque . . . . .	162
4.3.4.3	La représentation d'une agence . . . . .	164
4.3.4.4	La représentation d'un client . . . . .	165
4.3.4.5	La représentation d'un compte . . . . .	166
4.3.5	Implantation et perspectives . . . . .	167
4.3.5.1	L'adaptation de l'interpréteur CorbaScript . . . . .	167
4.3.5.2	Les différentes solutions de liaison avec le WWW . . . . .	168
4.3.5.3	Perspectives et travaux en cours . . . . .	170
4.4	Conclusion . . . . .	173
4.4.1	Une infrastructure globale de services en ligne . . . . .	173
4.4.2	La proposition CorbaWeb . . . . .	174
<b>Conclusion</b>		<b>177</b>
	Contexte . . . . .	177
	Propositions . . . . .	177
	Perspectives . . . . .	178
	Le langage CorbaScript . . . . .	178
	Des applications . . . . .	179
	Des pistes de recherche . . . . .	180
<b>A Exemples</b>		<b>183</b>
A.1	Un service de nommage . . . . .	183
A.2	Un service de calcul sur les nombres premiers . . . . .	188
A.3	Un système d'informations hospitalier multimédia . . . . .	191
<b>Bibliographie</b>		<b>200</b>

# Table des figures

1.1	La messagerie électronique MIME . . . . .	21
1.2	La navigation dans un graphe de documents du WWW . . . . .	22
1.3	Un service bancaire réparti . . . . .	24
1.4	Un service à grande échelle . . . . .	25
1.5	Transparence de la distribution . . . . .	30
1.6	Les domaines impliqués dans le support de services à grande échelle . . . . .	34
2.1	L'O.M.A. : la vision globale de l'O.M.G. . . . .	37
2.2	La séparation de l'interface et de l'implantation d'un objet . . . . .	38
2.3	L'interface d'une matrice . . . . .	38
2.4	Les types de données de l'OMG-IDL . . . . .	39
2.5	L'architecture Corba . . . . .	40
2.6	L'architecture répartie du service bancaire . . . . .	44
2.7	Le modèle d'un système bancaire . . . . .	45
2.8	La spécification du service bancaire en OMG-IDL . . . . .	47
2.9	Le processus de développement et déploiement du service bancaire avec Corba . . . . .	48
2.10	La hiérarchie des classes C++ générée par le compilateur IDL . . . . .	51
2.11	La transparence de l'invocation d'un objet distant . . . . .	55
2.12	La fonction C++ d'affichage de tous les comptes d'une banque . . . . .	56
2.13	L'interface IDL du service de nommage . . . . .	59
2.14	Le scénario d'utilisation d'un service de nommage . . . . .	61
2.15	L'environnement générique et dynamique GOODE . . . . .	67
3.1	L'interface IDL grille . . . . .	74
3.2	La procédure CorbaScript d'affichage de tous les comptes d'une banque . . . . .	97
3.3	Un exemple de fédération de contextes de nommage . . . . .	98
3.4	Le module de fédération de services de nommage (1/2) . . . . .	99
3.5	Le module de fédération de services de nommage (2/2) . . . . .	100
3.6	L'architecture de l'interpréteur CorbaScript . . . . .	102
3.7	La hiérarchie des classes d'implantation de CorbaScript . . . . .	104
3.8	La machine virtuelle d'exécution de CorbaScript . . . . .	107
4.1	Les principes de base du WWW . . . . .	116
4.2	Le navigateur WWW universel . . . . .	118
4.3	Les composants du WWW . . . . .	119
4.4	Où intégrer les objets Corba dans le WWW? . . . . .	123
4.5	L'intégration des protocoles de communication . . . . .	124
4.6	Une passerelle Corba statique et spécifique . . . . .	126
4.7	Une passerelle Corba générique . . . . .	126
4.8	La génération de documents dynamiques . . . . .	128

4.9	Des applets Corba dans le WWW . . . . .	130
4.10	L'environnement de CorbaWeb . . . . .	133
4.11	La liste des comptes débiteurs d'une agence . . . . .	142
4.12	La visualisation de la liste des comptes débiteurs d'une agence . . . . .	143
4.13	L'accès à un objet grille à travers le WWW . . . . .	144
4.14	L'invocation d'une opération sur une grille . . . . .	145
4.15	Le script d'interfaçage d'une grille dans le WWW . . . . .	146
4.16	Le script d'invocation des opérations sur une grille . . . . .	147
4.17	Une interface HTML sur une Agence . . . . .	149
4.18	Le méta script de génération automatique d'interfaces HTML . . . . .	150
4.19	L'interface WWW d'accès à tout objet Corba . . . . .	151
4.20	L'exécution d'une opération sur la Banque . . . . .	152
4.21	Le méta script d'exécution d'opérations . . . . .	152
4.22	Le méta script d'exécution de script . . . . .	153
4.23	L'exécution d'un script à travers le WWW . . . . .	154
4.24	Le résultat de l'exécution d'un script . . . . .	154
4.25	Le fonctionnement du méta script de représentation . . . . .	158
4.26	Le méta script de représentation des objets Corba . . . . .	159
4.27	Le méta script de représentation . . . . .	160
4.28	Le contenu d'un contexte de nommage Corba . . . . .	161
4.29	La navigation dans un contexte de nommage Corba . . . . .	161
4.30	Le script de représentation d'un contexte de nommage Corba . . . . .	162
4.31	La navigation dans un objet Banque . . . . .	163
4.32	Le script de représentation d'un objet Banque . . . . .	163
4.33	La navigation dans un objet Agence . . . . .	164
4.34	Le script de représentation d'un objet Agence . . . . .	164
4.35	La navigation dans un objet Client . . . . .	165
4.36	Le script de représentation d'un objet Client . . . . .	165
4.37	La navigation dans un objet Compte . . . . .	166
4.38	Le script de représentation d'un objet Compte . . . . .	166
4.39	CorbaWeb : l'adaptation de CorbaScript . . . . .	167
4.40	Diverses implantations de CorbaWeb . . . . .	169
4.41	Schéma d'une porte d'accès . . . . .	171
4.42	La coopération de deux passerelles CorbaWeb réparties . . . . .	172
A.1	La représentation d'un objet computer . . . . .	190
A.2	L'exécution d'une opération d'un objet computer . . . . .	190
A.3	Le système d'informations hospitalier multimédia . . . . .	191
A.4	La représentation d'un objet HOSPITALIER::hopital . . . . .	195
A.5	La représentation d'un objet HOSPITALIER::patient . . . . .	196
A.6	La représentation d'un objet HOSPITALIER::examen . . . . .	197
A.7	La représentation d'un objet HOSPITALIER::radiographie . . . . .	199

# Liste des tableaux

1.1	Les 7 couches du modèle OSI . . . . .	19
3.1	Liste des types de base de CorbaScript . . . . .	81
4.1	URL : Unification de la désignation des ressources Internet . . . . .	117
4.2	Les règles de correspondance entre le langage IDL et le langage HTML . . .	136
4.3	Le module HTML . . . . .	138
4.4	Le module CW . . . . .	139
4.5	Les variables HTTP . . . . .	140



# Introduction

Les grandes organisations telles que les compagnies de l'industrie de l'automobile, de l'aérospatiale, des télécommunications, de la défense et des finances ont besoin de très larges systèmes à grande échelle pour supporter leurs complexes activités. La majorité de ces applications, que nous désignons dans ce mémoire sous le vocable de services, sont distribuées sur de multiples machines fortement réparties géographiquement et très hétérogènes. A travers ces vastes services, les utilisateurs coopèrent à la réalisation des multiples activités des entreprises. Ces tâches évoluent fortement dans le temps pour suivre les changements dans les législations locales, nationales et même internationales, ou bien pour répondre à de nouvelles opportunités du marché. Les concepteurs de ces services doivent donc pouvoir répondre rapidement à ces changements en créant de nouvelles fonctionnalités ou en recombinaison celles existantes.

De manière générale, la réalisation d'un service à grande échelle est une tâche complexe car les développeurs doivent gérer de front un grand nombre de technologies telles que les méthodes de conception, les supports répartis ainsi que les interfaces Homme-Machine.

L'approche orientée objet répartie apporte des solutions à la conception et au support de tels systèmes. Les recherches académiques et industrielles menées dans les quinze dernières années sur les systèmes d'exploitation à objets distribués (Amoeba, SOS<sup>1</sup>, Guide) et plus récemment les efforts de normalisation des infrastructures de communication orientées objet (ODP<sup>3</sup>, CORBA<sup>4</sup>, TINA<sup>5</sup>) ont démontré la faisabilité du support de grandes applications. Celles-ci sont composées d'un grand nombre d'instances et de classes, mettant en oeuvre une grande variété et complexité d'interactions entre les objets. Elles supportent un grand nombre d'utilisateurs répartis sur de nombreuses machines et elles cachent les problèmes d'hétérogénéité (diversité du matériel, des environnements d'exécution ou des langages de programmation).

Si beaucoup d'attention a été portée à la «technologie» pour construire des systèmes complexes à grande échelle, il est à noter que les expériences «réelles» dans l'industrie sont encore peu nombreuses. Citons en exemple, le projet IRIDIUM de Motorola [Mot96] projetant la mise en place d'un réseau de téléphonie cellulaire mondial couvert par un réseau maillé d'une soixantaine de satellites. Le contrôle de ces satellites sera assuré par une infrastructure d'objets distribués conforme à la norme CORBA.

Ce manque de mise en pratique provient de deux constats. L'approche objet reste encore mal appréhendée par l'industrie. Néanmoins, les technologies objets y prennent une place de plus en plus importante. En outre, les principes de conception des services répartis (même à base d'objets) ne favorisent pas la réactivité des utilisateurs à répondre à leurs besoins spécifiques et évolutifs. Actuellement, l'utilisateur final n'a accès aux services qu'à

---

1. SOS : SOMIW<sup>2</sup> Operating System

3. ODP : Open Distributed Processing

4. CORBA : Common Object Request Broker Architecture

5. TINA : Telecommunications Information Networking Architecture

travers les outils conçus et prévus par les informaticiens. Si l'utilisateur désire réaliser une nouvelle tâche pour laquelle aucun outil n'est prévu, il doit attendre que le service informatique produise l'outil nécessaire. Cette production peut être lente car les infrastructures de support d'objets sont complexes à mettre en œuvre. Le point précédent restreint alors fortement l'accès aux services et brime l'intelligence et la créativité des utilisateurs.

De cette première réflexion se dégage la notion centrale de cette thèse : donner des «éléments techniques et méthodologiques pour favoriser l'accès à grande échelle à des services et objets répartis».

## Proposition

Notre objectif est de faciliter et de multiplier les possibilités d'accès à des services en fournissant *les éléments nécessaires pour que des postes clients puissent dialoguer dynamiquement avec des applicatifs distribués*. Nos travaux sont basés sur un support d'objets répartis (CORBA) pour la conception et la réalisation des services. CORBA nous permet de bénéficier de tous les apports des technologies objets pour la conception de services et de nous abstraire des problèmes inhérents à la réalisation de ceux-ci : le nommage, la distribution, l'hétérogénéité, la communication entre objets, ...

Cependant, CORBA est très complexe à utiliser et n'apporte pas de solution pour le déploiement, l'administration ou encore l'accès à grande échelle à des services disponibles sur un bus à objets. Pour ces différents cas de figure, l'approche statique (à l'aide des souches générées) n'est pas adaptée. La ligne directrice de nos travaux est d'utiliser les fonctionnalités dynamiques offertes par CORBA pour répondre aux problèmes présentés ci-dessus.

GOODE<sup>6</sup> est notre environnement générique et dynamique qui permet l'accès et l'utilisation de tout service CORBA. La problématique - technique et méthodologique - est de dissocier l'accès et la présentation de la partie fonctionnelle des applicatifs. Actuellement, GOODE est constitué de deux outils : – CorbaScript – un interpréteur de script qui permet d'invoquer toute opération définie sur un objet CORBA et – CorbaWeb – une passerelle générique d'accès aux objets CORBA au travers d'un navigateur World Wide Web<sup>7</sup>.

**CorbaScript** : Notre proposition est de favoriser l'exploitation de services orientés objet à travers un langage de script interprété, interactif, orienté objet et fournissant dynamiquement l'accès à tous les objets disponibles sur un bus Corba. Pour cela, nous utilisons l'interface d'invocation dynamique (DII<sup>8</sup>) conjointement avec les informations extraites du référentiel des interfaces (IR<sup>9</sup>).

Ce langage portable et interprété permet de naviguer, d'exécuter des méthodes, et de visualiser un ensemble distribué d'objets. Il sert d'intermédiaire entre les différentes utilisations et les applications distribuées en gérant les problèmes d'accès dynamiques (typage dynamique), de représentation et aussi d'hétérogénéité.

L'exploitation des services est alors grandement facilitée par le développement rapide de scripts répondant aux attentes des utilisateurs finaux mais aussi des administrateurs et informaticiens responsables de ces services.

---

6. GOODE: Generic Object-Oriented Dynamic Environment

7. pour simplifier, nous utiliserons aussi les termes *WWW* ou *Web*.

8. DII: Dynamic Invocation Interface

9. IR: Interface Repository

**CorbaWeb** : La passerelle CorbaWeb, basée sur CorbaScript, offre à la fois un moyen d'accès via le Web et un moyen d'extension du Web à de nouveaux services, et cela de manière générique contrairement aux solutions fermées proposées jusqu'ici. Cette intégration des mondes Web et CORBA fournit un nommage des objets par des URL<sup>10</sup> étendues. Elle permet ainsi la navigation dans des graphes complexes d'objets Corba, l'invocation d'opérations sur ces objets et la découverte des interfaces IDL<sup>11</sup> des objets. Un navigateur Web tout à fait standard peut accéder et dialoguer, via des objets, avec des applications distribuées.

Cet environnement permet ainsi la création de nouveaux services pour le WWW et l'accès à des services orientés objet Corba déjà créés et non conçus pour le WWW.

En résumé, notre proposition vise à :

- 1) permettre l'accès aux nombreuses sources d'informations et de services à partir de postes clients très divers,
- 2) généraliser un mode d'accès dynamique permettant de découpler les clients des applicatifs qu'ils utilisent,
- 3) développer l'intégration des technologies largement répandues pour multiplier les points d'accès.

## Plan détaillé

Cette thèse comprend quatre chapitres et une annexe :

- Le premier chapitre fait un tour d'horizon sur la notion de services à grande échelle. Dans un premier temps, nous introduisons la notion de service à grande échelle et nous l'illustrons sur divers exemples. Nous décrivons un service bancaire qui nous suivra tout au long de ce mémoire. A partir de ces exemples variés, nous dégagons les caractéristiques principales des services à grande échelle. Nous discutons ensuite des infrastructures de support de ces services. En conclusion, nous résumons les trois axes principaux à prendre en compte pour la réalisation d'un service : la conception, le déploiement et l'utilisation.
- Le deuxième chapitre est consacré à l'étude et à la critique du bus à objets répartis Corba sur lequel nous avons mené nos travaux.

Nous présentons la vision globale proposée par l'OMG<sup>12</sup> et les composantes du bus à objets répartis, l'Object Request Broker. Nous illustrons ensuite la complexité et les difficultés rencontrées pour réaliser au dessus de ce bus le service bancaire réparti présenté dans le premier chapitre. Cette étude est concrètement illustrée par des fragments de code IDL et C++ et aussi par la mise en œuvre du service de nommage standard de Corba. Dans la conclusion de ce chapitre, nous dégagons des éléments critiques sur le modèle de conception des applications utilisant Corba. Enfin, nous décrivons les mécanismes dynamiques que nous mettrons en œuvre dans les chapitres suivants pour favoriser l'accès aux objets Corba.

---

10. URL : Uniform Resource Locator ou Localisateur Uniforme de Ressources

11. IDL : Interface Definition Language

12. OMG : Object Management Group

- Dans le troisième chapitre, nous présentons CorbaScript, notre proposition de langage de script pour les objets Corba.

Dans un premier temps, nous analysons les intérêts d'un langage de script pour les objets répartis et nous en énumérons les bénéfices et les contextes d'utilisation possibles. Ensuite, nous survolons les constructions syntaxiques et sémantiques du langage CorbaScript. Nous illustrons ses possibilités dans le contexte du service bancaire et du service de nommage du second chapitre. Les points majeurs de l'implantation modulaire de l'interpréteur CorbaScript sont ensuite présentés. En conclusion, nous résumons les contextes dans lesquels le langage CorbaScript procure la souplesse et la flexibilité pour accéder à tout objet Corba.

- Le quatrième chapitre présente l'environnement CorbaWeb pour l'intégration générique et dynamique des objets Corba dans le WWW.

Après un rappel des éléments caractérisant le WWW, nous discutons de l'intérêt d'intégrer des objets dans le WWW et présentons les diverses approches possibles. Suite à cet état de l'art, nous présentons l'environnement CorbaWeb ainsi que l'ensemble de ses fonctionnalités pour favoriser l'intégration générique d'objets Corba dans le WWW. Nous illustrons sa mise en œuvre dans le cadre des services présentés précédemment. Finalement, nous concluons sur les apports de l'environnement CorbaWeb pour concevoir, déployer et accéder à des services à grande échelle.

- Une annexe présente divers exemples illustrant l'utilisation de notre environnement : l'implantation d'un service de nommage en CorbaScript, un service de calcul de nombres premiers accessible par CorbaWeb et un système d'information multimédia dans un contexte hospitalier.

## Contexte du travail

Mes travaux de recherche sont entrepris au sein de l'équipe GOAL (Groupe Objets et Acteurs de Lille) du LIFL (Laboratoire d'Informatique Fondamentale de Lille) dirigé par le Professeur Jean-Marc Geib.

Les travaux de cette équipe sont liés au projet OSACA (Open Software Architecture for Cooperative Applications) du programme Ganymède du Contrat de plan Etat-Région Nord/Pas-de-Calais. Ce projet vise à la conception d'une infrastructure ouverte pour les applications coopératives (applications dédiées à des groupes d'utilisateurs).

# Chapitre 1

## Les services à grande échelle

Ce chapitre fait un tour d'horizon de la notion de service à grande échelle. Nous illustrons quelques exemples de services actuellement disponibles pour déterminer les principales caractéristiques de ces services. Nous présentons ensuite des infrastructures pour les supporter. Enfin, nous résumons les trois axes principaux à prendre en compte pour la réalisation de services : la conception, le déploiement informatique et l'utilisation du service.

### 1.1 Introduction à la notion de service à grande échelle

Lors de ces dernières années, l'informatique a fait de grandes avancées sur le plan technologique : augmentation de la puissance des micro-processeurs, forte diminution du prix des micro-ordinateurs, déploiement de réseaux rapides à large échelle (régional, national) et finalement fédération de ces réseaux dans des structures mondiales de type Internet.

Ces avancées technologiques et la démocratisation de l'outil informatique ont permis l'éclosion d'une nouvelle catégorie d'applications : les services à grande échelle comme, par exemple, la messagerie électronique ou le World Wide Web. Ces services fournissent une grande variété de ressources (des sources d'informations et d'applications) utilisées par de nombreux utilisateurs répartis sur un ensemble de machines fortement hétérogènes.

De plus, ces services se distinguent des applications informatiques «classiques» par la diversité et la quantité des éléments technologiques et humains mis en œuvre. Aujourd'hui, un des défis les plus importants de l'informatique est de maîtriser la conception et le déploiement de telles infrastructures pour offrir un large éventail de services accessibles aussi bien par le grand public que par des utilisateurs «experts».

Pour définir et caractériser la notion de service informatisé à grande échelle, il nous paraît intéressant de faire l'analogie avec la notion de service dans le monde «réel». Dans ce mémoire, le mot «service» est utilisé dans le sens général suivant : «*une fonction d'utilité commune, publique et l'activité organisée qui la remplit*».

Cette définition s'applique à de nombreuses activités humaines comme la communication entre les individus (les postes, les télécommunications), la diffusion d'informations et/ou du savoir (la presse, la télévision, l'éducation) ou bien la sécurité, la santé et le confort des individus. Ces activités ont une fonction d'utilité commune à un grand nombre d'individus répartis géographiquement sur une région, un pays, un continent ou bien le monde.

Elles sont gérées par des organismes prestataires de services (par exemple La Poste, France Telecom, la Sécurité Sociale) dont l'infrastructure est répartie géographiquement

mais aussi administrativement. De plus, les organismes en charge d'une certaine fonction doivent souvent collaborer entre eux : dans le monde des télécommunications, cela se traduit par l'interconnexion des réseaux nationaux.

Cette coopération impose de fixer des règles de fonctionnement, appliquées par ces organismes. Ces services définissent aussi des procédures et des outils à destination des usagers comme la numérotation téléphonique, l'annuaire et le combiné de téléphone.

Transposons cette vision au domaine de l'informatique : un service à grande échelle offre alors un ensemble de fonctionnalités – des informations, des traitements sur ces informations – à un grand nombre d'utilisateurs répartis géographiquement. A travers ces services, les utilisateurs réalisent des activités communes par l'intermédiaire de leur ordinateur. Ces activités nécessitent de partager des informations et des ressources communes.

L'infrastructure d'un service est composée des différents éléments logiciels et matériels mis en œuvre pour réaliser le service : les interfaces d'interactions pour les utilisateurs, les logiciels de services (ou serveurs), les machines, les espaces de stockage, les réseaux, les protocoles de communication/dialogue entre les machines mais aussi entre les logiciels. La coopération à grande échelle entre ces éléments logiciels et matériels ne peut être atteinte que grâce à des efforts de normalisation, comme par exemple pour régler les problèmes d'hétérogénéité.

La réalisation de services à grande échelle est très complexe car elle est à l'intersection de divers domaines de l'informatique de la téléphonie et des loisirs. Elle nécessite de prendre en compte l'organisation des activités humaines : la planification des tâches et la coopération des individus. De nombreux travaux sont menés dans ce sens pour définir des modèles régissant l'activité humaine comme par exemple le travail coopératif [EW94]. Elle implique de structurer et de modéliser l'information (au sens large) partagée par les utilisateurs à l'aide de méthodologies de conception des systèmes d'informations (i.e. relationnel, objet). Un service à grande échelle met en œuvre une informatique distribuée composée de nombreuses machines et logiciels hétérogènes et communiquant à travers les réseaux. Les utilisateurs accèdent aux services à travers des interfaces Homme-Machines impliquant des problèmes d'ergonomie et d'utilisabilité.

Depuis longtemps, l'informatique répartie est un sujet de recherche très étudié : propositions de réseaux toujours plus rapides, plus fiables ou plus larges (ATM [Ro195], IPV6 [CD95]), paradigmes de communication entre logiciels (envoi de messages, appels de traitement distant et objets répartis [OHE95, OHE96]), systèmes d'exploitation distribués (Amoeba [MRTS90], SOS [SGH<sup>+</sup>89], Guide [BBD<sup>+</sup>91], Spring [RHKP95]) et bus logiciels ou «middleware» (DCE [Cha92, RKF92], CORBA [OMG95b, YD96]). Ce domaine offre une grande quantité de propositions et de choix pour supporter les services à grande échelle.

**Ainsi, le domaine de la réalisation d'un service à grande échelle englobe tout un ensemble de problématiques diverses. Dans cette thèse, nous nous consacrons à l'étude des moyens d'accès à des services et à la proposition de deux nouveaux moyens pour favoriser l'accès à des services composés d'objets répartis [MGG96e, Mer96].**

## 1.2 Quelques exemples de services à grande échelle

Nous dégageons quatre grandes catégories de services à grande échelle selon la nature de leur fonction : les services de l'infrastructure réseau, les services de communication

entre individus, les sources de diffusion d'informations et les sources d'applications. Nous présentons les normes et protocoles permettant à ces services de fonctionner à grande échelle, c'est à dire accessibles et utilisables par un maximum d'individus depuis n'importe quelle machine.

### 1.2.1 Les services de l'infrastructure réseau

Dans un système distribué, les machines communiquent par l'intermédiaire d'un réseau par échange de messages [Tan90]. Le réseau Internet est composé des supports physiques utilisés pour cette communication (par exemple , les câbles et équipements réseaux Ethernet) et des protocoles d'échange de messages sur ces câbles (par exemple IP<sup>1</sup> [Pos81], UDP<sup>2</sup> [Pos80], TCP<sup>3</sup>/IP [Pos80]). Afin d'assurer le routage des messages entre machines, celles-ci sont désignées par une adresse unique sur le réseau. De plus, un nommage symbolique des machines permet une désignation plus facilement compréhensible par des utilisateurs. La traduction des noms symboliques en adresses physiques est assurée par un service de localisation des machines.

La diversité des équipements rencontrés implique forcément l'hétérogénéité et donc la possible incompatibilité de ceux-ci. Afin de résoudre ce problème, le modèle OSI<sup>4</sup> [DZ83] spécifie 7 couches (cf. table 1.1) pour structurer et normaliser le service de communication.

	Couche	Description	Exemples
7	Application	Support des besoins d'applications spécifiques et définit souvent l'interface pour un service.	FTP, SMTP, NNTP, HTTP
6	Présentation	Codage des données dans un format indépendant des représentations utilisées par chaque ordinateur.	XDR, MIME, encryptage
5	Session	Communication entre processus et recouvrement des erreurs pour les communications en mode connecté.	RPC
4	Transport	Communication par envoi de messages vers des ports de communication.	TCP, UDP
3	Réseau	Transfert et routage de paquets de données entre ordinateurs sur des réseaux physiques différents.	X25, IP
2	Liaison de données	Transfert sans erreur entre équipements sur un même réseau physique.	Ethernet
1	Physique	Equipements physiques (cartes réseau, câbles) et type de signal utilisé.	X.21, Ethernet, FDDI, ATM

TAB. 1.1 - Les 7 couches du modèle OSI

Le standard de fait Internet de fédération des réseaux à l'échelle mondiale est la meilleure illustration d'une implantation possible de ce modèle. Il fournit des protocoles de communication entre machines, entre applications distribuées et des protocoles de codage de l'information. Le service de désignation symbolique d'Internet s'appelle alors le *Domain Naming Service* [Moc87a, Moc87b, Pos94]. Ce service maintient des tables d'associations entre des noms symboliques de machine (par exemple *malibu.lifl.fr*) et les adresses physiques sur le réseau. La simplicité de ce service n'est qu'apparente car la gestion à l'échelle

1. IP : Internet Protocol

2. UDP : User Datagram Protocol

3. TCP : Transmission Control Protocol

4. OSI: Open System Interconnection

mondiale de ces tables d'association se doit d'être efficace car elle ne doit pas pénaliser tous les autres services bâtis au dessus. Les tâches relatives au DNS sont : le découpage de l'espace de désignation en *domaine* et *sous-domaine* pour décentraliser l'administration et l'attribution des noms, la coopération d'une hiérarchie de serveurs DNS pour résoudre les recherches, la gestion d'un cache des adresses déjà résolues afin de diminuer les temps de recherche.

Les services de l'infrastructure réseau regroupent donc toutes les fonctions liées à la communication sur le réseau : la localisation des sites, le transport fiable et efficace des messages. La définition de normes nous permet de localiser et de communiquer avec n'importe quelle machine depuis n'importe quel autre site. Ces normes règlent ainsi les problèmes d'hétérogénéité et sont le socle sur lequel tous les autres services à grande échelle peuvent être bâtis.

### 1.2.1.1 La communication entre individus

Cette catégorie englobe les services à grande échelle dont l'objectif est de médiatiser le dialogue, la communication et l'échange de messages écrits, auditifs ou bien visuels entre des individus. La communication entre les individus se décline selon deux modes : *synchrone* ou *asynchrone*.

Nous parlons de communication synchrone lorsque les individus sont connectés en même temps et communiquent en direct. Le service Internet «Talk» permet à deux utilisateurs de dialoguer sur le réseau. Les services IRC, quant à eux, offrent des discussions de groupes sur des thèmes variés. Les audio- et visioconférences sont encore très peu utilisées car elles nécessitent des bandes passantes réseaux importantes pour avoir une qualité acceptable et sont donc chères. Toutefois, elles sont promises à un grand avenir avec l'apparition et la baisse des coûts des réseaux rapides (FDDI et ATM).

Les communications asynchrones permettent à des usagers de converser en temps (plus ou moins) différé. A l'opposé du mode précédent, la présence de tous les intervenants à un instant donné n'est pas obligatoire. Les exemples les plus répandus sont la messagerie électronique et les forums de discussion (ou News).

Développé au début des années 80, la messagerie électronique [Ros93] permet de médiatiser l'acheminement de messages entre usagers au sein d'une organisation dans le cas d'un réseau local<sup>5</sup> ou de part le monde grâce à Internet. A travers des outils personnels tel que *mailtool*, les utilisateurs consultent les messages de leur boîte aux lettres. Ces outils leur permettent également de rédiger et d'envoyer des messages aux autres utilisateurs.

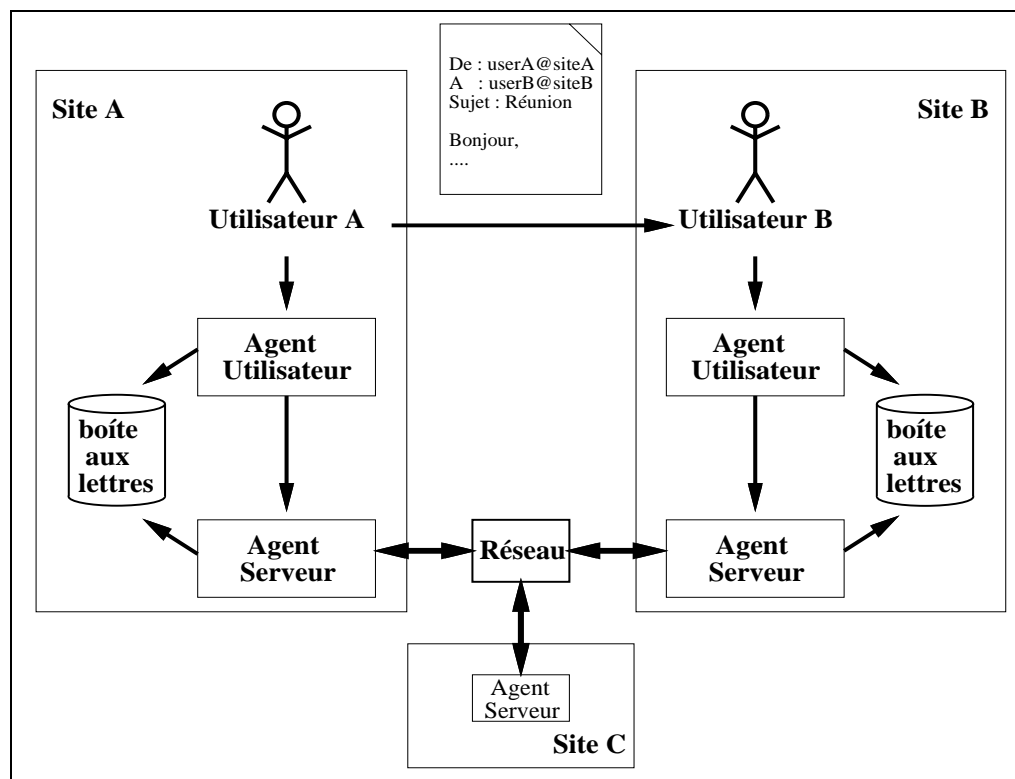
L'infrastructure sous-jacente de ce service est composée d'un ensemble de serveurs et de programmes utilisateurs qui coopèrent à l'acheminement des messages (cf. figure 1.1). Dans le contexte de la messagerie sur Internet (i.e. *email*), cette coopération est atteinte grâce à la définition du protocole d'échange des messages SMTP<sup>6</sup> [Pos82], de structuration des messages [Cro82] et de désignation des boîtes aux lettres des utilisateurs (i.e. leur adresse électronique). De plus, le format MIME<sup>7</sup> [BF93, Moo93] structure le contenu des messages multimédia : texte ascii, fichier postscript, image au format GIF, fichier sonore.

---

5. ou Intranet par opposition avec Internet.

6. SMTP : Simple Mail Transfer Protocol

7. MIME : Multipurpose Internet Mail Extensions

FIG. 1.1 - *La messagerie électronique MIME*

Pour la communication de groupes, on préférera les forums de discussions (ou News sur Internet). Grâce aux milliers de forums (ou « newsgroups »), des millions d'utilisateurs répartis dans le monde peuvent dialoguer, échanger des idées et des conseils, organiser leur travail sans se préoccuper par exemple des décalages horaires. Par l'intermédiaire d'un lecteur de News, l'utilisateur consulte et rédige des messages à destination de groupes de discussions. Ses messages sont alors diffusés à travers le monde via un réseau de serveurs de News dialoguant par le protocole NNTP<sup>8</sup> [KL86]. Le format MIME peut être utilisé pour transmettre des messages multimédias.

Un service à grande échelle nécessite la définition de protocoles normalisés pour faire interopérer des applications distribuées (leur permettre de dialoguer et de se comprendre). Dans le contexte de la communication entre individus, il est de plus nécessaire de pouvoir désigner ces individus (ou groupes) et de structurer leurs échanges de messages. Les protocoles SMTP, NNTP et le format MIME sont alors la réponse actuelle répondant à ces besoins dans le contexte à grande échelle d'Internet.

### 1.2.1.2 Les sources d'informations

Après la communication et le partage entre individus, l'accès à l'information est une autre fonction importante des services à grande échelle. Cette catégorie fournit aux usagers l'accès à des banques d'informations plus ou moins structurées depuis leur poste de travail.

Rapidement, à l'apparition des premiers systèmes distribués, les utilisateurs ont exprimé le besoin de partager et d'échanger leurs fichiers. FTP<sup>9</sup> [BBC<sup>+</sup>71, PR85] fut un

8. NNTP: Network News Transfer Protocol

9. FTP: File Transfer Protocol

des premiers services à offrir cette fonctionnalité : ce protocole permet la transmission de fichiers textes ou binaires entre deux machines. Puis, des systèmes tels que NFS<sup>10</sup> [Sun89] ont été développés pour rendre encore plus transparent ce partage en unifiant des systèmes de fichiers distribués dans une unique arborescence de fichiers. Couramment, NFS est plutôt utilisé sur des réseaux locaux mais il peut aussi s'employer à grande échelle car il repose sur les couches de communication d'Internet.

D'un autre côté, des services orientés documents se sont développés pour permettre aux utilisateurs d'accéder à de larges bases d'informations multimédia. Gopher[AML<sup>+</sup>93], créé à l'université du Minnesota, fut l'un des premiers services d'informations distribués. L'information est présentée sous forme d'une arborescence de menus. Les entrées pointent sur d'autres menus, sur des documents ou bien d'autres sites Gopher. Ainsi à travers une interface graphique, l'utilisateur navigue sur le réseau pour découvrir les informations qui l'intéressent et bien d'autres. Le navigateur envoie des requêtes au serveur à travers le protocole Gopher.

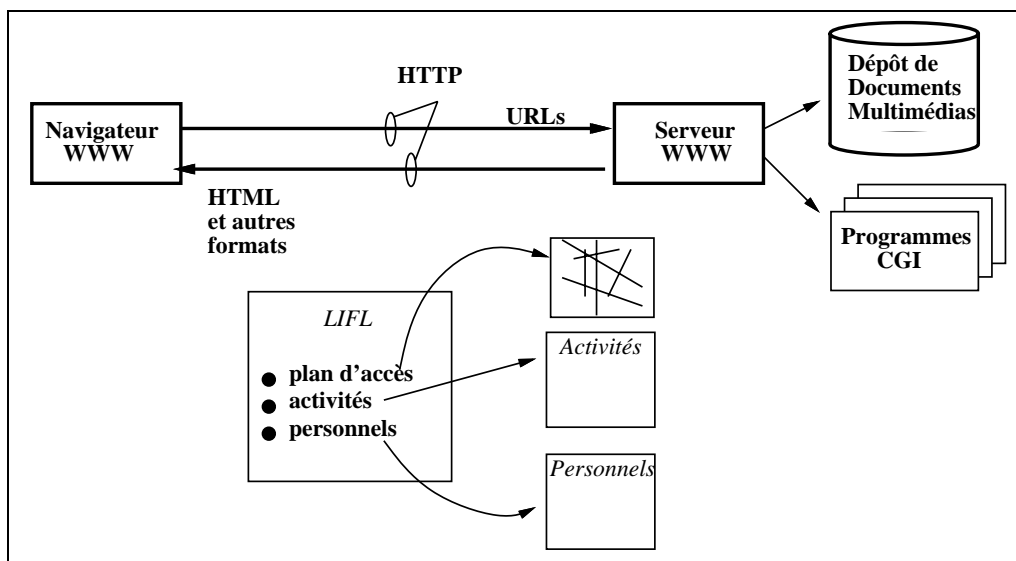


FIG. 1.2 - La navigation dans un graphe de documents du WWW

Plus récemment, le World Wide Web [BLC90] a repris ce principe de navigation en améliorant grandement la richesse et l'ergonomie de l'interface Homme-Machine. Dans le WWW (cf. figure 1.2), le navigateur communique avec les serveurs WWW par l'intermédiaire du protocole HTTP<sup>11</sup> [BLFF96]. Les documents sont codés avec le format MIME et sont désignés par des URL [Kun95, BLMM94]. **Nous reviendrons plus amplement sur le WWW dans le chapitre 4 où nous discuterons de l'intégration des objets Corba dans le WWW et de notre environnement CorbaWeb.**

Lorsqu'un utilisateur veut accéder à une source d'informations, il faut nécessairement qu'il connaisse sa localisation sur le réseau. Les services d'annuaires permettent de rechercher les sources d'informations à partir d'éléments les décrivant : thème, auteur, date, mots clés. De nombreuses variantes de ce type de services ont vu le jour sur Internet comme les services Wais, Archie, Veronica ou plus récemment les moteurs de recherche sur le World Wide Web tel qu'Altavista. Ces annuaires peuvent être complexes et mettent en œuvre d'immenses bases de données. Par exemple, le service Altavista parcourt tous les jours les

10. NFS : Network File System

11. HTTP : Hyper Text Transfer Protocol

documents disponibles sur le WWW et les messages postés dans les forums de discussions. Il indexe et stocke toutes ces informations dans une immense base de données accessible en ligne par le WWW.

Comme pour les services précédents, la définition de normes de protocoles, de formats et de désignations permettent la mise en place de sources d'informations à grande échelle accessibles aisément par tout utilisateur du réseau. Dans le WWW, les utilisateurs ne sont pas uniquement consommateurs, ils peuvent eux-mêmes assez rapidement rendre disponibles leurs propres documents. De plus, les utilisateurs peuvent retrouver les informations via les annuaires.

### 1.2.1.3 Les sources d'applications

Dans cette dernière catégorie, nous regroupons les services supportés par des applications distribuées. Elles offrent à l'utilisateur des services distants permettant, par exemple, d'effectuer un achat, de réserver un billet de transport, de consulter son compte bancaire, etc.

En France, l'exemple le plus répandu de telles sources d'applications est le Minitel. A travers un terminal semi-graphique, les utilisateurs accèdent à une grande variété de services. Sur Internet, il est aussi possible d'accéder à des applications fonctionnant en mode texte à travers le protocole Telnet [PR83]. Le WWW offre l'équivalent par l'intermédiaire des programmes CGI<sup>12</sup> [McC95, Rob96] exécutés sur les serveurs WWW. Ces applicatifs fournissent ainsi une interface utilisateur sur une base de données, un programme de calcul ou tout autre forme de traitement.

Mais ces solutions sont totalement centralisées : le poste client n'est qu'un terminal plus ou moins évolué (Telnet, Minitel ou navigateur Web) et tous les traitements sont réalisés dans l'applicatif serveur. De plus, les applicatifs sont développés spécifiquement pour s'interfacer avec un protocole d'accès (Telnet, Minitel ou CGI). Ils doivent réaliser toute la gestion des interactions avec les utilisateurs (contrôle de saisie, génération des écrans). Ils doivent aussi contenir tous les traitements dont les utilisateurs pourraient avoir besoin. En général, pour ces applicatifs, le concepteur ne peut pas clairement séparer la partie fonctionnelle de la partie présentation. Ces solutions sont une transposition à grande échelle de l'informatique centralisée d'il y a une vingtaine d'années. Néanmoins, de nouvelles technologies du Web telles que le langage Java permettent de déporter la partie présentation sur le poste du client (voir chapitre 4).

A la différence des services précédents, ce type de services manque cruellement de normes officielles de protocoles d'accès. La messagerie ou les forums favorisent la communication entre individus via des protocoles bien établis. L'accès à des sources d'informations est grandement facilité par le Web. Mais, il n'est pas encore trivial d'accéder à grande échelle à une source d'applications. De plus, les utilisateurs doivent développer du code spécifique pour rendre disponibles leurs propres applications. Il existe encore moins d'annuaires d'applications.

Néanmoins, de nombreuses couches intermédiaires («middleware»[Ber96]) et quelques travaux de standardisation tendent à définir des protocoles pour l'accès aux applications distribuées : l'appel de procédures distantes pour DCE et l'invocation d'objets distants pour Corba. Mais alors se pose le problème du déploiement de ce type d'infrastructures : comment les programmes des utilisateurs acquièrent-ils les mécanismes pour communiquer via ces bus logiciels ? Dans le chapitre 2, nous étudions plus précisément Corba et nous

---

12. CGI: Common Gateway Interface

montrons qu'il est complexe à mettre en œuvre. Même avec ces infrastructures, nous sommes bien obligés d'admettre qu'à l'heure actuelle, il n'y a encore que peu d'applications accessibles à grande échelle autrement que par le Minitel ou Telnet.

**La contribution de cette thèse porte principalement sur ce dernier type de services et présente une approche pour favoriser l'accès à des sources d'applications composées d'objets répartis. Nous illustrerons nos travaux sur un exemple de sources d'applications : un service bancaire réparti (cf. figure 1.3).**

Ce service met en œuvre une infrastructure à grande échelle composée de nombreuses machines hétérogènes interconnectées par des réseaux : les postes des utilisateurs et les serveurs répartis des diverses agences de la banque. les clients et les employés de la banque accèdent à des ressources et effectuent une grande variété de traitements sur celles-ci : consultation des comptes bancaires, opération de virement, interrogation et génération de tableaux de bord. Dans le chapitre 2, nous montrons comment construire un tel service avec le bus à objets répartis Corba et nous soulevons les problèmes liés au déploiement et à l'évolutivité de ce service. Le chapitre 3 nous permet de montrer comment le langage CorbaScript permet de favoriser l'accès à ces ressources pour supporter la multitude de tâches des utilisateurs. Dans le chapitre 4, nous illustrons comment l'environnement CorbaWeb permet l'accès à ces objets Corba à travers le WWW.

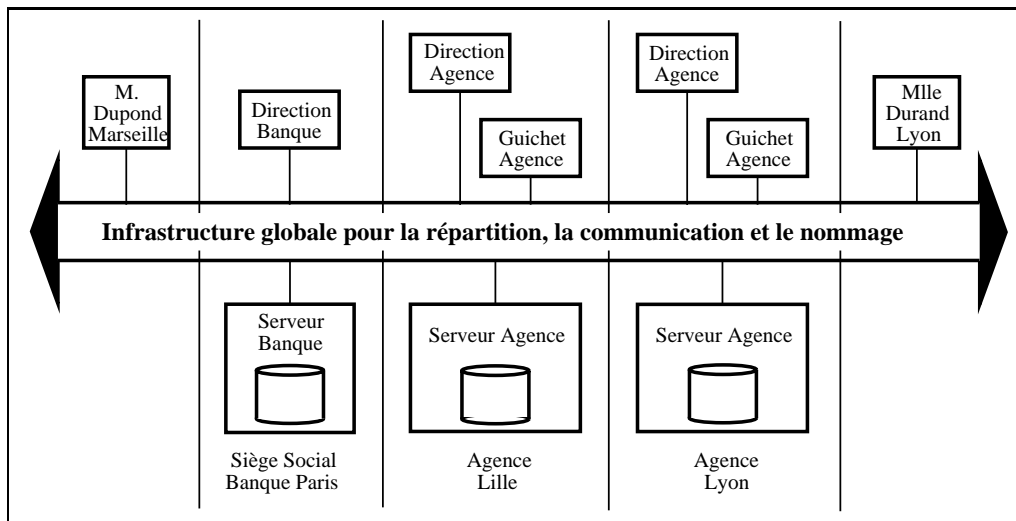


FIG. 1.3 - *Un service bancaire réparti*

### 1.3 Caractéristiques

Dans notre étude, nous nous concentrons sur les mécanismes d'accès par des utilisateurs à des ressources réparties fournies par des services. Dans ce contexte, la complexité des services à grande échelle provient d'un critère quantitatif – le grand nombre et l'importante diversité – des composantes des cinq niveaux suivants :

- **La nature du service** : celle-ci peut être fortement variable d'un service à un autre. Nous avons précédemment illustré quatre grandes catégories de services : les services de l'infrastructure réseau, les services de communication entre individus, les sources d'informations et les sources d'applicatifs.
- **Les interactions** : par l'intermédiaire d'un service, les utilisateurs communiquent entre eux et accèdent à des ressources partagées et réparties. Quatre modes d'interac-

tions peuvent être envisagés dans cette relation intervenant-ressource : un intervenant vers un ou des intervenants, un intervenant vers des ressources, des ressources vers des ressources et des ressources vers des intervenants. Ces interactions sont véhiculées par les relais de l'infrastructure.

- **Les intervenants** : un service à grande échelle est avant tout destiné à aider et à supporter les activités des utilisateurs et de leurs organisations (entreprises privées, universités ou administrations publiques). Les intervenants peuvent avoir des rôles et des tâches très diverses qui peuvent évoluer au cours du temps.
- **Les ressources** : celles-ci sont le support des activités des intervenants. Elles leur permettent de partager de l'information ou des traitements, de synchroniser leur tâches afin de mener une activité commune. Elles sont très diverses allant des fichiers documents jusqu'aux composants logiciels dédiés en passant par des bases de données, des boîtes aux lettres ou des groupes de discussions électroniques.
- **Les relais de l'infrastructure** : ceux-ci sont composés des *logiciels* s'exécutant sur des *machines* interconnectées par des *équipements de réseau*. Ces éléments transmettent les requêtes des utilisateurs vers les ressources. Ils assurent aussi la communication directe entre les intervenants. Les ressources réparties dialoguent entre-elles également à travers ces relais.

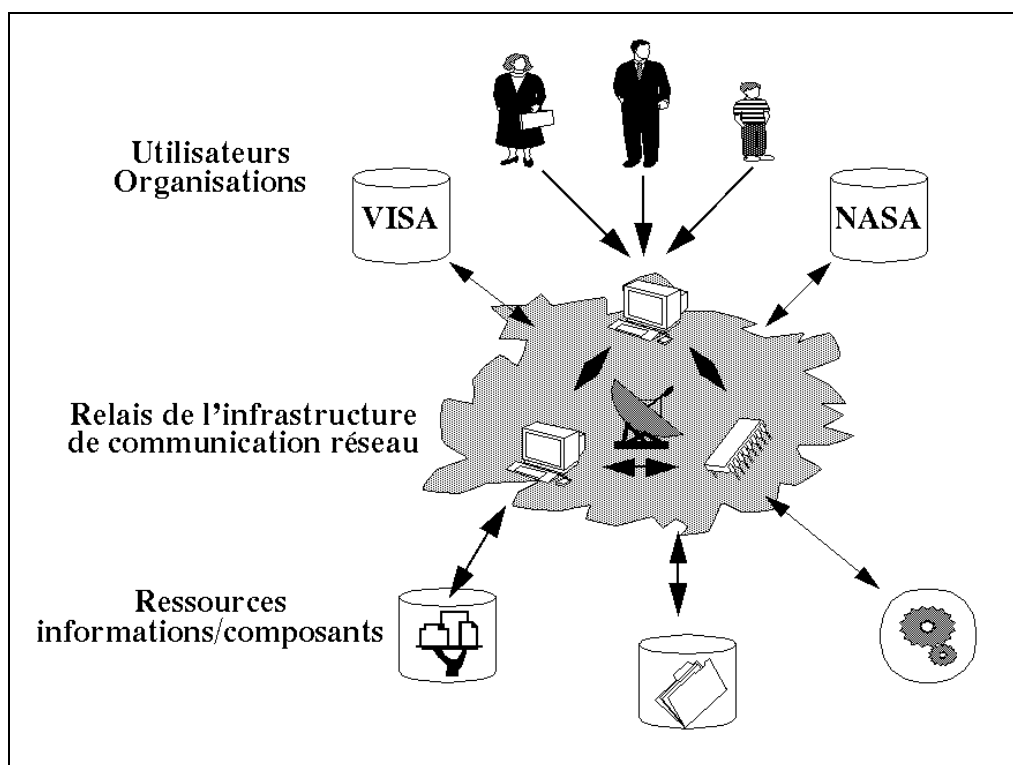


FIG. 1.4 - *Un service à grande échelle*

Dans les sous-sections suivantes, nous discutons de la diversité et de la variété de ces caractéristiques. *la variété des modes d'interactions, la variété des intervenants, la variété des ressources et la variété des relais.*

### 1.3.1 La variété des modes d'interactions

Dans un service à grande échelle, quatre modes d'interactions entre les intervenants et les ressources se dégagent :

1. **un utilisateur**<sup>13</sup> **communiquant avec d'autres utilisateurs** : ce mode d'interaction est principalement fourni par les services de communication interpersonnelle ou de groupe tels que la messagerie, les forums de discussions, l'audio- et la vidéo-conférence. Un utilisateur peut adresser un message à un ou des utilisateurs à travers les services de la messagerie ou des forums électroniques.
2. **un utilisateur vers des ressources** : c'est typiquement le cas dans les sources d'informations ou d'applications. L'utilisateur est à l'initiative de l'interaction avec les ressources : il charge un document via un serveur WWW ou il consulte son compte bancaire via le Minitel.
3. **l'interaction entre ressources** : ce cas de figure se présente lorsque les ressources communiquent entre elles de manière autonome afin de réaliser un traitement : par exemple, dans le projet Iridium<sup>14</sup>, les trajectoires des satellites sont contrôlées par des objets répartis entre les stations au sol et les satellites.
4. **une ressource vers un ou des individus** : dans ce contexte, les ressources contactent les utilisateurs soit pour les notifier de leurs changements ou pour accéder à des données personnelles aux utilisateurs (via des cartes à microprocesseur et des dossiers portables [Van97]).

Bien entendu, ces modes d'interactions peuvent être combinés au sein d'un même service à grande échelle comme dans l'exemple de l'édition coopérative de documents [DQV92, PSS94] :

- 1) la conversation des co-rédacteurs doit être médiatisée par un service de conférences pour les échanges textuels, sonores [Bar96] ou vidéos [Cla92],
- 2) les intervenants collaborent sur des documents stockés dans des serveurs,
- 3 et 4) les outils d'éditeurs doivent se synchroniser pour refléter aux utilisateurs les modifications des documents.

### 1.3.2 La variété des intervenants

Différents intervenants participent à un service à grande échelle : les usagers grand public auxquels le service est destiné, les administrateurs des systèmes informatiques participant à son fonctionnement, les organismes délivrant et responsables du service et finalement les concepteurs et développeurs. Tous ces intervenants ont des droits et devoirs différents vis-à-vis de ce service. Un service à grande échelle doit fournir les métaphores (outils, fonctionnalités du service) pour permettre à tous ces intervenants de réaliser leurs tâches.

Généralement, un intervenant utilise dans son activité plusieurs services : messagerie, source de documents et des applicatifs. Chacun de ces services est accessible par

---

13. ici, un utilisateur peut être aussi bien un individu qu'un organisme.

14. C'est un projet de Motorola pour créer un réseau de téléphonie cellulaire à l'échelle mondiale. Les communications sont transportées par 66 satellites en orbite géostationnaire. Pour plus d'informations, voir [Mot96].

un outil d'interface Homme-Machine spécifique : un lecteur de courrier électronique pour communiquer avec d'autres utilisateurs, un navigateur WWW pour consulter des données hypermédias, un collecticiel pour participer à un travail coopératif, ou bien un logiciel d'administration pour configurer un réseau. L'intervenant est donc confronté à une *grande variété d'ergonomie* allant à l'encontre des besoins de *transparence* et d'*uniformité* désirés par les utilisateurs. Par conséquent, cela entraîne une explosion du temps et des coûts dépensés pour sa formation.

De plus, les activités et les rôles de l'utilisateur peuvent évoluer dans le temps comme par exemple dans les collecticiels [Hoo95]. Les interfaces graphiques doivent supporter cette évolutivité [Cro95] pour autoriser les changements de rôles du simple utilisateur consultant des informations à l'expert utilisant des services complexes.

L'apparition d'interfaces graphiques à base d'objets tels que Windows95, OS/2 Warp et les navigateurs WWW (Mosaic, Netscape) nous offrent des canevas pour réaliser des interfaces utilisateurs *évolutives* et *adaptatives* durant l'exécution tout en respectant une *uniformité ergonomique*.

### 1.3.3 La variété des ressources

En observant l'ensemble des services à grande échelle, nous nous apercevons qu'il existe une multitude de types de ressources : des boîtes aux lettres, des forums de discussions, des messages, des documents et toute la diversité de ressources que pourraient fournir les sources d'applications.

Chaque service se caractérise aussi par la grande quantité des ressources qu'il rend disponibles aux utilisateurs. Par exemple, la messagerie électronique donne l'accès à des millions de boîtes aux lettres. Il y a des milliers de groupes de discussions dans les News. Des millions de documents sont disponibles sur le WWW. Les sources d'applications pourraient quant à elles fournir des millions de services divers et variés. Ainsi pour permettre l'accès à grande échelle à toutes leurs ressources, ces services doivent définir des normes pour la désignation, les protocoles d'accès et les formats des données :

- **Désignation** : pour pouvoir accéder à une ressource, les utilisateurs doivent pouvoir les désigner de manière globale<sup>15</sup>. De plus, cette désignation doit être compréhensible et manipulable par les utilisateurs. Par exemple dans la messagerie, chaque utilisateur possède une boîte aux lettres désignée par le couple : nom de connexion et nom du site où réside sa boîte. Les documents du WWW sont désignés par des URL codées sous la forme de chaînes de caractères. Cette désignation symbolique est très importante car ainsi, les utilisateurs peuvent interpréter le nom des ressources et construire eux-même des noms par analogie. Par exemple, un utilisateur voulant accéder au serveur WWW du LIFL pourra deviner l'URL de celui-ci par analogie avec d'autres serveurs : un serveur Web est souvent désigné par l'URL suivante *www.organisme.pays* donc il imaginera rapidement que l'adresse qu'il cherche est *www.lifl.fr*.
- **Protocole d'accès** : de même, il est nécessaire qu'un service à grande échelle définisse un protocole d'accès aux ressources. Par exemple, le protocole SMTP de la messagerie électronique contient des commandes pour émettre un message. Dans les News, NNTP fournit des primitives pour créer et détruire des groupes, pour envoyer des messages. Tandis que le protocole HTTP contient une commande pour demander

---

15. C'est à dire qu'à chaque ressource est associé un et un seul nom.

un document à un serveur WWW. Ces protocoles sont toujours spécifiques à un type de service car chaque service fournit des ressources bien spécifiques.

- **Format de données** : lors des échanges entre utilisateurs et ressources, il est nécessaire d’avoir un format commun pour structurer et représenter les données. Par exemple, le format MIME permet de structurer et codifier l’échange de données multimédias dans la messagerie électronique SMTP, les News ou sur le WWW.

Les services de base d’Internet ont donc nécessité la définition de normes pour permettre leurs utilisations à grande échelle. Cela a été possible car en général, chaque service fournit un nombre très limité de types de ressources : des boîtes et des messages dans la messagerie, des documents dans le WWW. Mais, comme les sources d’applications peuvent fournir une grande diversité de types de ressources (comme un compte bancaire, un catalogue, un algorithme de calcul), il est alors très difficile de les désigner, de définir un protocole d’accès et un format de données pour ce type de services. Cette disparité dans les ressources pourrait être estompée par la définition d’un *modèle fédérateur des ressources* (comme par exemple, les objets).

**Dans notre environnement CorbaWeb, nous proposons une solution à ce problème en utilisant les URL pour pouvoir désigner les objets Corba et le format MIME pour représenter les objets.**

#### 1.3.4 La variété des relais

Nous regroupons sous le terme *relais* l’ensemble des équipements matériels et logiciels qui permettent aux utilisateurs d’accéder aux ressources. Lorsque nous parlons de services (et/ou de systèmes) à grande échelle, la première pensée venant à l’esprit est de considérer le terme «grande échelle» sous l’angle de la répartition géographique des équipements matériels et logiciels. C’est bien entendu le cas dans la messagerie électronique car ce service est accessible depuis tout poste de travail connecté à Internet. Cette large répartition géographique implique les besoins suivants :

1. **Des réseaux à grande distance** : les WANs (Wide Area Network) sont nécessaires pour permettre la communication entre des machines géographiquement réparties. Depuis quelques années, Internet nous offre une infrastructure mondiale d’interconnexion de tous ces réseaux.
2. **Le masquage de l’hétérogénéité des relais** : ces nombreux relais mis en œuvre sont forcément hétérogènes : différents fournisseurs, différents types de processeurs, différents systèmes d’exploitation et une multitude d’implantations des logiciels utilisateurs. Cette hétérogénéité se retrouve aussi bien sur les postes de travail des utilisateurs, que dans l’infrastructure réseau et que sur les machines serveurs. La définition de *normes technologiques*, comme les protocoles de communication d’Internet (IP, DNS, TCP), permet de résoudre cette hétérogénéité.
3. **Des environnements de traitement réparti ouvert** : la coopération à grande échelle entre les logiciels nécessite la définition d’environnements de traitement réparti ouvert. Cette normalisation est soit de droit dans le cas de l’effort de l’*International Standardization Organisation* (ISO) pour définir la plate-forme répartie ouverte ou *Open Distributed Processing* (ODP[Lin94]), ou de fait comme le *Distributed Computing Environment*[RKF92] de l’OSF<sup>16</sup> ou le bus à objets répartis CORBA de l’OMG.

---

16. OSF : Open Software Foundation

L'hétérogénéité des relais (matériels et logiciels) doit donc être masquée pour faciliter et promouvoir le développement de services à grande échelle. Ainsi *des normes technologiques* et des *environnements de traitement répartis ouverts* ont été définis pour résoudre et aplanir ces différences pour permettre l'*interopérabilité* de ces relais dans leur participation à un service à grande échelle.

## 1.4 Supports pour la conception, le déploiement et l'utilisation

Dans les sections précédentes, nous avons montré les caractéristiques des services à grande échelle et quelques exemples significatifs de tels services.

La mise en place de services à grande échelle se heurte à de nombreux problèmes tant sur le plan technologique que sur le plan conceptuel. Le concepteur doit gérer un vaste ensemble de problèmes liés à la diversité et l'hétérogénéité du matériel (les stations des utilisateurs, les réseaux et les stations serveurs) ; la communication et l'échange de données entre les logiciels distribués. De plus, il faut en même temps concevoir les ressources, prévoir le déploiement de ces services sur l'infrastructure réseau et fournir les interfaces d'accès aux utilisateurs.

Actuellement, le concepteur ne dispose d'aucune méthodologie pour l'aider, le diriger dans ses investigations et il doit donc trouver des solutions technologiques «ad hoc» à ces problèmes. Bien entendu, de nombreux services de communications et d'informations fonctionnent correctement et ces problèmes ont été résolus : par exemple par la standardisation du format des messages échangés dans la messagerie électronique, les formats de documents MIME et d'adresses URL dans le World Wide Web et les protocoles d'accès (SMTP, NNTP, HTTP).

Cependant, le défi dans les années à venir sera de pouvoir concevoir et déployer des services du type applicatif afin d'offrir une vaste gamme de nouvelles ressources accessibles par tout utilisateur. Un des problèmes de ces services est la diversité des ressources et donc la difficulté de normaliser leur accès. Il ne faudrait pas remettre chaque fois en place une nouvelle infrastructure pour réaliser un nouveau service à grande échelle. Il serait donc nécessaire de factoriser les parties communes à tous ces futurs services et ne développer que la partie fonctionnelle de ceux-ci.

L'approche orientée objet a démontré ses qualités dans un grand nombre de domaines de l'informatique : les langages de programmation, les bases de données et les interfaces graphiques. Ces concepts (encapsulation, polymorphisme et héritage) permettent de structurer de larges et complexes applications en un graphe d'objets communicant par appel de méthodes. De plus, les objets ont été appliqués à l'informatique répartie avec succès.

### 1.4.1 Les objets répartis

Dans les quinze dernières années, de nombreux travaux ont été menés dans le domaine du support d'applications distribuées par objets au niveau des langages et des systèmes d'exploitation. L'objectif principal est alors d'offrir une transparence totale de la distribution des objets au-dessus des notions de sites, de processus, de messages échangés par les processus, de persistance des objets, d'invocations d'objets distants et des pannes logicielles et matérielles (cf. figure 1.5). Ces supports assurent au minimum la communication entre des objets distants c'est à dire localisés sur des sites différents. Cette transparence

est accompagnée d'un souci de généricité afin de pouvoir supporter un large éventail de modèles d'objets, de langages objets, de systèmes d'exploitation et de types de machines.

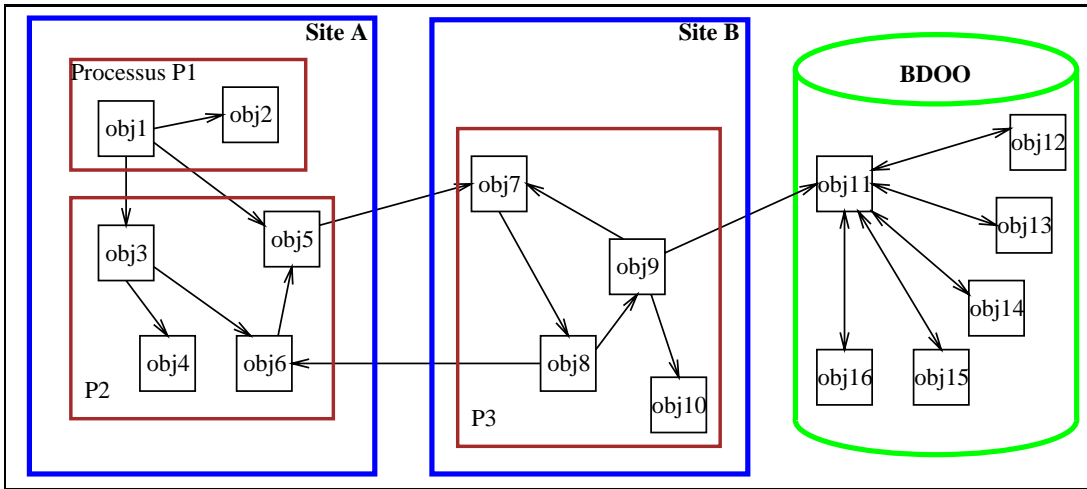


FIG. 1.5 - *Transparence de la distribution*

Dans le cadre général, un support de gestion d'objets distribués est très complexe si nous voulons qu'il prenne en charge tous les aspects de la distribution des objets. Les premiers travaux menés au milieu des années 1980 sur les objets distribués ont volontairement simplifié la problématique pour pouvoir l'étudier et l'expérimenter librement dans le contexte d'un langage unique. Le but de ces travaux fut d'intégrer des mécanismes pour supporter la distribution, le partage, la persistance, le parallélisme et/ou la synchronisation dans des langages à objets existants ou de définir de nouveaux langages à objets adaptés à la programmation distribuée.

Le choix d'un langage unique pour l'étude de la programmation distribuée par les objets permet de s'abstraire des problèmes d'interopérabilité entre langages et offre un cadre non limitatif pour étudier l'apport pour les développeurs de nouvelles constructions syntaxiques et sémantiques au niveau du langage et pour étudier les mécanismes à introduire dans le support d'exécution [Gra95]. Une liste exhaustive des supports à langage unique sort du cadre d'exploration de cette thèse, mais citons quelques exemples marquants.

Les précurseurs de la démarche de définition d'un nouveau langage ont été les projets Eden [ABLN85] dont le langage de programmation résultait d'une extension de Concurrent Euclid; et Argus [Lis85] qui étendait le langage CLU pour permettre l'exécution de transactions réparties. Emerald [BHJ<sup>+</sup>87] (successeur d'Eden) est un langage à objets sans classe et à typage statique qui proposa un modèle basé sur la migration des objets [JLHB88]: lors de l'invocation d'une opération sur un objet, celui-ci peut migrer vers le site distant du processus qui invoque l'opération.

Le langage Guide [BLR94]<sup>17</sup> explora de nombreux concepts au sein d'un langage unique: séparation entre la notion d'interface et la notion de classe, transparence de la persistance des objets, expression dans le langage de la protection des accès concurrents et des règles de partage des objets entre plusieurs utilisateurs. Le système Guide offre une machine virtuelle qui masque et unifie la mémoire persistante, la mémoire partagée et la communication inter-sites. Cette machine virtuelle a été implantée sur diverses architectures matérielles: réseau de machines Bull-PC, stations OSF/Mach ou stations Sun-OS.

17. développé entre 1986-1994 par l'équipe Bull-Imag de Grenoble dans le cadre du projet ESPRIT Co-ManDOS ("Construction and Management of Distributed Open Systems") de la communauté européenne.

Mais beaucoup d'autres projets se sont aussi inscrits dans cette lignée en définissant de nouveaux langages adaptés à la programmation distribuée par les objets (nous avons dans notre équipe défini et implanté le langage BOX [Gra95]).

De l'autre côté, de nombreux projets se sont intéressés à l'adaptation du support d'exécution de langages à objets existants. Les premiers travaux ont été menés sur la distribution du langage SmallTalk [Dec86, Ben87]. Ces travaux ont introduit le concept de *proxy* : pour invoquer un objet distant, un objet client fait appel à un objet-proxy local qui encapsule les communications sur le réseau. Plus récemment, de nouveaux langages se sont vus dotés de mécanismes de distribution comme le projet Network Objects [BNOW94] pour Modula-3 [Nel91] ou le mécanisme de «Remote Method Invocation» pour le langage Java [WRW96].

Les concepts introduits dans ces langages ont beaucoup influencé les recherches menées par la suite sur les supports d'objets distribués. Néanmoins, le principal défaut de ces travaux est de définir un noyau d'exécution spécifique à chaque langage et qui ne peut donc pas être réutilisé pour d'autres langages. Cette spécificité s'oppose aux besoins d'interopérabilité, de réutilisation de composants issus de langages différents, d'intégration d'applications patrimoines («legacy»). De plus, ces langages ne se sont pas imposés car il est très difficile voire impossible d'imposer à la communauté un langage unique.

Les travaux sur les objets distribués se sont également portés sur la définition de systèmes d'exploitation orientés objet généraux. Par exemple, SOS [SGH<sup>+</sup>89] a introduit les objets fragmentés permettant de répartir un service sur différents sites. Ainsi, les mandataires [Sha86] chez le client d'un service ne sont plus uniquement de simples talons d'appels à distance mais ils peuvent participer au service en assurant des fonctionnalités comme le cache local d'une partie de l'état de l'objet, l'exécution locale de certaines méthodes, la répartition de charge ou bien la tolérance aux pannes.

Comme autre exemple, citons Spring de Sun [Ma94, RHKP95] qui démontre la faisabilité d'un système d'exploitation totalement orienté objet. Il est basé sur un micro-noyau qui assure un minimum de fonctions : gestion de la mémoire virtuelle, de l'ordonnancement de flots d'exécution et la communication entre objets par le concept de *Doors*. Tous les autres services sont réalisés par des objets non systèmes. Un autre élément de sa réussite est d'offrir une compatibilité binaire avec Solaris, ainsi un grand nombre de programmes et d'outils sont réutilisables directement.

Les travaux plus récents et en cours s'orientent donc maintenant sur le déploiement à grande échelle. Le projet Globe de Andrew Tanenbaum [vSHvD<sup>+</sup>95, vSHT96, HvST96] est inspiré du concept des objets fragmentés de SOS. Il applique ce modèle à grande échelle sur Internet : fragmentation des services en objets, communication de groupes entre les fragments, réplication et cohérence des divers fragments formant un service, hiérarchie de serveurs de nommage des objets.

D'autres projets s'intéressent plus à la composition et à l'assemblage de composants logiciels hétérogènes pour former des applications distribuées. Citons par exemple, le langage OLAN [BAKR96, BABR96] du projet Sirac à Grenoble qui définit un langage de composants pour favoriser le déploiement d'applications coopératives.

Tous ces travaux ont démontré que les objets sont un excellent support pour les applications distribuées. Néanmoins, ces solutions se cantonnent toujours à une exploitation à petite échelle au sein d'un réseau local (sauf pour le projet Globe). Dans le contexte des services à grande échelle, il ne suffit pas de structurer les ressources et de définir des protocoles d'accès orientés objet, il faut en plus déployer les applications chez les utilisateurs et utiliser des normes pour la communication.

### 1.4.2 Architectures réparties ouvertes

Le mariage actuel entre les télécommunications, l'informatique, la vidéo et l'électronique grand public en un vaste ensemble généralement identifié sous le vocable de *technologies de l'information* [Rud93] nécessite la définition de normes pour les services à grande échelle. Ces normes doivent permettre d'interconnecter des applications distribuées au sein d'*architectures distribuées ouvertes* [Bro93, CDK94, Kha94, Sol94b]. Ces architectures doivent offrir un modèle global pour supporter les besoins d'uniformité de l'informatique distribuée : interopérabilité, transparence, efficacité, intelligence, distribution et réutilisabilité de composants interchangeable. Les buts sont alors de pouvoir agencer ces composants de différentes manières afin de produire rapidement de nouveaux services et de mettre à disposition des utilisateurs une grande variété de services adaptés à leurs besoins personnels.

Quelques efforts dans ce sens sont actuellement en cours. L'*International Standardization Organisation* (ISO) a défini un modèle de référence pour les systèmes répartis ouverts appelée *Open Distributed Processing Reference Model*. L'ODP-RM est un standard intégrateur des technologies existantes pour supporter tous types d'applications : l'ISO ne tente pas de tout réinventer comme ils l'ont fait auparavant avec le modèle OSI pour les réseaux. Cette architecture est générique et a comme objectifs la portabilité des applicatifs, l'interfonctionnement des composants logiciels les formant et la transparence de la répartition de ceux-ci.

Ce modèle propose une vision complètement objet (types et instances) de l'architecture globale. Il la décompose selon cinq points de vue<sup>18</sup> : Entreprise, Information, Traitement, Ingénierie et Technologie.

Le point de vue Entreprise définit les concepts pour caractériser les organisations et leurs interactions dans l'architecture globale. La normalisation des organisations est très difficile à atteindre et donc pour l'instant, ce point de vue ne définit pas beaucoup de règles décrivant l'interopérabilité entre organismes.

Il en va de même pour le point de vue Information qui doit permettre de spécifier à travers un langage les schémas d'informations des entreprises et leurs relations (i.e. structure des documents multimédia, bases de données et représentation des informations).

Le point de vue Traitement est au coeur du standard ODP-RM. Il définit comment structurer le système en objets fonctionnels. L'interface fonctionnelle des objets est clairement séparée de leur implantation. Les objets peuvent avoir plusieurs types d'interfaces : interface d'opérations, de signaux et de flots. Les interfaces spécifient le comportement mais aussi les contraintes d'environnement à respecter pour pouvoir utiliser ce type d'objets (par exemple disponibilité, sécurité, performance, fiabilité et qualité de service).

Le point de vue Ingénierie définit la manière de distribuer les objets et les mécanismes nécessaires à cette distribution. Ce point de vue réutilise tout le savoir-faire développé au cours des vingt dernières années en matière de systèmes d'exploitation distribués (orientés objet ou non) : talons d'emballage des invocations (i.e. souches des objets), protocoles de communication (i.e. RPC) et flots concurrents («multithreading»).

Le point de vue Technologique précise quelles technologies peuvent être appliquées pour construire cette architecture. La normalisation, par des objets, des technologies est une des ambitions d'ODP. Mais pour l'instant, ce point est faiblement spécifié dans le modèle de référence car il est très dépendant de l'évolution des technologies.

L'effort ODP est un projet ambitieux qui vise à spécifier les systèmes distribués pour

---

18. A l'origine, il y avait 7 points de vue.

les décennies à venir : de l'organisation des entreprises à l'interopérabilité des technologies. Cette normalisation est adaptée pour une vision d'ensemble et à long terme, mais elle est trop lente pour fournir des composants techniques exploitables tout de suite. Pour cela, l'ISO travaille en collaboration avec des consortiums de standardisation de fait : DAVIC dans le domaine de la télévision et des télécommunications, TINA-C dans celui des télécommunications, l'ANSA, l'OMG, COM, l'OSF, X/OPEN pour l'informatique en général. Nous consacrons le chapitre 2 à l'étude et la critique du standard de l'OMG : le bus à objets répartis CORBA (Common Object Request Broker Architecture). **Nous avons utilisé ce bus pour mener à bien nos expérimentations pour favoriser l'accès à des objets.**

### 1.4.3 Interfaces Homme-Machine

Les supports d'objets distribués et les standards en cours permettent de supporter une grande variété de services à grande échelle en encapsulant les ressources et les protocoles d'accès. Mais un autre élément est à prendre en compte : les utilisateurs ont besoin d'interfaces Homme-Machine pour pouvoir converser avec les services.

Ces interfaces doivent permettre aux utilisateurs d'effectuer toutes les opérations qu'ils désirent. Les services classiques d'Internet offrent un large éventail d'outils dans ce sens : nombreux lecteurs de courriers électroniques, divers navigateurs pour le WWW. De plus, les navigateurs intègrent la possibilité d'utiliser plusieurs services : Netscape Navigator est capable de communiquer avec des serveurs HTTP, FTP, Gopher, Wais, Telnet, NNTP, SMTP, etc. Cette intégration dans un seul outil est possible car les protocoles Internet sont assez simples et peu nombreux en fin de compte.

Grâce aux navigateurs, les utilisateurs ont une interface unique pour accéder à tous les services standards d'Internet. Ainsi, ils sont formés plus rapidement et à un coût moindre. Il serait intéressant que ce type d'interfaces leur permette ainsi d'utiliser les sources d'applications distribuées. Pour cela, les navigateurs doivent intégrer les protocoles d'accès à ces services. Mais vue la diversité des ressources applicatives, il est impossible de prévoir la conception des navigateurs pour tous les protocoles possibles.

Des technologies récentes telles que le langage portable Java permettent de télécharger à l'exécution des applications intégrant ces protocoles (voir chapitre 4). Ces applications constituent alors une interface Homme-Machine pour utiliser les services applicatifs distants. Cependant, l'utilisateur pourra uniquement réaliser les opérations prévues dans ces interfaces. Mais il ne pourra pas effectuer un traitement ou accéder à un service non prévu. Il serait intéressant de laisser plus de liberté aux utilisateurs en leur fournissant des outils génériques pour qu'ils puissent exprimer leurs besoins spécifiques.

Par analogie, dans les premières bases de données, l'utilisateur ne pouvait manipuler les données qu'à travers les procédures définies par les informaticiens. Dans les bases de données récentes, l'utilisateur peut lui-même mettre en place ses propres procédures à travers l'utilisation de langages d'interrogations (SQL, OQL, voir même le langage naturel). Ainsi, les informaticiens se concentrent uniquement sur la structure fonctionnelle des données et les services de base. Les utilisateurs finaux peuvent très facilement développer leurs propres traitements pour répondre rapidement à leurs besoins spécifiques.

## 1.5 Résumé

Au fil de ce chapitre, nous avons montré que le domaine des services à grande échelle est un domaine vaste contenant de nombreuses imbrications et technologies. Nous nous sommes limités uniquement à une vision : utilisateurs-infrastructure-ressources. Nous avons laissé de côté de nombreux autres aspects car ils n'interviennent pas dans notre problématique de l'accès à des ressources par des utilisateurs via une infrastructure de services. Cette problématique est à la croisée des travaux menés sur les méthodes de conception et de support des objets répartis, des architectures ouvertes pour leur déploiement et des interfaces Homme-Machine.

Un service à grande échelle se distingue par l'importance du facteur quantitatif : large répartition géographique, grand nombre et diversité des composantes (utilisateurs, relais, ressources) et variété des modes d'interactions. De plus, les concepteurs d'un service à grande échelle doivent faire face aux problèmes d'hétérogénéité, d'interopérabilité, d'intégration de technologies et doivent donc trouver des solutions pour résoudre le besoin d'uniformité qui favorisera l'accès. La figure 1.6 résume la problématique générale de la réalisation de services à grande échelle : il faut des méthodes pour les concevoir, des supports de l'informatique répartie pour les déployer et finalement des interfaces Homme-Machine pour pouvoir les utiliser.

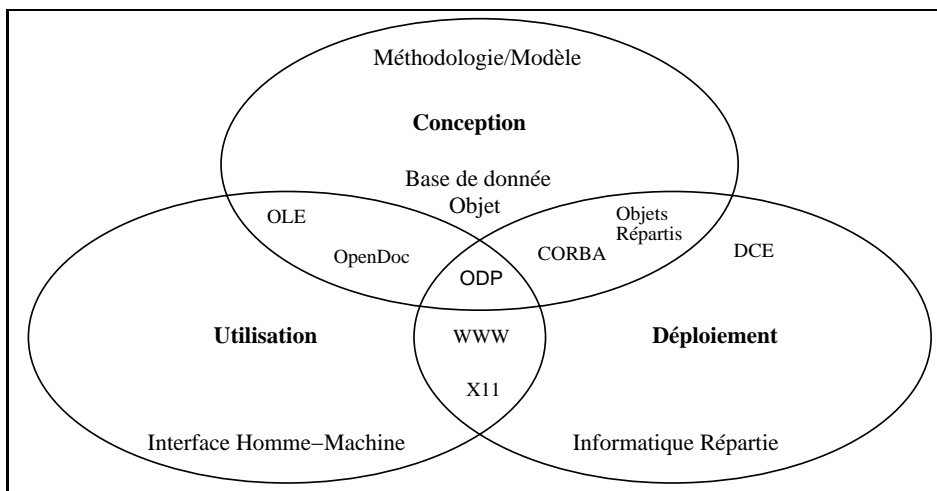


FIG. 1.6 - Les domaines impliqués dans le support de services à grande échelle

La réussite d'un service à grande échelle passe donc par la nécessité et la simplification de *l'intégration des technologies* sous-jacentes pour favoriser son déploiement. Par réussite, nous entendons le fait qu'il soit utilisé par le plus grand nombre d'individus, que ses fonctions satisfassent ces individus et qu'il incite ces individus à investir dans le service (par l'achat de logiciels, de matériels).

Pour la réalisation de services distribués et dans le contexte de notre étude, nous avons choisi :

- l'approche orientée objet pour la conception des services,
- une infrastructure basée sur un bus à objets répartis pour le déploiement,
- des outils génériques - CorbaScript et CorbaWeb - pour favoriser l'accès.

Avant de présenter ces outils génériques, nous présentons dans le chapitre suivant le bus à objets répartis Corba.

## Chapitre 2

# Etude et critiques de l'infrastructure Corba

Ce chapitre présente le bus à objets répartis Corba sur lequel nous avons mené nos travaux. Après la présentation de la vision globale proposée par l'OMG et les composantes du bus à objets répartis, l'Object Request Broker, nous illustrons la complexité et les difficultés rencontrées pour réaliser un service réparti au dessus de cette architecture à travers la conception d'un service bancaire. Dans la conclusion de ce chapitre, nous dégagons des éléments critiques sur le modèle de conception des applications utilisant Corba, puis nous précisons les bases techniques de notre approche pour favoriser l'accès dynamique à des services orientés objet.

### 2.1 Une vision globale

#### 2.1.1 La genèse de Corba

L'Object Management Group (OMG) est un consortium créé en 1989. Il regroupe plus de 600 membres de l'industrie de l'informatique : des fournisseurs de matériels comme Sun, HP, DEC, IBM; des fournisseurs de logiciels comme Iona Tech., Borland, Microsoft, Oracle; et des grands utilisateurs comme Boeing, Motorola, Alcatel.

L'objectif de l'OMG est de définir une norme pour l'intégration d'applications distribuées bénéficiant des qualités de l'approche orientée objet c'est à dire l'*encapsulation*, le *polymorphisme* et l'*héritage*. Le paradigme objet a fait ses preuves dans de nombreux domaines : les méthodes de conception [Car94], les langages de programmation [MNC<sup>+</sup>91], les bases de données [CK94], les composants logiciels [NT95] et les systèmes distribués [Sol94b, OHE96]. Mais ces différentes mises en œuvre sont souvent incompatibles entre elles à cause de divergences dans les concepts objet utilisés. La première phase du travail de l'OMG fut donc de définir son propre modèle de référence : l'*Object Management Architecture Guide* [GX92, Sol94a] décrivant les concepts objet utilisés et une architecture globale d'intégration de services orientés objet.

La deuxième étape fut la rédaction des spécifications de Corba pour *Common Object Request Broker Architecture* : le bus à objets répartis supportant le modèle de l'OMG [Vin93, Sie96b]. Ce bus permet l'interopérabilité entre des objets implantés avec divers langages sur différents systèmes d'exploitation. Pour assurer cette interopérabilité, les fonctionnalités de ces objets sont décrites grâce à un langage de description d'interface

l'OMG-IDL<sup>1</sup>. Ce langage est aussi utilisé pour décrire les composantes du bus. La version 1.0 [OMG91] définit les fonctionnalités, l'architecture et le mode d'utilisation de ce bus à partir du langage C. La version 2.0 [OMG95b] a complété cette spécification en introduisant deux nouveaux langages de programmation C++ et SmallTalk et en définissant une gamme de protocoles réseau pour rendre interopérables différentes implantations du bus Corba.

A l'heure actuelle, la «plomberie» de communication entre les objets étant établie [Sie96a], l'OMG s'intéresse à la spécification de services à valeur ajoutée et d'interfaces de flots (audio/vidéo) pour construire des applications distribuées portables et réutilisables.

### 2.1.2 L'architecture globale de l'OMG

Pour favoriser l'intégration d'applications réparties, l'OMG propose une architecture globale et modulaire (cf. figure 2.1). Cette architecture est globale car elle vise à prendre en compte tous les problèmes et aspects d'un environnement distribué aussi bien au niveau des composantes techniques pour réaliser les applications comme les transactions ou la persistance qu'au niveau des composantes proches de l'utilisateur comme l'interface graphique ou bien les documents composites. De plus, la modularité de ces composantes permet de construire une application en ne choisissant que les services strictement nécessaires. Cette architecture est composée de quatre grands types de services :

1. **Le négociateur de requêtes à objets** ou l'ORB<sup>2</sup> est le bus logiciel assurant le transport des requêtes entre des objets distribués. Il résout tous les problèmes d'hétérogénéité entre les différentes implantations de ces objets – langages de programmation, systèmes d'exploitation et formats des données des machines – et il assure la localisation des objets sur le bus. Sa fonction est donc de rendre le plus transparent possible l'accès aux objets Corba depuis n'importe quel site sur le bus. La section 2.2 présente plus en détails ses composantes.
2. **Les services communs** ou CORBA Services [OMG93, OMG94b, OMG94c, OMG95c] fournissent des abstractions des fonctions systèmes de base telles que le nommage, la persistance, le cycle de vie, les transactions, la sécurité. Actuellement seize services de base ont été ou sont en cours de spécification [OMG93, OMG95c]. Cette base permet d'écrire des applications indépendantes des mécanismes systèmes sous-jacents. Cette indépendance permet ainsi la portabilité des applications utilisant ces services. Dans notre exemple (section 2.5), nous illustrons l'utilisation du service de nommage des objets.
3. **Les utilitaires communs** ou CORBA Facilities [OMG94a, OMG95a] sont des canevas de composants logiciels orientés vers les activités des utilisateurs, décomposés en deux familles : d'un côté, les utilitaires horizontaux prennent en charge les fonctions communes à une grande variété d'applications telles que l'interface graphique, la gestion de l'information, l'administration du système et la gestion des tâches, de l'autre, les utilitaires verticaux modélisent les domaines de l'activité humaine tels que la santé, la finance, les télécommunications ou bien les échanges commerciaux.
4. **Les objets applicatifs** ou Object Applications sont les composants développés spécifiquement pour une application et ne seront donc pas standardisés. Dans notre

---

1. Ce langage a été adopté par ODP comme standard pour la description des interfaces.

2. ORB: Object Request Broker

exemple (section 2.3.3), nous définirons des interfaces pour nos objets du domaine bancaire.

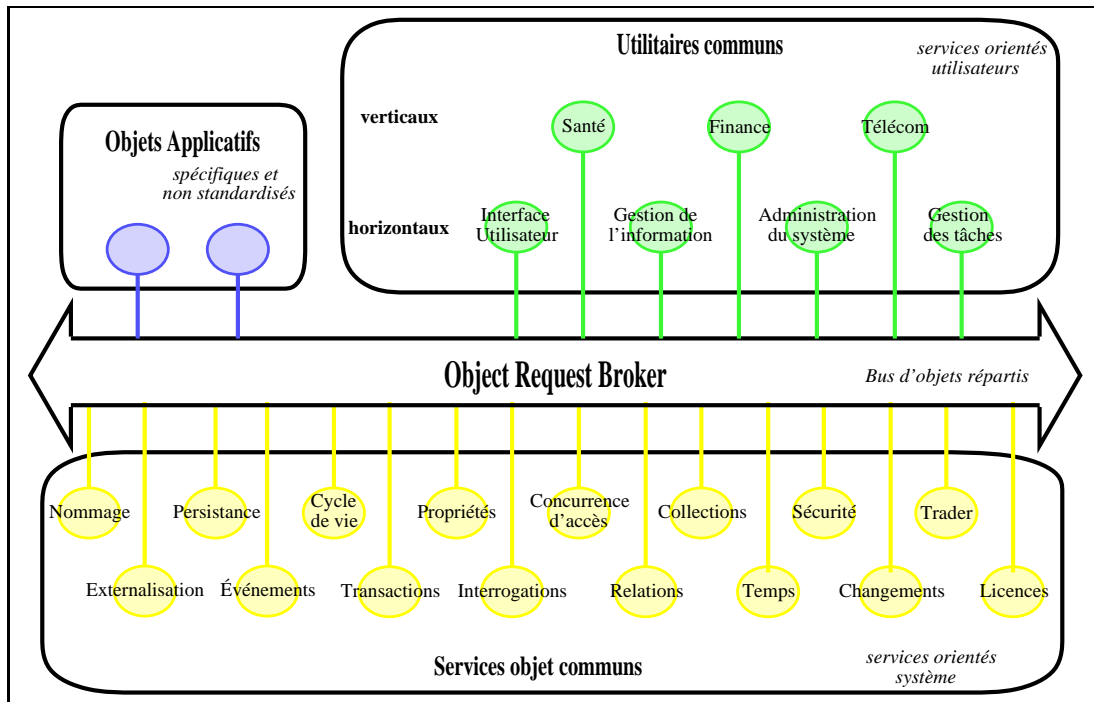


FIG. 2.1 - L'O.M.A. : la vision globale de l'O.M.G.

A travers cette vision « utopique », les membres de l'OMG prônent une industrie du composant logiciel équivalente à l'industrie des composants électroniques. Mais cela impose que tous ces services soient disponibles c'est à dire pas seulement spécifiés mais surtout implantés. Ce n'est pas le cas aujourd'hui car les ORBs actuellement disponibles ne fournissent, dans les meilleurs des cas, que quelques services communs (le nommage, le cycle de vie, les transactions) et nous sommes encore loin d'avoir à notre disposition les utilitaires communs. La non disponibilité de ces services est due à la lenteur intrinsèque de tout processus de normalisation : dans le cas de l'OMG, une spécification met une à deux années pour être acceptée par les membres et il faut ensuite attendre une à deux années supplémentaires pour voir les premières implantations.

Mais ce processus est en marche et il bouleverse la manière de construire les applications. Une application n'est plus développée de façon monolithique en intégrant toutes les fonctionnalités nécessaires mais plutôt par une approche modulaire de réutilisation et de composition des services de base (CORBA Services et CORBA Facilities). Une nouvelle application contient alors quelques objets spécifiques qui communiquent à travers le négociateur de requêtes à objets et utilisent fortement l'ensemble des services communs à toutes les applications.

### 2.1.3 Le langage de description d'interface IDL

Tout objet ou service Corba est défini par une interface décrite par le langage OMG-IDL. Ce langage est inspiré du langage C++ (un préprocesseur et une syntaxe similaire pour la description des types de données) mais il est tout de même indépendant de tout langage de programmation.

Une interface isole les applications utilisant un objet des détails d'implantation de cet objet (cf. figure 2.2) et permet ainsi de concevoir différentes implantations d'un même type d'objet.

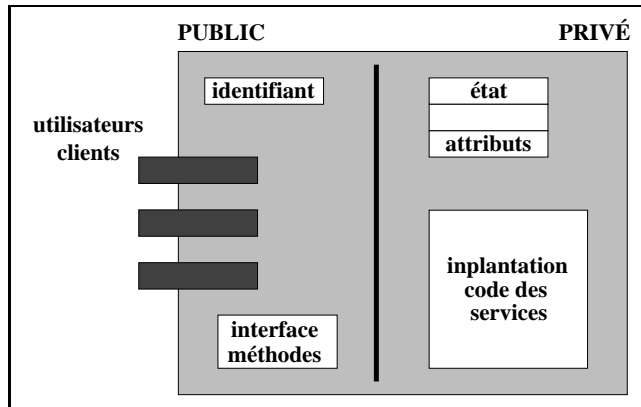


FIG. 2.2 - La séparation de l'interface et de l'implantation d'un objet

A travers l'interface d'une *matrice* (cf. figure 2.3), nous allons décrire les éléments caractéristiques du langage OMG-IDL : les concepts d'interfaces, d'attributs, d'opérations, de passages de paramètres, d'exceptions et le typage des données.

```

module math {
    typedef unsigned long Positif;
    struct coordonnee {
        Positif x, y;
    };
    exception mauvaiseCoordonnee {
        string message;
        coordonnee borne;
    };
    interface matrice {
        readonly attribut Positif hauteur, largeur;
        double lire (in coordonnee coord) raises (mauvaiseCoordonnee);
        void affecter (in coordonnee coord, in double valeur) raises (mauvaiseCoordonnee);
        oneway void remise_a_zero ();
        matrice creer_une_copie ();
    };
};

```

FIG. 2.3 - L'interface d'une matrice

Une interface *matrice* contient un ensemble d'opérations et d'attributs qui permettent de consulter et de changer l'état des instances de ce type. Une interface peut hériter de plusieurs autres interfaces créant ainsi une relation de sous-typage entre ces interfaces.

Un attribut n'est qu'une spécification décrivant une propriété d'un objet qui peut être seulement consultée (le mot clé *readonly* devant l'attribut *hauteur*) ou consultée et modifiée (c'est le cas par défaut).

Chaque définition d'une opération (*lire*) est caractérisée par une signature composée d'un nom, d'un type de retour, de paramètres et d'un ensemble d'exceptions.

Les paramètres sont spécifiés en décrivant leur mode de passage (*in*, *out* et *inout*), le type de valeur (*float*) ainsi que le nom formel du paramètre (*coord*, *valeur*).

Les opérations sont par défaut synchrones (*affecter*) impliquant que l'appelant attend la fin de l'exécution de l'opération avant de reprendre son activité. Mais elles peuvent être spécifiées comme asynchrones (le *oneway* devant l'opération *remise\_a\_zero*), dans ce cas,

l'appelant n'attend pas la fin de l'exécution de l'opération mais il ne recevra pas de valeur en retour ni même ne saura si l'opération a été réellement exécutée.

Les exceptions sont les mécanismes permettant à une implantation de signaler un problème d'exécution à ses appelants. Les exceptions peuvent être composées d'un ensemble structuré de valeurs : l'exception *mauvaiseCoordonnee* contient deux champs (*message* et *borne*) précisant à l'appelant la raison pour laquelle l'exception a été provoquée.

La notion de module permet de regrouper des déclarations dans un espace structuré de nom afin d'éviter les conflits de noms entre plusieurs descriptions (*math*).

Ces spécifications sont fortement typées en ce qui concerne le type de retour des opérations, le passage de paramètres ou le type des attributs (cf. figure 2.4) : 16 types scalaires de base (entier, réel, booléen, caractère, chaîne, octet), des types composés (des tableaux, des séquences, des structures, des unions) comme *coordonnee*, des types d'objets comme le retour de l'opération *creer\_une\_copie*, et deux types dynamiques de données (*TypeCode*, *Any* pour respectivement stocker la description d'un type et une valeur de n'importe quel type). Le définition de type *typedef* permet d'associer de nouveaux noms à des types existants (*Positif*).

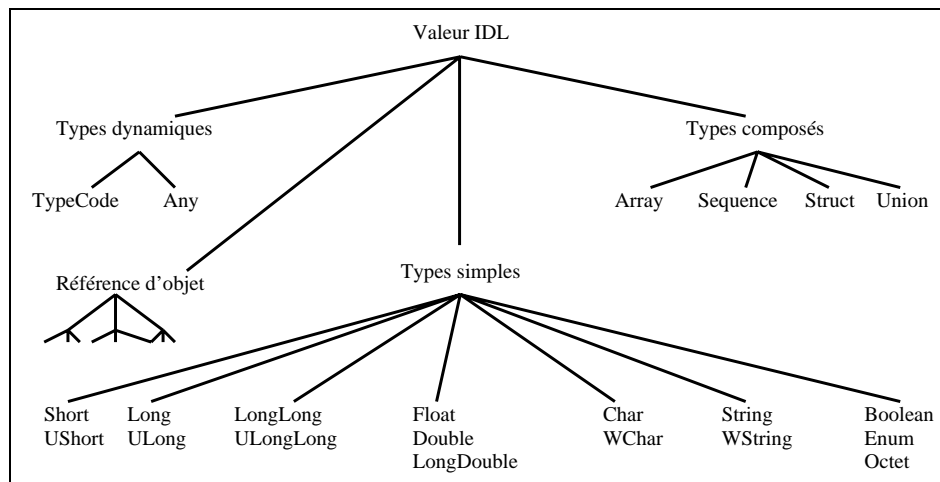


FIG. 2.4 - Les types de données de l'OMG-IDL

Comme tous les langages de description, l'OMG-IDL limite fortement le pouvoir d'expression des interfaces au minimum commun disponible dans tous les langages de programmation : une interface ne peut utiliser que les types disponibles en OMG-IDL.

#### 2.1.4 Les projections vers les langages de programmation

Une fois les spécifications décrites en IDL, il faut les *projeter* vers des langages de programmation pour pouvoir implanter les objets et réaliser les applications utilisant ces objets. Actuellement, des règles de projection du langage OMG-IDL ont été définies pour les langages de programmation suivants : le C [GX93], le C++ [OMG94d], SmallTalk [OMG95b] et Ada95 [OMG95d]. Au fur et à mesure des besoins des utilisateurs de Corba, d'autres projections seront normalisées ; les études en cours portent sur les langages Java, Cobol OO, Eiffel et CommonLisp.

Nous définissons donc une projection comme étant *la traduction dans un langage d'implantation des spécifications IDL d'un service orienté objet*. Pour utiliser ces traductions, le développeur doit se plier à un ensemble de règles de programmation définies par l'OMG pour chaque langage d'implantation.

A partir d'une description IDL, deux types de projections sont réalisées : la projection pour implanter les objets (appelée squelettes IDL) et celle pour les applications utilisant les objets (appelée souches IDL). Ces deux éléments permettent de séparer les applications du substrat de communication c'est à dire le négociateur de requêtes à objets.

## 2.2 Le négociateur de requêtes à objets

Le négociateur de requêtes à objets (Object Request Broker) est le coeur de l'architecture proposée par l'OMG : il fournit la transparence de la localisation ainsi que le transport des requêtes entre les objets. Il doit gérer l'hétérogénéité technique entre ces objets : les différences entre les langages supports et les formats de données.

### 2.2.1 L'architecture

Comme Corba n'est qu'une spécification d'un service de communication réparti orienté objet, son architecture ne précise aucun détail d'implantation mais seulement un ensemble de blocs fonctionnels (cf. figure 2.5) prenant en charge une partie de la communication entre les objets. Ces six blocs fonctionnels composant Corba sont les suivants :

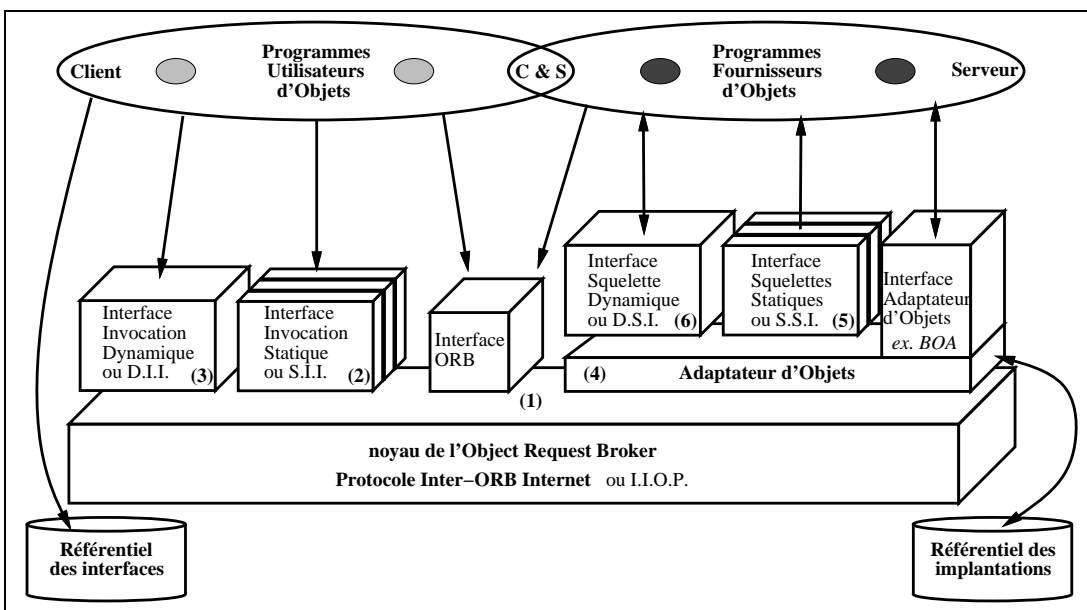


FIG. 2.5 - L'architecture Corba

- (Fig. 2.5, (1)) **Le noyau de communication** assure le transport des invocations entre les objets selon diverses techniques dépendantes de la localisation : une couche de communication réseau si les objets sont situés sur des machines différentes, des outils de communication inter-processus (des IPCs<sup>3</sup> telles que les tubes, les files de messages ou bien la mémoire partagée) lorsque les objets sont sur la même machine ou bien l'appel procédural si les objets sont dans le même espace d'adressage. Ce noyau n'est directement accessible des applications que par l'intermédiaire de l'*interface vers l'ORB*. Elle fournit les primitives d'initialisation et de paramétrage de l'environnement Corba, d'instanciation de références ainsi que de gestion de la mémoire.

3. IPC : Inter-Process Communication

- (Fig. 2.5, (2)) **L’interface d’invocation statique** est le résultat de la projection des descriptions IDL dans un langage de programmation donné. A travers cette interface, les programmes clients invoquent les opérations des objets de manière transparente. Ces souches servent à emballer les invocations et délèguent leur transport au noyau de communication. A chaque interface IDL est associée une interface d’invocation statique et le développeur doit suivre les règles de projections pour pouvoir utiliser ces interfaces.
- (Fig. 2.5, (3)) **L’interface d’invocation dynamique** permet de construire dynamiquement des requêtes vers les objets. Cette interface est plus complexe à utiliser que la précédente car le développeur doit spécifier à l’exécution chaque élément de la requête : objet cible, nom de l’opération ainsi que chaque paramètre. Cette information de typage est accessible à l’exécution à travers le **référentiel des interfaces** que nous décrirons dans la section 2.2.2. Cette complexité apporte de la souplesse lorsque le programme client ne possède pas les souches des objets qu’il veut invoquer. C’est typiquement le cas des interpréteurs de commandes qui ne peuvent pas connaître toutes les souches de tous les objets d’un système réparti.
- (Fig. 2.5, (4)) **L’adaptateur d’objets** est le bloc fonctionnel auquel le noyau de communication délègue l’exécution des requêtes. Corba vise une grande variété de supports des objets, des processus aux bases de données. Ainsi, l’adaptateur isole ces supports du substrat de communication en prenant en charge les problèmes tels que l’activation des processus supportant les objets ou le contrôle des accès. Les informations nécessaires sont stockées dans le **référentiel des implantations**. Actuellement, seul le Basic Object Adapter (BOA), un adaptateur pour processus, a été spécifié.
- (Fig. 2.5, (5)) **L’interface de squelettes statiques** s’occupe de déballer les requêtes pour les transmettre aux objets d’implantations et d’emballer les résultats. Les souches et les squelettes sont générés automatiquement par le compilateur IDL pour éviter aux développeurs de s’occuper des détails techniques du transport des invocations.
- (Fig. 2.5, (6)) **L’interface de squelette dynamique** est l’équivalent pour les serveurs d’objets de l’interface d’invocation dynamique pour les clients, elle permet de recevoir les requêtes sans disposer a priori des squelettes de déballage. Cette interface est utilisée pour implanter des passerelles entre un négociateur et un autre environnement. Nous utilisons cette interface pour pouvoir implanter des objets Corba dans notre langage de script CorbaScript (voir chapitre 3).

Cette architecture fonctionne clairement sur un mode client-serveur. Cependant, un programme peut être à la fois client d’objets distants et fournisseur d’objets. Concrètement pour le développeur, l’ensemble de ces interfaces est décrit dans le langage OMG-IDL et est accessible depuis son langage de programmation à travers une ou des bibliothèques de programmation.

### 2.2.2 Les référentiels

Corba propose actuellement deux référentiels l’un pour décrire les implantations des objets et l’autre pour décrire les interfaces des objets.

**Le référentiel des implantations** contient l'ensemble des informations décrivant l'implantation des objets : le nom de l'exécutable contenant le code de réalisation des objets, la politique d'activation de ces exécutables, les droits d'accès aux serveurs et à leurs objets. Il peut contenir aussi des informations pour l'administration, la gestion, l'audit ou bien le débogage des objets. Actuellement ce référentiel n'a pas encore été spécifié par l'OMG et il est donc spécifique à chaque implantation de Corba.

**Le référentiel des interfaces** contient les interfaces IDL de tous les objets Corba. Ces métadonnées sont stockées sous forme d'un graphe d'objets Corba accessibles à travers une quarantaine d'interfaces IDL (*ModuleDef*, *InterfaceDef*, *OperationDef*, *AttributDef*, etc). Chacun de ces types représente un élément sémantique du langage IDL par exemple les objets de type *InterfaceDef* contiennent des informations décrivant une interface IDL. Le parcours de ce graphe permet d'extraire les informations nécessaires par exemple à un navigateur d'interfaces IDL ou bien à un programme invoquant dynamiquement des objets.

Actuellement, peu d'ORBs fournissent un référentiel des interfaces complet et il existe encore moins d'outils exploitant ces méta-données. Notre proposition de langage de script pour Corba est une illustration de l'utilisation d'un tel référentiel (voir section 3.3.4).

### 2.2.3 L'interopérabilité

Corba 1.x mettait l'accent sur la portabilité des applications pour faire interopérer des objets au sein d'un même négociateur. Corba 2.0 a apporté les éléments pour faire interopérer des objets à travers diverses implantations du négociateur en formalisant les passerelles entre ORBs et en définissant un ensemble de protocoles réseaux (GIOP<sup>4</sup>).

L'une des implantations de GIOP repose sur le protocole TCP/IP d'Internet. Ce protocole, appelé IIOP<sup>5</sup>, permet à des objets situés n'importe où sur Internet et sur n'importe quel ORB de pouvoir dialoguer. Sans entrer dans les détails de ce protocole [OMG95b], ses principales caractéristiques sont de définir un format d'échange des données (CDR<sup>6</sup>), un format des références d'objets (IOR<sup>7</sup>) et une structuration des messages transmis en mode connecté et fiable.

IIOP a toutes les chances de devenir un standard de fait comme le protocole universel d'accès aux objets répartis : Netscape l'a intégré dans les dernières versions de son navigateur (Netscape ONE [Net96]) et le Web Consortium envisage de le substituer au protocole HTTP [W3C96b].

### 2.2.4 Quelques exemples d'ORBs

La norme Corba spécifie uniquement les interfaces de programmation des services offerts par un négociateur de requêtes à objets, par l'intermédiaire du langage OMG-IDL et des règles de projections vers les langages de programmation. Les choix d'implantation de ces services sont laissés totalement aux concepteurs d'ORB. Ainsi, nous trouvons aujourd'hui un large éventail d'implantations du négociateur de requêtes à objets Corba dont

---

4. GIOP : General Inter-ORB Protocol

5. IIOP : Internet Inter-ORB Protocol

6. CDR : Common Data Representation

7. IOR : Interoperable Object Reference

voici quelques exemples significatifs :

1. **ILU** est une bibliothèque de communication orientée objet développé par Xerox[Jan96] et compatible avec Corba 2.0 (IDL, projections et IIOP). Ses points forts sont d'être gratuit, multi-langages (C, C++, Python, Java, CommonLisp), multi-plates-formes et multi-protocoles (IIOP, ONC-RPC, Xerox Courrier, HTTP, ...).
2. **Electra** est issu des travaux de recherche de Silvano Maiffeis [Sil95] sur l'intégration des communications de groupe dans un ORB. Des groupes d'objets peuvent être répliqués pour offrir la tolérance aux pannes et la répartition de la charge.
3. **Orbix** est l'une des premières implantations commerciales de Corba produite par Iona tech. [ION95c, ION95a, ION95b] suite aux travaux de recherche du Trinity College dans le cadre du projet ESPRIT CoManDOS<sup>8</sup>. Cet ORB est constitué d'une bibliothèque liée à chaque exécutable et d'un démon tournant sur chaque machine. Les forces d'Orbix sont d'être largement diffusés, multi-langages (C++, SmallTalk, Ada95, Java), multi-plates-formes (des PC-Windows aux systèmes temps réel) et finalement d'offrir une large palette d'extensions (liaison avec les BDs, traitement transactionnel, communication de groupes avec Isis[II94] ou OrbixTalk).
4. **Spring** est un micro-noyau orienté objet développé par Sun [Ma94, RHKP95] (une des composantes de l'offre NEO). Spring n'est pas compatible avec Corba 2.0 mais il utilise lui aussi le langage IDL pour décrire les interfaces des objets (applicatifs et systèmes) et les souches chez les clients comme représentantes des objets. Sa force réside dans le fait que le substrat de communication est le système d'exploitation, obtenant ainsi des temps d'invocations très performants entre objets sur une même machine. Les invocations se font à travers des portes (*doors*) qui sont une abstraction orientée objet des mécanismes de communication inter-processus.

Ces exemples sont représentatifs de la variété des choix d'implantation d'un négociateur de requêtes à objets allant d'une bibliothèque de communication orientée objet dans le cas de ILU, Electra et Orbix à un micro-noyau orienté objet dans le cas de Spring. Plus d'une vingtaine d'ORBs sont désormais disponibles sur le marché aussi bien conçus par les constructeurs de machines (IBM, Sun, Dec) que par de petites sociétés telles que Iona tech. ou Visigenic.

Notre choix s'est porté sur Orbix car au début de ce travail c'était l'une des seules implantations de Corba offrant, sur le papier<sup>9</sup>, le support de l'invocation dynamique et du référentiel des interfaces.

## 2.3 Un exemple de système bancaire

L'architecture proposée par l'OMG et son bus à objets Corba paraît répondre aux besoins des concepteurs de service réparti : la transparence des communications entre objets hétérogènes et répartis et une vaste gamme de services réutilisables (les services et les utilitaires communs). Mais ce bus répond-t-il à toutes les attentes des concepteurs ? Quels problèmes ou difficultés doivent-ils encore affronter ?

---

<sup>8</sup>. "Construction and Management of Distributed Open Systems" soutenu par la communauté européenne [Pro97].

<sup>9</sup>. au cours de l'implantation de nos prototypes, nous nous sommes rendus compte de certaines faiblesses dans l'implantation de ces composants. Mais, nous restons encore persuader de l'intérêt de ce choix.

Pour répondre à ces questions, il nous paraît intéressant d'illustrer la mise en œuvre d'une plate-forme Corba sur un exemple significatif de service réparti.

Pour cela, nous présentons le processus de conception, d'implantation et de déploiement d'un service bancaire réparti sur Internet supporté par le négociateur de requêtes à objets Orbix. A travers cet exemple, nous illustrons la description des interfaces IDL, l'implantation des interfaces dans un langage de programmation (dans notre cas C++), la conception des serveurs d'objets et des programmes utilisateurs et finalement les problèmes de liaison entre ces programmes distribués.

Au fil de cette présentation, nous dégagons les bénéfices engendrés par l'utilisation de Corba mais aussi les problèmes auxquels les concepteurs doivent faire face. Cette analyse est illustrée concrètement par des fragments représentatifs de code IDL et C++, l'ensemble du code de ce service n'est pas présenté car il est trop long et inutile à notre discours. Dans la conclusion de ce chapitre, nous dégagons des critiques sur le modèle de conception des applications Corba.

### 2.3.1 L'architecture de ce service

Dans le chapitre 1, nous avons dégagé les facteurs de complexité d'un service à grande échelle distribué : le grand nombre et la diversité des relais, des utilisateurs et des ressources accessibles. L'exemple du service bancaire illustre la complexité de l'architecture matérielle et logicielle (cf. figure 2.6) sur les points suivants :

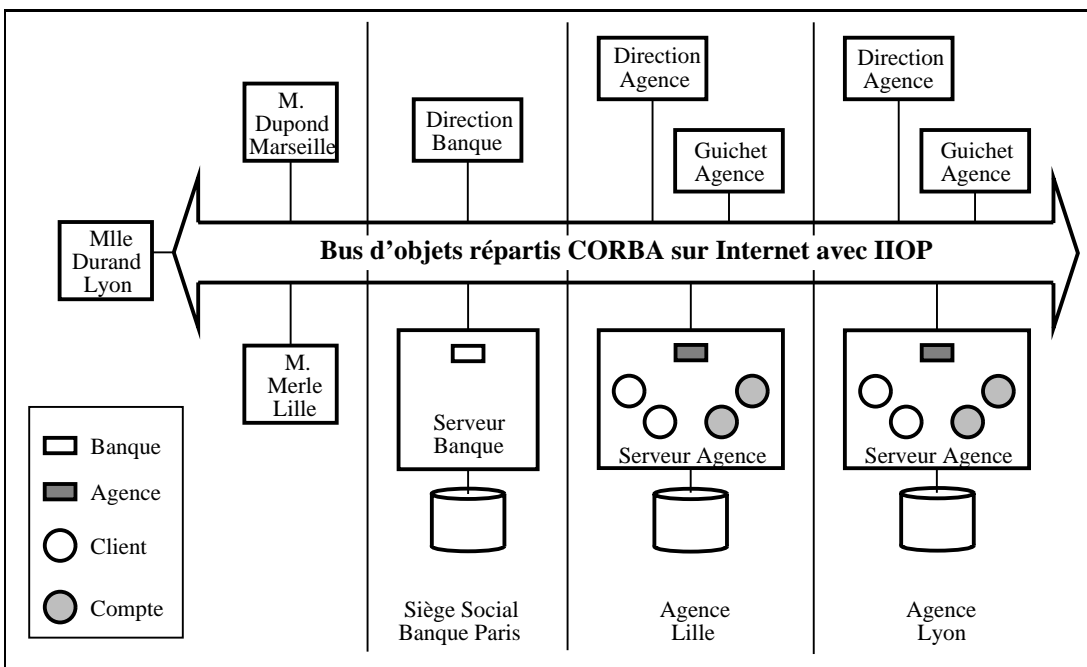


FIG. 2.6 - L'architecture répartie du service bancaire

**La distribution géographique et la diversité matérielle** : ce service de réseau bancaire est réparti sur l'ensemble des sites formés par le siège social (Paris), les agences (Lille, Lyon), les distributeurs d'argent et les stations de consultation à domicile. Les sites du siège social et des agences peuvent être composés d'un vaste parc hétérogène de stations de travail et de machines serveurs pour stocker les informations.

**La variété des intervenants et des ressources** : ce service doit fournir l'accès aux ressources à un grand nombre de clients de la banque mais aussi à tous les employés de la banque. Les rôles des employés sont très variés : du directeur aux conseillers financiers en passant par les analystes financiers, les informaticiens. Ces intervenants agissent sur le service à travers une multitude d'agents utilisateurs : une borne de consultation de comptes, un distributeur d'argent, un tableur pour l'analyse financière ...

L'implantation «réelle» d'un tel service doit prendre en compte de nombreux aspects tels que la persistance des objets, la sécurité des communications, le contrôle d'accès aux objets et les transactions. Nous avons écarté ici ces problèmes car notre travail s'intéresse principalement aux mécanismes d'accès aux objets.

### 2.3.2 Le modèle objet

Comme cet exemple a un objectif pédagogique, nous avons délibérément simplifié le modèle objet des ressources proposées par ce service. Il se compose donc de cinq types d'objets :

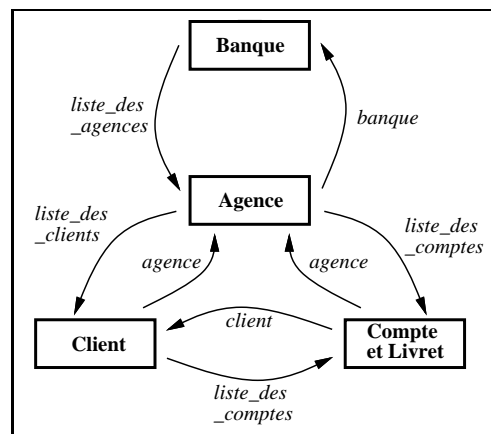


FIG. 2.7 - Le modèle d'un système bancaire

1. **Le compte** : ce type d'objets encapsule l'état d'un compte bancaire. Il fournit des attributs tels que le solde, le possesseur du compte et l'agence gérant le compte. Les deux opérations de base sont le débit et le crédit du solde.
2. **Le livret** : ce type nous sert à illustrer l'héritage avec IDL. Un livret possède les mêmes opérations et attributs qu'un compte mais en plus il possède un *taux* d'intérêt.
3. **Le client** : ce type représente les clients d'une banque. Il maintient la fiche signalétique du client contenant son nom, ses coordonnées, un lien sur l'agence gérant ce client et la liste de ses comptes. Les comptes sont créés et détruits par l'intermédiaire de cette interface.
4. **L'agence** : elle maintient les informations caractérisant une agence comme un lien sur la banque, l'adresse, la liste locale des comptes et des clients. De plus, elle fournit les opérations pour créer de nouveaux clients ou pour en supprimer.
5. **La banque** : cet objet maintient la liste de ses agences permettant ainsi d'accéder transitivement à tous les objets précédents.

Même simplistes, les objets de ce service forment un graphe assez complexe offrant de multiples chemins de parcours. La figure 2.7 montre les liens entre les objets de ce graphe. Chaque parcours du graphe peut définir un type d'activité différent, par exemple : un analyste financier du siège social de la banque traversera le graphe à la recherche des clients ayant assez d'argent sur leur compte pour investir, un directeur d'agence fera la chasse aux clients débiteurs, un conseiller exécutera les ordres de virements sur les comptes de ses clients et un distributeur d'argent appliquera des opérations de retrait sur les comptes.

### 2.3.3 La spécification en IDL

Le texte de la figure 2.8 est la spécification en IDL des types d'objets que nous venons de présenter. Dans le langage OMG-IDL, tous les types doivent avoir un nom et ce nom doit être unique comme par exemple la définition de type *numCompte* ou le type énuméré *sexe*. Pour assurer cette unicité, le module *bancaire* définit un espace de nommage pour les descriptions de notre service afin d'éviter tout conflit de noms avec d'autres services. Ces noms de types sont directement accessibles à l'intérieur du module ou doivent être préfixés du nom du module depuis l'extérieur (i.e. *bancaire::adresse*).

Les séquences (*desComptes*), les structures (*adresse*) définissent des valeurs composées servant au passage de paramètres ou au retour de résultat. Cela permet de transférer des structures de données complexes entre les clients et les serveurs mais ces données sont limitées à des types plats (sans pointeur). Donc le seul moyen de transférer une structure de données complexe telle qu'un arbre ou une liste chaînée est de l'aplatir dans une séquence d'éléments structurés.

L'exception *probleme* permet à un objet de signaler un problème sur l'appel d'une opération. Elle est composée de deux champs (*sur\_quoi* et *raison*) pour préciser la nature de l'erreur rencontrée.

Les cinq types de notre modèle objet sont représentés par les cinq interfaces suivantes : *compte*, *livret*, *client*, *agence* et *banque*. L'interface *compte* contient quatre attributs (*numero*, *client*, *agence* et *solde*) pour consulter l'état des instances et les deux opérations pour modifier le solde (*credit* et *debit*). Un *livret* est une spécialisation d'un *compte* d'où le lien d'héritage entre ces deux interfaces. L'interface *client* contient quelques attributs pour accéder à l'état mais surtout des opérations pour créer de nouveaux comptes (*creer\_un\_compte* et *creer\_un\_livret*), consulter et modifier la liste des comptes (*rechercher\_le\_compte*, *liste\_des\_comptes* et *destruire\_un\_compte*). L'opération *virement* simule une transaction bancaire entre deux comptes d'un même client. L'exception *probleme* est retournée en cas d'impossibilité de réaliser les opérations invoquées comme lorsque le numéro d'un compte, le nom d'un client ou d'une agence sont inconnus.

Les interfaces *agence* et *banque* sont construites sur le même schéma : des attributs pour l'état et des opérations pour consulter les listes des comptes, des clients et des agences. Cependant, l'interface *banque* ne permet pas la création des agences, elle autorise uniquement l'enregistrement des agences.

Cette différence provient du fait que pour créer un objet avec Corba, il faut soit invoquer une opération sur un autre objet ou soit instancier cet objet au démarrage du système. Un objet *banque* s'exécutant au siège social de Paris est incapable de créer un objet *agence* à Lille, il ne peut le faire qu'en invoquant une opération d'un autre objet s'exécutant à Lille. Mais alors comment connaît-il cet objet ? Afin d'éviter ce problème de démarrage, nous avons inversé le protocole : une nouvelle agence se fait connaître de la banque en invoquant l'opération *insérer\_une\_agence* (voir la section 2.4.3).

```

// Spécification IDL du service Bancaire
module bancaire {
    // Description modulaire du service

    interface banque;
    interface agence;
    interface client;
    interface compte;
    interface livret;

    // Quelques types utiles pour le passage de paramètres

    typedef string numCompte;
    typedef sequence<compte> desComptes;
    typedef sequence<client> desClients;

    struct UneAgence {
        string ville;
        agence agence;
    };
    typedef sequence<UneAgence> desAgences;

    struct adresse {
        string rue;
        string ville;
        string telephone;
    };

    enum sexe {masculin, feminin};

    // Une seule exception pour signaler les problèmes
    //
    exception probleme {
        string sur_quoi;
        string raison;
    };

    interface compte {
        readonly attribute numCompte numero;
        readonly attribute client client;
        readonly attribute agence agence;
        readonly attribute float solde;

        void credit (in float montant);
        void debit (in float montant);
    };

    interface livret : compte {
        // Le type Livret est sous-type de Compte
        readonly attribute float taux;
    };

    interface client {
        // Description du type Client
        readonly attribute string nom;
        readonly attribute sexe sexe;
        attribute adresse adresse;
        readonly attribute agence agence;

        compte creer_un_compte (in float solde);
        livret creer_un_livret (in float solde, in float taux);
        compte rechercher_le_compte (in numCompte compte) raises (probleme);
        desComptes liste_des_comptes ();
        void detruire_un_compte (in numCompte compte) raises (probleme);

        void virement (in numCompte c1, in numCompte c2, in float montant) raises (probleme) ;
    };

    interface agence {
        // Description du type Agence
        readonly attribute adresse adresse;
        readonly attribute banque banque;

        desComptes liste_des_comptes ();
        compte rechercher_le_compte (in numCompte compte) raises (probleme);

        client creer_un_client(in string nom, in sexe s, in bancaire::adresse a) raises(probleme);
        client rechercher_le_client (in string client) raises (probleme);
        desClients liste_des_clients ();
        void detruire_un_client (in string client) raises (probleme);
    };

    interface banque {
        // Description du type Banque
        readonly attribute string nom;

        void inserer_une_agence (in string ville, in agence a) raises (probleme);
        void retirer_une_agence (in string ville) raises (probleme);
        agence rechercher_l_agence (in string ville) raises (probleme);
        desAgences liste_des_agences ();
    };
};

```

FIG. 2.8 - La spécification du service bancaire en OMG-IDL

### 2.3.4 Le processus de développement et de déploiement

Les interfaces IDL du service étant décrites, il est maintenant nécessaire de les projeter vers un ou des langages de programmation pour implanter les objets et réaliser les programmes clients mettant en œuvre ces objets. La figure 2.9 illustre le processus standard de développement et de déploiement à suivre :

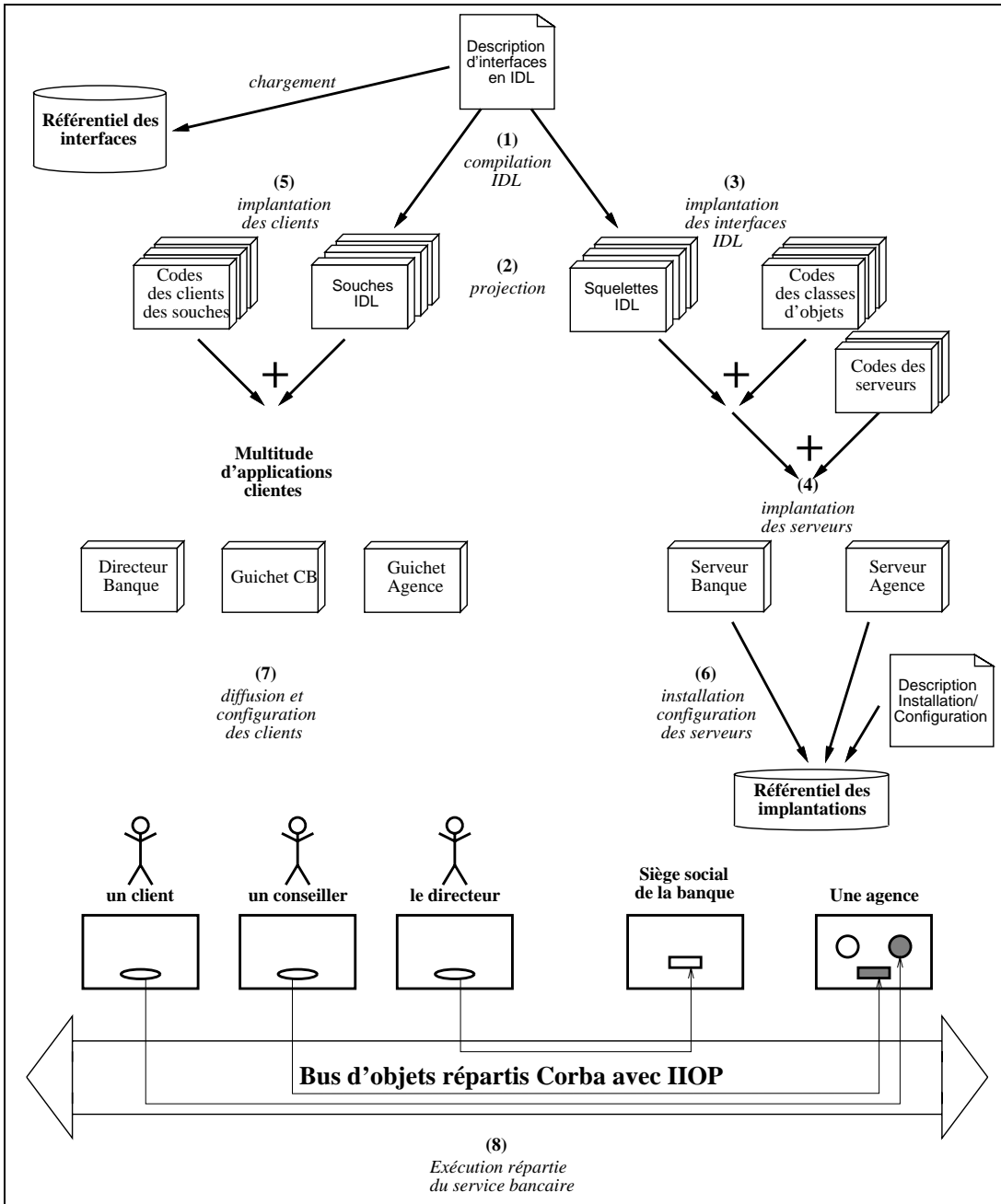


FIG. 2.9 - Le processus de développement et déploiement du service bancaire avec Corba

- 1. La compilation des interfaces IDL** : les interfaces des objets sont décrites avec la syntaxe du langage IDL et sont stockées dans des fichiers texte. Le compilateur IDL prend en entrée un tel fichier et opère un contrôle syntaxique et sémantique des interfaces contenues dans ce fichier. Ce compilateur peut aussi charger ces inter-

faces dans le référentiel des interfaces. C'est la partie frontale commune à tous les compilateurs IDL.

2. **La projection vers les langages de programmation** : le compilateur IDL génère en sortie le code des souches qui sera utilisé par les programmes clients des types d'objets décrits dans le fichier IDL et le code des squelettes pour les programmes serveurs implantant ces types. Cette projection est spécifique à chaque langage de programmation, ainsi un environnement Corba fournit un compilateur IDL pour chacun des langages supportés.
3. **L'implantation des interfaces IDL** : en complétant ou en réutilisant le code squelette généré, le développeur peut implanter ces objets dans le langage qu'il a choisi. Il doit tout de même se plier aux règles de projection vers ce langage.
4. **L'implantation des serveurs d'objets** : le développeur doit écrire les programmes serveurs qui incluent son implantation des objets et les squelettes générés. Dans notre exemple, il y a deux types de serveurs : le serveur « Banque » pour le siège social et le serveur « Agence » pour les agences. Ces programmes contiennent le code pour se connecter à l'ORB, instancier les objets racines du serveur, rendre publiques les références sur ces objets et se mettre en attente de requêtes pour ces objets.
5. **L'implantation des clients d'objets** : le développeur écrit un ensemble de programmes clients qui agissent sur les objets en les parcourant et en invoquant des opérations. Ces programmes incluent le code des souches, le code pour l'interface Utilisateur-Machine (GUI) et le code applicatif.
6. **L'installation et la configuration des serveurs** : cette phase consiste à installer dans le référentiel des implantations les serveurs « Banque » et « Agence » pour automatiser leur activation lorsque des requêtes arrivent pour leurs objets. De plus, il faut configurer les serveurs « Agence » pour qu'ils sachent auprès de quel serveur « Banque », ils devront s'enregistrer à l'initialisation.
7. **La diffusion et la configuration des clients** : une fois les programmes clients mis au point, il est nécessaire de diffuser ces exécutables sur les sites de leur future utilisation et de les configurer pour qu'ils sachent où se trouvent les serveurs Banque et Agence.
8. **L'exécution répartie du service bancaire** : enfin, l'exploitation du service réparti peut commencer : l'ORB assure alors les communications entre les processus et les objets de ce système d'information.

Si notre choix de répartition du modèle objet amène à réaliser deux types de serveurs pour la banque et les agences, il faut tout de même développer une **multitude de programmes clients des objets** en fonction des rôles et activités de chacun des utilisateurs, comme un client de la banque, un conseiller financier ou le directeur.

Afin de ne pas surcharger notre exemple, nous utilisons uniquement la projection vers le langage C++ définie dans la spécification de Corba 2.0 [OMG95b]. Mais quel que soit le langage employé avec Corba, le développeur doit toujours suivre le cycle de développement que nous venons de présenter.

Dans la suite de ce chapitre, nous étudions l'implantation du service bancaire : la section 1.4 discute de l'implantation interne des serveurs et des clients de ce service tandis que la section 1.5 discute de la liaison à l'exécution entre les programmes clients et les

serveurs. Cette analyse nous permet de dégager en section 1.6 des éléments critiques sur les mécanismes de construction des applications Corba.

## 2.4 La réalisation des serveurs et des clients

Dans cette section, nous allons présenter les éléments caractéristiques du développement du code des serveurs et des clients. Les structures de données utilisées pour implanter les objets, la gestion de la persistance, de la concurrence et du contrôle d'accès des objets ne seront pas détaillées dans la suite, car elles ne comportent aucun élément spécifique à Corba et peuvent être considérées comme des «détails» d'implantation dans le contexte de notre propos<sup>10</sup>.

### 2.4.1 La projection des interfaces IDL

La projection du langage IDL vers le langage de programmation C++ traduit chaque définition IDL en une définition C++ équivalente; l'objectif étant de ne pas trop changer les habitudes des développeurs C++. Cette projection génère l'interface d'invocation statique (les souches IDL) pour l'invocation distante des objets depuis les programmes clients et l'interface de squelettes statiques (les squelettes IDL) pour implanter les objets dans les programmes serveurs.

Chacune de ces interfaces est représentée par un ensemble de définitions C++ contenues dans un fichier d'entête à inclure dans les programmes conçus par les développeurs. L'implantation de ces définitions est stockée dans un fichier C++ à compiler. Comme cette implantation est dépendante de l'ORB utilisé, nous n'entrerons pas dans les détails de ce code produit automatiquement par le compilateur IDL.

Du point de vue du développeur, un module IDL est transformé en un espace de noms C++ , notion définie dans la dernière spécification de ce langage, ou en utilisant des classes imbriquées pour les anciens compilateurs C++. Une interface IDL est traduite par une classe C++, ses opérations et attributs deviennent alors des fonctions membres.

L'exemple suivant illustre la projection en C++ de l'interface du service bancaire générée par le compilateur IDL d'Orbix<sup>11</sup> : le module *bancaire* est traduit en une classe C++, les interfaces *compte* et *livret* sont traduites en deux classes C++ et leurs opérations et attributs par des fonctions membres (*solde* et *credit*).

```
class bancaire {
public:
    class compte;
    typedef compte* compte_ptr;
    class compte_var { /*... pointeur intelligent pour la gestion de la mémoire ...*/};

    class compte : public virtual CORBA::Object {
public:
        virtual bancaire::numCompte numero ();
        virtual bancaire::client_ptr client ();
        virtual bancaire::agence_ptr agence ();
        virtual CORBA::Float solde ();
        virtual void credit (CORBA::Float montant);
        virtual void debit (CORBA::Float montant);

        /* autres éléments spécifiques à l'ORB utilisé */
    };
};
```

10. Pour implanter ces éléments, nous pourrions faire appel aux services communs de Corba lorsqu'ils seront disponibles.

11. Les règles de projection définies par l'OMG peuvent être interprétées et implantées d'une autre manière avec un autre ORB.

```

};

class _sk_compte : public virtual compte {
public:
    /* des primitives pour débiller les requêtes */
    /* elles sont invoquées par l'adaptateur lorsqu'une requête arrive pour l'objet */
};

class livret : public virtual compte, public virtual CORBA::Object { /* ... */ };

class _sk_livret : public virtual livret, public virtual _sk_compte { /* ... */ };

// les projections pour les autres déclarations telles que banque, agence, client, ...
};

```

La classe C++ `bancaire::compte` implante la souche d'emballage des requêtes pour pouvoir invoquer tout objet Corba répondant à l'interface IDL `compte`. Cette classe hérite de la classe `CORBA::Object` qui encapsule les informations décrivant la localisation de l'objet comme son adresse sur le réseau.

La classe `bancaire::_sk_compte`<sup>12</sup> implante le comportement du squelette, c'est à dire les mécanismes de débilage des requêtes transportées par l'ORB. La dérivation de cette classe permet alors d'implanter réellement les objets de type `compte`.

De plus, la projection C++ définit pour chaque interface un type simple (extension `_ptr`) pour référencer les objets et un type de référence «intelligente»<sup>13</sup> (extension `_var`) assurant la désallocation automatique de la mémoire par l'utilisation de compteurs de références.

La relation d'héritage entre la souche `bancaire::compte` et le squelette `bancaire::_sk_compte` est une commodité pour bénéficier du polymorphisme du C++ (héritage + fonctions virtuelles). Cela permet alors de référencer à travers le type `bancaire::compte_ptr` aussi bien un compte distant représenté localement par une instance de la classe `bancaire::compte` qu'un compte local représenté par une instance d'une sous-classe du squelette. C'est ainsi que la transparence de localisation est assurée par un ORB utilisé en C++.

Les classes `bancaire::livret` et `bancaire::_sk_livret` sont générées suivant les mêmes règles de projection. La relation d'héritage définie entre les interfaces IDL `compte` et `livret` est préservée en C++ à travers les relations d'héritage entre les classes souches et les classes squelettes.

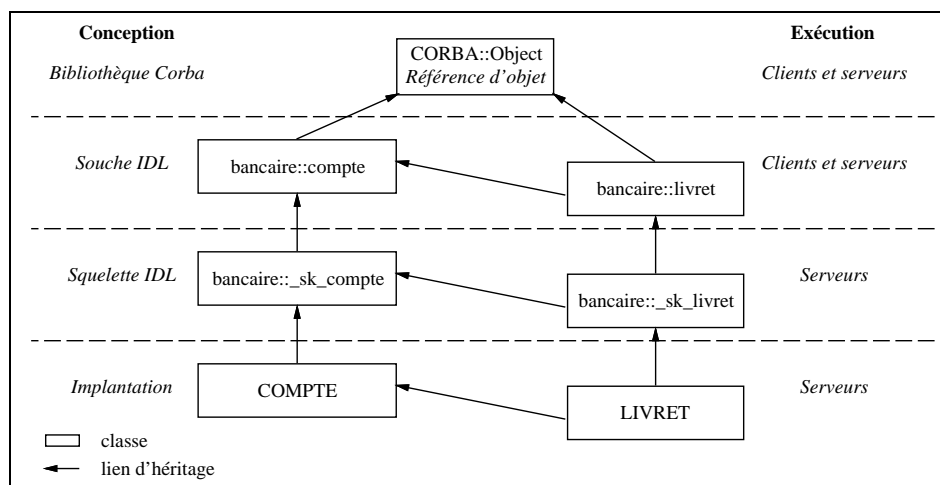


FIG. 2.10 - La hiérarchie des classes C++ générée par le compilateur IDL

12. ou `compteBOAImpl` pour la projection C++ actuelle d'Orbix.

13. des «smart pointer» [Ede92, Str87].

En résumé, la figure 2.10 schématise le graphe des relations d'héritage entre quelques classes composant l'interface d'invocation statique, l'interface de squelettes statiques et l'implantation du service bancaire. Le code de ces classes est alors contenu soit par tous les programmes clients et serveurs (i.e. *CORBA::Object*, *bancaire::compte* et *bancaire::livret* mais aussi *bancaire::client*, *bancaire::agence* et *bancaire::banque*) ou soit uniquement par les serveurs (i.e. *bancaire::\_sk\_compte*, *bancaire::\_sk\_livret*, *bancaire::\_sk\_client*, *bancaire::\_sk\_agence*, *bancaire::\_sk\_banque* et les classes d'implantation des objets).

La plupart des compilateurs IDL produisent le code pour toutes les souches et tous les squelettes pour une description donnée, impliquant alors des applications serveurs et clientes très volumineuses même si elles n'utilisent qu'une petite portion des souches et des squelettes. Pour diminuer ce gaspillage, il faut que le développeur divise lui-même son fichier de description IDL en plusieurs fichiers autonomes afin de pouvoir inclure dans ses applications le minimum nécessaire; ou bien il devra attendre l'arrivée d'un compilateur IDL «intelligent» permettant de sélectionner les souches et squelettes nécessaires au développement.

Ce problème n'est pas très gênant pour le développement de petites applications client-serveur. Mais si nous nous mettons à utiliser intensément les services communs, comme le préconise l'approche de l'OMG, nous obtiendrons de volumineux exécutable dont la majeure partie sera constituée de code souche et/ou squelette.

## 2.4.2 L'implantation des classes d'objets

La classe *COMPTE*, présentée ci-dessous et héritant de *\_sk\_compte*, est une implantation possible de l'interface *compte*. Elle contient quelques variables d'instance (*\_client*, *\_agence* et *\_solde*) et l'implantation des opérations et attributs IDL.

Les attributs IDL ne servent qu'à spécifier les propriétés d'un objet Corba, ils peuvent très bien réaliser un traitement très complexe sur l'objet mais ici, ils retournent simplement une copie des variables d'instance C++.

```
class COMPTE : public virtual bancaire::_sk_compte {
    char* _numero;
    bancaire::client_var _client;
    bancaire::agence_var _agence;
    CORBA::Float _solde;

public: // implantation des opérations et des attributs de l'interface bancaire::compte
    virtual bancaire::numCompte numero () { return CORBA::string_dupl (_numero); }
    virtual bancaire::client_ptr client () { return bancaire::client::_duplicate (_client); }
    virtual bancaire::agence_ptr agence () { return bancaire::agence::_duplicate (_agence); }
    virtual CORBA::Float solde () { return _solde; }
    virtual void credit (CORBA::Float montant) { _solde += montant; }
    virtual void debit (CORBA::Float montant) { _solde -= montant; }

    // ... reste de l'implantation comme le constructeur C++ ou la gestion de la persistance.
};
```

La projection en C++ impose pour le retour de résultat de dupliquer les références d'objet, les chaînes de caractères ainsi que les types composés (*struct*). Le non-respect de ces règles est une source certaine d'erreurs d'exécution. Ces règles conduisent aussi au fait suivant: les objets sont **développés spécifiquement pour Corba**. Bien entendu, ils peuvent toujours encapsuler du code déjà existant par l'intermédiaire d'objets encapsulateurs (ou «object wrapper techniques» [MZ95]).

L'exemple suivant illustre la possibilité d'hériter à la fois d'une classe squelette et d'une implantation: la classe *LIVRET* hérite du squelette pour l'interface *livret* et en même

temps de l'implantation de la classe *COMPTE*. Cette technique permet de réutiliser des projections IDL pour implanter les objets mais aussi des implantations déjà réalisées.

```
class LIVRET : public virtual bancaire::_sk_livret, public virtual COMPTE {
// ... implantation ...
};
```

Toutes les interfaces sont implantées en itérant le procédé que nous venons de décrire : les classes *BANQUE*, *AGENCE* et *CLIENT* implantent respectivement les interfaces *bancaire::banque*, *bancaire::agence* et *bancaire::client*.

### 2.4.3 La création et destruction des objets

Corba ne fournit que l'invocation de requêtes sur des objets. Ainsi pour créer un nouvel objet, il faut s'adresser à un objet existant pour lui demander de faire cette création, ces objets sont appelés dans la terminologie ODP des «usines». Le fragment suivant illustre l'implantation de l'opération *creer\_un\_compte* dans la classe *CLIENT*. Suite à la création d'un *COMPTE*, cette instance est stockée dans une structure de données pour garder un lien sur cet objet. Finalement, la fonction retourne à l'appelant une copie de la référence sur ce nouvel objet.

```
bancaire::compte_ptr CLIENT::creer_un_compte (CORBA::Float montant_initial)
{
    COMPTE* le_nouveau_compte = new COMPTE (montant_initial, this, _agence /*...*/);
    // enregistrement dans une structure de donnée par exemple un dictionnaire
    return bancaire::compte::_duplicate (le_nouveau_compte);
}
```

L'appel du constructeur *COMPTE* crée une instance dans le même espace d'adressage que l'objet *CLIENT* invoqué; par conséquent, les objets sont centralisés dans leurs serveurs de création. La gestion de la répartition des nouvelles instances est totalement à la charge des développeurs : ils doivent prévoir à l'avance la distribution des objets «usines» et coder leur coopération pour répartir les instances à créer.

La destruction des objets est aussi à la charge des développeurs, cependant quelques ORBs implantent plus ou moins un ramasse-miettes réparti. L'exemple suivant illustre la destruction d'un objet client. Le développeur doit gérer explicitement la cohérence interne de ces structures de données pour ne plus référencer des objets détruits.

```
void AGENCE::destruire_un_client (const char* nom_client)
{
    CLIENT* le_client = /* recherche dans la liste des clients */;
    if ( le_client == 0 ) { // si pas trouvé
        throw bancaire::probleme (nom_client, "ce nom ne désigne pas un client de cette agence");
    }
    // retirer cette référence de la liste des clients et détruire tous les comptes de ce client
    delete le_client;
}
```

La gestion du cycle de vie des objets n'est donc pas prise en charge par Corba, elle fait partie des «détails» d'implantation des objets. L'OMG a spécifié les interfaces d'un service commun de gestion du cycle de vie des objets. Mais le problème reste complet car il faut tout de même implanter ces interfaces.

### 2.4.4 L'initialisation des serveurs

Une nouvelle question se pose alors : qui crée les objets «usines»? Ces objets sont instanciés au démarrage des serveurs comme illustré dans le fragment de code suivant. La fonction principale d'un serveur initialise localement l'environnement Corba, crée l'ensemble de ses objets racines, diffuse les références de ses objets auprès des clients (voir la section suivante) et signale sa présence auprès de l'adaptateur de processus par l'appel de l'opération *impl\_is\_ready* du BOA. A partir de ce moment, le serveur est en attente de requêtes adressées à ses objets. Le noyau Corba gère automatiquement le routage des requêtes vers les objets, leur décodage étant réalisé par les squelettes.

```
int main (int argc, char** argv)
{
    CORBA::ORB_ptr orb = CORBA::ORB_init (argc,argv,0);
    CORBA::BOA_ptr boa = orb->BOA_init (argc,argv,0);

    BANQUE* la_banque = new BANQUE (/* des paramètres */);
    // diffusion aux clients de cette référence (voir section 2.5)

    boa->impl_is_ready ("NomDuServeurBanque");
    return 0;
}
```

Dans l'architecture du service bancaire, la mise en place d'une nouvelle agence nécessite la mise en œuvre d'une procédure d'initialisation plus complexe que pour le serveur *Banque* : la *banque* peut difficilement créer de nouveaux objets *agences* localisés sur des sites distants (se reporter à la section 2.4.3) donc les *agences* doivent s'enregistrer auprès de la *banque* par l'invocation de l'opération *insérer\_une\_agence*. Cette opération doit être faite uniquement lors de l'installation de la nouvelle agence, c'est à dire à l'instanciation par le serveur *Agence* d'un objet *AGENCE*.

Plus généralement, ce problème dans la phase de démarrage (le «bootstrap») se pose dans toute implantation d'un service réparti composé d'un ensemble de serveurs activés de manière asynchrone. Les objets racines doivent s'auto-référencer pour former un graphe de ressources distribuées. Les spécifications IDL doivent être construites en conséquence et contiennent donc des opérations pour réaliser ces interconnexions. De plus, les serveurs doivent intégrer l'implantation de l'initialisation grâce par exemple à la lecture de fichiers contenant les informations de démarrage.

Si nous appliquons la démarche de l'OMG, il suffit de définir des interfaces IDL pour répondre à ce service commun à toutes les applications. Cette solution ne fait que repousser le problème car alors comment implanter ces interfaces?

### 2.4.5 L'invocation d'opérations sur des objets distants

Une fois que les objets Corba sont implantés par des serveurs, il faut développer les applications pour les utilisateurs finaux de ces objets. Le service bancaire contient une multitude de tels programmes, un ou plusieurs pour chaque type d'activité : le directeur d'une agence, l'analyste financier, le distributeur automatique d'argent ou encore la consultation de comptes à domicile.

Ces programmes réalisent principalement une interface Homme-Machine pour représenter l'état des objets du service bancaire mais aussi pour appliquer des opérations sur ces objets. Nous allons par la suite nous pencher sur deux aspects de ces programmes spécifiques de l'utilisation de Corba c'est à dire l'invocation d'opérations sur des objets distants et la navigation dans un graphe d'objets.

L'exemple suivant illustre l'invocation de l'opération *debit* et la lecture de l'attribut *solde* d'un objet de type *compte*. Après avoir obtenu une référence sur un *compte* bancaire (voir la section suivante), ce fragment de code utilise l'objet *compte* aussi simplement et naturellement que tout objet C++ local.

```

CORBA::compte_var un_compte = obtenir_un_compte (); // détaillé dans la section suivante 2.5

cout << "Ancien solde du compte = " << un_compte->solde() << endl;
un_compte->debit (1000);
cout << "Nouveau solde du compte = " << un_compte->solde() << endl;

```

Cette transparence est assurée par l'utilisation des souches Corba. Une souche encapsule dans un objet local à l'appelant tous les mécanismes pour invoquer l'objet distant comme le stockage des informations de localisation de l'objet distant dans une référence, l'emballage des arguments dans une requête, et l'envoi de la requête à l'ORB (cf. figure 2.11). A cette transparence, Corba ajoute l'interopérabilité entre les divers langages de programmation à travers les règles de projection de l'OMG-IDL.

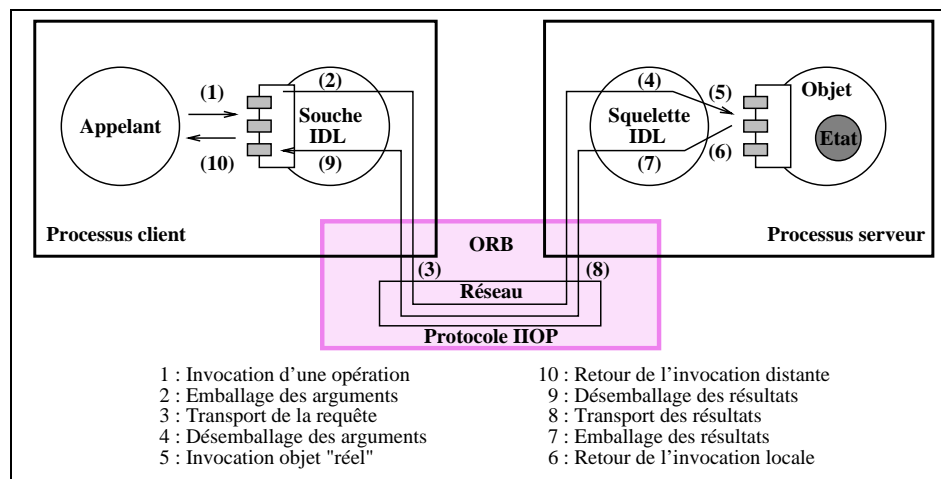


FIG. 2.11 - La transparence de l'invocation d'un objet distant

Ce mécanisme d'invocation d'objets distants est en fait une extension de l'appel de procédures distantes (RPC[Nel81, BN84]) dans le contexte des objets. Il était déjà mis en œuvre, il y a une dizaine d'années, dans Distributed SmallTalk [Dec86] et dans le principe général de «Proxy» [Sha86]. De nombreux travaux de recherche ont depuis amélioré ce principe en rendant les souches plus intelligentes. Afin de réduire les communications réseau, une souche peut se comporter comme un cache local d'une partie de l'état de l'objet distant. Les délais de réponse sont ainsi améliorés mais cela introduit de nouveaux problèmes de cohérence. Une souche peut aussi exécuter localement certains traitements plutôt que de les déléguer à l'objet distant. Le système à objets répartis SOS illustre parfaitement ces possibilités d'extensions à travers le concept d'objets fragmentés [SGH<sup>+</sup>89]. Néanmoins, même si Corba ne spécifie qu'un simple mécanisme d'invocations distantes, il n'interdit pas l'implantation de souches intelligentes : par exemple, Orbix fournit la possibilité par héritage de redéfinir le comportement des souches générées pour introduire des mécanismes de cache.

A l'opposé de ce mécanisme de transfert des activités vers l'objet, les travaux s'intéressant aux mémoires virtuellement partagées et réparties visent à rapprocher l'objet des activités l'utilisant. Ce type de modèle de programmation est très séduisant mais pose

encore de nombreux problèmes d'implantation comme par exemple le choix de la politique de cohérence des diverses copies distribuées d'un même objet. A première vue, l'architecture de Corba n'est pas conçue pour intégrer ce modèle d'objets répartis mais à ma connaissance, aucune étude n'a été menée démontrant ou infirmant cette affirmation.

### 2.4.6 La navigation dans le graphe

Les programmes clients plus complexes vont quant à eux naviguer dans le graphe des objets. L'exemple 2.12 illustre une fonction écrite C++ qui affiche le solde de tous les comptes de tous les clients de toutes les agences d'une banque. Ce parcours du graphe s'effectue hiérarchiquement à partir de l'objet racine *banque* en invoquant successivement les opérations *liste\_des\_agences*, *liste\_des\_clients* et *liste\_des\_comptes*. Chacune de ces opérations retourne un ensemble de références d'objets Corba. L'ORB instancie automatiquement les souches locales représentant ces objets distants et établit dynamiquement la connexion réseau vers les serveurs de ces objets. Ainsi, une fonction C++ peut de manière transparente (c'est à dire sans intervention de la part du développeur) naviguer à travers les différents serveurs : dans notre exemple, le serveur *Banque* et les serveurs *Agences*. L'utilisation du protocole IIOP permet de plus de distribuer ces serveurs sur Internet et même d'utiliser différents ORBs pour former un vaste bus à objets répartis.

```

#include <iostream.h>    // entrée/sortie dans les flux standards
#include <bancaire.h>    // génération des souches C++ pour le module bancaire

void afficher_tous_les_comptes_d_une_banque ( bancaire::banque_ptr banque )
{
    cout << "Ville de l'Agence\tNom du Client\tNuméro du Compte\tSolde Courant" << endl;

    bancaire::desAgences_var agences ( banque->liste_des_agences ( ) );
    for ( int ia=0; ia<agences->length(); ia++ ) {
        bancaire::agence_ptr agence = agences[ia].agence;

        bancaire::adresse_var adresse = agence->adresse();
        const char* ville = adresse->ville;

        bancaire::desClients_var clients ( agence->liste_des_clients ( ) );
        for ( int ic=0; ic<clients->length(); ic++ ) {
            bancaire::client_ptr client = clients[ic];

            CORBA::String_var nom_client = client->nom();

            bancaire::desComptes_var comptes ( client->liste_des_comptes ( ) );
            for ( int cmp=0; cmp<comptes->length(); cmp++ ) {
                bancaire::compte_ptr compte = comptes[cmp];

                CORBA::String_var numero = compte->numero();
                cout << ville << "\t\t\t" << nom_client << "\t\t" << numero
                    << "\t\t\t" << compte->solde() << endl;
            }
        }
    }
}

```

FIG. 2.12 - La fonction C++ d'affichage de tous les comptes d'une banque

Dans la section 3.2.5, nous reprenons cet exemple pour illustrer la souplesse et la simplicité que le langage CorbaScript peut apporter aux utilisateurs pour exploiter et naviguer dans tout graphe d'objets Corba.

## 2.5 La liaison entre les serveurs et les clients

Dans cette section, nous allons aborder la liaison entre les programmes serveurs et clients à travers l'étude de deux mécanismes fournis par Corba que sont la liaison directe par instanciation d'une référence d'objet et les services d'annuaire tel que le service de nommage.

### 2.5.1 La liaison directe

Nous avons vu que lorsqu'une opération retourne une référence d'objet, l'ORB instancie alors automatiquement une souche vers cet objet distant. Mais comment obtenir la première référence vers l'objet banque? Pour cela, l'ORB fournit des opérations pour transformer une référence d'objet en chaîne de caractères et inversement.

Ainsi, le serveur «Banque» peut diffuser la référence sur son objet banque grâce à l'opération *object\_to\_string* définie pour toutes les références dans la classe *CORBA::Object*. Une fois que la référence d'objet est convertie en chaîne, le serveur peut la diffuser en la stockant par exemple dans un fichier.

```
char* chaine = la_banque->object_to_string ();
// diffusion de la référence aux clients par exemple
// par la sauvegarde de cette chaîne dans un fichier
delete [] chaine; // la libération de la chaîne est à la charge du programme
```

Les programmes clients doivent lire le fichier pour obtenir la référence sur la banque. La primitive *string\_to\_object* permet alors d'instancier un objet souche à partir d'une chaîne. Ensuite, le client doit convertir cet objet vers le type de souche qu'il veut utiliser (opération de *narrowing* équivalente à l'affectation inverse du langage Eiffel[Mey92]).

```
char chaine [512];
// lecture de la chaîne stockée par exemple dans un fichier
CORBA::Object_var objet = orb->string_to_object (chaine);
bancaire::banque_var la_banque = bancaire::banque::_narrow (objet);
// le programme peut maintenant invoquer les opérations de la banque
```

Avec le format de référence IOR du protocole IIOP, un client peut ainsi se connecter à n'importe quel objet sur Internet. Mais comme nous venons de le voir ce mécanisme ne propose aucune solution pour la diffusion des références: n'importe quel moyen de communication peut être utilisé comme des fichiers, le courrier électronique ou les forums de discussion.

Ce mécanisme n'est donc pas totalement transparent car il nécessite de fixer une convention entre les clients et les serveurs sur le support de diffusion employé.

### 2.5.2 Les services d'annuaire

La solution envisagée dans la plupart des systèmes distribués à grande échelle est d'employer des services d'annuaire équivalent aux annuaires téléphoniques. Ils permettent de retrouver des objets à partir d'informations les caractérisant comme un nom ou une liste de propriétés. Nous trouvons deux catégories de tels services :

- **Les services de nommage** sont l'équivalent des pages blanches des annuaires. A partir d'un nom, ces services renvoient la référence sur l'objet désiré. Citons comme exemple le DNS pour retrouver les adresses physiques des machines dans le contexte d'un réseau.

- **Les services de recherche** sont l'équivalent des pages jaunes. A partir d'un ensemble de caractéristiques, ces services renvoient la liste des objets correspondants. Un moteur de recherche tel que Altavista est un exemple de tel service dans le WWW.

Les services communs Corba proposent le service de nommage ou Naming Service et le service de recherche ou Trader Service pour répondre au besoin de diffusion des références entre les processus serveurs et les processus clients. Nous allons détailler le service de nommage symbolique pour illustrer son utilisation dans notre contexte.

### 2.5.3 Le service de nommage

Le service de nommage de ressources est un élément clé dans tous les systèmes informatiques. Il permet de retrouver les ressources en leur associant des noms symboliques manipulables par des programmes mais aussi directement par les utilisateurs comme l'arborescence de fichiers Unix. De nombreuses implantations d'un tel service sont disponibles, citons par exemple CDS de DCE, X.500 de l'ISO ou NIS+ de Sun. Mais elles sont toujours dédiées à un environnement de ressources spécifiques et ne sont donc que rarement compatibles entre elles. Cette incompatibilité est liée à la nature divergente des ressources gérées et aux différentes conventions de nommage de ces ressources [RNP93]. Par exemple, les noms désignant des fichiers dans les systèmes d'exploitation ont des structures diverses : le caractère '/' pour séparer les noms de répertoire sous Unix tandis que MS-DOS utilise le '\'; ou encore la longueur des noms limitée à 8+3 caractères sous MSDOS.

Pour éviter ces écueils, le service de nommage de Corba est défini dans le langage OMG-IDL pour séparer l'interface du service de ses implantations possibles (interopérabilité de diverses implantations existantes). Il gère des associations entre des noms symboliques et des références d'objet pour fournir un nommage uniforme de tout type de ressource (encapsulé dans des objets). Finalement, il fixe une convention de structuration des noms et des chemins d'accès.

Son objectif est donc de permettre aux utilisateurs et à leurs programmes de retrouver dynamiquement à l'exécution les objets qui leurs sont nécessaires. Ce service est constitué d'un graphe de contextes de nommage interconnectés (ce graphe pouvant former un espace de nommage réparti); chaque contexte gère une liste d'associations nom-référence. A l'intérieur d'un contexte, chaque nom doit être unique mais un objet peut être désigné par plusieurs noms dans un ou plusieurs contextes. Ce service est composé d'un ensemble de définitions OMG-IDL (cf. figure 2.13), regroupées dans le module *CosNaming*, spécifiant les concepts suivants :

1. **La convention de nommage** : *CosNaming::Name* est la définition IDL pour représenter les noms des associations ou chemins d'accès dans le graphe des contextes. Un chemin est représenté par une *sequence* de noms (*CosNaming::NameComponent*). Les n-1 premiers éléments indiquent le chemin à parcourir à travers les contextes liés. Le dernier nom désigne une association contenue dans le dernier contexte atteint par le parcours.
2. **Les contextes de nommage** : *CosNaming::NamingContext* est l'interface IDL modélisant les contextes de nommage. Elle fournit les opérations de base pour ajouter (*bind*), modifier (*rebind*), retrouver (*resolve*) et détruire (*unbind*) des associations, pour créer de nouveaux contextes (*new\_context* et *bind\_new\_context*), pour connecter des contextes existants (*bind\_context* et *rebind\_context*) et finalement pour les détruire (*destroy*).

```

// Spécification IDL du service de Nommage
module CosNaming {
    //          Quelques types utiles pour le passage de paramètres
    typedef string Istring;
    struct NameComponent {
        Istring      id;
        Istring      kind;
    };
    typedef sequence<NameComponent> Name;
    enum BindingType {nobject, ncontext};
    struct Binding {
        Name          binding_name;
        BindingType   binding_type;
    };
    typedef sequence <Binding> BindingList;
    interface BindingIterator;
    interface NamingContext;
    // Un Contexte de nommage
    interface NamingContext {
        // Quelques types d'exception pour signaler les problèmes
        //
        enum NotFoundReason {missing_node, not_context, not_object};
        exception NotFound {
            NotFoundReason why;
            Name           rest_of_name;
        };
        exception CannotProceed {
            NamingContext cxt;
            Name           rest_of_name;
        };
        exception InvalidName {};
        exception AlreadyBound {};
        exception NotEmpty {};
        // Les opérations applicables sur un Contexte de nommage
        void bind (in Name n, in Object obj)
            raises (NotFound, CannotProceed, InvalidName, AlreadyBound);
        void rebind (in Name n, in Object obj)
            raises (NotFound, CannotProceed, InvalidName);
        void bind_context (in Name n, in NamingContext nc)
            raises (NotFound, CannotProceed, InvalidName, AlreadyBound);
        void rebind_context (in Name n, in NamingContext nc)
            raises (NotFound, CannotProceed, InvalidName);
        Object resolve (in Name n)
            raises (NotFound, CannotProceed, InvalidName);
        void unbind (in Name n)
            raises (NotFound, CannotProceed, InvalidName);
        NamingContext new_context ();
        NamingContext bind_new_context (in Name n)
            raises (NotFound, CannotProceed, InvalidName, AlreadyBound);
        void destroy ()
            raises (NotEmpty);
        void list (in unsigned long how_many, out BindingList bl, out BindingIterator bi);
    };
    // Itérateur pour consulter un Contexte de nommage
    interface BindingIterator {
        boolean next_one (out Binding b);
        boolean next_n (in unsigned long how_many, out BindingList bl);
        void destroy ();
    };
};

```

FIG. 2.13 - L'interface IDL du service de nommage

3. **La consultation des associations** : l'opération *list* permet de consulter le contenu d'un contexte, représenté par la séquence *CosNaming::BindingList* d'éléments structurés *CosNaming::Binding*. Si ce contenu est trop important alors une nouvelle instance de type *CosNaming::BindingIterator* permet de consulter le reste de ce contenu (*next\_one* et *next\_n*). Corba n'intégrant pas par défaut de ramasse-miettes, la destruction de cet objet est à la charge de l'application consultant les contextes (opération *destroy*).
4. **Le traitement des erreurs** : les cinq types d'exceptions *CosNaming::NotFound*, *CosNaming::CannotProceed*, *CosNaming::InvalidName*, *CosNaming::AlreadyBound* et *CosNaming::NotEmpty* permettent aux contextes de signaler à leurs appelants les problèmes liés à une mauvaise utilisation des opérations lorsqu'un nom ne correspond à aucune association lors d'une recherche. Elles signalent également l'impossibilité d'effectuer une opération, un nom contenant des caractères non autorisés, un nom déjà utilisé par une autre association ou bien la destruction d'un contexte non vide.

Le service de nommage Corba spécifie uniquement les interfaces d'un service d'annuaire. L'organisation de cet espace de nommage est laissé totalement à la charge des utilisateurs/administrateurs. Diverses configurations sont alors possibles de l'organisation hiérarchique à la fédération d'espaces de nommage.

Un espace de nommage peut être réparti sur de nombreux sites, il suffit alors de relier les divers contextes présents sur ces sites. En étendant ce scénario au niveau d'Internet, cette fédération de contextes formerait alors un service d'annuaire à grande échelle équivalent au DNS. Les problèmes d'hétérogénéité et d'interopérabilité entre ces contextes fortement répartis seraient résolus par le protocole IIOP. A travers ce vaste service, nous pourrions retrouver tout objet Corba disponible sur le réseau.

#### 2.5.4 Le scénario d'utilisation du service de nommage

Ainsi, le service de nommage permet à des applications clientes de retrouver des objets gérés par des applications serveurs. La figure 2.14 représente le scénario standard d'utilisation de ce service :

1. le serveur obtient une référence sur un contexte de nommage,
2. il crée un objet racine,
3. il invoque l'opération *bind* du contexte de nommage pour créer une association entre un nom et la référence vers l'objet créé,
4. le client obtient une référence sur un contexte de nommage,
5. il invoque l'opération *resolve* pour retrouver la référence de l'objet,
6. le retour de cette opération crée localement une référence sur l'objet serveur,
7. le client doit faire une conversion explicite vers le type qu'il désire utiliser et,
8. finalement, le client peut invoquer les opérations de l'objet serveur.

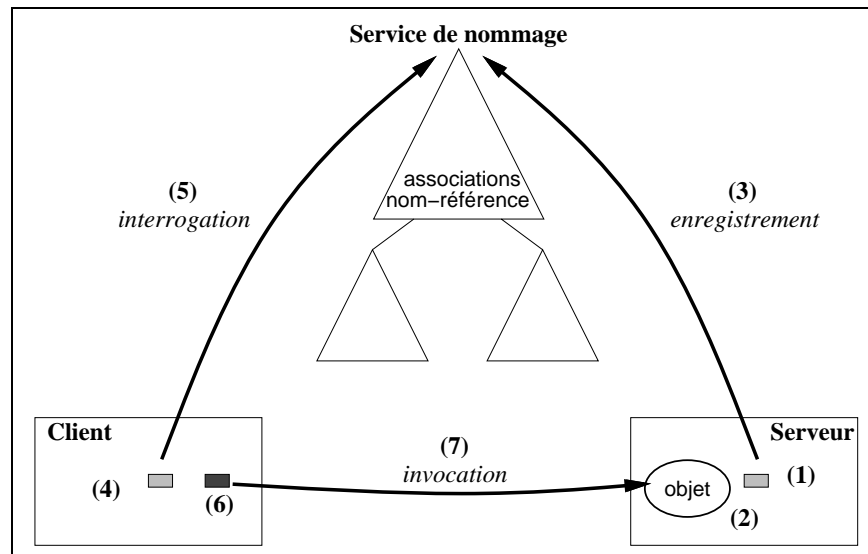


FIG. 2.14 - Le scénario d'utilisation d'un service de nommage

Ce scénario impose que le client et le serveur se soient mis d'accord sur le nom qui sert à retrouver la référence de l'objet. Si le client ne connaît pas ce nom, il peut toujours consulter (opération *list*) et naviguer dans les contextes de nommage pour découvrir où est entreposé l'objet désiré. Les paragraphes suivants illustrent concrètement comment un programme Corba utilise ces contextes de nommage.

#### 2.5.4.1 Référencer un contexte initial

Pour pouvoir utiliser un contexte de nommage, un programme doit obtenir une référence sur ce contexte. Nous sommes encore ici en présence d'un problème d'initialisation (de «bootstrap»); l'ORB fournit donc une opération (*resolve\_initial\_references*) pour obtenir les références sur les services de base tels que le service de nommage («*NamingService*») et le référentiel des interfaces (i.e. des objets notoires).

```
CORBA::Object_var objet = orb->resolve_initial_references ("NamingService");
CosNaming::NamingContext_var NS = CosNaming::NamingContext::_narrow (objet);
```

Cette opération de l'ORB renvoie une référence d'objet du type *CORBA::Object* qui doit être convertie en une instance de type contexte de nommage. A partir de cette référence, l'utilisateur peut découvrir tous les autres services et objets disponibles sur le bus à condition qu'ils aient été enregistrés par leur serveur respectif.

#### 2.5.4.2 Enregistrer une référence

Une fois qu'un serveur a obtenu la référence de son contexte de nommage initial, il peut enregistrer tous ses objets racines sous différents noms et dans différents contextes de nommage transitivement accessibles. L'exemple suivant complète le code de la fonction principale du serveur «Banque» et réalise l'enregistrement de l'instance *la\_banque* sous le nom symbolique *votre\_banque* dans le service de nommage désigné par la variable *NS*.

```
CosNaming::Name_var nom = new CosNaming::Name (1);
nom[0].id = "votre_banque";    nom[0].kind = 0;
try {
    NS->bind (nom, la_banque);
} catch (...) { /* traitement des exceptions */ }
```

L'opération *bind* peut renvoyer des exceptions en cas de problèmes comme par exemple si le nom est déjà utilisé ou bien si le contexte ne peut plus contenir de nouvelles associations (en raison d'une limitation dans son implantation).

### 2.5.4.3 Obtenir une référence

Les programmes clients obtiennent simplement les références des objets désirés en invoquant l'opération de résolution (*resolve*) sur leur contexte initial (*NS*) et peuvent ensuite directement invoquer l'objet après la nécessaire conversion de type (l'opération de *narrowing*).

```
CosNaming::Name_var nom = new CosNaming::Name (1);
nom[0].id = "votre_banque";    nom[0].kind = 0;
CORBA::Object_var objet = NS->resolve (nom); /* traitement des exceptions ? */
bancaire::banque_var la_banque = bancaire::banque::_narrow (objet);
```

La liaison entre un serveur d'agence et ses programmes clients est réalisée de la même manière : le serveur «Agence» enregistre sous un certain nom son objet racine *agence* et les programmes clients de l'agence retrouvent cet objet par l'intermédiaire du service de nommage.

### 2.5.4.4 Configurer et administrer

Dans le scénario présenté, les objets *banque* et *agence* sont enregistrés à la racine de l'espace de nommage. Comme le service de nommage sera utilisé par différentes applications distribuées, il est nécessaire de structurer l'espace de nommage en créant par exemple des contextes pour chaque application. Le fragment suivant invoque l'opération *bind\_new\_context* de création d'un nouveau contexte connecté au contexte initial. Le serveur «Banque» n'a plus alors qu'à enregistrer son objet racine par le chemin «*bancaire votre\_banque*».

```
CosNaming::Name_var nom = new CosNaming::Name (1);
nom[0].id = "bancaire";    nom[0].kind = 0;
NS->bind_new_context (nom);
```

Pour réaliser ces opérations de configuration et d'administration de l'espace de nommage, un service de nommage est souvent fourni avec un ensemble d'outils déjà écrits. Par exemple, l'implantation livrée par Orbix (appelée OrbixNames [ION96a]) est composée d'un exécutable serveur de contextes et de neuf exécutables clients. Ceux-ci sont *putns*, *catns*, *reputns* et *rmns* pour respectivement créer, résoudre, modifier et supprimer une association, *lsns* pour lister le contenu d'un contexte, *putnewncns* et *newncns* pour créer un nouveau contexte connecté ou non, *putncns* pour interconnecter des contextes préexistants et finalement *reputncns* pour modifier les connexions entre contextes de nommage. Mais ces outils ne permettent d'effectuer que les opérations de base; les développeurs doivent composer avec ces outils préexistants ou en construire de nouveaux afin de réaliser des tâches plus complexes telles que déplacer une association d'un contexte vers un autre ou bien naviguer dans l'espace de nommage.

Cependant la séparation entre les interfaces standardisées d'un service et leurs diverses implantations ouvre la voie à la vision «utopique» de l'OMG d'une industrie du composant logiciel où nous pourrions construire nos systèmes logiciels en composant avec l'offre de plusieurs fournisseurs : par exemple, un service de nommage fourni par Sun avec un navigateur d'espace de nommage fourni par Netscape.

## 2.6 Critiques et conclusion

### 2.6.1 La complexité de Corba

Comme nous venons de le voir à travers la réalisation du service bancaire relativement simple (il n'est composé que de cinq types d'objets et il met en œuvre uniquement le service de nommage), un environnement Corba est complexe à utiliser, mais il en va de même de la plupart des supports distribués.

Si la séparation entre les interfaces et les implantations est conceptuellement très bénéfique, le langage OMG-IDL limite fortement le pouvoir d'expression des développeurs et les projections imposent un surplus de travail non négligeable pour implanter et utiliser les objets.

La lecture des fragments de code précédents n'est pas aisée car nous n'avons pas présenté toutes les subtilités des règles de projection vers le langage C++ mais cette complexité serait toujours présente quel que soit le ou les langages de programmation utilisés avec Corba. Il est alors clair que le développement d'applications avec Corba ne peut être réalisé que par des **développeurs «experts»** maîtrisant parfaitement leurs langages de programmation ainsi que les règles de projection.

Corba apporte tout de même la transparence des communications entre les objets mais c'est la seule transparence assurée; le nommage, la persistance, le cycle de vie, les transactions, la sécurité sont des aspects que doivent gérer les développeurs à travers l'utilisation des services et des utilitaires communs Corba. L'architecture OMA peut être ainsi comparée à l'architecture des micro-noyaux : un substrat d'exécution minimal sur lequel sont posés les autres services du système d'exploitation tels que la gestion de la mémoire, des fichiers ou des communications.

### 2.6.2 L'évolutivité des services

Dans le contexte d'un service réparti à grande échelle, par exemple le service bancaire que nous venons de décrire, les activités des utilisateurs peuvent fortement changer au fil du temps impliquant la mise en place de nouvelles procédures d'exploitation du système d'informations formé par les objets répartis. De plus, la création de nouveaux objets peut être nécessaire pour répondre aux changements dans les entreprises, dans les législations locales, nationales ou voire même internationales. Les informaticiens concevant et/ou administrant ces services doivent répondre rapidement à ces demandes de changement pour d'un côté satisfaire les besoins des utilisateurs et de l'autre côté justifier auprès des dirigeants de l'entreprise l'intérêt de construire le service à l'aide de Corba.

La prise en compte de ces changements se traduit par la conception de nouvelles spécifications IDL ou la modification de celles existantes nécessitant la révision du code complexe des programmes serveurs ainsi que des programmes clients. Ces révisions ralentissent fortement l'évolutivité des services répartis.

Quoique très imparfaite, l'infrastructure Corba a toutes les chances de devenir dans les années à venir le bus d'intégration d'applications réparties favorisant l'interopérabilité entre objets hétérogènes sur le réseau Internet. Mais alors dans ce contexte, la question principale que nous posons est la suivante : **«comment répondre à la variété et à l'évolutivité des services répartis orientés objet de manière souple et rapide»** :

- **La variété des types d'objets** : l'approche objet pousse à modulariser, décomposer fortement les ressources; ainsi un système d'informations sera composé de millions d'instances et de milliers d'interfaces.

- **L'apparition de nouvelles activités** : les utilisateurs ont des activités, des tâches et des rôles qui évoluent très rapidement en fonction des besoins des entreprises et des changements du monde «réel».
- **Le besoin de nouvelles ressources** : l'apparition de nouvelles activités nécessite, pour les prendre en charge, la conception de nouveaux objets ou la révision des objets existants.

Pour satisfaire ces besoins en terme de variété et d'évolution dans les descriptions IDL des services répartis, Corba n'offre actuellement qu'une seule réponse : concevoir de nombreux programmes clients, les réviser en cas de changements ou bien en produire de nouveaux pour supporter les nouvelles activités. Mais la complexité du développement avec Corba rend difficile et ralentit cette production car elle nécessite des développeurs «chevronnés». Cette inflation sera donc gérée difficilement par les services informatiques des entreprises et donc ne favorisera pas l'utilisation des objets pour structurer les services répartis. Cependant, ces problèmes ne sont pas spécifiques à Corba mais à tous les environnements de distribution d'applications orientés objet ou non.

Notre exemple bancaire illustre parfaitement cette inflation. Il est composé seulement de deux types de serveurs mais surtout de nombreux types de processus clients pour invoquer, administrer, configurer, créer, détruire, connecter et naviguer dans le graphe d'objets formant le système d'informations réparti d'une banque.

### 2.6.3 Les mécanismes dynamiques de Corba

Comme nous l'avons vu dans ce chapitre, l'approche par souches IDL crée un lien statique entre les programmes clients et les interfaces IDL des objets d'un service. Ainsi lorsque les interfaces évoluent, il faut revoir tous les programmes clients et les serveurs du système. La complexité de Corba et le nombre volumineux de ces programmes rendent ce travail long et laborieux.

D'un autre côté, Corba offre un ensemble de mécanismes pour exploiter et implanter dynamiquement des objets répartis tels que l'**interface d'invocation dynamique** (DII), le **référentiel des interfaces** (IR) et l'**interface de squelettes dynamiques** (DSI<sup>14</sup>). Jusqu'à présent ces mécanismes sont sous-exploités par les développeurs de services répartis avec Corba car leurs intérêts ne sont pas encore bien mesurés, il n'y a pas non plus de guide méthodologique pour les exploiter et donc ils restent très complexes à mettre en œuvre.

A l'origine, le DSI fut défini pour permettre la mise en place de passerelles entre différents ORBs. Lorsque un ORB non-IIOP<sup>15</sup> veut invoquer des objets d'un autre ORB IIOP, il est nécessaire de transformer les requêtes de son format propriétaire vers le format IIOP. Ces conversions de protocoles sont effectuées via une passerelle : les requêtes arrivent dans le format propriétaire et elles sont recodées dans le format IIOP. Deux types de passerelles sont alors envisageables. Les passerelles statiques reçoivent les requêtes via des squelettes IDL mais alors soit elles ne traitent qu'un nombre limité de types IDL définis à la compilation de la passerelle, soit elles doivent contenir tous les squelettes possibles. Par contre, les passerelles dynamiques reçoivent les requêtes via le DSI et peuvent ainsi convertir toute requête. Le DSI sert aussi à connecter un langage, pour lequel aucune règle

---

14. DSI: Dynamic Skeleton Interface

15. c'est à dire un ORB qui implante un protocole propriétaire pour transporter les requêtes entre des objets répartis sur le réseau.

de projection n'est définie, avec un ORB : il reçoit les requêtes à travers une unique API. Mais comme le DSI est très récent, il n'est pas encore implanté dans beaucoup d'ORBs et il est encore moins utilisé par des applications.

Pour illustrer les mécanismes dynamiques, nous prenons un exemple simple d'invocations dynamiques d'un objet Corba. L'analyse de cet exemple nous permettra de dégager la difficulté d'utilisation de ces mécanismes. L'exemple ci-dessus illustre l'invocation bloquante de l'opération *débit* et la consultation de l'attribut *solde* avec attente différée sur un objet *compte*.

```

CORBA::Object_var un_compte = un_moyen_quelconque_pour_obtenir_sa_reference ();
// créer d'une requête
CORBA::Request_var requete = un_compte->request ("debit");
// fixer les arguments et le type du résultat
requete->add_arg_in("montant") <<= v;
requete->set_return_type (CORBA::_tc_void);
// invoquer la requête en mode bloquant synchrone
requete->invoke ();
// au retour, traiter les exceptions si nécessaire

requete = un_compte->request("_get_solde");
requete->set_return_type (CORBA::_tc_float);
// invoquer la requête avec attente de résultat différée
requete->send_deferred ();
// faire autre chose pendant que la requête est traitée par l'objet
requete->get_response ();
CORBA::Float solde;
requete->return_value () >>= solde;

```

Corba fournit donc un mécanisme, encapsulé dans la classe *CORBA::Request*, pour construire à l'exécution des requêtes vers des objets Corba. Cette classe possède des méthodes pour fixer chacune des informations caractérisant une requête : l'objet cible, le nom de l'opération, la liste des paramètres et le type du résultat. Plusieurs moyens existent pour créer une requête, le plus simple étant d'invoquer la méthode *\_request("nom de l'opération")* sur une référence d'objet. Dans le cas de l'accès à un attribut, il faut préciser s'il s'agit d'une consultation (*\_get\_*) ou d'une modification (*\_set\_*). Ainsi, les deux premières caractéristiques de la requête sont fixées.

Ensuite, chaque paramètre doit être ajouté à la requête en précisant son mode de passage (*add\_arg\_in*, *add\_arg\_out* et *add\_arg\_inout*), sa valeur ainsi que son type et optionnellement le nom formel du paramètre. Pour simplifier un peu cette tâche, Corba surcharge l'opérateur C++ (i.e. *<<=*) pour fixer les valeurs des types IDL de base. Le type du résultat peut être fixé par *set\_return\_type*. Corba fournit un ensemble de constantes pour les types de base IDL (*CORBA::\_tc\_void*, *CORBA::\_tc\_short*, ...). Ces métadonnées de typage sont représentées par des objets de la classe *CORBA::TypeCode*.

Enfin, Corba permet d'invoquer les opérations selon trois modes : l'appel synchrone et bloquant jusqu'au retour de résultat (*invoke*), l'appel asynchrone<sup>16</sup> sans attente de résultat (*send\_oneway*) et finalement l'appel avec attente de résultat différée (*send\_deferred*) où la synchronisation est explicite et bloquante avec *get\_response* ou par attente active avec *poll\_response*. Ce dernier mode peut être aussi réalisé en invocation statique en utilisant plusieurs flots d'exécution concurrents (« multithreading »). Le résultat peut alors être extrait de la requête par *return\_value* et stocké dans une variable grâce aux opérateurs d'extractions (i.e. *>>=*).

Mais, à bien y regarder, ce code n'a rien de dynamique car le développeur connaissait parfaitement à la compilation la signature des opérations que ce code invoquera à

16. Corba ne spécifie pas clairement la sémantique de l'appel asynchrone, ainsi il peut être fiable ou non, légèrement bloquant ou non selon les implantations de Corba utilisées.

l'exécution. Il aurait donc pu utiliser des souches IDL plus simples d'emploi et qui permettent de vérifier le typage à la compilation. Toutefois, nous avons pris cet exemple uniquement pour illustrer le principe d'utilisation de l'interface DII. De plus, nous avons laissé de côté quelques autres classes du DII à savoir *CORBA::Any*, *CORBA::NamedValue* et *CORBA::NVList* permettant aussi de spécifier les informations des requêtes.

Néanmoins, l'interface DII est réellement utile lorsque le développeur ne connaît aucune information sur la signature des opérations que son programme invoquera; par exemple, dans un environnement général de navigation dans des objets («object browser»). Dans ce contexte, le programme devra consulter le référentiel des interfaces pour découvrir quelles sont les signatures des opérations applicables sur un objet : le nom de l'opération, le type du résultat et des paramètres. A partir de ces métadonnées, le navigateur affichera une représentation visuelle de l'interface de l'objet. L'utilisateur choisira l'opération qu'il veut invoquer et remplira les paramètres. Le navigateur pourra alors construire une requête à partir des données saisies par l'utilisateur et lui afficher les résultats.

Nous constatons alors que le code de tels programmes doit être générique car ils doivent permettre l'invocation de n'importe quelle opération et donc ils doivent savoir gérer n'importe quelle forme de signature. Corba ne fournit que la mécanique de base pour réaliser de telles applications mais beaucoup d'efforts doivent être fournis pour implanter cette généralité. Ainsi, dans notre projet GOODE (Generic Object-Oriented Dynamic Environment), nous tentons de définir un environnement et de fournir à terme une boîte à outils pour la conception de telles applications génériques [MGG96a].

#### 2.6.4 Vers une approche générique et dynamique

Les idées maîtresses de l'environnement générique et dynamique GOODE sont : la conception de services composés d'objets répartis, leur déploiement à travers un bus à objets répartis et l'utilisation de ces services à travers des outils génériques. Le principal atout de notre approche est de ne pas construire une multitude de clients Corba spécifiques à chaque tâche et rôle des utilisateurs comme c'est couramment le cas lorsque l'on développe un service réparti. Pour cet environnement, nous choisissons le langage IDL pour la conception des objets composant les services, le bus à objets répartis Corba pour leur déploiement et les agents génériques CorbaScript et CorbaWeb pour l'utilisation de ces services. La figure 2.15 compare notre approche générique et dynamique par rapport à l'approche statique de conception et d'exploitation d'applications réparties couramment employée avec Corba.

Dans l'approche classique, un utilisateur ne peut accéder à un service que si un agent spécifique a été développé pour satisfaire ce besoin d'accès. Si cet agent n'existe pas, l'utilisateur doit attendre qu'un informaticien «expert» en Corba implante cet agent. Cela entraîne forcément des délais et peut être pénalisant pour la tâche que veut accomplir l'utilisateur. Techniquement, cet agent utilise les souches IDL pour émettre des requêtes sur le bus Corba. Ces requêtes sont reçues par les squelettes IDL des objets serveurs.

Par opposition, l'approche développée dans cette thèse favorise l'accès aux objets par l'intermédiaire d'outils génériques et dynamiques reflétant immédiatement les modifications apportées aux objets serveurs. Ces outils favorisent la navigation dans les graphes d'objets et exploitent dynamiquement les métadonnées fournies par les référentiels des interfaces. L'utilisation conjointe de ces métadonnées et des mécanismes dynamiques (DII et DSI) permet d'accéder directement aux nouvelles ressources sans passer par un long et pénible cycle de développement. Techniquement, un tel agent générique est construit au dessus des mécanismes dynamiques de Corba à savoir le DII pour émettre des requêtes à

destination de n'importe quel objet du bus, le DSI pour recevoir n'importe quel type de requête à destination de nos agents, l'IR pour découvrir dynamiquement les informations de typage et pouvoir mettre en œuvre intelligemment les deux mécanismes précédents.

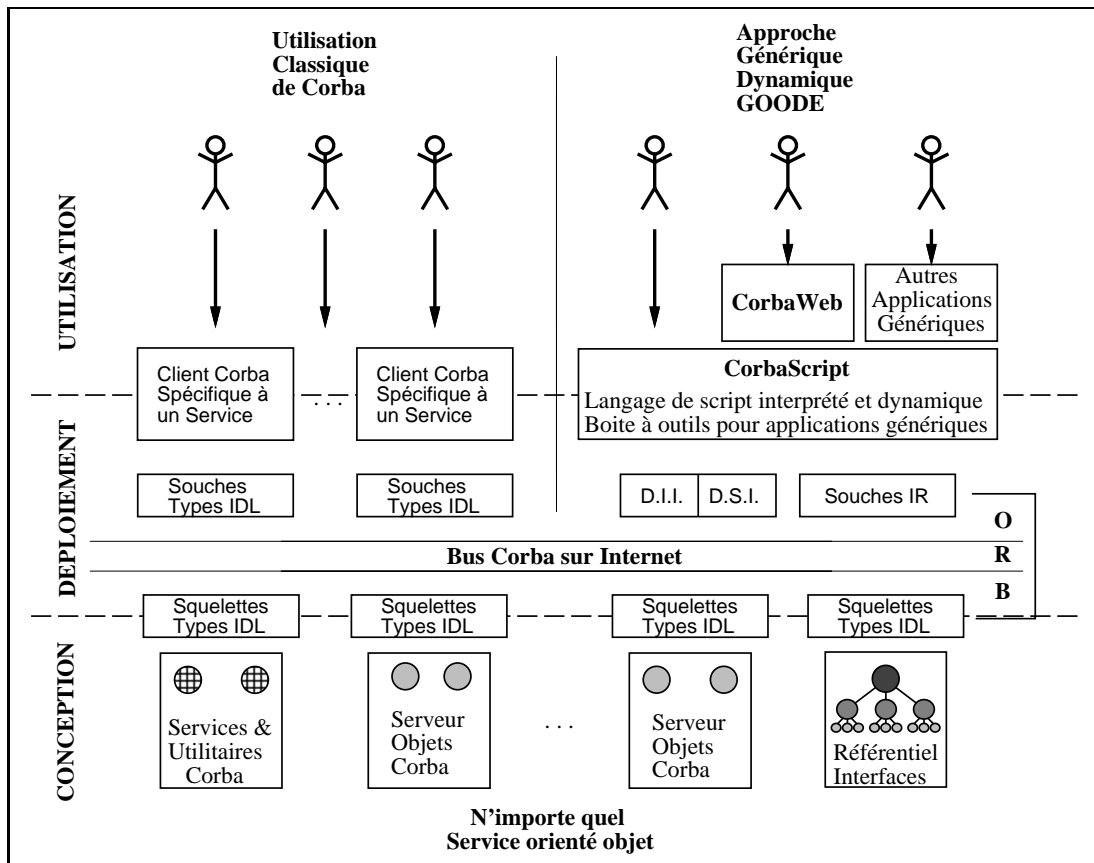


FIG. 2.15 - *L'environnement générique et dynamique GOODE*

Dans les chapitres suivants, nous présentons les deux outils qui composent actuellement l'environnement GOODE : CorbaScript et CorbaWeb. CorbaScript est un langage de script orienté objet pour utiliser dynamiquement les objets Corba. Il fournit aussi une boîte à outils pour construire des nouvelles applications génériques (i.e. le moteur de l'interpréteur est réutilisable). CorbaWeb est un environnement de navigation dans tout graphe d'objets Corba à travers l'interface utilisateur du WWW.



## Chapitre 3

# CorbaScript : un langage de script pour Corba

Comme nous l'avons vu au chapitre 2, Corba est très complexe à utiliser. Toutefois, il a de nombreuses chances de devenir le support pour les services orientés objet (OMG-IDL, ORB et IIOP). Mais alors dans ce contexte, comment faciliter et promouvoir la conception de services orientés objet ?

Notre objectif est de favoriser l'exploitation de services orientés objet à travers un langage de script interprété, interactif, orienté objet et fournissant dynamiquement l'accès à tous les objets disponibles sur un bus Corba. L'exploitation des services est alors grandement facilitée par le développement rapide de scripts répondant aux attentes des utilisateurs finaux mais aussi des administrateurs et informaticiens responsables de ces services.

Le plan de ce chapitre est le suivant : dans un premier temps, nous présentons l'analyse des intérêts d'un langage de script pour les objets répartis et la définition de ses fonctionnalités. ensuite nous étudions des constructions syntaxiques et sémantiques proposées dans le langage CorbaScript. Nous illustrons CorbaScript dans le contexte du service bancaire et du service de nommage (voir chapitre 2). Enfin, nous discutons des points majeurs de l'implantation de CorbaScript. La conclusion résume les contextes d'utilisation du langage CorbaScript.

### 3.1 Pourquoi des scripts ?

Les scripts sont utilisés dans de nombreux contextes informatiques dont voici quelques exemples : l'interface d'accès aux systèmes d'exploitation comme les «shells» Unix (comme sh et csh [GldOA92], Perl [WS91]), la production rapide de programmes avec des langages interprétés (comme Tcl/Tk [Ous94], Python [WvRA96]), l'exploitation des bases de données à travers les langages d'interrogation (comme SQL [ISO92], OQL [CK94]) et l'intégration d'applications de bureautique avec Visual Basic dans le monde Windows.

Actuellement, il n'existe que peu d'environnements de ce type qui soient satisfaisants pour les objets répartis Corba. Dans la section 3.1.4, nous présenterons et critiquerons deux exemples de ce qui a été réalisé jusqu'à présent : TclDii [ASG<sup>+</sup>94] et Python+ILU [Jan95]. Notre langage CorbaScript vise donc à combler ce manque.

Mais quelles doivent-être les fonctionnalités d'un tel langage et pour quelles utilisations ? Afin de répondre à ces deux questions, il convient d'étudier les environnements de script existants afin de dégager leurs caractéristiques, leurs intérêts ainsi que les bénéfices

qu'ils apportent.

### 3.1.1 Interactivité et interprétation

Un environnement de script est par nature interactif et se doit donc d'être interprété.

L'**interactivité** d'un environnement de script permet aux utilisateurs d'accéder, de manipuler, de modifier des ressources immédiatement sans passer par un long processus de développement : l'utilisateur saisit sa ou ses commandes, elles sont exécutées directement et il en perçoit instantanément les résultats comme c'est le cas lorsque l'on lance un processus sous Unix par l'intermédiaire d'un «shell».

Cette instantanéité implique alors que l'exécution des scripts soit **interprétée**. Un environnement de script est alors constitué d'un interpréteur fonctionnant sur le schéma suivant : l'utilisateur saisit ses commandes ou scripts, l'interpréteur analyse et vérifie la syntaxe des commandes, cette analyse produit une forme interne représentant la commande et finalement l'interpréteur exécute cette représentation en agissant sur les ressources à travers des interfaces de programmation (API). Nous reprendrons ce schéma pour expliquer le fonctionnement de l'interpréteur du langage CorbaScript.

Ce mode interactif est très souple pour exploiter un environnement lorsque l'utilisateur veut directement obtenir un résultat. Mais il devient très lourd d'utilisation pour réaliser des tâches répétitives et complexes : l'utilisateur doit alors ressaisir toute la suite des commandes nécessaires, cette opération pouvant malheureusement introduire des erreurs.

Pour cette raison, la plupart des environnements de script permettent l'enregistrement de suites de commandes afin de les rejouer ultérieurement. Cela est souvent réalisé par l'emploi de fichiers stockant les scripts. Ces scripts peuvent alors être considérés comme de nouveaux exécutables fonctionnant sur le mode traitement par lots («batch») avec ou sans l'intervention de l'utilisateur. De plus, certains environnements permettent de composer ou de réutiliser des scripts pour en former d'autres plus complexes : par exemple les scripts de configuration dans les environnements Unix.

### 3.1.2 Survol des environnements de script

Afin de mieux nous rendre compte de la simplicité d'utilisation des scripts, nous allons observer leur utilité dans les environnements informatiques classiques tels que les systèmes d'exploitation, les langages de programmation, les bases de données et les applications généralistes en dégagant les points importants les caractérisant.

#### 3.1.2.1 Les systèmes d'exploitation

Les systèmes d'exploitation modernes sont toujours accompagnés d'un ou de plusieurs langages de commandes, par exemple, il existe de nombreux interpréteurs de commandes pour le système Unix (comme *sh*, *ksh*, *csh*).

Ces langages permettent de piloter, d'administrer et de configurer les ressources fournies par ce système : créer un utilisateur, déplacer des fichiers, lancer un ensemble de processus et bien d'autres tâches. Sans l'existence de ces langages<sup>1</sup>, nous devrions, pour chaque tâche, écrire un programme en C utilisant l'API d'Unix (comme *open*, *read*, *fork*).

Les scripts permettent de réaliser des tâches complexes en assemblant des exécutables et des scripts existants par l'intermédiaire, par exemple, du mécanisme de tubes Unix. Mais cette **composition de scripts** ne permet que de manipuler les deux types de ressources de

---

1. Heureusement pour nous, ce n'est pas le cas !

base disponibles sur un système «à la» Unix : des fichiers et des processus. Dans le contexte des objets répartis, il faut pouvoir **accéder à des milliers de types de ressources différentes**.

De plus, ces langages ont rarement un **pouvoir d'expression important** car ils n'offrent qu'un nombre très limité de types de données (entier, chaîne) et peu d'instructions et de constructions syntaxiques pour écrire des scripts complexes et réutilisables.

### 3.1.2.2 Les langages de programmation

Pour répondre à ce manque, quelques langages de script intègrent la puissance d'expression de langages de haut niveau : une grande variété des types de données, des procédures, des modules, des concepts de l'approche orientée objet et/ou de la programmation fonctionnelle comme c'est le cas pour le langage Python [Lut96]. Ces langages de scripts deviennent alors de véritables langages de programmation offrant un cycle de développement favorisant le **prototypage rapide d'applications** (comme Tcl/Tk [Ous94], Python [vR96], SmallTalk [GR83]).

Comme ces langages sont interprétés, leur moteur d'exécution peut apporter de nombreuses fonctionnalités aux utilisateurs : le **typage dynamique** vérifié à l'exécution, le **ramasse-miettes local** gérant automatiquement la désallocation de la mémoire et la **documentation en ligne**. Ces fonctionnalités améliorent la productivité des utilisateurs mais entraînent forcément un surcoût à l'exécution.

A l'opposé d'un langage de commandes dédié à un système d'exploitation, l'interpréteur de tels langages de programmation assure la portabilité des scripts sur différents environnements d'exécution (matériels et systèmes d'exploitation), citons comme exemple le langage Perl<sup>2</sup> [WS91] pour écrire des scripts indépendants du système de fichiers sous-jacent. Pour les objets répartis, il faut que le langage de script soit indépendant le plus possible des spécificités du bus Corba utilisé pour pouvoir obtenir cette **portabilité des scripts**.

Java, le langage portable orienté objet défini par Sun [GM95], ne peut être inclus dans cette catégorie des langages de programmation de scripts : un programme Java est compilé pour produire un exécutable portable (du pseudo code ou ByteCode Java), ce pseudo code sera ensuite chargé dynamiquement et interprété par la machine virtuelle Java incluse par exemple dans un navigateur WWW. Dans le chapitre 4, nous reviendrons tout de même sur l'intérêt que peut offrir Java pour accéder à des objets Corba à travers le WWW.

### 3.1.2.3 Les bases de données

Les principes de conception d'applications et de services distribués (même à base d'objets voir chapitre 2) impliquent que l'utilisateur final n'a accès à ces services qu'à travers les outils conçus et prévus par les informaticiens. Par analogie, dans les premières bases de données, l'utilisateur ne pouvait manipuler les données qu'à travers les procédures définies par les informaticiens. Ce point restreint alors fortement l'accès aux services et données et brime l'intelligence et la créativité des utilisateurs à répondre leurs besoins spécifiques.

Dans les bases de données relationnelles, l'utilisateur peut lui-même mettre en place ses propres procédures à travers l'utilisation de langages d'interrogations (SQL [ISO92], OQL [CK94]). Ainsi, les informaticiens ne se concentrent plus que sur le schéma fonctionnel des données (tables et relations) et sur les procédures de base du système d'informations,

---

2. Perl : Pratical Extraction and Report Language

les utilisateurs finaux pouvant très facilement développer leurs propres traitements pour **répondre rapidement à leurs besoins spécifiques**.

Pour satisfaire ces besoins, l'utilisateur construit interactivement ou par le biais d'interfaces graphiques des requêtes d'exploitation des bases de données en composant avec les cinq opérations du modèle relationnel (traduites en SQL par SELECT, INSERT, DELETE, UPDATE, CREATE). Dans les bases de données orientées objet et avec OQL, l'utilisateur peut en plus invoquer les méthodes définies sur les objets.

Notre proposition est donc d'introduire cette flexibilité dans les services orientés objet conçus avec Corba. Les informaticiens pourront concevoir toutes sortes de services en les encapsulant dans des objets répartis et interopérables sur le réseau. Les utilisateurs pourront utiliser ces objets soit par des programmes spécifiquement développés pour répondre aux besoins du service ou soit par des scripts invoquant dynamiquement ces objets. Ainsi, l'utilisateur, qui doit réaliser une nouvelle tâche, peut écrire un script sans attendre que son service informatique prenne en compte ses besoins en lui développant un nouveau programme.

#### 3.1.2.4 Les applications généralistes

En observant le contexte des applications généralistes, nous nous apercevons que la plupart de ces applications intègrent un langage de script ou de macro-commande (comme Visual Basic dans Excel, Lisp dans Emacs). Plus ou moins facilement, ces langages offrent la possibilité de paramétrer, d'adapter et/ou de personnaliser ces applications aux besoins des utilisateurs. Cette flexibilité et cette souplesse autorisent donc à faire du «sur-mesure» à partir d'applications de type «prêt-à-porter».

Les environnements d'intégration d'applications, comme Windows ou OpenDoc [OHE96], vont encore plus loin dans cette souplesse car les utilisateurs peuvent construire de nouvelles applications à partir d'applications préexistantes. Ces nouvelles applications sont des scripts, par exemple en Visual Basic, qui assemblent les composants OLE existants dans les applications généralistes.

Si nous voulons bénéficier de la vision «utopique» de l'OMG d'une industrie du composant logiciel interopérable, il faudra bien disposer d'**une «colle» logicielle pour assembler ces composants**. Etant données les difficultés pour utiliser les objets Corba (se reporter section 2.4), les langages de programmation classiques ne peuvent pas apporter une réponse satisfaisante. Pour pallier à ce problème, nous proposons un nouveau langage basé sur le paradigme de script pour accéder dynamiquement à tout type d'objets Corba.

#### 3.1.3 Des scripts pour les objets répartis Corba

Nous venons de voir les intérêts et les bénéfices que les utilisateurs peuvent tirer de l'emploi de langages de script dans des environnements classiques comme les systèmes d'exploitation, les bases de données et les applications généralistes. Les scripts favorisent l'accès et l'exploitation de ces ressources sans nécessiter l'écriture de programmes complexes, nous apportant ainsi les bénéfices suivants :

- **La simplicité d'utilisation** : un script est souvent plus facile à réaliser et plus concis (typage dynamique, ramasse-miettes) que son équivalent écrit dans un langage de programmation classique. La simplicité des langages de script permet aux utilisateurs même débutants de concevoir des petits programmes répondant à leurs besoins.

- **L’augmentation de la productivité** : cette facilité rend alors leur production plus souple et plus rapide, l’utilisateur pouvant prototyper ses scripts en mode interactif et ensuite les exploiter en mode traitement par lots. Cela favorise aussi l’échange de scripts entre utilisateurs, ils peuvent les adapter, les étudier pour répondre à leurs besoins.
- **La réduction des coûts** : la simplicité et la productivité entraînent respectivement la réduction des coûts de formation des utilisateurs et des coûts d’exploitation des environnements informatiques.

Dans le chapitre 2, nous avons vu que la conception d’un service avec Corba est complexe. Un service nécessite la production d’un grand nombre de programmes clients des objets, pour les utilisateurs finaux ainsi que pour la bonne marche du service comme par exemple, les outils de configuration et d’administration du service de nommage.

Nous pensons que beaucoup de ces programmes peuvent être remplacés avantageusement par des scripts invoquant les objets constituant un service. Ces scripts améliorent fortement la mise en œuvre des objets répartis Corba dans les phases suivantes du cycle de développement :

- **La conception et le prototypage** : durant la phase de conception d’un service réparti, deux problèmes importants se posent : le choix des interfaces IDL et le choix de la répartition des objets. Actuellement, il n’existe pas de solution miracle pour répondre à ces deux problèmes, seuls l’expérience et le savoir-faire permettent de trouver les « bons » choix. Dans ces conditions, il est nécessaire de pouvoir prototyper rapidement afin d’évaluer ces choix fondamentaux. Mais prototyper dans un langage compilé tel que le C++ implique un cycle de développement complexe et coûteux d’où l’intérêt d’employer un langage interprété ayant un cycle de développement plus court pour réaliser des maquettes du service.
- **Le développement et les tests** : durant le développement d’une application client/serveur orientée objet avec Corba, les développeurs doivent écrire de nombreux morceaux de programmes pour vérifier la validité et le bon fonctionnement de leurs objets Corba. Ces codes de test sont difficiles à mettre au point à cause de la complexité des règles de projections. De plus, ils deviennent inutiles lorsque les composants sont correctement implantés. Dans ce contexte, un interpréteur de commandes permet alors de gagner beaucoup de temps et d’efforts. Il devient possible de tester immédiatement et interactivement les implantations des objets durant leurs développements.
- **La configuration et l’administration** : la plupart des services et des canevas d’objets nécessitent de nombreux programmes clients pour configurer, administrer et connecter les objets les constituant (comme le service de nommage). Ce nombre de programmes clients est souvent dépendant du nombre d’opérations décrites par les interfaces IDL de ces objets. Généralement, ces programmes sont conçus avec l’approche statique des souches IDL. Un langage de script dynamique est alors une excellente alternative pour supporter cette multitude de programmes car ils pourront être écrits en quelques instructions et évoluer rapidement pour répondre aux besoins des administrateurs de ces services.
- **L’utilisation des composants** : l’utilisateur expérimenté peut lui-même concevoir des scripts pour répondre à ses besoins spécifiques. Ainsi, à partir des composants disponibles sur le bus, il extrait les informations qui lui sont pertinentes sans devoir passer par son service informatique.

- **L’assemblage de composants logiciels** : les scripts peuvent être utilisés pour assembler des composants existants afin d’en créer de nouveaux. Ces nouveaux composants encapsulent l’ensemble des fonctionnalités des composants connectés et fournissent de nouvelles fonctionnalités. Ainsi, nous obtenons une «colle logicielle» pour construire de nouveaux objets par agrégation d’objets existants. De plus, ces nouveaux composants peuvent être utilisés depuis des applications Corba comme des objets classiques.
- **L’évolution** : même si les composants évoluent ou si de nouveaux apparaissent, l’emploi de scripts permet de les découvrir dynamiquement à l’exécution et donc de pouvoir les utiliser dès qu’ils sont disponibles.

Comme nous venons de le voir, un langage de script peut rendre de nombreux services au cours du cycle de vie d’un service orienté objet réparti. Ces différents usages impliquent alors que le langage de script fournisse les mécanismes pour découvrir, invoquer et naviguer dans des objets Corba et pour pouvoir implanter des objets par l’intermédiaire de scripts. La navigation et l’exploitation de vastes graphes d’objets de types disparates impliquent d’acquérir dynamiquement les souches des types rencontrés car le langage de script ne peut pas connaître à l’avance tous les types IDL.

### 3.1.4 Liaisons entre les langages existants et les objets répartis

Les langages de script pourraient donc favoriser l’exploitation d’objets Corba. Mais comment intégrer dans de tels langages l’accès à ces objets ? L’architecture Corba propose deux interfaces pour invoquer les objets : l’interface d’invocation statique (S.I.I.) constituée de souches IDL générées par un compilateur IDL et l’interface d’invocation dynamique (D.I.I.) qui peut être considérée comme une souche générique d’accès à tout objet Corba. Cette intégration doit aussi définir une projection du langage IDL vers le langage de script afin de pouvoir manipuler les définitions Corba à partir des scripts.

Ces deux approches ont été mises en œuvre respectivement dans les environnements Python+ILU [Jan95] et TclDii [ASG<sup>+</sup>94]. Nous allons étudier ces deux environnements et montrer pourquoi ils ne sont pas totalement satisfaisants. Pour illustrer l’utilisation de ces deux environnements, nous prenons l’exemple d’interface IDL suivant (cf. figure 3.1).

```

// nom du fichier IDL 'grille.idl'

typedef unsigned short Positif;
exception mauvaiseCoordonnee {
    Positif x, y;
};
interface grille {
    attribute Positif hauteur, largeur;
    float consulter (in Positif x, in Positif y)
        raises (mauvaiseCoordonnee);
    void modifier (in Positif x, in Positif y, in float valeur)
        raises (mauvaiseCoordonnee);
};

```

FIG. 3.1 - L’interface IDL grille

Cette interface décrit simplement des objets de type *grille* avec deux opérations pour *consulter* et *modifier* l’état de la grille et deux attributs pour obtenir les propriétés *hauteur* et *largeur*. L’exception *mauvaiseCoordonnee* est retournée par les opérations *consulter* et

*modifier* lorsque les coordonnées passées en paramètres sont incorrectes (débordement des limites de la grille).

### 3.1.4.1 Projection vers un langage de script

Le négociateur de requêtes à objets ILU<sup>3</sup>[Jan96] est un bus compatible Corba (du point de vue d'IDL, des projections et d'IIOP) conçu au Xerox PARC auquel il ajoute de nombreuses fonctionnalités : le langage de description ISL<sup>4</sup> plus puissant que l'OMG-IDL, un ramasse-miettes réparti et de nombreux protocoles de transports. Cette implantation se distingue par le nombre de langages interfacés avec l'ORB : C, C++, Modula-3, Java, Common Lisp ainsi que Python. Python [WvRA96, vR96] est un langage interprété portable développé par Guido van Rossum au CWI à Amsterdam dont les principales caractéristiques sont l'intégration dans un même langage du typage dynamique, d'un ramasse-miettes local, de la modularité (modules de code chargés dynamiquement à l'exécution), des concepts de la programmation fonctionnelle (fonctions de lambda calcul) et orientée objet (classes, héritage, polymorphisme et instances).

Les règles de projection d'ILU vers le langage interprété Python [Jan95] définissent la traduction des constructions IDL (module, interface, opération, ...) en constructions Python (module, classe, fonction). Le compilateur *python-stubber* génère deux modules Python l'un contenant l'interface d'invocation statique où chaque interface IDL est traduite en une classe souche écrite en Python, et l'autre pour l'interface des squelettes statiques où chaque interface IDL est traduite en une classe squelette écrite en Python. Par instantiation des classes souches, les programmes clients peuvent invoquer des objets Corba distants écrits dans n'importe quel langage. Par héritage des classes squelettes, un programme Python peut implanter des objets Corba accessibles depuis n'importe quel lieu sur le bus.

L'exemple suivant affiche les propriétés de la grille (comme *largeur* et *hauteur*), puis il modifie la valeur d'une case de la grille et enfin affiche la valeur de la case modifiée.

```
import ilu, grille

refGrille = ilu.LookupObject ("référence_objet_sous_forme_d_une_chaine", grille.grille)

h = refGrille.hauteur()
l = refGrille.largeur()
print "la taille de la grille est de ", h, 'x', l, " cases"

refGrille.modifier (10, 10, 100)

valeur = refGrille.consulter (10,10)
print "grille[10,10]=", valeur
```

La puissance d'expression du langage Python associée aux règles de projection de Python-ILU permet d'invoquer les objets Corba très naturellement sans dérouter les développeurs Python. L'invocation d'opérations et la consultation d'attributs s'expriment par la «classique» notation pointée définie dans ce langage (comme *refGrille.hauteur()*). Pour pouvoir utiliser un type d'objets Corba, il suffit d'importer un module Python contenant l'interface d'invocation statique générée par le compilateur *python-stubber* (*import grille*). La fonction *LookupObject* du module *ilu* est l'équivalent de l'opération *string\_to\_object* définie par l'OMG pour convertir une chaîne en une référence d'objet<sup>5</sup>.

---

3. ILU : Inter-Language Unification

4. ISL : Interface Specification Language

5. en compliquant un peu l'exemple, nous aurions pu aussi utiliser le service de nommage.

De plus, la liaison Python-ILU permet d'implanter les objets Corba par simple héritage des classes squelettes générées par le compilateur IDL. Sans entrer dans les spécificités de la syntaxe du langage Python, nous pouvons voir qu'il est relativement simple d'implanter un objet Corba en Python. La classe d'implantation *MaClasseGrille* hérite de la classe squelette *grille\_\_skel.grille*. La méthode `__init__` permet de définir le traitement à faire à l'instanciation d'un objet (comme la dernière ligne de l'exemple suivant). Ensuite, chaque opération (ou attribut) est codée par une méthode Python.

```
import grille__skel

class MaClasseGrille (grille__skel.grille):
    def __init__ (self,hauteur,largeur):
        self.h = hauteur
        self.l = largeur
        self.valeurs = # une structure de données quelconque

    def hauteur (self):
        return self.h

    def largeur (self):
        return self.l

    def consulter (self,x,y):
        return self.valeurs[x][y]

    def modifier (self,x,y,valeur):
        self.valeurs[x][y] = valeur

uneInstance = MaClasseGrille (100,100)
```

Comme on peut le constater, un langage interprété de haut niveau comme Python simplifie grandement l'utilisation et l'implantation d'objets. Il élimine tous les problèmes liés au typage statique et à la gestion mémoire que nous avons rencontrés avec, par exemple, le langage compilé C++. Mais cette approche par projection ne résout que partiellement les difficultés d'exploitation d'un service orienté objet :

- **La modification des interfaces** : en cas de modifications des interfaces IDL, même mineures comme par exemple changer pour la grille le type *float* en *double*, il est nécessaire de régénérer les souches et squelettes Python pour pouvoir utiliser ces objets modifiés. Au cas où les souches ne seraient pas régénérées, le noyau de communication Corba détectera à l'exécution les invocations qui ne sont plus correctement typées.
- **La liaison statique** : si l'utilisateur navigue à travers les objets Corba par l'intermédiaire d'un service de nommage, il ne peut utiliser que les types dont il dispose dans un module souche car il doit explicitement importer ces modules. En fait, il lui est impossible de découvrir dynamiquement d'autres types pour lesquels il ne disposerait pas d'un module souche.

En résumé, nous tirons les conclusions suivantes : notre langage de script doit fournir une notation simple pour l'accès aux objets mais il doit surtout supporter l'évolution et la navigation dynamique dans des graphes complexes d'objets. L'approche par génération de souches statiques n'est pas la solution adéquate, il faut plutôt se tourner vers les fonctionnalités dynamiques de Corba comme l'interface d'invocation dynamique pour construire les requêtes à la volée et le référentiel des interfaces pour contrôler le typage des invocations.

### 3.1.4.2 Intégration de l'invocation dynamique dans un langage de script

Comme nous l'exposons depuis le début de cette thèse, Corba est principalement utilisé pour réaliser des applications interopérables et réparties en mettant en œuvre l'invocation statique à travers des souches IDL pré-générées. Peu de travaux, aussi bien dans le milieu de la recherche que dans l'industrie, ont expérimenté l'approche dynamique (l'invocation dynamique) et la réflexivité (le référentiel des interfaces) offertes par un bus Corba.

Le projet de recherche Web\* [ASM<sup>+</sup>95] du laboratoire de recherche CERC (Concurrent Engineering Research Center de la West Virginia University aux U.S.A.) travaille sur l'accès à de grands ensembles d'objets répartis (actuellement des systèmes d'informations pour la santé) à travers l'interface graphique, simple et universelle du WWW. Une des composantes de leur environnement est l'outil TclDii [ASG<sup>+</sup>94] une interface entre le langage de script Tcl [Ous94] et l'interface d'invocation dynamique du bus Corba. Cette intégration permet à des scripts Tcl d'invoquer dynamiquement n'importe quel objet Corba mais ne permet pas d'en implanter comme avec Python+ILU. La version actuelle est réalisée sur Orbix, et les concepteurs ont l'intention de porter TclDii sur d'autres ORBs comme VisiBroker (ex-Orbeline).

Un script Tcl est composé d'une suite de commandes de base comme l'affectation d'une variable (*set*), l'affichage d'un message (*puts*), la déclaration de procédures et le contrôle de flots (*if*). L'exécution d'un script Tcl est relativement simple et peut être décrite comme suit<sup>6</sup>. A chaque nom de commande est associé une fonction C qui réalise le traitement. En analysant le script, l'interpréteur détermine le prochain nom de commande à exécuter et appelle la fonction C associée. Lorsque la fonction C est appelée, elle reçoit en paramètre la liste des chaînes de caractères qui suivent le nom de la commande : par exemple la commande *set* reçoit le nom de la variable à affecter et la nouvelle valeur. De nouvelles commandes peuvent être ajoutées à l'interpréteur grâce à l'interface de programmation du moteur de l'interpréteur pour associer une fonction C à un nom.

TclDii utilise ce mécanisme d'extension du langage Tcl pour introduire de nouvelles commandes dédiées à l'utilisation de Corba. La principale commande est *orb\_call* qui permet d'invoquer n'importe quel objet Corba par l'intermédiaire de l'invocation dynamique. Cette commande prend en paramètres la référence de l'objet à invoquer, le nom de l'opération (ou attribut), le type IDL de retour et la liste des arguments (type IDL, valeur). TclDii définit pour chaque type IDL une représentation sous forme de chaîne : les flottants IDL sont codés par "f", les entiers non signés par "us", le type void par "v" et ainsi de suite pour tous les types IDL<sup>7</sup>. Ainsi, notre exemple précédent se traduit alors en TclDii de la manière suivante :

```
set refGrille "une_référence_d_objet_sous_la_forme_d_une_chaine"

set h [orb_call $refGrille _get_hauteur us]
set l [orb_call $refGrille _get_largeur us]
puts "la taille de la grille est de $h x $l cases"

orb_call $refGrille modifier v us 10 us 10 f 100

set valeur [orb_call $refGrille consulter f 10 us 10 us]
puts "grille[10,10]=$valeur"
```

Outre le faible pouvoir d'expression du langage Tcl et son manque de typage, l'approche TclDii n'est pas satisfaisante car elle ne masque pas du tout la complexité de l'interface

6. Nous simplifions le mode de fonctionnement réel de Tcl pour alléger cette présentation.

7. En fin de compte, TclDii reprend directement la représentation interne des TypeCodes IDL d'Orbix.

d'invocation dynamique : l'utilisateur doit connaître les interfaces IDL des objets qu'il veut invoquer et il doit préciser les informations de typage (les chaînes TypeCode spécifiques à Orbix) pour construire les requêtes. Conscient de ce problème et des limites intrinsèques de Tcl, les concepteurs de TclDii envisagent d'interfacer l'invocation dynamique avec le langage interprété Scheme Lisp (cette nouvelle interface est appelée ScmDii).

Mais analysons pourquoi cette approche est décevante par rapport à la simplicité que Python+ILU apporte. Premièrement, l'accès à des objets s'exprime plus naturellement en employant la notation pointée (*objet.opération(paramètres)*) plutôt qu'une notation procédurale (*commande objet opération paramètres*). Deuxièmement, il faut que le langage de script ait une puissance d'expression convenable et un nombre de types significatif pour pouvoir aisément écrire des scripts qui accèdent à des spécifications IDL fortement typées. Troisièmement, l'utilisateur ne devrait pas avoir à préciser les informations de typage des requêtes Corba comme c'est le cas avec TclDii et sûrement avec le futur ScmDii.

En conclusion, nous considérons qu'utiliser uniquement l'invocation dynamique pour accéder aux objets n'est pas suffisant. Le langage de script doit utiliser les métadonnées de typage contenues dans le référentiel des interfaces. Ceci facilitera la tâche des utilisateurs et permettra de contrôler, avant émission, la conformité des invocations.

### 3.1.5 Pourquoi un nouveau langage ?

Comme nous l'avons déjà signalé, le nombre de langages de script fournissant une interface vers Corba est très réduit, les propositions actuelles ne nous conviennent pas car soit elles sont basées sur une projection statique (Python+ILU) limitant l'exploitation de grands graphes d'objets complexes, soit elles utilisent l'invocation dynamique mais ne masquent pas sa complexité (TclDii) limitant fortement la simplicité d'utilisation. Mais alors pour accéder aux objets Corba par l'intermédiaire de scripts, devons-nous étendre un langage de script existant ou construire de toute pièce un nouveau langage ?

Pour satisfaire nos besoins, un langage de script existant devait intégrer des concepts objet pour implanter des objets Corba et une notation orientée objet pour accéder aux objets distants. De plus, son noyau d'exécution devait fournir des mécanismes d'extension pour que nous puissions mettre en œuvre l'interface d'invocation dynamique (DII) et l'interface de squelettes dynamiques (DSI) de Corba. Peu de langages existants réunissent toutes ces fonctionnalités et leurs mécanismes d'extension sont souvent trop limités pour intégrer de manière transparente les complexes mécanismes dynamiques de Corba.

Comme nous étions en pleine phase d'expérimentation et nous ne voulions pas être trop contraints, nous avons choisi de définir notre propre langage. Ce nouveau langage allie la puissance d'expression de Python+ILU avec la liaison dynamique de TclDii. Il utilise le référentiel des interfaces pour réaliser le contrôle dynamique du typage. Les invocations sont effectuées à travers l'interface d'invocation dynamique. Nous présentons les constructions syntaxiques de ce nouveau langage dans la section suivante, mais à titre de comparaison, voici comment se traduit en CorbaScript l'exemple précédent :

```
refGrille = grille ("référence_objet_sous_forme_d_une_chaine")

h = refGrille.hauteur
l = refGrille.largeur
print "la taille de la grille est de ", h, 'x', l, " cases\n"

refGrille.modifier (10, 10, 100)

valeur = refGrille.consulter (10,10)
print "grille[10,10]=", valeur
```

Précisons quelques points : les variables sont créées à leur première utilisation, toutes les définitions IDL (*grille*) contenues dans le référentiel des interfaces sont automatiquement accessibles depuis les scripts, l'appel d'opération ou la consultation d'un attribut sont exprimés par la notation pointée. Ces invocations sont réalisées grâce à l'interface d'invocation dynamique (DII).

De plus, toute interface IDL peut être implantée en CorbaScript sans nécessiter la génération de squelette. Le décodage et la vérification des appels sont réalisés par le moteur de l'interpréteur en mettant en œuvre l'interface de squelettes dynamiques (DSI).

```
class MaClasseGrille (grille) {
  proc __MaClasseGrille__ (self, hauteur, largeur) {
    self.__grille__ ()
    self.h = hauteur
    self.l = largeur
    self.valeurs = # une structure de donnée quelconque pour implanter une matrice
  }

  proc _get_hauteur (self)           { return self.h }
  proc _get_largeur (self)          { return self.l }
  proc consulter (self,x,y)         { return self.valeurs[x][y] }
  proc modifier (self,x,y,valeur) { self.valeurs[x][y] = valeur }
}

uneInstance = MaClasseGrille (100,100)
```

Nous reviendrons dans la section suivante sur les détails syntaxiques de ce langage comme la construction d'une classe (*MaClasseGrille*), l'héritage (*grille*), l'initialisation d'une instance (*\_\_MaClasseGrille\_\_*), les méthodes (*\_get\_hauteur*, *\_get\_largeur*, *consulter*, *modifier*) et l'instanciation d'une classe.

## 3.2 Survol du langage CorbaScript

L'introduction précédente sur les langages de script nous a amenés à définir et caractériser leurs intérêts pour l'accès dynamique à des objets Corba. Nous avons justifié la nécessité de définir un nouveau langage de script dédié à Corba. Dans cette section, nous définissons les caractéristiques de ce nouveau langage CorbaScript et détaillons ses constructions sémantiques et syntaxiques.

### 3.2.1 Les fonctionnalités de CorbaScript

La création d'un nouveau langage nécessite toujours de définir les fonctionnalités et les constructions syntaxiques. Le problème crucial est de définir la puissance d'expression du langage.

Le premier prototype du langage CorbaScript permettait seulement d'exprimer des séquences d'instructions invoquant les objets Corba. Au fur et à mesure de nos expérimentations, nous avons introduit de nouvelles constructions : des procédures, des modules puis des classes. Dans le second et actuel prototype, nous avons jugé la liste de fonctionnalités suivantes comme étant intéressantes et nécessaires<sup>8</sup> pour la conception de scripts pour les objets Corba :

- **Interactivité et interprétation** : CorbaScript est un interpréteur de script accessible en ligne de commandes (interactivement) ou en mode traitement par lots grâce à des fichiers de scripts. Un script est une suite d'instructions telles que l'affichage

---

8. De nombreuses fonctionnalités manquent mais l'essentiel est présent.

d'une valeur, l'appel d'une opération, l'affectation d'une variable, le contrôle de flots et la gestion des exceptions.

- **Puissance d'expression** : ce langage dispose de nombreux types de données et constructions algorithmiques (variable, test, boucle) pour exprimer des scripts complexes. Toute valeur CorbaScript est un objet, la notation pointée permet d'exprimer l'appel d'une opération, la consultation ou la modification d'un attribut de l'objet. Quelques types de base sont cablés dans l'interpréteur : les entiers, les chaînes, les tableaux, les types de base IDL... Mais toute déclaration de type IDL est aussi accessible par l'intermédiaire du référentiel des interfaces.
- **Accès à tout objet Corba** : de plus, à partir d'un script, l'utilisateur peut invoquer les opérations, consulter ou modifier les attributs de n'importe quel objet Corba distant. L'invocation est réalisée à travers l'interface d'invocation dynamique de Corba (DII).
- **Vérification dynamique du typage** : la conformité des expressions est vérifiée dynamiquement à l'exécution grâce au mécanisme de typage dynamique. Les invocations Corba sont vérifiées grâce au référentiel des interfaces qui stocke la signature de toutes les opérations de tous les types d'objets.
- **Modularité et procédures** : les procédures permettent de factoriser les scripts en entités réutilisables. Le retour de résultat et les paramètres des procédures sont non typés. Ces procédures peuvent être regroupées dans des modules téléchargeables et partagés.
- **Classes et instances** : un script manipule des valeurs locales et des objets distants se comportant ainsi uniquement comme un programme Corba client. Une autre fonctionnalité de CorbaScript est la possibilité d'implanter de nouveaux types d'objets (locaux ou Corba). Il intègre les concepts objet tels que les classes, l'héritage et le polymorphisme. Ainsi, un script peut devenir un serveur d'objets Corba accessible par tout programme Corba compilé et aussi par des scripts.
- **Ramasse-miettes local** : l'utilisateur est affranchi des problèmes de désallocation de la mémoire grâce au mécanisme de ramasse-miettes local inclu dans l'interpréteur CorbaScript.
- **Extensibilité** : les traitements spécifiques ou nécessitant de bonnes performances peuvent être codés dans le langage C++. L'interpréteur CorbaScript est un canevas extensible auquel nous pouvons ajouter de nouveaux types d'objets directement accessibles depuis les scripts.
- **Aide en ligne** : l'utilisateur peut consulter interactivement les définitions IDL pour découvrir quelles sont les opérations d'une interface, les champs d'une structure ou bien le contenu d'un module. Cette aide en ligne améliore la productivité des utilisateurs.

Cette liste de fonctionnalités montre que CorbaScript est un langage de programmation de haut niveau permettant de développer des scripts complexes. Dans la suite de cette section, nous allons passer en revue les constructions offertes dans CorbaScript telles que : les *expressions*, les *instructions*, les *procédures*, les *modules* et les *classes*.

### 3.2.2 Les expressions CorbaScript

Après avoir lancé l'interpréteur CorbaScript (*cssh*) en mode interactif, l'utilisateur peut saisir ses scripts.

```
le_prompt_unix> cssh
CorbaScript 1.0 (August 1 1996)
Copyright 1996 LIFL, France
>>>
```

Les scripts peuvent alors manipuler toutes sortes d'expressions comme des constantes, les types Corba de base, des variables, des expressions booléennes, arithmétiques et de comparaison, des tableaux, les types décrits en OMG-IDL, des références d'objets et des invocations d'opérations et d'attributs.

#### 3.2.2.1 Les valeurs et types de base

Le tableau 3.1 énumère tous les types de valeurs disponibles de base dans CorbaScript. Les cinq premiers types représentent les types primitifs classiques à tout langage : les entiers, les réels, les booléens, les caractères et les chaînes. Ces types sont cablés dans la syntaxe du langage pour optimiser les traitements.

La création de valeurs IDL se fait par l'intermédiaire de fonctions spécifiques (*CORBA.Long*, *CORBA.String*) cablées dans l'interpréteur. Cette syntaxe est un peu lourde mais nécessaire pour pouvoir faire la différence entre un entier 16 bits non signé et un entier 32 bits qui sont deux types de données distincts pour Corba. Cependant la majeure partie du temps, l'utilisateur n'a pas à préciser cette information de typage car l'interpréteur fera automatiquement les conversions de type nécessaires.

Type de valeur	Type CorbaScript	Exemple de valeur
constante entière	long	10
constante réelle	double	3.14
constante booléenne	boolean	<i>true</i> et <i>false</i>
constante caractère	char	'c'
constante chaîne	string	"bonjour"
type IDL null	CORBA.Null	CORBA.Null()
type IDL void	CORBA.Void	CORBA.Void()
entier IDL signé sur 16 bits	CORBA.Short	CORBA.Short(-16)
entier IDL non signé sur 16 bits	CORBA.UShort	CORBA.UShort(16)
entier IDL signé sur 32 bits	CORBA.Long	CORBA.Long(-32)
entier IDL non signé sur 32 bits	CORBA.Ulong	CORBA.UShort(32)
réel IDL sur 32 bits	CORBA.Float	CORBA.Float(32.32)
réel IDL sur 64 bits	CORBA.Double	CORBA.Double(64.64)
booléen IDL	CORBA.Boolean	CORBA.Boolean(true)
octet IDL	CORBA.Octet	CORBA.Octet(100)
caractère IDL	CORBA.Char	CORBA.Char('c')
chaîne de caractères IDL	CORBA.String	CORBA.String("une chaîne")
type Any IDL	CORBA.Any	CORBA.Any(CORBA.Long(10))
type TypeCode IDL	CORBA.TypeCode	CORBA.TypeCode(CORBA.String)
Référence d'objet IDL	CORBA.Object	pas de constructeur

TAB. 3.1 - Liste des types de base de CorbaScript

Corba dispose de deux types de métadonnées *TypeCode* et *Any* pour respectivement stocker un type IDL et une valeur Corba typée. Les références d'objet n'ont pas de constructeurs; nous verrons par la suite comment référencer un objet Corba.

Les valeurs CorbaScript sont représentées par des objets et les types sont eux-mêmes des valeurs CorbaScript. Chacun de ces objets possède une propriété (*\_type*) référant son type CorbaScript. Les chaînes possèdent en plus une propriété indiquant leur taille (*length*). L'opération *\_toString()* disponible sur tous les objets CorbaScript permet d'obtenir la représentation d'un objet sous forme d'une chaîne de caractères.

```
>>> 10._type
< type long >
>>> "chaîne"._type
< type string >
>>> "une chaîne".length
10
>>> v = 3.14
>>> v._toString()
"3.14"
>>> v._toString()._type
< type string >
```

### 3.2.2.2 Les variables

Toute valeur CorbaScript peut être affectée à une variable par l'opérateur `=`. Il n'y a pas besoin de déclarer les variables car elles sont créées à leur première utilisation.

```
>>> une_variable = "une valeur"
>>> une_variable
"une valeur"
>>>
>>> v = CORBA.Long
>>> v(10)
CORBA.Long(10)
>>>
```

Les variables sont uniquement des noms symboliques pouvant désigner n'importe quelle valeur CorbaScript (une donnée ou un type) un peu comme en SmallTalk ou en Python où tout est objet. Ainsi dans l'exemple ci-dessus, les deux symboles (*CORBA.Long* et *v*) désignent le même objet CorbaScript (c'est à dire la représentation interne à l'interpréteur du type IDL entier signé sur 32 bits) et s'utilisent donc de la même manière.

Lorsqu'un objet CorbaScript n'est plus accessible par une variable ou bien une structure interne de l'interpréteur, il est alors automatiquement récupéré par le ramasse-miettes.

### 3.2.2.3 Les opérateurs

Comme dans tous les langages de haut niveau, CorbaScript fournit une gamme complète d'opérateurs pour construire des expressions complexes. Les opérandes d'une expression sont convertis vers des valeurs compatibles lorsque leurs types sont différents. CorbaScript dispose des opérateurs suivants :

1. **Les opérateurs arithmétiques** sont l'addition (`+`), la soustraction (`-`), la multiplication (`*`), la division (`/`), la division entière (*div*) et le modulo (*mod*). Les chaînes supportent uniquement l'opérateur d'addition.
2. **Les opérateurs de comparaisons** sont l'égalité (`==`), l'inégalité (`!=`), l'infériorité (`<` et `<=`) et la supériorité (`>` et `>=`). Ces opérateurs produisent toujours un résultat booléen.

3. **Les opérateurs logiques** sont le «et» logique (*and* ou *&&*), le «ou» logique (*or* ou *//*) et la négation (*not*). Les opérandes de ces opérateurs ne peuvent être que des expressions booléennes.
4. **L'opérateur d'évaluation de chaîne** (i.e. *eval*) permet d'évaluer une chaîne contenant un script et retourne la dernière valeur produite par cette chaîne.

Les quelques lignes suivantes illustrent ces opérateurs, les ambiguïtés sont réglées par le parenthésage des expressions.

```
>>> "bonjour le monde" + '!'
"bonjour le monde!"
>>>
>>> 100 != 200
true
>>> "chaîne"._type == string
true
>>> ( "chaîne" < "chaîne2" ) and not (100 == 10 * 10)
false
>>> eval ("10 / 2")
5
```

### 3.2.2.4 Les tableaux

Les tableaux sont un autre type de base de CorbaScript permettant de créer des structures de données complexes. L'exemple suivant illustre leurs possibilités : ils peuvent contenir des valeurs de types différents (simples mais aussi d'autres tableaux), la propriété *length* indique leur taille et ils sont dynamiquement modifiables (*append* pour ajouter un élément, l'opérateur *//* pour modifier un élément).

```
>>> t = [ 1, 'c', "texte" ]
>>> t
[1, 'c', "texte"]
>>> t.length
3
>>> t1 = t
>>> t.append(3.14)
>>> t1
[1, 'c', "texte", 3.14]
>>> t + [ t1 ]
[1, 'c', "texte", 3.14, [1, 'c', "texte", 3.14]]
>>> t[0] = t[0] + 10
>>> t
[11, 'c', "texte", 3.14]
```

### 3.2.2.5 Les types IDL

CorbaScript permet d'accéder à tous les types décrits en OMG-IDL (les définitions de types, les structures, les unions, les séquences et les tableaux). Cependant, ces définitions IDL doivent nécessairement être enregistrées dans le référentiel des interfaces. Lorsque l'utilisateur saisit un nom de type IDL, l'interpréteur va rechercher, dans le référentiel des interfaces de Corba, la description de ce type de données. S'il la trouve, il crée une représentation interne de cette description et la stocke dans le cache des types IDL déjà chargés. La représentation interne est alors un objet CorbaScript directement utilisable dans les scripts. Prenons l'exemple de description IDL suivant :

```
typedef double Reel;
struct Complexe {
    Reel partie_reelle, partie_imaginaire;
};
typedef sequence<Complexe> listeDeComplexes;
```

L'utilisateur peut directement exploiter ces types IDL depuis ses scripts sans passer par une phase de génération de souches IDL. L'interpréteur charge de manière transparente la description de ces types depuis le référentiel et permet alors à l'utilisateur de les exploiter naturellement.

```
>>> r = Reel (10)
>>> c = Complexe (1,r)
>>> c.partie_imaginaire
10
>>> liste = listeDeComplexes ( [1,2], c, [5,6])
>>> liste.length
3
>>> liste[1]
Complexe(1,10)
>>>
>>> Complexe
< OMG-IDL struct Complexe {
    Reel partie_reelle;
    Reel partie_imaginaire;
}; >
```

Dans cet exemple, nous pouvons voir que la notation tableau permet aussi de construire des valeurs IDL composées comme les structures, les unions, les séquences et les tableaux IDL. Comme tous les types sont des objets CorbaScript, l'utilisateur peut les évaluer pour découvrir leur description IDL et donc comprendre comment se servir d'un type.

Pour accéder à une définition contenue dans un module IDL ou dans une interface IDL, il suffit d'indiquer le chemin complet désignant la définition comme par exemple *CosNaming::Name* pour accéder à la définition de la séquence *Name* contenue dans le module *CosNaming*. Plus généralement, la syntaxe est *espace1::...::espaceN::définition* où *espace* désigne un espace de nommage (module ou interface IDL) des déclarations IDL.

### 3.2.2.6 Les objets répartis Corba

Jusqu'ici, nous avons uniquement fait un survol des constructions d'expressions CorbaScript. Ces expressions sont purement locales à l'espace mémoire d'une exécution de l'interpréteur. Mais elles sont nécessaires pour avoir un pouvoir d'expression convenable pour écrire des scripts. Nous allons voir maintenant comment invoquer les objets Corba distants<sup>9</sup>.

**Obtenir une référence** Pour pouvoir invoquer une opération sur un objet, il faut avoir une référence sur cet objet. Nous avons discuté dans le chapitre 2 des différentes manières d'obtenir une référence: la liaison directe à partir d'une chaîne (*object\_to\_string* et *string\_to\_object*), les services d'annuaire ou plus généralement l'invocation d'une opération retournant une référence. Toutes ces solutions sont accessibles depuis CorbaScript.

La syntaxe pour créer une liaison directe est la suivante **une\_interface ("une \_référence\_d\_objet")**. Le nom de l'interface référence un type d'interface IDL enregistré dans le référentiel des interfaces, ce nom peut aussi désigner une interface contenue dans un module (le séparateur ::). La chaîne code la localisation de l'objet sur le bus logiciel.

Comme CorbaScript est actuellement implanté sur Orbix, nous utilisons pour désigner les objets le format employé par Orbix. Chaque objet Orbix est désigné par un triplet d'informations: le nom de machine où l'objet est localisé, le nom du serveur implantant

---

9. Ce qui est tout de même l'objectif de CorbaScript!

l'objet et finalement le nom de l'objet à l'intérieur du serveur. Ainsi, l'expression CorbaScript `grille("UneGrille:ServeurDeGrilles:duff.lifl.fr")` permet de se connecter dynamiquement à l'objet de nom *UneGrille* contenu dans le serveur *ServeurDeGrilles* sur la machine *duff.lifl.fr*, cet objet est du type IDL *grille*.

CorbaScript offre aussi deux autres formes de codage pour référencer un objet : le format interne d'Orbix par exemple `:\duff.lifl.fr:ServeurdeGrilles:UneGrille::IR:grille` et le format IIOP de Corba 2.0 tel que `IOR:03647...62453`.

La première syntaxe est simple et naturelle pour tout utilisateur d'Orbix, mais les deux suivantes sont moins évidentes. Cependant, il n'utilisera que rarement ces notations car il obtiendra les objets qui lui sont nécessaires par l'intermédiaire des services d'annuaire. Dans la section 3.2.5, nous illustrons comment utiliser le Naming Service de Corba avec CorbaScript pour obtenir les objets Corba.

Ces diverses variantes permettent donc d'obtenir une référence vers l'objet Corba désiré. L'interpréteur charge, depuis le référentiel des interfaces, la description de l'interface IDL de l'objet. Il crée alors dynamiquement une souche pour invoquer n'importe quelle opération ou attribut de manière transparente. Ces souches sont conservées dans le cache interne de CorbaScript pour ne plus consulter ultérieurement le référentiel. Ce cache est stocké temporairement dans l'espace mémoire de chaque exécution de l'interpréteur.

**Invoquer les opérations, consulter et modifier les attributs** Reprenons notre précédente interface IDL *grille* afin d'illustrer l'utilisation des objets Corba depuis CorbaScript.

```
// nom du fichier IDL 'grille.idl'

typedef unsigned short Positif;
exception mauvaiseCoordonnee {
    Positif x, y;
};
interface grille {
    attribute Positif hauteur, largeur;
    float consulter (in Positif x, in Positif y)
        raises (mauvaiseCoordonnee);
    void modifier (in Positif x, in Positif y, in float valeur)
        raises (mauvaiseCoordonnee);
};
```

La notation pointée permet alors d'invoquer les opérations, de consulter et de modifier les attributs définis dans l'interface des objets *grille* comme on peut le voir dans l'exemple suivant.

```
>>> refGrille = grille ("MaGrille:ServeurdeGrilles:duff.lifl.fr")
>>>
>>> refGrille.modifier (10, 10, 160)
>>> refGrille.consulter (10,10)
160
>>> h = refGrille.hauteur
>>> h
100
>>> refGrille.largeur = 60
>>> refGrille.largeur
60
```

Nous pouvons remarquer que l'interpréteur convertit automatiquement les valeurs constantes vers les types IDL nécessaires. Dans l'exemple précédent, les valeurs *10* et *60* sont transformées en valeurs IDL de type *unsigned short* et la valeur *160* est convertie en un *float* IDL. Ce mécanisme d'inférence de type simplifie l'écriture des scripts pour les

appels avec passage de paramètres en mode *in*. Pour les paramètres en mode *out* ou *inout*, il est nécessaire de passer des variables qui serviront à stocker les valeurs retournées par l'opération.

**Tout est objet** Comme nous l'avons déjà précisé précédemment, toute valeur CorbaScript possède une propriété (*\_type*) indiquant son type CorbaScript. Cela s'applique aussi aux références d'objet Corba, le type est un objet CorbaScript encapsulant l'interface IDL. L'exemple suivant illustre la réflexivité du langage CorbaScript où tout est objet aussi bien les données, les types mais aussi les opérations et les attributs.

```
>>> t = refGrille._type           # obtenir le type CorbaScript de la référence 'refGrille'
>>> t == grille                   # le type de 'refGrille' est-il bien 'grille'
true                              # eh bien oui ! c'est normal tout est objet.
>>> t                             # évaluons le type
< OMG-IDL interface grille {
  attribute unsigned short hauteur ;
  attribute unsigned short largeur ;
  float consulter (in unsigned short x, in unsigned short y) ;
  void modifier (in unsigned short x, in unsigned short y, in float valeur) ;
} >
>>> t.consulter                  # évaluons une opération
< OMG-IDL operation float grille::consulter (in unsigned short x, in unsigned short y) >
```

Cette réflexivité offre une aide en ligne pour découvrir comment utiliser un service aussi bien pour les utilisateurs novices que pour les développeurs «chevronnés» de scripts.

### 3.2.3 La structuration des scripts

CorbaScript permet donc de construire des expressions complexes accédant dynamiquement à tout type d'objets Corba. Comme un script ne peut pas se limiter à une suite d'expressions, Nous présentons dans la suite les instructions CorbaScript pour l'affichage des valeurs, le contrôle des flots d'exécution, le traitement des exceptions, la création de procédures et de modules de codes réutilisables.

#### 3.2.3.1 L'affichage

Dans le mode interactif, l'interpréteur évalue puis affiche chaque expression entrée. Cependant un script peut aussi afficher un ensemble de valeurs grâce à l'instruction *print*. Les valeurs sont séparées par des virgules. La mise en forme des affichages (retour à la ligne, tabulation) est réalisée par l'emploi de caractères spéciaux (comme *\n*, *\t*, ...).

#### 3.2.3.2 L'alternative

La construction suivante permet d'exprimer des alternatives. La condition doit être une expression booléenne. La partie *else* est facultative.

```
if ( <condition_booléenne> ) {
  <suite d'instructions> si la condition est vraie
} [ else {
  <suite d'instructions> si la condition est fausse
}
]
```

### 3.2.3.3 La boucle conditionnelle

La construction suivante exprime les boucles conditionnelles. La suite d'instructions est exécutée tant que la condition est vraie.

```
while ( <condition_booléenne> ) {
    <suite d'instructions> tant que la condition est vraie
}
```

### 3.2.3.4 L'itération

La construction *for v in <expression\_énumérable> { <suite d'instructions> }* exprime l'itération d'une suite d'instructions sur un ensemble de valeurs énumérables. A chaque itération, la variable *v* référence la prochaine valeur de l'expression énumérable (un intervalle de valeurs, une chaîne, un tableau ou une séquence IDL). Le script suivant montre les possibilités de cette construction :

```
>>> a = [ "Lundi", "Mardi", "Mercredi", "Jeudi", "Vendredi", "Samedi", "Dimanche" ]
>>> for i in a { print i , ' ' }
Lundi Mardi Mercredi Jeudi Vendredi Samedi Dimanche
>>> for i in "une chaîne" { print i , ' ' }
u n e   c h a î n e
>>> for i in range (1,10) { print i , ' ' }
1 2 3 4 5 6 7 8 9 10
```

### 3.2.3.5 Les exceptions

CorbaScript permet aux développeurs de contrôler les exceptions par l'intermédiaire d'une syntaxe similaire à celle du langage C++. Un bloc *try { ... }* peut entourer une suite d'instructions susceptibles de provoquer une exception. Comme les exceptions CorbaScript sont typées, le développeur peut associer un traitement d'exceptions différent pour chaque type d'exceptions attendu. La construction suivante montre la syntaxe générale du traitement des exceptions.

```
try {
    <suite d'instructions>
} [ catch (T1 v) {
    <instructions exécutées si une exception de type T1 est rencontrée>
} ]
[ catch (T2 v) {
    <instructions exécutées si une exception de type T2 est rencontrée>
} ]
[ catchany (v) {
    <instructions exécutées pour toutes les autres exceptions
        non traitées par les catch précédents>
} ]
```

Si une exception de type *T2* est provoquée dans le bloc *try { ... }* alors l'exécution est détournée vers le gérant du type d'exceptions *T2* (bloc *catch (T2 v) { ... }* où *T2* précise le type de l'exception attendue et *v* la variable où sera stockée cette exception). Pour choisir le gérant le plus adapté, l'interpréteur parcourt la liste des gérants. Lorsqu'aucun gérant n'est trouvé, l'interpréteur exécute le bloc optionnel *catchany* de traitement par défaut de toutes exceptions. Si ce bloc n'est pas présent, l'exception est reportée vers le bloc *try*

englobant. Finalement, s'il n'y a aucun gérant pour le type d'exception provoquée alors l'interpréteur arrête l'exécution en cours et il signale l'exception à l'utilisateur.

L'interpréteur dispose de quelques types d'exceptions pour signaler les problèmes internes tels qu'un problème de mauvais typage (*3 / "chaîne"*) et une opération non définie pour un objet (*10.hauteur*). Mais le traitement des exceptions est surtout intéressant lorsque l'on invoque une opération d'un objet Corba. De la même manière que toutes les définitions IDL sont accessibles, un script a la visibilité sur tous les types d'exceptions définis dans le référentiel des interfaces.

L'instruction *throw* permet de provoquer une exception. Elle prend en paramètre n'importe quel type de valeurs CorbaScript donc évidemment toute définition d'exceptions IDL.

```
// définition en IDL d'une exception
exception MonException {
    string champ_chaine;
    long champ_entier;
};

# utilisation en CorbaScript
try {
    # quelques instructions
    throw MonException ("valeur de la chaine", 10)
    # d'autres instructions
} catch ( MonException e ) {
    print "une exception de type MonException a été provoquée\n"
    print "    champ_chaine = ", e.champ_chaine, "\n"
    print "    champ_entier = ", e.champ_entier, "\n"
} catchany (e) {
    print "Une exception non prévue a été provoquée\n"
}
```

### 3.2.3.6 Les procédures

Les scripts peuvent être structurés par l'intermédiaire de procédures. La déclaration d'une procédure se fait par le mot clé *proc* suivi du nom de la procédure et de la liste des paramètres formels. Ces paramètres formels ne sont pas typés et peuvent avoir une valeur par défaut (valeur évaluée à la déclaration de la procédure). Le corps de la procédure est une suite d'instructions encadrées dans un bloc *{}*. L'instruction *return* permet de retourner un résultat à l'appelant de la procédure et d'interrompre l'exécution de la procédure. Ce résultat peut être une valeur CorbaScript quelconque.

```
proc nom_de_la_procedure (p1, p2, p3 = valeur_par_défaut)
{
    < suite d'instructions >
    return <valeur retour>
}
```

Les paramètres formels sont des variables locales à la procédure, elles peuvent aussi bien être lues que modifiées. De nouvelles variables locales peuvent être définies<sup>10</sup> n'importe où à l'intérieur de la procédure. Une variable locale n'existe que pendant la durée de l'exécution de la procédure.

Cependant, les variables globales sont accessibles en lecture. Lorsqu'une variable globale *A* est masquée par une variable locale, *A* sera toujours accessible en passant par l'espace global des variables (*global.A*).

10. En fait, CorbaScript ne contient pas de construction pour définir les variables, il suffit d'affecter une valeur à une variable pour la créer.

L'exemple suivant illustre une procédure de tri de tableaux (*descendant*, *ascendant*, *trier*). Comme les arguments d'un appel de procédure sont passés par référence (tout est objet), cette procédure modifie le *tableau* passé en paramètre.

```

proc descendant (a,b) { return a < b }
proc ascendant (a,b) { return a > b }

proc trier (tableau, critere_de_tri = ascendant) {
  for i in range (0, tableau.length - 2) {
    for j in range (i+1, tableau.length - 1) {
      if critere_de_tri (tableau[i], tableau[j]) {
        temp = tableau[i]
        tableau[i] = tableau[j]
        tableau[j] = temp
      }
    }
  }
}

>>> t = [4,6,124,6543,1]
>>> trier (t)
>>> t
[1 , 4 , 6 , 124 , 6543]
>>> trier (t, descendant)
>>> t
[6543 , 124 , 6 , 4 , 1]
>>> t = ['f', 'd', 'e', 'a']
>>> trier (t)
>>> t
['a', 'd', 'e', 'f']

```

Comme CorbaScript n'est pas syntaxiquement typé, la procédure précédente peut trier un tableau de n'importe quel type d'éléments comparables (des entiers, des chaînes, des booléens, ...). De plus, les procédures CorbaScript sont elles-mêmes des valeurs CorbaScript et peuvent donc être passées en paramètre (*descendant* et *ascendant*). Cela permet dans l'exemple précédent de préciser le critère de tri des éléments du tableau.

### 3.2.3.7 Les modules

Le mode interactif est très intéressant pour mettre au point des scripts ou pour exploiter dynamiquement des objets Corba. Mais tout ce que l'utilisateur a saisi est perdu lorsque l'interpréteur est arrêté. Pour réaliser des activités complexes et répétitives, l'utilisateur peut stocker ses scripts dans des fichiers. Ces fichiers sont alors considérés comme étant des modules CorbaScript. Ces modules sont constitués d'instructions, de variables, de procédures et aussi de classes (présentées dans la section suivante). Les modules peuvent être réutilisés en mode interactif, en mode traitement par lots ou bien dans d'autres modules. Le module *'math'* présenté ci-dessous contient deux fonctions mathématiques d'usage courant (i.e. *fac* pour calculer la factorielle d'un nombre entier et *fib* pour évaluer une suite de Fibonacci), l'affectation d'une variable (*une\_variable*) et une instruction d'affichage.

```

# module 'math' : quelques fonctions mathématiques

proc fac (n) {
  result = 1
  for i in range (1,n) {
    result = result * n
  }
  return result
}

proc fib(n) {
  result = []

```

```

a = 0
b = 1
while ( b < n ) {
  result.append (b)
  tmp = b
  b = a + b
  a = tmp
}
return result
}

une_variable = 10
print "Le module 'math' est maintenant chargé\n"

```

CorbaScript offre deux mécanismes pour charger un module : l'exécution et l'importation. La commande `exec("nom_d_un_fichier")` exécute le script contenu dans le fichier indiqué. A la fin de la commande `exec`, toutes les déclarations de procédures et de variables sont conservées et peuvent être utilisées par la suite. Dans l'exemple suivant, la procédure `fac` est automatiquement disponible après l'exécution du module. Ce mécanisme est comparable à l'inclusion d'un fichier en C++ : chaque fois que l'on exécute le fichier, il est rechargé en mémoire et analysé par l'interpréteur, ensuite, l'interpréteur exécute le contenu analysé. Ainsi, ce mécanisme ne permet pas de partager des données entre plusieurs modules car chacun aura sa propre copie des variables et de procédures du module à partager. L'exemple suivant illustre ce mécanisme d'exécution : le module est exécuté une première fois, il affiche un message et son contenu est automatiquement disponible, à la seconde exécution, le contenu de `une_variable` est alors réinitialisé.

```

>>> exec ("math.cs")                # illustration de l'exécution
Le module 'math' est maintenant chargé
>>> fac(10)
3628800
>>> une_variable = 20
>>> exec ("math.cs")
Le module 'math' est maintenant chargé
>>> une_variable
10
>>> import math                      # illustration de l'importation
Le module 'math' est maintenant chargé
>>> math.fib(100)
[1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]
>>> math.une_variable = 20
>>> import math
>>> math.une_variable
20

```

Par contre, l'importation de modules permet de partager des informations entre modules. Un module importé par la commande `import` est chargé en mémoire et exécuté une seule fois à la première importation. L'interpréteur gère un cache des modules chargés, ainsi les importations suivantes ne réexécutent pas le module permettant ainsi de partager le code mais aussi les données d'un module. La variable d'environnement `CSPATH` permet aux utilisateurs de préciser la liste des chemins d'accès où rechercher les modules à importer. Après importation, le contenu du module est accessible par la notation pointée précédée par le nom du module (`math.fib(100)`). Le dernier forme ainsi un espace de noms bien délimité évitant alors les possibles conflits de noms entre plusieurs modules. Comme nous le voyons dans l'exemple précédent, nous avons délibérément choisi de ne pas restreindre l'accès en écriture à ces espaces (`math.une_variable = 20`).

### 3.2.4 Des classes et des instances

L'architecture Corba permet de concevoir des programmes Corba qui sont clients d'objets distants mais aussi fournisseurs d'objets Corba. Tous les mécanismes précédents permettent d'écrire toutes sortes de scripts pour accéder à des objets mais uniquement en mode client. Aussi, nous avons introduit dans CorbaScript la notion de classe pour permettre à un script de devenir fournisseur d'objets.

#### 3.2.4.1 Quelques précisions sur les concepts objet de CorbaScript

Actuellement, il n'y a pas de consensus bien défini sur ce que doit être et sur ce que doit fournir un langage orienté objet. Nous avons choisi de munir CorbaScript des fonctionnalités orientées objet suivantes. Une classe définit l'ensemble de ses méthodes et de ses attributs de classe ainsi que les méthodes de ses instances. Elle peut hériter d'une ou plusieurs autres classes (héritage multiple et répété). Toutes les méthodes d'instance peuvent être redéfinies ou renommées dans les sous-classes. Chacune de ces méthodes prend un argument explicite référençant l'instance. Les instances sont créées à partir de classes. Les attributs d'une instance ne sont pas typés ni même déclarés. Ces attributs sont créés au fur et à mesure des besoins des méthodes d'instance. Et finalement, toutes les méthodes et attributs sont publics. L'exemple ci-dessous présente la construction syntaxique de toutes ces fonctionnalités :

```
class uneClasse [ ( liste_des_sur_classes ) ]
{
    # initialisation d'un attribut de classe
    un_attribut_de_classe = valeur

    # déclaration d'une méthode de classe
    proc une_methode_de_classe (arguments) {}

    # initialisation d'une instance à sa création
    proc __UneClasse__ (l_instance, arguments) {
        l_instance.attribut = valeur_d_initialisation
    }

    # déclaration d'une méthode d'instance
    proc methode_d_instance (l_instance,arguments) {}

    # renommage d'une méthode héritée d'une sur-classe
    l_ancienne_version_d_une_methode = une_methode_d_une_des_sur_classes

    # redéfinition d'une méthode d'une sur-classe
    proc une_methode_d_une_sur_classes (l_instance, arguments)
    {
        # nouveau traitement et invocation de la version précédente
        l_instance.l_ancienne_version_d_une_methode (arguments)
    }
}
```

Lorsque nous avons dû faire les choix pour introduire les classes en CorbaScript, nous ne voulions pas compliquer la syntaxe du langage en introduisant des constructions trop

spécifiques et nouvelles. Nous avons donc choisi de considérer une classe comme une portée de déclarations. Ainsi, la syntaxe d'appel d'une méthode ou d'accès à un attribut d'une classe est similaire à celle de l'appel à une procédure d'un module c'est à dire la notation pointée. De même, l'invocation d'une méthode sur une instance utilise cette notation. Les modules, les classes et les instances sont en fait des espaces de noms associant à chacun de ses symboles une valeur CorbaScript qui peut être une donnée (un entier, une instance) ou bien un traitement (une procédure pour un module, une méthode pour une classe). En fait, dans CorbaScript, une variable n'est qu'un symbole permettant aux utilisateurs de référencer les objets CorbaScript : des valeurs mais aussi des modules, des procédures, des classes et des instances. Cette vision «tout objet» a été fortement inspirée par la souplesse rencontrée dans les langages SmallTalk et Python.

### 3.2.4.2 Un exemple simple d'utilisation des classes

L'exemple suivant illustre la réalisation d'une classe de points 2D. Cette classe contient la procédure d'initialisation des instances de la classe (`__point2D__`), deux méthodes d'instances (`translater` et `deplacer`) et une méthode de classe pour obtenir le point origine du repère du plan (`origine`).

```
class point2D {
  proc __point2D__ (self, x, y) {
    self.x = x
    self.y = y
  }
  proc translater (self, dx, dy) {
    self.x = self.x + dx
    self.y = self.y + dy
  }
  proc deplacer (self, p) {
    self.x = p.x
    self.y = p.y
  }
  proc afficher (self) {
    print "c'est un point2D (" , self.x, ', ', self.y, ")\n"
  }
  proc origine () {
    return point2D (0,0)
  }
}

>>> p = point2D (10,20)
>>> p.x * p.y
200
>>> p.translater (10,10)
>>> p.afficher ()
c'est un point2D (20,30)
>>> p.deplacer ( point2D.origine() )
>>> p.afficher ()
c'est un point2D (0,0)
>>> p
< point2D instance
  x = 0
  y = 0
>
>>> proc comparaison_de_2_points (p1,p2) {
  return p1.x > p2.x && p1.y > p2.y
}
>>> t = [ point2D(20,20), p, point2D(10,10)]
>>> import TRI
>>> TRI.trier (t, comparaison_de_2_points)
>>> for i in t { i.afficher() }
c'est un point2D (0,0)
c'est un point2D (10,10)
c'est un point2D (20,20)
```

L'instanciation d'un objet se fait simplement en indiquant le nom de la classe suivi des paramètres de la procédure d'initialisation (*point2D(10,20)*). La procédure `__point2D__` crée et initialise alors les attributs de l'instance. La notation pointée permet de consulter les attributs (*p.x*) et d'invoquer les méthodes d'une instance (*p.afficher()*). L'invocation d'une méthode de classe est naturellement exprimée par la syntaxe suivante *classe.methode(arguments)* comme dans l'exemple *point2D.origine()*. Par défaut, en interactif, lorsque l'on évalue une instance, l'interpréteur affiche la liste des attributs ainsi que leur valeur. Dans l'exemple ci-dessus, nous pouvons remarquer que comme CorbaScript n'est pas typé syntaxiquement, il est relativement aisé de réutiliser notre procédure de tri précédemment décrite et stockée dans un module *TRI*.

### 3.2.4.3 L'héritage et la recherche des méthodes

Illustrons maintenant l'héritage. La classe suivante *point2DColore* est une spécialisation de la classe *point2D*. Elle définit une méthode d'initialisation `__point2DColore__` qui réutilise celle définie dans la sur-classe `__point2D__`. La méthode *afficher* est redéfinie pour afficher l'information supplémentaire. Cette méthode fait appel à l'ancienne version définie dans la classe *point2D*. CorbaScript permet d'accéder directement aux méthodes des sur-classes (*point2D.afficher(self)*). De même, de nouveaux alias peuvent référencer ces méthodes (*ancienne\_version\_afficher = point2D.afficher*).

```
class point2DColore (point2D) {
  proc __point2DColore__ (self, x, y, c) {
    self.__point2D__ (x,y)
    self.c = c
  }

  ancienne_version_afficher = point2D.afficher

  proc afficher (self) {
    point2D.afficher (self) # ou alors self.ancienne_version_afficher ()
    print "mais en plus il est coloré en ", self.c, "\n"
  }
}
```

La politique de recherche des méthodes en CorbaScript est relativement simple et est basée sur un parcours ordonné en profondeur d'abord et de gauche à droite du graphe des classes héritées. Lorsqu'une méthode *M* est appliquée sur une instance, le nom *M* est recherché dans l'espace de noms de la classe de l'instance. Si ce nom est inconnu dans cet espace alors les sur-classes sont parcourues dans l'ordre fixé à la déclaration de la classe. Par exemple pour une classe *D* héritant de trois classes *A*, *B* et *C* (i.e. *class D(A,B,C) {}*), la recherche de méthodes parcourt d'abord l'espace de déclarations formé par *D*, puis *A*, puis *B* et *C* jusqu'à trouver cette méthode. Bien sûr, si *A*, *B* ou *C* ont des sur-classes, elles sont récursivement parcourues.

Des conflits de noms peuvent apparaître lorsqu'une classe possède une méthode définie sur deux ou plusieurs branches de l'arbre d'héritage. La recherche de méthodes de CorbaScript va trouver la première version mais le développeur peut vouloir utiliser une des autres versions. L'accès direct à une méthode d'une sur-classe et la définition d'alias permettent de résoudre ces conflits respectivement de manière explicite à la C++ où c'est l'utilisateur d'une classe qui résout les conflits ou de manière implicite à la Eiffel où c'est le développeur de la classe qui les résout.

### 3.2.4.4 L'implantation d'objets Corba

Reprenons notre interface IDL *grille* pour présenter comment des objets Corba peuvent être simplement implantés.

```
// nom du fichier IDL 'grille.idl'

typedef unsigned short Positif;
exception mauvaiseCoordonnee {
    Positif x, y;
};
interface grille {
    attribute Positif hauteur, largeur;
    float consulter (in Positif x, in Positif y)
        raises (mauvaiseCoordonnee);
    void modifier (in Positif x, in Positif y, in float valeur)
        raises (mauvaiseCoordonnee);
};
```

Cette interface peut alors être implantée en CorbaScript par l'intermédiaire de la classe *MaClasseGrille* héritant de l'interface *grille*. Lorsqu'une classe CorbaScript hérite d'une interface, l'interpréteur crée automatiquement un squelette dynamique pour décoder les requêtes Corba à destination des instances de cette classe.

```
class MaClasseGrille (grille) {
    proc __MaClasseGrille__ (self, nom, hauteur=10,largeur=10) {
        self.__grille__ (nom)
        self.h = hauteur
        self.l = largeur
        self.reinitialiser_valeurs()
    }
    proc reinitialiser_valeurs (self) {
        self.valeurs = []
        for i in range (1,self.h) {
            tmp = []
            for j in range (1,self.l) { tmp.append(0) }
            self.valeurs.append (tmp)
        }
    }
    proc _get_hauteur (self) {
        return self.h
    }
    proc _set_hauteur (self,h) {
        self.h = h
        self.reinitialiser_valeurs ()
    }
    proc _get_largeur (self) {
        return self.l
    }
    proc _set_largeur (self,l) {
        self.l = l
        self.reinitialiser_valeurs ()
    }
    proc consulter (self,x,y) {
        try {
            return self.valeurs[x][y]
        } catchany (v) {
            throw mauvaiseCoordonnee(x,y)
        }
    }
    proc modifier (self,x,y,valeur) {
        try {
            self.valeurs[x][y] = valeur
        } catchany (v) {
            throw mauvaiseCoordonnee(x,y)
        }
    }
}
```

```

}

uneInstance = MaClasseGrille ("UneGrille",100,100)

```

Nous avons vu précédemment que chaque objet Orbix est désigné par un identificateur unique composé du nom de la machine, du nom du serveur et finalement d'un nom local de l'objet au sein de ce serveur. Cet identificateur est utilisé pour initialiser les références et permet ainsi à Orbix de router les invocations vers les objets destinataires. CorbaScript fournit les mécanismes pour fixer ces informations. Lorsque l'utilisateur lance l'interpréteur, il peut préciser le nom de serveur associé à cet interpréteur (i.e. *cssh -s nom\_de\_serveur*). Ensuite, lorsqu'il crée un objet Corba, il peut fixer son nom local en invoquant la méthode d'initialisation de l'interface héritée (*self.\_\_grille\_\_(nom)*). Si le *nom* est une chaîne vide, Orbix allouera automatiquement un nom unique à cet objet (par défaut, Orbix gère un compteur pour créer les noms uniques).

Par exemple, si l'utilisateur a lancé l'interpréteur avec le nom de serveur *ServeurDeGrilles*, la variable *uneInstance* référence un objet Corba d'interface IDL *grille* dont le nom Orbix local est *UneGrille*. Cet objet est alors accessible depuis n'importe quel autre interpréteur distant grâce à la référence suivante : *grille("UneGrille:ServeurDeGrilles:machine")* (*machine* devant être remplacée par la machine sur laquelle a été créé cet objet).

Mais lorsqu'un serveur gère de nombreux objets, il est très difficile d'associer un nom à chacun d'eux. Dans ce cas, il suffit d'explicitement nommer les objets racines et de laisser Orbix numéroter les autres objets.

Les attributs des instances *MaClasseGrille* sont *h* pour stocker la hauteur de la grille, *l* pour la largeur et *valeurs* pour la structure de données contenant les valeurs de la grille (i.e. un tableau de tableaux CorbaScript). La classe *MaClasseGrille* doit alors implanter les opérations et attributs de l'interface *grille* héritée.

Pour les opérations, il suffit de définir des méthodes d'instances avec le même nom que celui de l'opération et avec autant d'arguments que l'opération a de paramètres (il faut tout de même en ajouter un de plus pour le passage explicite de la référence de l'instance). Les deux méthodes *consulter* et *modifier* illustrent les mécanismes de traitement des exceptions. Si les coordonnées *x* et *y* sont en dehors de la grille alors l'accès au tableau provoque une exception de débordement de tableau. Cette exception est transformée en une exception IDL définie dans la signature des opérations IDL.

CorbaScript ne permet pas la surcharge des méthodes d'instances. Ainsi pour les attributs IDL, il est nécessaire de définir deux méthodes l'une pour consulter l'attribut (*\_get\_hauteur*) et l'autre pour le modifier (*\_set\_hauteur*). Ces méthodes retournent simplement un attribut de l'objet mais elles auraient pu faire un traitement plus complexe si nécessaire.

### 3.2.5 Quelques exemples

Reprenons maintenant les exemples du chapitre 2. Nous allons voir que nous pouvons simplement et rapidement développer des scripts pour manipuler ces objets préexistants. Nous illustrons cette souplesse sur le service de nommage et le service bancaire.

#### 3.2.5.1 Référencer le service de nommage

Avec CorbaScript, un utilisateur peut interactivement accéder et manipuler le service de nommage. Pour cela, il lui suffit de créer une référence du type IDL *CosNaming::NamingContext*. L'exemple ci-dessous illustre les deux mécanismes pour se connecter

à ce service.

```
>>> NS = CosNaming::NamingContext ("root:NS")
>>> NS = CORBA.ORB.resolve_initial_references ("NamingService")
```

Grâce à la première méthode, l'utilisateur indique la localisation exacte du contexte initial du service de nommage. Par défaut dans l'environnement Orbix, le service de nommage est géré par un serveur dont le nom symbolique est *NS* et l'objet contexte initial a le nom *root*. Ainsi, la référence CorbaScript *CosNaming::NamingContext("root:NS")* permet de se connecter au contexte initial de la machine locale.

La seconde méthode sera implantée dès que nous aurons la prochaine version d'Orbix (la 2.1). Nous introduirons un nouvel objet CorbaScript (i.e. accessible par *CORBA.ORB*) pour représenter l'ORB. Il disposera d'une opération pour invoquer le service d'initialisation de Corba qui permet d'obtenir les références des services de base (les objets «notoires» de la section 2.5.4.1).

### 3.2.5.2 Enregistrer un objet dans un contexte

Une fois cette connexion réalisée, l'utilisateur peut enregistrer les objets qu'il crée interactivement. Le script suivant crée une instance de la grille implantée par la classe *MaClasseGrille*. Puis il l'enregistre dans le contexte initial sous le nom *grille*.

```
>>> la_grille = MaClasseGrille ("",100,100)
>>> NS.bind ([["grille", ""], la_grille])
```

Le premier paramètre de l'opération *bind* de l'interface IDL des contextes est du type *CosNaming::Name*. Ce type est une séquence de structures de type *CosNaming::NameComponent*. L'utilisateur n'a pas besoin d'indiquer ces types car l'interpréteur de CorbaScript effectue automatiquement les conversions des tableaux vers les types IDL composés. Le premier crochet permet d'indiquer le début de la séquence tandis que le second indique le début de la structure. Si le tableau imbriqué ne contenait pas deux éléments, CorbaScript aurait provoqué une exception. Ainsi, l'invocation d'opérations à paramètres de type complexe est nettement simplifiée en CorbaScript par rapport, par exemple, au langage C++.

### 3.2.5.3 Obtenir un objet depuis un contexte

Nous avons enregistré précédemment l'objet banque dans le service de nommage (voir section 2.5.4.2). Pour retrouver cet objet, il suffit donc d'invoquer l'opération *bind* du contexte de nommage. Dans le script suivant, l'utilisateur précise le chemin pour obtenir cet objet. Le tableau passé en paramètre contient donc deux éléments : le nom du sous-contexte et le nom associé à la référence dans celui-ci.

```
>>> la_banque = NS.resolve ( [ ["bancaire",""], ["banque", "" ] ] )
```

Lorsque le service de nommage renvoie la réponse, CorbaScript crée automatiquement une référence d'objet pour le type dynamique de cette réponse. Si l'interpréteur ne connaît pas encore le type *bancaire::banque*, il charge sa définition depuis le référentiel des interfaces. Ainsi, l'utilisateur peut directement appliquer les opérations supportées par cette interface sans nécessiter une quelconque conversion de type comme cela est nécessaire en C++.

### 3.2.5.4 Appliquer des traitements sur un compte

Interactivement, l'utilisateur peut naviguer dans le graphe constitué par les objets. Cette navigation est réalisée en invoquant les opérations ou en consultant les attributs de ceux-ci. Le script ci-dessous effectue un débit sur le premier compte de M. Dupond de l'agence de LILLE.

```
>>> l_agence = la_banque.rechercher_l_agence ("LILLE")
>>> le_client = l_agence.rechercher_le_client ("Dupond")
>>> le_compte = le_client.liste_des_comptes()[0]
>>> le_compte.solde
15000
>>> le_compte.debit (1000)
>>> le_compte.solde
14000
```

### 3.2.5.5 Parcourir le graphe des objets du service bancaire

Jusqu'à présent, nous avons toujours appliqué des traitements rudimentaires sur les objets Corba. Pour des traitements plus ambitieux, il est nécessaire et utile de les stocker dans des modules. Le fragment de script (cf. figure 3.2) est contenu dans le module *BANCAIRE*. Cette procédure parcourt récursivement tous les comptes de tous les clients de toutes les agences d'une banque. Ce parcours itère sur la *liste\_des\_agences()*, la *liste\_des\_clients()* puis sur la *liste\_des\_comptes()*. Elle affiche l'état de chacun des comptes : nom de l'agence, nom du client, numéro du compte et solde courant. Afin d'optimiser légèrement le traitement et ainsi réduire le nombre d'invocations Corba, la consultation des informations pertinentes est faite une seule fois dans chaque boucle : les variables *ville* et *nom\_client*. Le lecteur pourra faire la comparaison avec l'écriture en C++ du même traitement (voir section 2.4.6).

```
proc afficher_tous_les_comptes_d_une_banque (banque) {
  print "Ville de l'Agence\tNom du Client\tNuméro du Compte\tSolde Courant\n"
  agences = banque.liste_des_agences ()
  for agence in agences {
    adresse = agence.adresse
    ville = adresse.ville
    clients = agence.liste_des_clients ()
    for client in clients {
      nom_client = client.nom
      comptes = client.liste_des_comptes ()
      for compte in comptes {
        print ville, "\t\t\t", nom_client, "\t\t\t",
              compte.numero, "\t\t\t", compte.solde, "\n"
      } } } }
}
```

FIG. 3.2 - La procédure CorbaScript d'affichage de tous les comptes d'une banque

```
>>> import BANCAIRE
>>> BANCAIRE.afficher_tous_les_comptes_d_une_banque (la_banque)
Ville de l'agence      Nom du Client      Numéro du Compte   Solde Courant
LILLE                  Dupont              C_1                 14000
LILLE                  Dupont              C_2                 -200
PARIS                  Durand              C_10                790
...
```

L'utilisateur doit donc charger le module pour disposer du traitement et il peut ensuite l'exécuter sur l'objet banque qu'il a obtenu auprès du service de nommage. Ainsi, de nombreux traitements peuvent être implantés sans que l'utilisateur soit un expert Corba. Il est tout de même nécessaire qu'il connaisse le langage CorbaScript et les interfaces

IDL des objets auxquels il veut accéder. Mais, l'écriture de scripts Corba nous paraît nettement plus conviviale que l'écriture de programmes Corba dans un langage compilé. Les utilisateurs peuvent rapidement répondre à leurs besoins spécifiques et s'échanger des scripts lorsque leurs activités ont des points en commun.

### 3.2.5.6 Fédérer des services de nommage

Comme dernière illustration de ce survol du langage CorbaScript, nous avons choisi de réaliser un utilitaire de fédération de services de nommage répartis. Dans cet exemple, nous prenons comme hypothèse que chaque machine dispose d'un service de nommage exécuté localement. L'objectif de l'utilitaire est de fédérer ces espaces de nommage indépendants en un unique espace. La fédération est réalisée de la manière suivante : un contexte de nommage (appelé contexte de fédération) dans chaque espace contient les références des espaces de nommage des autres machines. Lorsqu'une nouvelle machine est ajoutée à la fédération, nous créons le contexte de fédération local. Il est initialisé en recopiant le contenu du contexte de fédération d'une des machines de la fédération. Ensuite, une référence sur le contexte initial de la nouvelle machine est insérée dans le contexte de fédération de chacune des machines. Ainsi, nous établissons un maillage complet reliant tous les espaces de nommage.

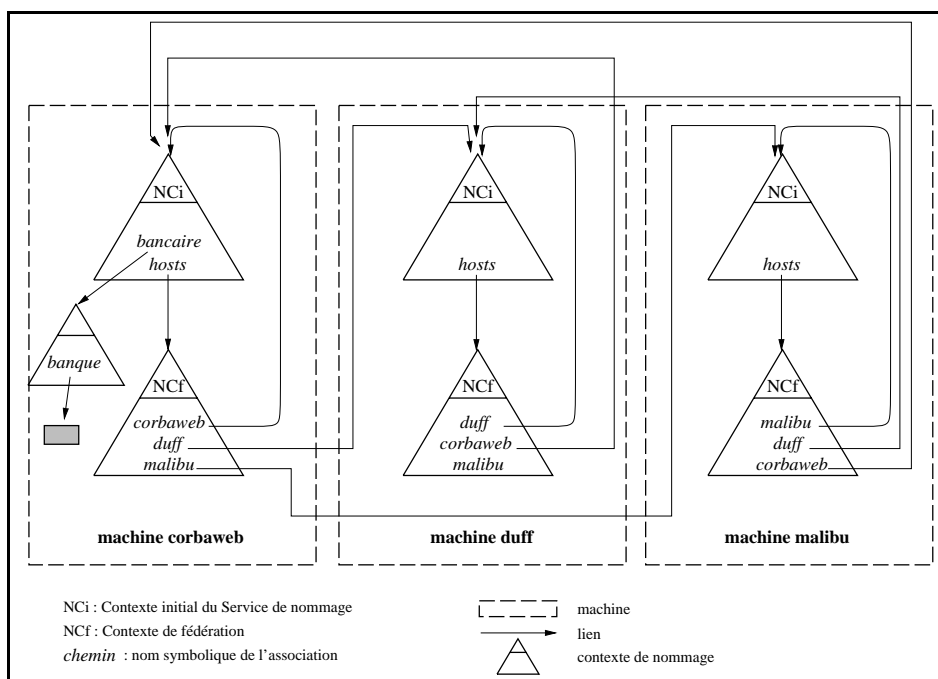


FIG. 3.3 - Un exemple de fédération de contextes de nommage

La figure 3.3 illustre la fédération de trois espaces de nommage localisés sur les machines *corbaweb*, *duff* et *malibu*. Chacun de ces espaces contient un contexte de fédération appelé *hosts*. Ce contexte contient la liste des références des services de nommage des autres machines et une référence sur le contexte initial local.

```
>>> import FEDERER
>>> federe = FEDERER.federer
>>> federe ("corbaweb")
>>> federe ("duff", "corbaweb")
>>> federe ("malibu", "duff")
```

Le script précédent illustre comment est créée la fédération de la figure 3.3. L'utilitaire de fédération est implémenté par la procédure *federer* du module *FEDERER*. Pour simplifier les commandes suivantes, l'utilisateur peut créer des alias (i.e. *federe*) car tout est objet en CorbaScript. Lorsque l'utilitaire est invoqué avec un seul paramètre, il se contente de créer le contexte de fédération sur la machine passée en paramètre. Ce procédé n'est utilisé que pour initialiser la première machine de la fédération (i.e. *federe ("corbaweb")*). Lorsque l'on veut ajouter une nouvelle machine dans la fédération, il suffit d'indiquer son nom et le nom d'une des machines de la fédération. La procédure crée alors automatiquement les liens entre l'espace de nommage de la nouvelle machine et ceux des machines déjà présentes dans la fédération.

```
# Module FEDERER pour fédérer un ensemble de service de nommage répartis

# nom du contexte permettant d'accéder aux autres services de nommage de la fédération
NOM_FEDERATION = "hosts"
# référence du contexte initial d'un service de nommage
REFERENCE_INITIALE = "root:NS:"

# obtenir le service de nommage localisé sur la 'machine'
proc obtenir_contexte_initial (machine) {
  return CosNaming::NamingContext (REFERENCE_INITIALE + machine)
}
# obtenir le contexte de fédération
proc obtenir_contexte_federation (NS) {
  return NS.resolve ( [ [NOM_FEDERATION, "" ] ] )
}

# installation d'une machine dans une fédération
proc federer (machine, depuis_machine = Void) {
  NS = obtenir_contexte_initial (machine)
  # créer le contexte de fédération pour cette machine
  CF = NS.bind_new_context ( [ [NOM_FEDERATION, "" ] ] )
  # relier le contexte de fédération au service de nommage de la machine
  CF.bind_context ( [ [machine, "" ] ] , NS)
  # si une machine distante est donnée en paramètre
  if (depuis_machine != Void) {
    # obtenir le contexte de fédération de la machine distant
    CFI = obtenir_contexte_federation (obtenir_contexte_initial (depuis_machine))
    # obtenir le contenu de ce contexte de fédération
    CFI.list (255, blist, biterateur)
    # itérer sur ce contenu
    for b in blist {
      # nom de la machine distante
      nom_distant = b.binding_name[0].id
      # obtenir le service de nommage de chaque machine
      ns_distant = CFI.resolve ( [ [ nom_distant, "" ] ] )
      # obtenir son contexte de fédération
      cf_distant = obtenir_contexte_federation (ns_distant)
      # relier le contexte de fédération distant avec le service de nommage de machine
      cf_distant.bind_context ( [ [machine, "" ] ] , NS)
      # relier le contexte de fédération avec le service de nommage de machine distante
      CF.bind_context ( [ [nom_distant, "" ] ] , ns_distant)
    }
  }
}
```

FIG. 3.4 - Le module de fédération de services de nommage (1/2)

Le code du module est contenu dans la figure 3.4. La procédure *federer* obtient la référence du service de nommage de la machine à ajouter dans la fédération (i.e. procédure *obtenir\_contexte\_initial*). Elle crée le contexte de fédération (i.e. opération *bind\_new\_context*). Ce contexte est relié au contexte initial par l'opération *bind\_context*. Ensuite, la procédure consulte le contenu du contexte de fédération de la machine servant à faire l'initialisation (i.e. *list*). Elle crée alors les liaisons entre le contexte de fédération en cours d'initialisation

et le contexte initial des machines de la fédération. Finalement, chaque contexte de fédération des autres machines est relié au contexte initial de la machine en cours de traitement. Ces liaisons établissent ainsi le maillage complet du graphe des espaces.

De plus, ce module contient la procédure *retirer* qui permet de retirer une machine de la fédération (cf. figure 3.5). Pour cela, elle supprime dans tous les contextes de fédération des autres machines la liaison sur la machine courante par l'opération *unbind*. Puis finalement, elle détruit le contexte de fédération de la machine courante par l'opération *destroy*.

```
# retirer une machine de la fédération
proc retirer (machine) {
  NS = obtenir_contexte_initial(machine)
  CF = obtenir_contexte_federation (NS)
  CF.list (255, blist, biterateur)
  for b in blist {
    nom = b.binding_name[0].id
    if ( nom != machine ) {
      cf_distant = obtenir_contexte_federation (CF.resolve ( [ [ nom, "" ] ] ))
      cf_distant.unbind ( [ [machine, "" ] ] )
    }
    CF.unbind ( [ [nom, "" ] ] )
  }
  CF.destroy ()
  NS.unbind ( [ ["hosts", "" ] ] )
}
```

FIG. 3.5 - *Le module de fédération de services de nommage (2/2)*

Ce dernier module nous a permis de voir que l'utilisateur (final, développeur ou administrateur) peut réaliser des tâches complexes sur des objets Corba répartis : consulter, modifier et assembler des composants par l'intermédiaire de la «**colle logicielle**» CorbaScript.

### 3.2.6 Résumé des fonctionnalités de CorbaScript

Ce survol du langage CorbaScript nous a permis de voir toutes les fonctionnalités offertes. Ce langage contient de nombreuses constructions pour pouvoir développer des scripts clients de tout type d'objets Corba mais aussi des scripts fournisseurs d'objets Corba. En résumé, CorbaScript est un «réel» langage de programmation de haut niveau, comparable à SmallTalk ou Python, offrant les fonctionnalités suivantes :

- l'interactivité et l'interprétation,
- la puissance d'expression,
- l'accès à tout objet Corba et à toute spécification IDL,
- la vérification dynamique du typage,
- la modularité et les procédures,
- les classes et les instances,
- le ramasse-miettes local, et
- l'aide en ligne.

Ces fonctionnalités ont été illustrées sur quelques exemples de scripts CorbaScript. Ces exemples montrent la simplicité et la souplesse du langage CorbaScript. Evidemment, ces scripts sont relativement simples et implantent des traitements modestes mais le lecteur trouvera des exemples plus complexes de l'utilisation de CorbaScript dans le chapitre suivant et en annexes.

Dans le chapitre suivant, nous présentons l'environnement CorbaWeb pour l'intégration générique des objets Corba dans le WWW. Cette intégration est réalisée à travers des scripts CorbaScript qui illustrent comment accéder par le WWW au service standard de nommage Corba, au service bancaire défini dans le chapitre 2 et à l'exemple de grille exposé dans ce chapitre. De plus, les annexes contiennent les sources d'une implantation du service de nommage écrite en CorbaScript ainsi qu'un module d'aide à l'administration de ce service. Mais avant cela, nous allons maintenant discuter de l'implantation de CorbaScript.

### 3.3 Implantation

Après la présentation de l'intérêt d'un langage de script pour les objets Corba et le survol des fonctionnalités de CorbaScript, nous allons maintenant décrire l'implantation de l'interpréteur de CorbaScript. Nous nous concentrerons uniquement sur quelques caractéristiques clés : l'architecture modulaire de CorbaScript, un survol des principales classes de l'interpréteur, quelques informations sur la liaison et les problèmes rencontrés avec Orbix et finalement, nous discuterons des performances actuelles de l'interpréteur.

#### 3.3.1 L'architecture

A la suite du développement du premier prototype expérimental de CorbaScript, nous nous sommes aperçus de la complexité du travail que nous voulions réaliser. CorbaScript comprend un langage complet de programmation (typage dynamique des données, procédures, modules et classes) et la liaison avec les mécanismes dynamiques de l'environnement Orbix (le référentiel des interfaces et les interfaces dynamiques DII et DSI). Lorsque nous avons commencé à réaliser la deuxième version de CorbaScript, nous avons fait le pari de modulariser la réalisation en espérant pouvoir la maintenir plus aisément et pouvoir envisager de la porter sur d'autres ORBs.

La figure 3.6 représente l'architecture actuelle de l'environnement CorbaScript. Celle-ci est décomposée en quatre parties distinctes :

- **Le noyau de l'interpréteur** compose le coeur de CorbaScript, il est constitué de l'analyseur lexical et syntaxique du langage générant la représentation interne des scripts sous forme de pseudo-code (équivalent d'un P-Code des anciens compilateurs Pascal ou du ByteCode produit par le compilateur Java). L'interpréteur de pseudo-code est basé sur une machine à pile. Il contient les valeurs CorbaScript de base (entier, chaîne, tableau, procédure, module, classe et instance), les espaces de variables et de symboles ainsi que le cache des modules chargés. Le ramasse-miettes local est pris en charge par ce noyau et est actuellement basé sur des compteurs de références.
- **L'environnement Orbix** nous fournit le négociateur de requêtes à objets Corba. Nous utilisons les souches C++ pour accéder aux métadonnées du référentiel des interfaces, l'interface d'invocation dynamique DII pour construire les requêtes à destination de tout objet Corba et l'interface de squelettes dynamiques DSI pour planter des objets Corba à l'aide de script CorbaScript.

- **Le module CorbaTools** permet d'adapter, de corriger et d'isoler les fonctionnalités d'Orbix du code de notre interpréteur. La norme Corba 2.0 n'est pas encore complètement ni correctement implantée dans Orbix : le référentiel est défini en IDL par les interfaces non conformes à Corba 2.0, le DII ne fonctionne pas correctement et le DSI n'est pas encore implanté. Cette couche permet donc d'isoler notre code des spécificités d'Orbix et devrait nous permettre de faciliter le portage de CorbaScript vers d'autres ORBs.
- **Le module CorbaScript** étend le noyau de base pour lui ajouter les fonctionnalités d'accès à Corba. Ces principales fonctions sont de consulter le référentiel des interfaces pour obtenir les métadonnées (types IDL, signatures des interfaces), d'assurer un cache de ces informations afin de limiter les accès au référentiel, de vérifier et de contrôler les arguments des invocations d'objets distants Corba mais aussi les requêtes reçues par les objets Corba implantés en CorbaScript. Grâce au module précédent, ce module n'est pas dépendant d'Orbix et donc doit pouvoir utiliser n'importe quel ORB.

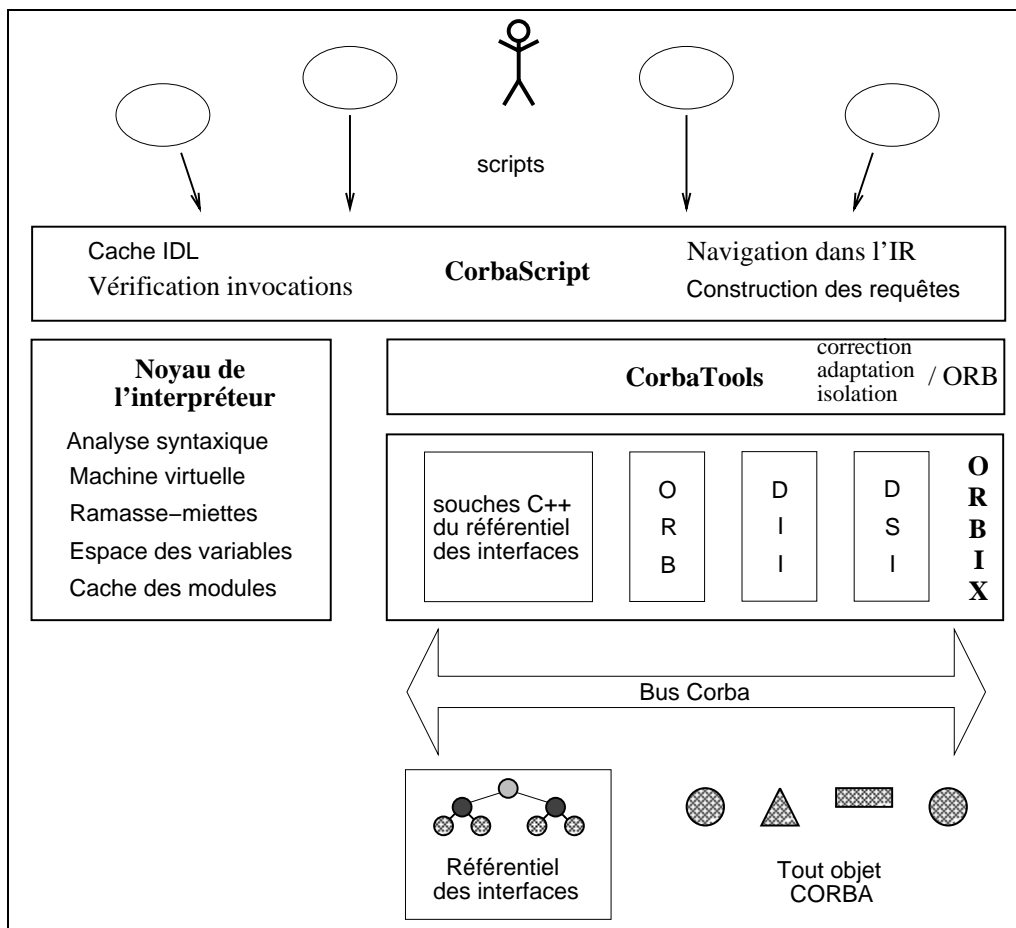


FIG. 3.6 - L'architecture de l'interpréteur CorbaScript

Grâce à cette architecture modulaire, les mécanismes de base du langage CorbaScript sont totalement isolés de l'ORB sous-jacent. Cette isolation nous a permis de faire évoluer le langage indépendamment des mécanismes spécifiques à Corba, par exemple en introduisant de nouvelles constructions syntaxiques ou de nouveaux types de valeurs

CorbaScript. D'un autre côté, cette architecture nous permet d'envisager l'utilisation d'un autre environnement ORB sans avoir à tout remettre en question.

Malheureusement, nous n'avons pas pu expérimenter ce choix d'architecture sur un autre ORB car actuellement, il n'existe pas beaucoup d'ORBs disponibles et encore moins d'ORBs fournissant un référentiel des interfaces, élément primordial pour le fonctionnement de notre langage de script. Les fournisseurs d'ORBs n'ont pas encore compris l'intérêt des métadonnées et des mécanismes dynamiques de Corba. Car premièrement, la majorité des applications distribuées actuelles sont construites selon le modèle client-serveur des souches-squelettes. Deuxièmement, les mécanismes dynamiques sont complexes à implanter dans un ORB. Finalement, peu de travaux de recherche (ou industriels) ont démontré les intérêts et les bénéfices que nous pouvons tirer de ces mécanismes dynamiques. Dans cette nouvelle phase exploratoire des objets répartis, CorbaScript et CorbaWeb (présentés dans le chapitre 4) apportent des éléments de réponse pour favoriser l'accès dynamique à tout objet Corba.

### 3.3.2 La hiérarchie des classes

L'interpréteur CorbaScript est écrit en C++. Ce langage s'est imposé de lui-même au début de nos travaux car Orbix supportait uniquement ce langage de programmation. De plus, le C++ offre des concepts orientés objet pour structurer et modulariser des applications complexes. Cependant, CorbaScript pourrait être écrit dans n'importe quel langage offrant l'héritage simple et une liaison avec Corba comme par exemple les langages SmallTalk, Java et Ada95 utilisables avec Orbix.

Actuellement, le code de l'interpréteur est structuré par les trois modules présentés précédemment (le noyau de l'interpréteur, le module CorbaScript et le module CorbaTools). Ces trois modules sont constitués d'environ 27.000 lignes de code C++ décomposés en une centaine de classes. La hiérarchie des classes principales est présentée dans la figure 3.7. Ce schéma présente uniquement les classes du noyau de l'interpréteur et l'extension pour accéder à Corba.

#### 3.3.2.1 La machine virtuelle de l'interpréteur

CorbaScript est un langage interprété mais il utilise une technique de compilation pour pouvoir exécuter les scripts. Avant d'exécuter les scripts, ceux-ci sont transformés en une représentation interne (notre pseudo-code) qui sera exécutée par une machine virtuelle à pile orientée objet.

Cette machine virtuelle utilise une pile de valeurs pour stocker les opérandes des expressions à évaluer et pour réaliser le passage de paramètres lors d'un appel de méthode ou de procédure. Cette machine est orientée objet car la pile ne contient que des références sur des objets CorbaScript (instances d'une sous-classe de *CS\_Object*). L'exécution effective des instructions de la machine est déléguée aux objets contenus sur la pile. Cette délégation est réalisée par un ensemble de méthodes virtuelles définies dans la classe *CS\_Object* et redéfinies dans ses sous-classes afin d'implanter l'exécution effective du traitement. Chacune de ces méthodes virtuelles correspond à une instruction du pseudo-code : les opérateurs, l'appel de procédure, l'invocation de méthode, la consultation d'un attribut.

Comme les traitements ne sont pas cablés dans la machine virtuelle mais dans les classes d'objets CorbaScript, nous pouvons aisément ajouter de nouveaux types de valeurs ou de nouveaux concepts dans l'interpréteur sans avoir à modifier la machine virtuelle ou la syntaxe.

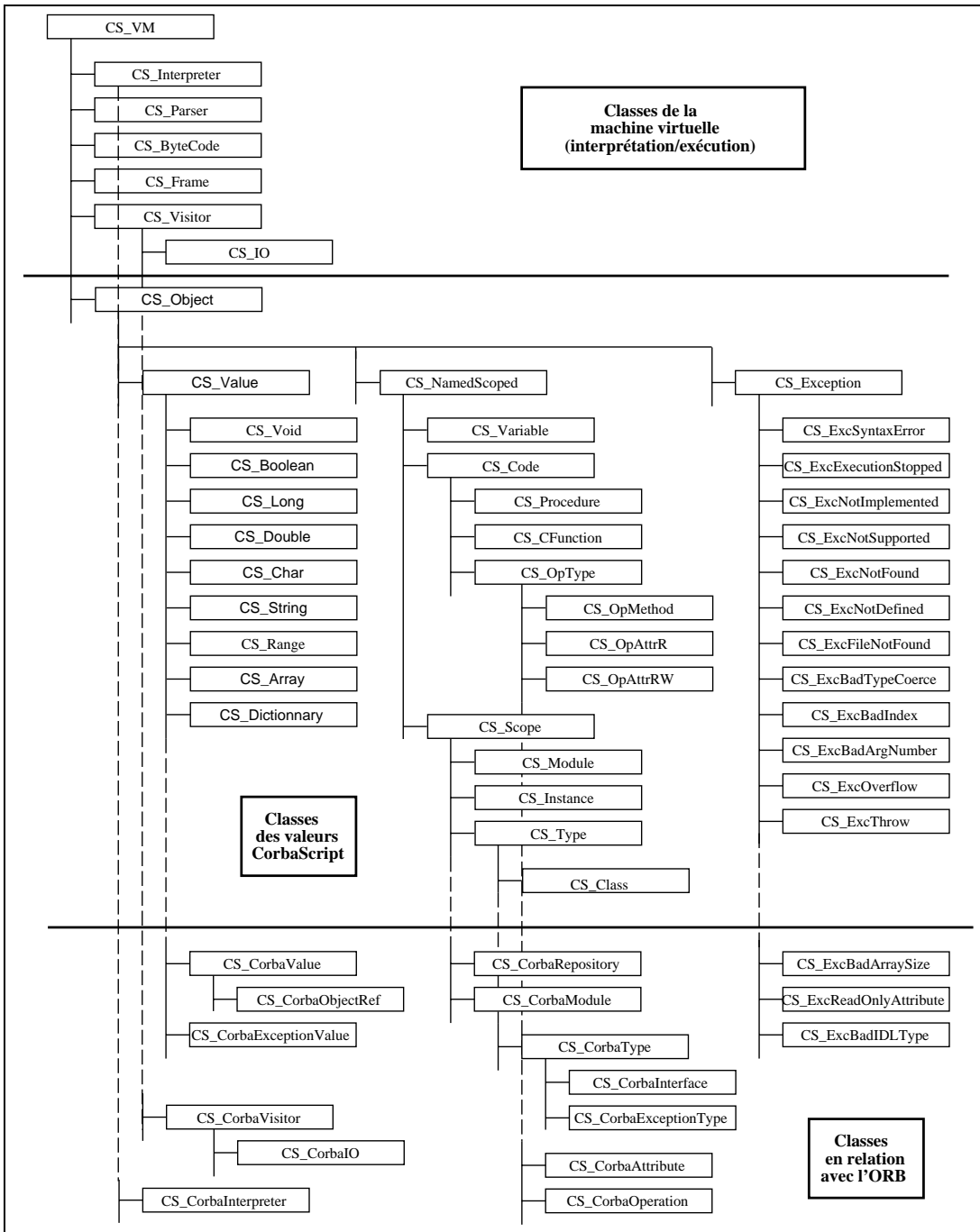


FIG. 3.7 - La hiérarchie des classes d'implantation de CorbaScript

Les classes suivantes implantent les mécanismes permettant d'analyser, de compiler et d'exécuter les scripts. La classe *CS\_VM* est la classe de base de notre hiérarchie. Elle intègre un compteur de référence pour gérer le ramasse-miettes. *CS\_Interpreter* encapsule la boucle principale de l'interpréteur. *CS\_Parser* contient l'analyseur lexical écrit avec l'outil Flex<sup>11</sup> [Pax95] et l'analyseur syntaxique écrit avec l'outil Yacc. Les règles de production Yacc pour notre langage génèrent le pseudo-code stocké dans une instance *CS\_ByteCode*. Ce pseudo-code est exécuté par la machine à pile encapsulée dans la classe *CS\_Frame*. Les valeurs stockées sur cette pile sont des instances d'une sous-classe de *CS\_Object*.

11. Flex est un générateur d'analyseurs lexicaux pour le langage C++.

### 3.3.2.2 Les valeurs de base CorbaScript

Chaque instance de *CS\_Object* contient un attribut qui référence le type de cette valeur. C'est cette information qui est retournée lorsque l'on utilise la propriété *\_type* en CorbaScript. Les types sont eux-mêmes des objets CorbaScript sur lesquels un script peut appliquer des traitements, par exemple, la comparaison de deux types (*10.\_type == long*).

Les classes de données CorbaScript sont toutes des sous-classes de la classe *CS\_Value* : *CS\_Void* représente une valeur vide pour les procédures ne retournant pas de résultat, *CS\_Boolean* encapsule une valeur booléenne, *CS\_Long* pour les valeurs entières, *CS\_Double* pour les valeurs réelles, *CS\_Char* pour les valeurs caractères, *CS\_String* pour les chaînes de caractères, *CS\_Range* pour les intervalles de valeurs et *CS\_Array* pour les tableaux polymorphes. Les dictionnaires ne sont pas pour l'instant syntaxiquement accessibles mais sont utilisés en interne pour stocker les associations «nom de variable - valeur CorbaScript».

Pour la liaison avec Corba, nous avons ajouté trois classes. *CS\_CorbaValue* stocke une valeur IDL quelconque dans un attribut de type *CORBA::Any*. *CS\_CorbaObjectRef* stocke une référence sur un objet Corba distant. Quant à *CS\_CorbaExceptionValue*, elle permet de mémoriser une exception Corba.

Toutes ces classes redéfinissent les méthodes virtuelles de la classe *CS\_Object* pour répondre aux instructions de la machine virtuelle par exemple *CS\_Boolean* redéfinit uniquement le comportement des opérateurs booléens, *CS\_Array* redéfinit l'opération *[]* et l'addition de tableaux tandis que *CS\_CorbaObjectRef* redéfinit l'invocation de méthode et l'accès aux attributs pour mettre en œuvre l'invocation dynamique de Corba.

L'approche modulaire, que nous avons choisie, est très intéressante car elle permet de bien isoler les traitements applicables à chaque type de données CorbaScript. Ainsi, la maintenance corrective du code est très rapide à effectuer lorsque l'on connaît parfaitement la hiérarchie des classes.

### 3.3.2.3 Les exceptions CorbaScript

Finalement, les exceptions sont elles-même représentées par des objets. Actuellement, nous avons défini 15 types d'exceptions pour refléter tous les problèmes que peut rencontrer l'interpréteur tels qu'une erreur de syntaxe (*CS\_ExcSyntaxError*), un symbole non trouvé dans un espace de noms (*CS\_ExcNotFound*), la lecture d'un fichier non disponible (*CS\_ExcFileNotFound* provoqué par l'importation ou l'exécution d'un module), une incompatibilité de type entre deux opérandes lors d'une addition (*CS\_BadTypeCoerce*), un mauvais nombre d'arguments pour l'appel d'une procédure (*CS\_BadArgNumber*), une exception provoquée par l'utilisateur (*CS\_Throw*) et bien d'autres encore.

Toutes ces exceptions peuvent être interceptées dans les scripts par l'intermédiaire des gérants d'exceptions. Les scripts peuvent ainsi prendre des décisions pour corriger le problème. Dans certains cas, cette correction est possible : par exemple si un script tente d'exécuter un fichier inconnu, il pourra toujours exécuter un autre fichier script. Dans d'autres cas, la correction est quasiment impossible lorsque l'interpréteur signale une erreur de syntaxe par exemple.

### 3.3.2.4 Les objets CorbaScript nommés

Dans CorbaScript, tout est objet aussi bien les données et exceptions présentées précédemment que les procédures, les modules, les classes et les instances.

Chacun de ces objets est instance d'une classe du noyau de CorbaScript : *CS\_Variable* encapsule une variable du langage, *CS\_Procedure* encapsule les informations décrivant une procédure (son nom, le nombre d'arguments, les valeurs par défaut et le pseudo-code à exécuter), *CS\_Scope* encapsule un espace de symboles associant à chacun d'eux un objet CorbaScript, *CS\_Module* est la représentation interne d'un module et il référence l'ensemble de ses variables, de ses procédures et de ses classes, *CS\_Classe* encapsule la définition d'une classe et maintient la liste des classes héritées et la liste des méthodes de classe et d'instances, *CS\_Instance* contient un dictionnaire associant à chaque nom d'attribut une valeur CorbaScript. Les diverses classes intermédiaires (*CS\_NamedScoped*, *CS\_Code* et *CS\_Scope*) permettent de factoriser des traitements et attributs communs.

La classe *CS\_Type* représente les types CorbaScript. Chaque objet CorbaScript référence une instance de cette classe ou d'une sous-classe. Ces instances conservent un dictionnaire des opérations et des propriétés accessibles pour un type de données. Comme ces traitements sont souvent de très bas niveau, ils sont réalisés en C++. Par exemple, l'opération d'ajout d'un élément dans un tableau CorbaScript (par exemple *t.append(10)*) est codé en C++ car elle manipule la structure interne de l'objet tableau. Les classes *CS\_OpMethod*, *CS\_OpAttrR* et *CS\_OpAttrRW* permettent d'encapsuler ces traitements C++ dans des objets CorbaScript. Cette isolation entre les traitements et la machine virtuelle d'exécution nous permet d'ajouter aisément de nouveaux traitements sans modifier la machine virtuelle.

Par cette approche, nous avons aussi pu isoler les spécificités de l'invocation des objets Corba et le stockage des métadonnées obtenues auprès du référentiel des interfaces dans un nombre réduit de classes CorbaScript. La classe *CS\_CorbaRepository* gère le cache des spécifications IDL chargées depuis le référentiel des interfaces. Les modules IDL sont représentés par des instances de *CS\_CorbaModule* et contiennent un dictionnaire (par la classe héritée *CS\_Scope*) du contenu du module IDL. Les interfaces IDL sont représentées par des instances de *CS\_CorbaInterface* qui contiennent un dictionnaire de tous les attributs (i.e. *CS\_CorbaAttribute* et de toutes les opérations (i.e. *CS\_CorbaOperation*). Les autres types IDL sont gérés par les deux classes *CS\_CorbaType* et *CS\_CorbaExceptionType*.

### 3.3.3 Le fonctionnement de l'interpréteur

Etant donnée la quantité de sources et le nombre de classes C++, il m'est impossible de décrire complètement le fonctionnement interne de l'interpréteur dans tous les cas de figure. Nous allons plutôt présenter, sur un exemple simple de script, les différentes opérations qu'effectue l'interpréteur CorbaScript. Cet exemple contient la déclaration d'une procédure *P*, l'affectation d'une variable *V*, la création d'une référence sur un objet Corba d'interface *grille*, une expression contenant l'accès au symbole global *true* et la consultation de l'attribut *hauteur*.

```
proc P () { print "ceci est une procédure\n"}
V = 10
ref = grille ("uneGrille:serveurGrille:corbaweb.lifl.fr")
print ref.hauteur < V == true
```

La figure 3.8 représente l'état de l'interpréteur après l'exécution du script précédent. Les numéros entre parenthèses réfèrent les diverses opérations réalisées par l'interpréteur au fil de l'exécution du script.

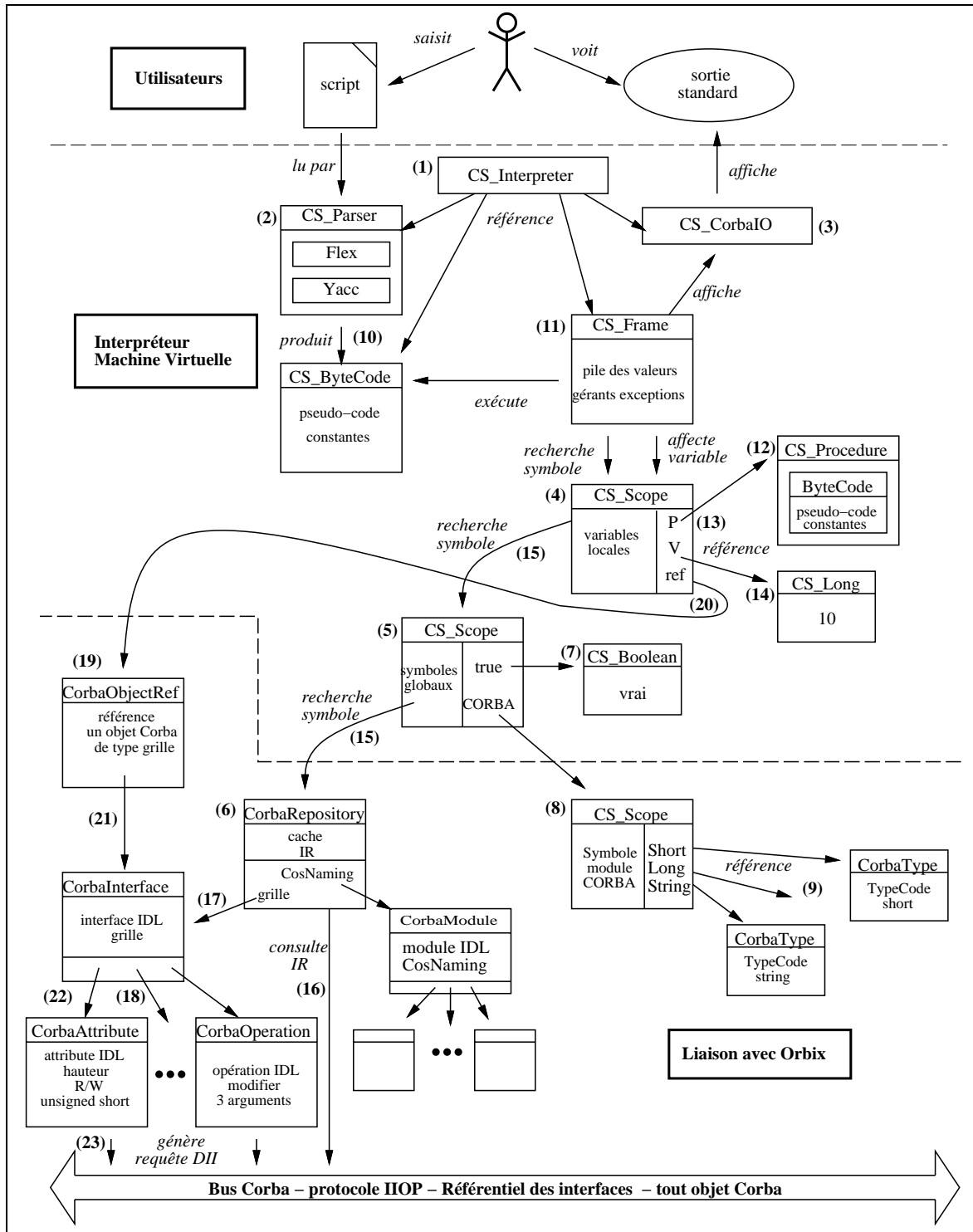


FIG. 3.8 - La machine virtuelle d'exécution de CorbaScript

Lorsque l'utilisateur lance CorbaScript, une instance de `CS_Interpreter` est créée (1). Cet objet crée l'analyseur de script `CS_Parser` (2), un objet `CS_CorbaIO` pour afficher les résultats de l'exécution du script (3), ainsi que l'espace des variables locales à l'exécution du script (4), l'espace des symboles globaux (5) et le cache du référentiel des interfaces (6). Tous ces espaces sont vides au démarrage et ils sont chaînés entre eux pour permettre la recherche des symboles. L'espace global (5) est initialisé avec tous les symboles prédé-

finis dans l'interpréteur : le symbole *true* référence un objet *CS\_Boolean* (7), le symbole *CORBA* référence un autre espace de nom (8) contenant tous les types IDL sous forme d'objets CorbaScript (9). D'autres symboles sont disponibles comme *false*, *long*, *exec*, ...

L'analyseur lexical et syntaxique est implanté avec les outils Flex et Yacc. Les règles de production Yacc génèrent le pseudo-code dans un objet *CS\_ByteCode* (10). Une fois l'analyse terminée, l'interpréteur crée la machine virtuelle à pile *CS\_Frame* et lance l'exécution du pseudo-code (11). La machine virtuelle exécute les instructions du pseudo-code, stocke les variables dans l'espace local (4) et recherche les symboles en parcourant les espaces de déclarations (4), (5) et (6).

La première instruction du pseudo-code demande la création de la procédure *P*. Cette procédure est représentée par un objet *CS\_Procedure* (12) encapsulant le pseudo-code de cette procédure. Ensuite, cet objet est inséré dans l'espace local (13). Le même scénario est réalisé pour exécuter la deuxième instruction du script : un objet *CS\_Long* est créé puis il est inséré dans l'espace local des variables (14).

L'exécution de la troisième instruction est plus complexe car l'interpréteur va d'abord chercher le symbole *grille* en parcourant successivement les espaces de variables locaux puis globaux puis le cache du référentiel (15). Comme cet objet cache ne connaît pas encore l'interface IDL *grille*, il consulte le référentiel pour découvrir ce type IDL (16). En retour si ce type existe, le cache crée une représentation *CS\_CorbaInterface* locale de l'interface IDL (17) et charge aussi tous les attributs et opérations de la *grille* (18). Si cette interface avait hérité d'autres interfaces IDL elles auraient toutes été chargées récursivement depuis le référentiel. L'objet interface (17) est alors inséré dans le cache pour optimiser les prochaines recherches. Si le nom recherché est un module IDL alors le cache crée un objet *CS\_Module* et charge récursivement toutes les déclarations contenues dans le module IDL comme par exemple pour le module *CosNaming*.

Ensuite, l'interpréteur exécute l'instruction d'appel procédural sur l'objet interface (18) en lui fournissant en paramètre la chaîne référençant l'objet Corba *grille* désiré. L'interface crée alors un *CS\_CorbaObjectRef* encapsulant la référence sur l'objet Corba (19). Cet objet est alors inséré dans l'espace des variables locales sous le nom *ref* (20).

La dernière instruction du script est l'affichage d'une expression. Lors de l'évaluation de ces expressions, les symboles sont recherchés dans les espaces de variables. Ainsi, la recherche du symbole *ref* retourne l'objet (19). Cet objet reçoit une demande de lecture de l'attribut *hauteur* de la part de la machine virtuelle. Il délègue ensuite l'exécution de cette lecture à son interface (21). L'interface recherche dans son dictionnaire le symbole *hauteur* et délègue l'invocation à l'objet associé (22). Ainsi, l'objet attribut *hauteur* va construire une requête DII pour aller consulter la valeur de l'attribut *hauteur* de l'objet Corba distant. Lorsque le résultat revient, il est encapsulé dans un objet *CS\_CorbaValue*. Cet objet se retrouve au sommet de la pile de la machine virtuelle qui peut alors continuer l'évaluation de l'expression et afficher le résultat grâce à l'objet d'affichage (3).

La modularité de l'implantation de CorbaScript rend très complexe l'explication de son fonctionnement. Chaque objet joue un rôle dans l'exécution d'un script et ses objets forment un graphe complexe.

### 3.3.4 La liaison avec Corba

Cette partie va décrire quelques éléments de l'implantation en rapport avec Corba et présenter les difficultés que nous avons rencontrées durant notre implantation.

### 3.3.4.1 Le cache du référentiel des interfaces

CorbaScript comprend quelques classes pour représenter localement les métadonnées stockées dans le référentiel des interfaces. Nous avons opté pour la mise en place d'un cache de ces métadonnées dans l'espace mémoire de l'interpréteur pour optimiser les traitements. Ce choix provient des expériences menées lors du premier prototype de CorbaScript.

Le référentiel des interfaces (ou IR) d'Orbix est un serveur d'objets que l'on peut installer sur n'importe quel site où s'exécute l'ORB. Pour accéder aux objets contenus dans ce référentiel, Orbix fournit une bibliothèque contenant les souches C++ des interfaces IDL décrivant ce référentiel. Ainsi la consultation d'un objet du IR est réalisée par un appel de méthode distante transporté par l'ORB. Dans notre premier prototype, nous n'avions pas mis de cache, ainsi chaque requête sur un objet Corba nécessitait une requête vers le IR pour pouvoir contrôler et inférer les paramètres de la requête. Le cache nous permet donc de diminuer par deux le trafic sur le bus réparti par rapport à notre premier prototype de CorbaScript.

Mais l'inconvénient est que si l'on modifie dynamiquement à l'exécution certaines métadonnées contenues dans le IR, nous sommes pour l'instant incapables de détecter ces changements et de mettre à jour le cache. La seule solution actuelle consiste à arrêter les interpréteurs et à les relancer. Ce coût n'est pas trop important et nous pouvons le supprimer en rajoutant dans l'interpréteur des primitives pour maintenir le cache, comme détruire explicitement les objets CorbaScript contenus dans le cache ou alors leur imposer de recharger les métadonnées contenues dans le IR.

Un problème plus crucial provient du fait que le IR d'Orbix n'est pas actuellement à la norme Corba 2.0. Cela nous obligera à modifier le code du module CorbaScript et du module CorbaTools lorsque Iona fournira un IR à la norme ou lorsque nous voudrons porter CorbaScript sur un autre ORB.

De plus, le IR actuel n'est pas correctement implanté et certaines de ces méthodes retournent de mauvaises métadonnées qui impliquent alors un mauvais fonctionnement de notre interpréteur. Par exemple, si une définition IDL est contenue dans un module IDL alors la métadonnée la décrivant ne contient pas le nom de ce module ou lorsqu'un attribut spécifique comme résultat une référence d'objet, nous obtenons de l'IR une métadonnée spécifiant que l'attribut retourne une structure. Ces erreurs ont été transmises à Iona et seront peut-être un jour corrigées. Dans le module CorbaTools, nous avons corrigé certaines de ces erreurs.

### 3.3.4.2 L'interface d'invocation dynamique ou DII

L'invocation des objets Corba depuis les scripts CorbaScript est implantée en utilisant l'interface d'invocation dynamique (DII). Nous avons ajouté des mécanismes de contrôle des invocations exprimées dans les scripts. Ce contrôle est réalisé grâce aux données stockées dans le cache de l'IR.

```
>>> grille ("UneGrille:UnServeur:UneMachine").modifier (10, 10, 100)
```

Le pseudo-code ci-dessous explique le scénario de réalisation de l'invocation de l'opération *modifier* sur un objet de type *grille*.

```
chargement dans le cache du IR de la description de l'interface IDL 'grille'
ainsi que de toutes ses opérations et de tous ses attributs.
```

```
rechercher si l'opération 'modifier' existe dans le cache de l'interface 'grille'
si non correct alors provoquer une exception CS_ExcNotFound
```

```

vérifier le bon nombre d'arguments sur la pile d'exécution (CS_Frame)
si non alors provoquer une exception CS_ExcBadArgNumber

pour chaque argument déposé sur la pile {
  vérifier le mode de passage
  si mode = 'in' alors continuer
  si mode = 'out' ou = 'inout' alors l'argument sur la pile doit être une variable
  si problème alors provoquer une exception

  vérifier le type de l'argument par rapport à la signature IDL de l'opération
  mais aussi conversion de l'argument vers une valeur IDL si nécessaire et si possible
  si problème de typage ou de conversion alors provoquer une exception CS_ExcBadIDLType
}

créer une requête CORBA::Request
ajouter chaque argument à la requête

invoquer la requête et attendre la fin de l'exécution de l'opération
si exception Corba alors créer et provoquer une exception CorbaScript

pour chaque paramètre en mode 'out' ou 'inout' {
  extraire le paramètre retourné de la requête

  si le type de la valeur extraite n'est pas connu de l'interpréteur
  alors utiliser le cache du IR pour découvrir ce type IDL

  transformer cette valeur en un objet CS_CorbaValue
  stocker cet objet dans la variable passée en argument sur la pile
}

transformer la valeur de retour de l'opération en un objet CS_CorbaValue
puis stocker cet objet sur le sommet de la pile

```

En fait ce pseudo-code est dispersé dans les différentes classes assurant la liaison avec Corba : *CS\_CorbaRepository*, *CS\_CorbaOperation*, *CS\_CorbaValue* et *CS\_CorbaType*.

La principale difficulté technique que nous avons rencontrée provient du fait que l'implantation du DII dans Orbix n'est pas complète. Lorsqu'Orbix doit emballer ou déballer une structure IDL dans ou depuis une requête, il utilise des fonctions générées dans les souches IDL par le compilateur IDL. Cela fonctionne parfaitement quand le programme Corba utilisant le DII est lié avec ces souches. Mais dans notre contexte, c'était inacceptable car nous voulions réaliser un interpréteur générique capable d'accéder à n'importe quel objet Corba sans pour cela devoir intégrer les souches IDL.

Devant ce problème technique, nous avons dû réimplanter totalement le mécanisme d'emballage et de déballage des requêtes DII. Ce travail a été réalisé lors de deux mémoires de DEA [Cha95, Hor96] que nous avons encadrés. Il est le résultat d'une longue et fastidieuse analyse des souches générées par le compilateur IDL d'Orbix pour reproduire dynamiquement le comportement des fonctions d'emballage et de déballage générées statiquement.

### 3.3.4.3 L'interface de squelettes dynamiques ou DSI

Pour implanter des objets Corba à l'aide de scripts CorbaScript, nous utilisons l'interface de squelettes dynamiques (DSI). Mais comme elle n'était pas encore disponible dans Orbix 2.0, nous avons dû l'implanter totalement en nous basant fortement sur le travail que nous avons fait pour le DII. Le DSI nécessite de pouvoir décoder une requête Corba à destination du serveur, d'analyser le contenu de cette requête en vérifiant sa conformité à la signature de l'opération IDL correspondante.

Pour le décodage des arguments et le codage des résultats, nous avons réutilisé le code

que nous avons implanté pour le DII. Nous avons dans le même temps longuement analysé les squelettes IDL pour reproduire dynamiquement leur fonctionnement. Une fois cette interface implantée, nous l'avons connectée aux mécanismes de classes du langage CorbaScript afin de rendre transparente cette interface auprès des utilisateurs de CorbaScript. Le pseudo-code suivant illustre comment une requête Corba est transformée en un appel de méthode sur une instance CorbaScript.

```

réception d'une requête Corba à travers l'interface DSI

vérifier si elle est conforme à la définition contenue dans le cache IR
    (nombre d'arguments, type et mode de passage)

rechercher si l'instance CorbaScript implante cette opération
si non alors retourner une exception Corba pour signaler le problème

créer un contexte d'exécution :
    une pile d'exécution de la machine virtuelle (CS_Frame)
    un espace de variables locales pour l'exécution de la méthode

déballer les paramètres de la requête
et les empiler sur la pile d'exécution

exécuter le pseudo-code de la méthode de l'instance écrit en CorbaScript

emballer les résultats (retour et paramètres 'out' et 'inout') dans la requête
retourner la requête au client Corba.

```

Dans les mois à venir, Iona doit livrer une version d'Orbix contenant le DSI que nous pourrions alors utiliser si elle n'a pas les mêmes contraintes que son DII.

### 3.3.5 Synthèse et performance

L'approche modulaire que nous avons choisie pour l'implantation du langage CorbaScript nous a permis de bien isoler les différentes parties de l'interpréteur : le noyau principal de l'interpréteur, le module CorbaScript de liaison avec Corba et finalement le module CorbaTools qui isole notre code des spécificités d'Orbix et corrige certaines de ces imperfections (IR, DII et DSI mal ou non implantés). Cette structuration permet de maintenir et d'étendre aisément notre langage.

L'objectif du langage CorbaScript est de favoriser l'accès à tout objet Corba en insistant sur la souplesse et la facilité d'utilisation. Sa syntaxe et son emploi ont été étudiés pour simplifier le développement de scripts par des utilisateurs «experts» ou non. Le moteur de l'interpréteur cache complètement la complexité intrinsèque à Corba. Mais cette simplicité d'emploi se traduit par une implantation complexe dont nous n'avons illustré que quelques éléments significatifs.

La complexité et la modularité de CorbaScript impliquent des performances faibles. Nous n'avons pas encore eu le temps d'étudier les optimisations que nous pourrions faire. A titre d'exemple, nous avons comparé l'exécution d'un programme écrit en C++ et le même écrit en CorbaScript : la version CorbaScript va huit fois moins vite que la version C++. Cette performance de l'interpréteur n'est pas excellente mais il faut prendre en considération les éléments suivants : 1) le script est interprété tandis que le programme est compilé, le typage est vérifié à l'exécution tandis qu'il est vérifié à la compilation en C++; 2) l'interpréteur accède au référentiel des interfaces pour obtenir les types IDL tandis que le programme C++ n'a pas à payer ce coût; 3) il utilise l'invocation dynamique plutôt que des souches pré-générées. Mais si l'on compare ses médiocres performances par rapport à d'autres langages interprétés comme SmallTalk ou bien Python alors le coût à payer

avec CorbaScript n'est pas trop exorbitant par rapport à la souplesse et la simplicité d'utilisation.

Cependant, il sera nécessaire dans l'avenir d'optimiser notre interpréteur mais pour cela, il faudra étudier les coûts de chacune des parties de notre implantation afin de ne pas remettre trop en cause notre approche modulaire.

### 3.4 Conclusion

Grâce à la transparence offerte par CorbaScript, les scripts peuvent invoquer les opérations, consulter ou modifier les attributs de n'importe quel objet Corba. De plus, le chargement dynamique et le cache des modules permettent de structurer ces scripts en entités facilement réutilisables. Ces entités permettent d'écrire rapidement des groupes de procédures pour exploiter un service et de les réutiliser pour construire une multitude de clients pour ce service répondant aux besoins spécifiques de chaque développeur mais aussi de chaque utilisateur.

CorbaScript offre suffisamment de constructions syntaxiques et d'entités sémantiques, telles que les expressions, les nombreux types de données de base et tous les types exprimés en IDL, les modules, les procédures, les classes et les instances, pour développer rapidement des programmes clients et des serveurs d'objet Corba. CorbaScript remplit ainsi les objectifs que nous avons définis dans la section 3.1.3 :

- **La conception et le prototypage** : de part sa nature interprétée et son typage dynamique, CorbaScript permet rapidement de prototyper des services orientés objet, de fixer des choix conceptuels au niveau des interfaces IDL et d'évaluer des choix de répartition des objets. Le prototypage peut très bien être réalisé dans un langage compilé comme C++ mais au prix d'importants efforts de programmation et d'un cycle de développement très lourd. Au contraire, CorbaScript offre un accès naturel et interactif aux objets.
- **Le développement et les tests** : même si l'on développe ses services avec un langage compilé, CorbaScript peut alors servir pour écrire des scripts de tests de validité de ces objets. Ces scripts peuvent être conservés pour pouvoir être rejoués plus tard. De plus, ils peuvent être générés automatiquement à partir du référentiel des interfaces et d'informations sur la sémantique des interfaces automatisant ainsi les tests.
- **La configuration et l'administration** : Assembler, connecter, configurer, créer, détruire des objets Corba devient un jeu d'enfant avec le langage CorbaScript. Son mode interactif permet d'invoquer directement les objets et de réaliser des opérations complexes sur les graphes d'objets.
- **L'utilisation des composants** : l'utilisateur expérimenté peut lui-même concevoir des scripts pour répondre à ses besoins spécifiques. Il extrait ainsi, des composants disponibles sur le bus, les informations qui lui sont pertinentes sans devoir passer par son service informatique.
- **L'assemblage de composants logiciels** : CorbaScript peut être utilisé comme une «colle logicielle» pour assembler des composants. Ces agrégats peuvent eux-mêmes constituer de nouveaux objets Corba. Ces nouveaux objets sont alors utilisés par toute application compilée mais aussi par d'autres scripts.

- **L'évolution** : si les composants évoluent ou si de nouveaux apparaissent, l'emploi de scripts permet de les découvrir dynamiquement à l'exécution et donc de pouvoir les utiliser dès qu'ils sont disponibles. Les modifications mineures ne nécessitent pas forcément de réécrire les scripts.

Au cours de ce chapitre, nous avons montré que le langage CorbaScript fournit les fonctionnalités pour accéder simplement et dynamiquement à tout service orienté objet Corba. De plus, le moteur du langage peut être étendu et adapté à d'autres contextes d'applications génériques. Dans le chapitre suivant, nous illustrons cette souplesse et cette flexibilité à travers l'environnement CorbaWeb : la mise en œuvre du langage CorbaScript pour réaliser une passerelle générique d'intégration des objets Corba dans le World-Wide Web.



## Chapitre 4

# CorbaWeb : l'intégration des objets Corba dans le WWW

Dans ce chapitre, nous allons aborder l'intégration des objets Corba dans le World Wide Web. Après un rappel des éléments caractérisant le WWW, nous discutons de l'intérêt d'intégrer des objets dans le WWW et présentons les diverses approches possibles. Suite à cet état de l'art, nous présentons notre environnement CorbaWeb. Cet environnement offre un ensemble de fonctionnalités extensibles pour favoriser l'intégration transparente d'objets Corba dans le WWW. Il permet ainsi la navigation dans des graphes d'objets Corba complexes, l'invocation d'opérations sur ces objets et la découverte des interfaces IDL des objets. Cet environnement est réalisé avec le langage CorbaScript présenté dans le chapitre précédent montrant ainsi ses possibilités pour favoriser l'exploitation de tout objet Corba. Cet environnement permet ainsi la création de nouveaux services pour le WWW et l'accès à des services orientés objet Corba déjà créés et non conçus pour le WWW.

### 4.1 Le World Wide Web

#### 4.1.1 Les principes

Le World Wide Web est un système d'informations hypermédias distribué sur Internet. Il a vu le jour en 1989 à l'initiative de Tim Berners-Lee [BLC90, BLCL<sup>+</sup>94] pour supporter la communication d'informations au sein de la communauté de scientifiques du CERN. La définition officielle du Web est la suivante : « *wide-area hypermedia information retrieval initiative aiming to give universal access to a large universe of documents* ». Ce système permet de connecter de larges bases d'informations multimédias hétérogènes et réparties grâce à la métaphore de documents hypertextes. La navigation des utilisateurs est l'élément clé du WWW offrant ainsi un mode non linéaire de consultation des documents. Aujourd'hui, le mot « document » englobe aussi toute autre ressource informatique : des bases de données distantes, des traitements distants ainsi que des codes mobiles.

Le Web repose sur un modèle client-serveur très simple dans lequel des clients (appelés navigateurs) permettent aux utilisateurs de naviguer dans un graphe de documents multimédias – tels que des textes, des images, du son, de la vidéo – stockés dans des serveurs Web (cf. figure 4.1) mais aussi dans d'autres types de serveurs Internet (ftp, mail, news, gopher, ...).

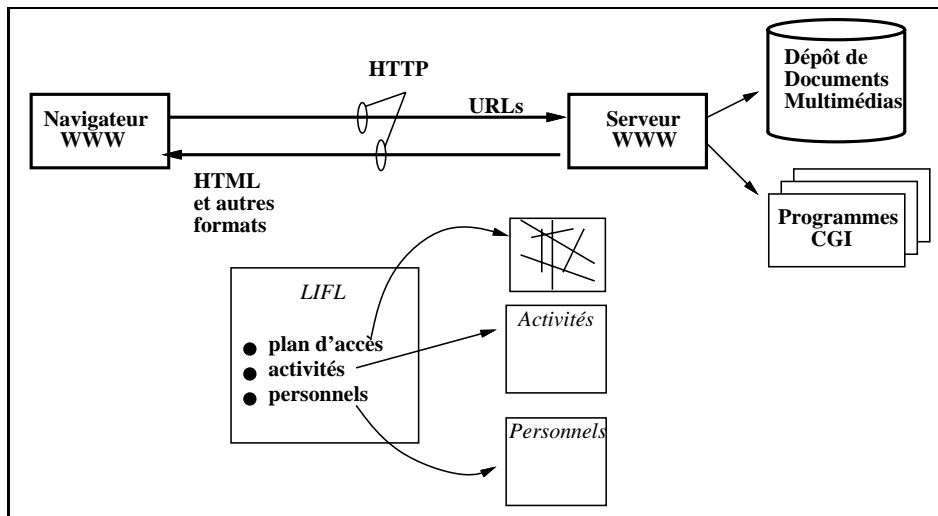


FIG. 4.1 - Les principes de base du WWW

Les clients et les serveurs Web communiquent par le protocole HTTP<sup>1</sup> [BLFF96]). De nombreux clients peuvent se connecter à un serveur car la version courante 1.0 de ce protocole est sans état évitant ainsi de consommer des ressources dans le serveur mais au prix d'une certaine inefficacité des communications.

Un navigateur se connecte à un serveur en utilisant les fonctions de communication de TCP/IP. Il lui envoie ensuite une requête HTTP : un ordre codé en ASCII contenant l'opération désirée, l'identificateur de la ressource ainsi que des paramètres. Le serveur évalue la requête et retourne le document demandé. Ensuite, le navigateur et le serveur coupent la connexion (une extension du protocole permet de garder cette connexion ouverte).

Les documents statiques sont stockés dans des dépôts de documents par exemple un système de fichiers ou une base de données. Les documents peuvent aussi être générés dynamiquement à la demande par des programmes externes appelés scripts CGI<sup>2</sup> [McC95]. Le protocole CGI est un standard pour interfacier des programmes externes avec les serveurs HTTP. Ce protocole définit le format des données échangées entre le serveur et les programmes externes via des variables d'environnement. Ces programmes peuvent être écrits dans divers langages de programmation (comme les langages de commande Unix, Perl, C). Un navigateur Web communique des informations à ces programmes à partir de formulaires (menus, zones de saisie) contenus dans les documents. Notre environnement CorbaWeb utilise ce mécanisme pour accéder aux objets Corba depuis un navigateur (*c.f.* section 4.3.5.2).

#### 4.1.2 Les ressources et leur désignation

Les documents retournés par le protocole HTTP sont typés grâce au format MIME [BF93, Moo93]. Ainsi, lorsque le navigateur reçoit un document, il détermine le format et applique la méthode de présentation adaptée à ce format : afficher une image GIF, jouer un son numérique ou bien représenter un document hypertexte.

Les documents hypertextes sont décrits avec le langage HTML<sup>3</sup> [BLC95]. Ce langage permet de décrire la structure interne et hypertexte des documents indépendamment de la manière dont le navigateur présentera ces documents aux utilisateurs. La syntaxe d'HTML

1. HTTP : HyperText Transfer Protocol

2. CGI : Common Gateway Interface

3. HTML : HyperText Markup Language

est une application (ou une DTD<sup>4</sup>) du langage standardisé de description de structure de document SGML<sup>5</sup> [ISO86]. SGML est un méta-langage de documents : un langage pour décrire des langages de description de documents.

Un document HTML est un texte ASCII agrémenté d'un ensemble de tags. Ces tags permettent de spécifier la structure interne du document : le titre, les sections, les sous-sections, des listes, des tableaux; la mise en forme du texte telles que la typographie (italiques, gras), la couleur; la structure hypermédia du document comprenant les images incluses, les ancres de navigation, les cadres (ou *frames*). Les ancres HTML servent à exprimer les relations hypertextes entre les documents. Une ancre peut être placée n'importe où dans un document et peut référencer n'importe quel autre document du Web. Le navigateur représente les ancres graphiquement par une zone sensitive que l'utilisateur sélectionne pour naviguer d'un document à un autre.

Dans le contexte à grande échelle du WWW se pose le problème de l'identification des ressources qui sont stockées dans des serveurs sur des machines très hétérogènes et accessibles via divers protocoles de communication. Pour résoudre ce problème, les concepteurs du WWW ont proposé un format standard pour identifier les ressources appelées URI<sup>6</sup> [BL94]. Ce format se décline en différentes variantes selon l'information contenue dans les URIs.

Les URL<sup>7</sup> [Kun95, BLMM94] contiennent des informations décrivant la localisation physique et le protocole d'accès à la ressource sur Internet. Le tableau 4.1 donne un aperçu de quelques formats d'URL pour l'accès aux services principaux d'Internet. Les URL sont donc composées du protocole Internet d'accès suivi du nom du serveur (ou son adresse Internet) et du nom local de la ressource sur le serveur.

Services	Protocole - Site - Nom de la ressource
Mail	<i>mailto:nom_destinataire@site_internet</i>
News	<i>news:nom_du_groupe</i>
Fichiers locaux	<i>file:///chemin_d_accès/nom_du_fichier</i>
FTP	<i>ftp://site_internet/chemin_d_accès/nom_du_fichier</i>
Gopher	<i>gopher://site_internet/chemin_d_accès/nom_du_fichier</i>
WWW	<i>http://site_internet/chemin_d_accès/nom_du_fichier#marque</i> <i>http://site_internet:port/cgi-bin/executable</i>

TAB. 4.1 - URL : Unification de la désignation des ressources Internet

D'un autre côté, les URNs<sup>8</sup> [SM94] décrivent l'identification des ressources par des informations symboliques indépendantes de la localisation. Par exemple, l'URN *internal-gopher-menu* est interprétée par les navigateurs comme l'icône représentant un menu gopher et ce logo n'a pas besoin d'être téléchargé car il est intégré dans le navigateur. Si le navigateur ne dispose pas localement de la ressource, il contacte un service d'annuaire pour transformer l'URN en l'URL, et ensuite il accède à la ressource grâce à l'URL retournée. Ainsi, la même URN peut conduire à des ressources physiquement différentes permettant de répliquer une ressource pour réduire les temps d'accès, de répartir la charge d'accès entre plusieurs serveurs ou bien de fournir une ressource dans la langue de l'utilisateur.

4. DTD : Document Type Definition

5. SGML : Standard Generalized Markup Language

6. URI : Uniform Resource Identifier

7. URL : Uniform Resource Locator ou Localisateur Uniforme de Ressources

8. URN : Uniform Resource Name

### 4.1.3 L'interface universelle d'accès

Le Web a littéralement explosé pour devenir l'outil Internet utilisé par des millions d'utilisateurs à partir de 1993 lorsque Marc Andreessen développa à l'Université de l'Illinois à Urbana Champaign la première version de *Mosaic* un navigateur Web interactif et convivial sous X11. Depuis, de nombreux autres navigateurs ont vu le jour sur toutes les plateformes matérielles : les plus répandus étant *Netscape* et l'*Internet Explorer* de Microsoft.

Ainsi, à travers l'interface de ces navigateurs, l'utilisateur peut converser avec différents types de services sur Internet (cf. figure 4.2) : la messagerie électronique, les forums de discussion, les serveurs FTP, les serveurs WWW ou des serveurs de cache WWW. Le navigateur supporte les protocoles spécifiques à ces services (SMTP<sup>9</sup>, NNTP<sup>10</sup>, FTP<sup>11</sup>, HTTP).

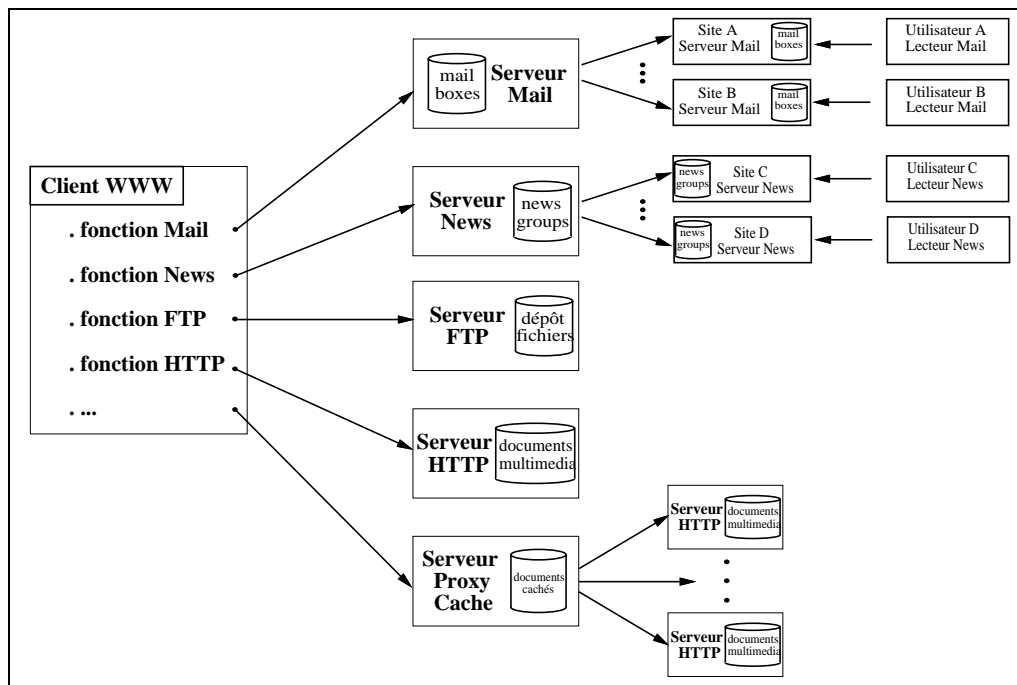


FIG. 4.2 - *Le navigateur WWW universel*

De plus, les navigateurs récents offrent de multiples possibilités d'extension (cf. figure 4.3) aussi bien pour fournir de nouvelles fonctions que pour améliorer l'interactivité :

- **Les visualisateurs externes** : en standard, les navigateurs gèrent un nombre limité de formats MIME (HTML, GIF et quelques autres formats d'image). Par configuration, l'utilisateur peut fournir des visualisateurs externes pour représenter les formats non reconnus par son navigateur pour par exemple afficher des fichiers Postscript, jouer des sons, des vidéos ou voir toute autre sorte de fichiers.
- **Les «plugs-in» clients** : à la différence des visualisateurs externes, ces modules permettent d'étendre les navigateurs et peuvent accéder aux ressources du navigateur comme les fenêtres. Ces modules communiquent avec le navigateur par une API<sup>12</sup>

9. SMTP : Simple Mail Transfer Protocol [Pos82, Cro82]

10. NNTP : Network News Transfer Protocol [KL86]

11. FTP : File Transfer Protocol [BBC<sup>+</sup>71, PR85]

12. API : Application Programming Interface

(comme l'API CCI [Tho95] définie pour Mosaic ou les interfaces de programmation Java pour Netscape [Cor96a]) et sont donc spécifiques à chaque navigateur. Netscape est le navigateur pour lequel le plus de modules ont été écrits : visualisateurs de documents, audio et téléconférences, tableurs ...

- **Le téléchargement de code mobile** : Un code mobile est une application téléchargée depuis un serveur et qui s'exécute dans le contexte d'une page HTML. Ces applications sont écrites dans des langages interprétés et portables comme Java [GM95] (dans ce cas, ce sont des applets), JavaScript dans Netscape, Python [WvRA96], Safe-Tcl [OLW96], Caml [Rou96] ou Obliq [BN96]. Selon le langage, le navigateur télécharge soit un source à interpréter comme dans le cas JavaScript, ou soit un pseudo-code portable à exécuter comme dans le cas de Java [LY96]. Comme le code peut provenir de n'importe quelle source, la sécurité est le problème crucial que doivent gérer les interpréteurs de ces langages.

La prise en compte de nombreux formats de documents, de nombreux protocoles d'accès et l'extensibilité des navigateurs (par des modules ou par téléchargement) offre une interface utilisateur universelle d'accès à de multiples services Internet répondant aux besoins de *transparence* que nous avons exposés dans le chapitre 1. De plus, le coût et la durée de la formation des nouveaux utilisateurs sont alors fortement réduits.

#### 4.1.4 Un serveur général de ressources

L'infrastructure WWW propose donc un large éventail de fonctionnalités du côté des navigateurs mais aussi du côté des serveurs. De nombreuses solutions s'offrent aux fournisseurs de services pour étendre les fonctionnalités des serveurs (cf. figure 4.3) : les modules d'extension ou «plug-in» serveurs, les interpréteurs de «servplets» (applications encapsulées dans un document Web et exécutées sur le site du serveur), les programmes externes interfacés par le protocole CGI. Ces extensions sont accessibles par l'intermédiaire d'URL.

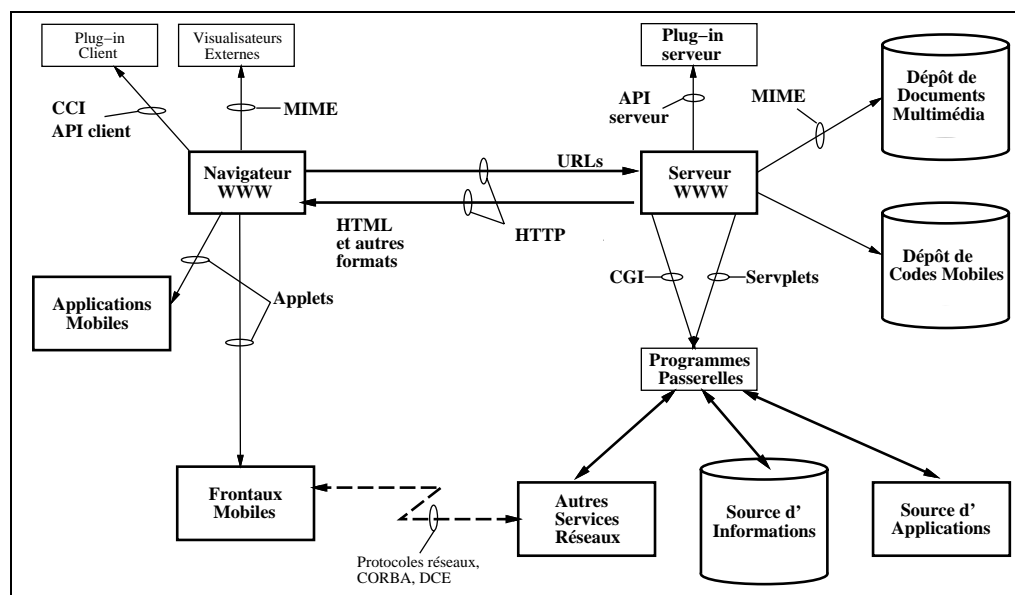


FIG. 4.3 - Les composants du WWW

Un serveur HTTP peut donc être considéré comme un serveur général de ressources [EKL<sup>+</sup>96] :

- **Un dépôt de documents multimédias** : les fournisseurs de services peuvent délivrer à leurs utilisateurs de larges bases d'informations hypermédias et distribuées. Chaque serveur stocke dans son système local de fichiers un ensemble de documents multimédias typés par l'intermédiaire du format MIME.
- **Un dépôt de codes mobiles** : grâce à la centralisation du stockage du code des «applets», les fournisseurs de services peuvent rapidement mettre à jour les applications exécutées sur les postes des utilisateurs alliant ainsi les bénéfices de l'informatique répartie pour l'exécution et de l'informatique centralisée pour la maintenance. De plus, les applets peuvent directement dialoguer avec des services de serveurs.
- **Un serveur de sources d'informations** : par l'intermédiaire des modules d'extension, les fournisseurs peuvent rendre disponible une partie de leurs bases de données locales. Ces programmes interrogent les sources d'informations puis génèrent des documents HTML représentant les résultats des interrogations.
- **Un serveur de sources d'applications** : ces modules peuvent aussi accéder à des applications s'exécutant sur le serveur pour réaliser des traitements plus complexes. L'intégration d'objets Corba du côté d'un serveur WWW entre dans ce cas de figure.

#### 4.1.5 Synthèse et critiques

Le WWW est devenu en quelques années l'environnement distribué le plus répandu et le plus utilisé : des millions de personnes l'utilisent tous les jours dans leur travail quotidien pour rechercher et découvrir des informations. Chaque jour voit l'apparition de nouveaux serveurs d'informations et de services. Cette explosion provient de la simplicité et de la souplesse des concepts du WWW : le protocole HTTP d'échange de documents, les URL pour désigner les ressources réparties, le format MIME pour typer les documents et les navigateurs comme **interface universelle d'accès**. Les extensions des serveurs permettent d'accéder à des applications patrimoines («*legacy applications*») comme des bases de données ou d'autres programmes. Les utilisateurs naviguent et découvrent les informations qui leur sont nécessaires à travers **la métaphore de documents hypermédias** mais ils peuvent aussi agir par l'intermédiaire des **documents actifs** contenant des formulaires HTML ou des applications téléchargées. Mais à ces qualités, nous pouvons opposer les défauts suivants :

- **Le syndrome du serveur monolithique** : comme les serveurs intègrent de plus en plus d'informations et de services, ils deviennent de plus en plus volumineux et difficiles à maintenir. A bien y regarder, cela nous ramène trente ans en arrière à l'époque de l'informatique centralisée : des serveurs contenant tous les traitements et des postes clients servant uniquement de terminaux de consultation. Bien sûr, le WWW a ajouté le multimédia et la navigation entre les serveurs répartis mais le modèle client-serveur est exploité à son minimum.
- **Des documents statiques** : actuellement, la majorité des serveurs WWW ne sont que des dépôts de documents d'informations ou de publicités : la présentation d'une entreprise, de ses produits, de son personnel, ou bien les rapports d'activités et les publications d'un laboratoire de recherche. Ces documents mettent en forme et

contiennent une copie des informations contenues dans le système d'informations de ces organismes. Ils sont rédigés manuellement ou générés automatiquement par des moulinettes, mais ils sont souvent en décalage et non à jour par rapport à ces systèmes d'informations. Ces documents sont donc des ressources statiques, en lecture seule et orientés fichier. La modification de ces ressources ne fait pas partie intégrante du WWW.

- **Le développement «ad hoc» de ressources dynamiques** : pour résoudre ce problème, le WWW offre la possibilité de générer à la volée des documents que ce soit par l'intermédiaire des scripts CGI, des interpréteurs de «servplets» ou des modules d'extension des serveurs. Les concepteurs de documents dynamiques doivent affronter deux problèmes : choisir un mécanisme parmi la multitude offerte par les serveurs et ensuite accéder à leur système d'informations. Leurs développements reposent alors principalement sur des mécanismes propriétaires à chaque serveur et de plus, ils mettent à chaque fois en place des solutions «ad hoc». Cependant, l'accès à des bases de données commence à être bien maîtrisé par exemple avec des produits commerciaux comme O2Web [O2W96] ou le WebServer d'Oracle [OWS96]. Mais l'accès à d'autres types de système pose encore de nombreux problèmes de choix.
- **L'accès à des applications réparties ou distantes** : le WWW intègre de nombreux protocoles pour accéder aux principales applications Internet : la messagerie, les forums, les serveurs FTP, Gopher, Wais ou l'exécution d'applications distantes par Telnet. Mais l'accès direct à des applications non prévues pour le WWW comme des services DCE, des bases de données ou des objets Corba (c.f. section 4.2) reste difficile à mettre en place : comment par exemple interfacier un service de nommage Corba avec le WWW ? Le téléchargement de codes mobiles offre des éléments de réponse pour concevoir de nouvelles applications réellement client-serveur où le navigateur n'est pas uniquement un terminal. Mais le développement de ces codes reste encore complexe et «ad hoc», et nous devons bien constater que la plupart des applets réalisées à ce jour sont uniquement des animations graphiques et rarement des applications complexes qui accèdent à des services distants.

Malgré ces critiques, le WWW peut être considéré comme le système d'exploitation orienté objet (OOOS) le plus répandu à ce jour, car tel que Andrew Black et Jonathan Walpole l'ont montré dans [BW94], de nombreuses fonctionnalités des serveurs HTTP peuvent être comparées à celles d'un OOOS : un *nommage uniforme* des ressources via les URL, des *objets persistants* stockés dans le système de fichier de la machine hôte des serveurs, un *méta-protocole* HTTP avec un ensemble d'opérations de base pour invoquer les ressources, et l'*extensibilité* via le protocole CGI ou les APIs des serveurs.

Mais le WWW manque cruellement d'un **modèle uniforme de structuration des ressources**. Si les documents sont bien adaptés pour des informations non structurées ou semi-structurées, ils ne permettent pas d'encapsuler des ressources structurées et évolutives. Le WWW deviendra un véritable OOOS lorsque des millions de services pourront y être incorporés aisément. Nous allons voir dans la section suivante que l'intégration des objets Corba dans le WWW peut apporter des éléments de réponse.

## 4.2 Des objets Corba dans le WWW

Le WWW et Corba ont été conçus et développés durant la même période mais ils se sont longtemps mutuellement ignorés. Cela provient du fait que leurs objectifs à long terme ne sont pas très éloignés et ils sont donc en concurrence : tous les deux définissent une infrastructure logicielle pour permettre de construire des systèmes d'informations et de services répartis à grande échelle et accessibles pour le plus grand nombre d'utilisateurs depuis des postes de travail très hétérogènes.

Face au choix entre ces deux infrastructures, les concepteurs de services répartis s'orientent de plus en plus vers une association de Corba et du Web pour bénéficier du meilleur de ces deux mondes : l'interface d'accès universelle du WWW et l'approche structurée et modulaire des objets Corba.

### 4.2.1 L'intégration du WWW et de Corba

Durant cette année, le mariage du WWW et de Corba fut au coeur des discussions de nombreuses conférences internationales comme à OOPSLA'95 et 96, à WWW4 et WWW5 [MGG96b], à COOTS'96 [MGG96c] et à l'atelier de travail organisé conjointement par l'OMG et le W3C [MGG96d]. Plus qu'une simple mode, cette intégration permet d'envisager une infrastructure globale de services répartis orientés objet et accessibles à travers une interface Homme-Machine conviviale et largement répandue, dont les éléments principaux sont :

- **La modularité des services** : les objets Corba sont définis par des interfaces IDL clairement spécifiées et indépendantes des mécanismes de mise en œuvre pour leur implantation et pour leur utilisation. Au contraire, les ressources WWW, autres que les documents, sont construites sur des mécanismes «ad hoc» comme par exemple les scripts CGI. Le WWW pourrait pleinement profiter des concepts de Corba pour modulariser, structurer et répartir les ressources.
- **L'interface graphique universelle** : plutôt que de concevoir de nouvelles interfaces graphiques spécifiques à chaque service Corba, les navigateurs WWW constituent une excellente alternative. L'accès à toute information WWW ou à tout service Corba pourrait être réalisé à travers une interface utilisateur universelle. Cela réduit les coûts de développement mais aussi les coûts de formation des utilisateurs.
- **La navigation dans une toile d'objets** : la simplicité d'utilisation du WWW provient de la métaphore de navigation qui est aisément compréhensible par les utilisateurs. Cette métaphore pourrait être appliquée pour naviguer dans des graphes d'objets complexes. Les objets sont alors représentés par des documents. Les liens entre les objets sont matérialisés par des liens hypertextes. L'utilisateur invoque les opérations sur des objets à travers des documents actifs (formulaires HTML ou codes mobiles téléchargés).

Une telle infrastructure allie alors les bénéfices des mondes Corba et WWW aussi bien du point de vue des développeurs que des utilisateurs : Corba pour le développement de services répartis orientés objet hétérogènes et interopérables, le WWW pour l'utilisation conviviale de ces services par la navigation dans des documents hypermédias et actifs. Le mariage de Corba et du WWW constitue alors le support logiciel (le middleware) des vastes «réseaux client/serveur intergalactiques» décrits dans [OHE96].

De nombreuses voies peuvent être envisagées pour réaliser cette intégration. Dans la suite de cette section, nous allons présenter les différents types d'intégration du WWW et Corba (cf. figure 4.4) au sein des trois composantes du WWW :

- **Au sein de l'infrastructure réseau** : le WWW et Corba définissent tous les deux des protocoles de transport de requêtes orientées objet respectivement HTTP et IIOP. Ils pourraient interopérer pour favoriser l'accès à la toile d'objets répartis.
- **Du côté des serveurs** : les objets Corba offrent de bonnes caractéristiques pour encapsuler et structurer de nouvelles ressources du WWW.
- **Du côté des navigateurs** : les navigateurs peuvent acquérir les mécanismes pour accéder à des objets Corba distants et contenir eux-mêmes des objets Corba grâce au téléchargement d'«ORBlets»[OHE96], des applications Corba mobiles.

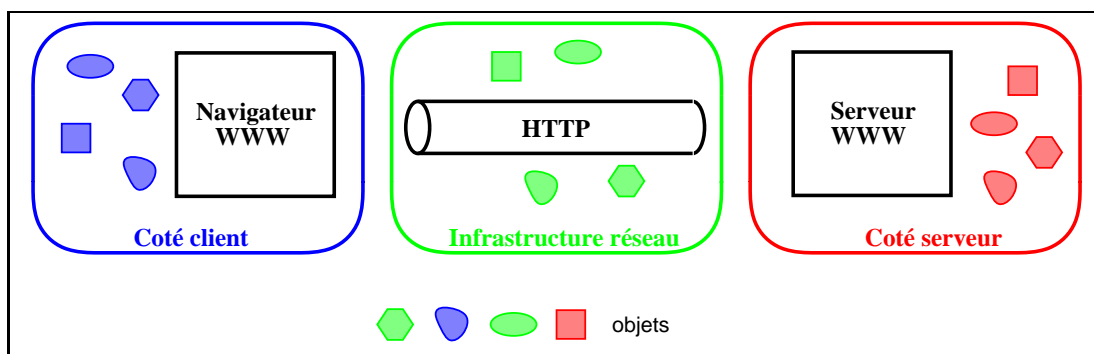


FIG. 4.4 - Où intégrer les objets Corba dans le WWW?

#### 4.2.2 Interopérabilité des protocoles HTTP et IIOP

HTTP est un protocole client/serveur de type RPC pour accéder aux ressources. Il ne définit qu'un petit nombre de méthodes applicables sur les ressources (GET, POST, PUT, DELETE, etc). Si ces méthodes peuvent suffire pour des documents, elles sont trop limitées pour accéder à des ressources plus complexes. De plus, le codage ASCII de HTTP limite les types de données que nous pouvons passer en paramètres de ces méthodes : il ne permet que de transporter des chaînes de caractères devant être ensuite interprétées par les ressources. Au contraire, le protocole IIOP permet de transporter n'importe quelle requête dont la signature est fortement typée par l'intermédiaire du langage IDL. Ainsi, HTTP peut être avantageusement remplacé par IIOP mais ce remplacement implique de revoir tous les serveurs et les clients WWW. Une autre solution est d'évoluer progressivement de HTTP vers IIOP comme illustré sur la figure 4.5.

Le groupe ANSA étudie l'intégration de Corba dans le WWW et l'interopérabilité entre les protocoles du WWW (HTTP, FTP, ...) et le protocole IIOP dans le cadre du projet ISF<sup>13</sup> et ANSAweb [ER94]. Un de leur premier travail dans ce domaine fut d'étudier les différentes étapes pour passer d'un WWW sous HTTP à un WWW sous IIOP [REM<sup>+</sup>95]. Dans un premier temps, ils ont expérimenté le transport de requêtes WWW par le protocole IIOP. Pour conserver l'existant, c'est à dire les navigateurs et les serveurs, ils ont développé deux passerelles l'une étant un proxy HTTP convertissant les requêtes HTTP émises par le

13. Le projet ISF pour Information Services Framework a comme objectif d'étudier et de proposer une infrastructure de déploiement de services orientés objet à grande échelle.

navigateur en requêtes IIOP (1), l'autre étant une passerelle convertissant les requêtes IIOP en requêtes HTTP délivrées au serveur WWW (2). Cette étude leur a permis d'évaluer différentes spécifications en IDL du protocole HTTP. Lors d'une deuxième phase, ils ont conçu un serveur IIOP natif : un serveur WWW modifié pour accepter à la fois des requêtes HTTP et IIOP (3). La dernière étape fut de réaliser un navigateur IIOP natif, c'est à dire un navigateur qui sache émettre des requêtes IIOP (4). L'ANSA a ainsi fait le tour du problème de la transition d'HTTP vers IIOP et a bien énoncé les différentes étapes pour réussir ce passage.

Quoique très intéressant, ce travail reste limité dans ses perspectives car il ne permet que la communication entre des entités à gros grain (les serveurs, les clients) grâce à une spécification de HTTP en IDL. Il ne permet pas d'accéder à d'autres types d'objets Corba comme un service d'annuaire ou notre application bancaire répartie. Pour aller plus loin, une autre piste peut être envisagée en intégrant dans les navigateurs un nouveau type d'URL pour le protocole IIOP (5). La syntaxe de ces URL pourrait être la suivante *iiop://référence\_objet/* et *iiop://référence\_objet/opération/arg1,arg2,arg3* pour respectivement visualiser un objet et appliquer une opération sur cet objet. Mais à notre connaissance, aucun travail n'a évalué les bénéfices et les défauts d'une telle approche. Cependant comme IIOP nécessite de typer les arguments, il y a beaucoup de chances pour que la syntaxe des URL soit très complexe car elle devra contenir les informations de typage traduites sous la forme de chaînes de caractères (nous risquons alors de tomber dans les mêmes défauts que ceux de TclDii, voir section 3.1.4.2).

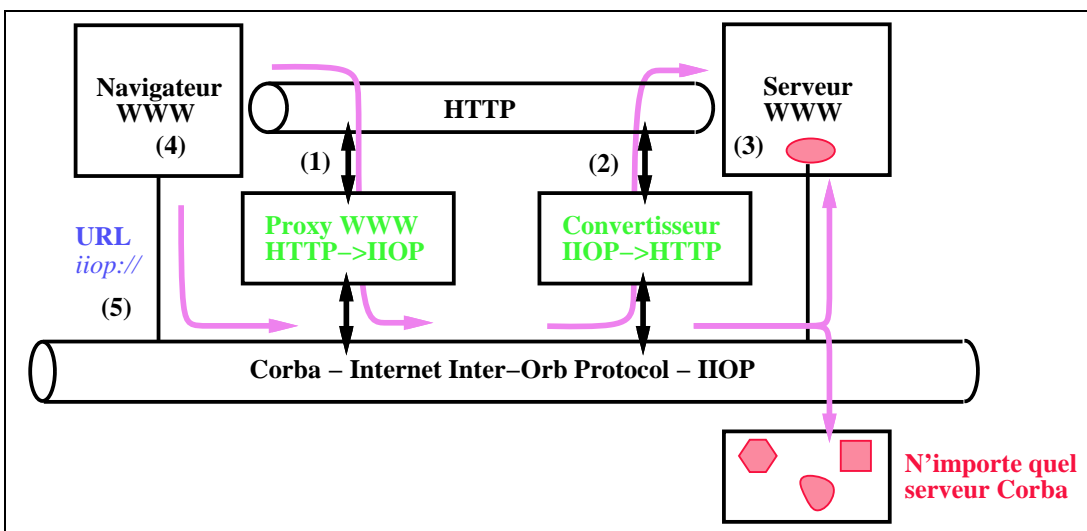


FIG. 4.5 - L'intégration des protocoles de communication

L'interopérabilité des protocoles HTTP et IIOP permet de fusionner les mondes Corba et WWW en résolvant les problèmes techniques de conversion entre ces protocoles. Cependant cette approche n'est pas la solution pour permettre l'intégration transparente de milliers de nouveaux types de services dans le WWW.

### 4.2.3 Des objets dans les serveurs

L'intégration des technologies objet dans les serveurs WWW permet de mieux structurer et de rendre disponibles de nouveaux types de ressources complexes. Diverses pistes ont été ou sont encore étudiées : des *ressources basées sur les technologies objet*, des *passerelles Corba* et des *documents dynamiques*.

#### 4.2.3.1 Des ressources basées sur les technologies objet

Les serveurs WWW deviennent de plus en plus complexes car ils intègrent de nombreuses fonctionnalités : le contrôle d'accès, l'audit des connexions, la gestion des types MIMEs, le besoin de ressources structurées. Afin de modulariser ces fonctionnalités, les serveurs récents utilisent les concepts de la programmation orientée objet.

Les modules du serveur Apache [Apa96a] permettent d'ajouter des extensions développées en C, une de ces extensions intègre l'interpréteur du langage Python pour pouvoir développer de nouvelles ressources. Comme Python est un langage orienté objet, les nouvelles ressources WWW sont alors encapsulées dans des objets Python comme dans l'environnement Python Object Publisher [Ful96]. L'interface fonctionnelle de ces objets contient des méthodes pour le WWW : *Get*, *Post*, *Destroy*, ...

Le W3C a développé une nouvelle version de son serveur HTTP appelé Jigsaw [W3C96a] (Sun a aussi développé son propre serveur WWW appelé Jeeves [SMI96]). Ce serveur est entièrement écrit avec le langage orienté objet Java et permet d'ajouter de nouvelles classes d'objets pour gérer de nouvelles ressources. Comme le support d'exécution de Java autorise le chargement dynamique de codes, ces nouvelles ressources peuvent être instanciées sans arrêter le serveur.

Ces deux solutions visent surtout à pallier l'inefficacité du protocole CGI : un nouveau processus est créé pour chaque requête adressée à un script CGI. L'intégration des ressources dans le serveur permet d'éliminer ce coût. De plus, les ressources peuvent encapsuler un état persistant entre chaque requête. Cependant, de tels serveurs deviennent rapidement très complexes car ils incorporent de plus en plus de ressources et seront difficilement administrables.

D'un autre côté, le projet AnsaWeb a travaillé sur la simplification et l'automatisation de la conception des scripts CGI. En standard, les scripts CGI sont accessibles à travers des formulaires HTML et ils reçoivent leurs paramètres sur l'entrée standard et dans des variables d'environnements. Le développeur doit donc écrire les formulaires et décoder les paramètres CGI. Ce travail répétitif peut être source de nombreuses erreurs et il pourrait donc être automatisé afin de développer rapidement et sans erreurs de nombreux scripts d'accès à de nouveaux services. L'ANSA a donc développé un environnement de développement de scripts CGI [Edw95a]. Chaque script est décrit par une interface IDL et leur compilateur IDL [Edw95b] génère automatiquement les formulaires HTML et le code de décodage des scripts CGI. Les développeurs n'ont plus qu'à se concentrer sur le code de traitement des requêtes.

Les mécanismes que nous venons de présenter améliorent les performances ou réduisent les temps de développement en se basant sur les concepts de la programmation orientée objet. Mais l'intégration d'un nouvel objet nécessite toujours des développements «ad hoc» : de nouveaux scripts CGI ou de nouvelles classes dans les serveurs. De plus, ces ressources objets sont conçues spécialement pour le WWW et sont souvent localisées sur le serveur WWW. Ces solutions n'apportent aucune solution générique pour intégrer simplement des millions d'objets Corba non prévus pour le WWW.

#### 4.2.3.2 Des passerelles Corba

Actuellement, pour accéder à une ressource externe et non standard depuis un serveur WWW, la seule solution est de développer un ensemble de programmes qui agissent comme une passerelle entre la ressource et le serveur. Si une ressource n'a pas de telles passerelles alors elle est inaccessible depuis les navigateurs. Ces passerelles peuvent être implantées

par des scripts CGI ou bien par l'intermédiaire d'une API serveur. Ainsi pour accéder à des objets Corba s'exécutant sur le site du serveur (même machine ou même domaine), il est nécessaire de développer de telles passerelles. Néanmoins, deux types de passerelles peuvent être considérés :

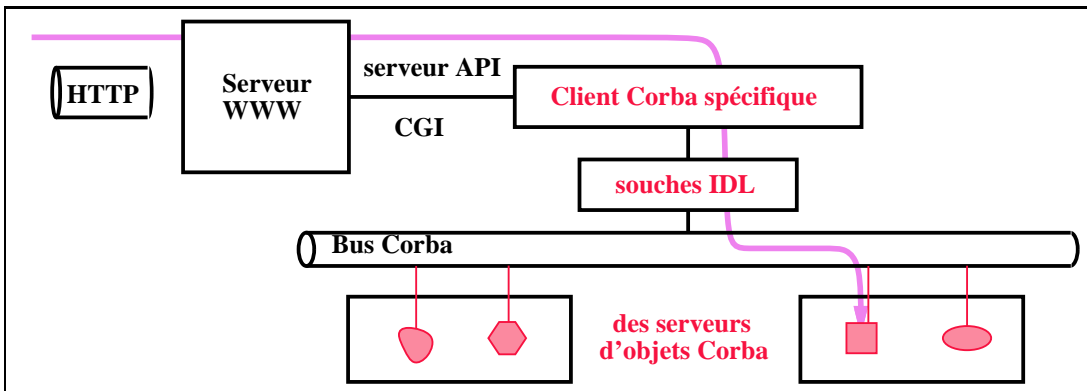


FIG. 4.6 - Une passerelle Corba statique et spécifique

- **Les passerelles statiques** sont des programmes Corba spécifiques qui invoquent les opérations des objets Corba à travers des souches IDL prégénérées. Cette approche est illustrée par la figure 4.6. Mais cela ressemble à la programmation standard de passerelle et nécessite des développements spécifiques pour chaque type d'objets. Lorsqu'un nouveau type d'objets est ajouté, il faut concevoir de nouvelles passerelles. La modification des interfaces IDL oblige à modifier les passerelles utilisant ces spécifications.

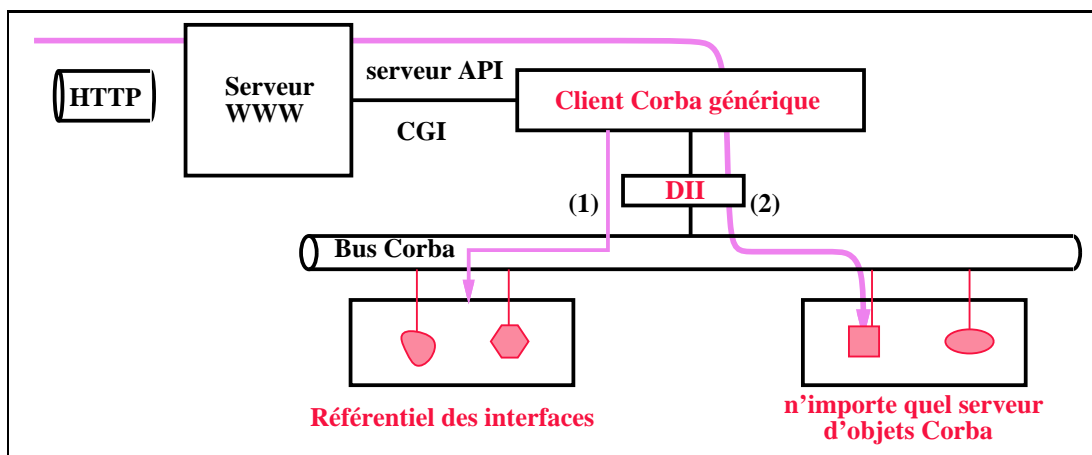


FIG. 4.7 - Une passerelle Corba générique

- **Les passerelles génériques** permettent aux navigateurs WWW d'accéder, de visualiser et d'invoquer n'importe quel objet Corba. Une passerelle générique (cf. figure 4.7) ne connaît pas à l'avance tous les types d'objets à sa conception. Les types sont découverts dynamiquement par l'intermédiaire du référentiel des interfaces (1) ou par n'importe quel autre gestionnaire de types. Les objets sont ensuite dynamiquement invoqués (2) grâce à l'interface d'invocation dynamique (DII). Cela nous permet de naviguer à travers les objets aussi aisément que les utilisateurs naviguent

dans des documents WWW. De plus, l'invocation d'opérations sur les objets peut être faite par des formulaires automatiquement générés à partir des métadonnées du référentiel des interfaces (IR). A notre connaissance, seul notre environnement CorbaWeb présenté dans la section 4.3 a mis en œuvre cette approche.

Ainsi grâce à ces deux types de passerelles, tout objet Corba peut devenir une ressource WWW : les services et utilitaires de base de Corba (comme le service de nommage), des services distribués (c.f. notre service bancaire du chapitre 2) ou bien d'autres objets Corba. Les nouvelles ressources sont alors encapsulées dans des objets Corba et deviennent accessibles aussi bien par le bus HTTP que par le bus IOP. De plus, ces ressources ne sont plus centralisées sur la machine du serveur mais au contraire elles peuvent être distribuées sur l'ensemble des machines du site du serveur afin d'assurer par exemple l'équilibre de charge, la réplication et la tolérance aux pannes. Bien sûr, comme les passerelles sont exécutées sur le site du serveur, elles forment alors un goulot d'étranglement mais rien n'empêche de dupliquer les serveurs pour multiplier le nombre de points d'accès à ces passerelles et donc aux objets Corba. Nous verrons dans la section 4.2.4 que ce problème peut aussi être résolu en téléchargeant du code mobile Corba sur les postes des clients.

Cependant, la principale difficulté à résoudre pour ces deux types de passerelles est de permettre aux utilisateurs de naviguer dans les graphes formés par les objets Corba comme par exemple visiter un contexte de nommage pour retrouver l'objet désiré, une instance de banque, puis ensuite à partir de cette banque accéder aux agences, aux clients et aux comptes bancaires.

Afin de mieux comprendre cette difficulté, prenons le scénario suivant. Nous définissons une passerelle pour chaque type d'objets Corba : une pour le service de nommage et une pour chaque type d'objets du service bancaire (i.e. *banque*, *agence*, *client* et *compte*). Chacune de ces passerelles génère la représentation en HTML des objets dont elle est en charge en consultant l'état de ces objets. Lorsque la passerelle pour la *banque* doit afficher la liste des *agences*, elle va générer un lien hypertexte pour chaque référence de cette liste. Ce lien référence la passerelle permettant de visualiser les *agences*. Ainsi, les passerelles sont statiquement liées par le graphe des relations entre les interfaces IDL. Mais comme le service de nommage peut stocker n'importe quelle référence d'objets Corba, sa passerelle doit connaître la liste de toutes les passerelles disponibles. Ainsi si nous ajoutons un nouveau type d'objets, nous devons écrire une passerelle pour ce type mais aussi modifier la passerelle du service de nommage.

Nous pensons qu'écrire ces passerelles selon l'approche statique limite fortement les possibilités d'évolution et d'extension. Car chaque modification ou ajout dans les spécifications IDL oblige la modification d'un grand nombre de passerelles. Nous pensons que l'approche par passerelles dynamiques permet de résoudre cette difficulté et nous illustrerons dans CorbaWeb la solution que nous avons envisagée à travers le méta script de représentation.

### 4.2.3.3 Des documents dynamiques

La dernière possibilité envisageable du côté des serveurs est la génération de documents dynamiques (cf. figure 4.8). Classiquement, les documents stockés dans un serveur WWW sont statiques et contenus dans des fichiers. Lorsqu'un client accède à de tels documents, le serveur se contente de lire les fichiers et de les transmettre en réponse aux requêtes HTTP. Par contre, les documents dynamiques sont analysés sur le serveur avant d'être transférés (par exemple, le mécanisme «server include» fourni par la majorité des serveurs HTTP).

Cette analyse complète certaines parties du document en allant extraire des informations du système d'informations. Nous pouvons très bien envisager que ce système soit implanté par des objets répartis Corba. Ces zones sont marquées dans le document par des tags spécifiques et ainsi les analyseurs de tels documents peuvent reconnaître les parties statiques des parties dynamiques. Le document stocké dans le serveur est un patron des documents à générer. Ces patrons sont stockés dans un dépôt spécifique ou dans l'arborescence de fichiers du serveur WWW.

Par exemple, l'environnement Web\* [ASM<sup>+</sup>95] est basé sur cette approche. Il définit des pages de présentation (*layout pages*) pour générer les documents dynamiques. Ces pages sont de simples documents HTML dans lesquels du code TclDii [ASG<sup>+</sup>94] a été inséré pour invoquer les objets Corba à travers l'interface d'invocation dynamique (DII) d'Orbix. Ces codes sont délimités par les caractères '[' et ']'. L'interpréteur Web\* de pages de présentation est un programme CGI qui intègre l'interpréteur Tcl, l'extension TclDii et l'interface avec le protocole CGI pour transformer les paramètres CGI en variables Tcl. Cet interpréteur lit les pages de présentation et remplace les codes TclDii rencontrés par les résultats de leurs exécutions. De plus, le WWW étant par défaut sans état, Web\* a étudié un mécanisme pour transférer des informations (variables Tcl) au cours de la navigation entre les pages afin de garder la trace des transactions en cours. Pour cela, il stocke l'état des variables Tcl persistantes dans les URL qu'il génère. Cet environnement est exploité pour naviguer dans un système d'informations hospitalier modélisé par des objets Corba encapsulant des bases de données telles que Oracle.

L'inconvénient majeur de cette approche est de fortement limiter l'exploitation des objets Corba. Les utilisateurs ne peuvent accéder qu'aux pages qui ont été définies. Cet environnement permet d'invoquer les objets Corba depuis les pages mais ne permet pas aux utilisateurs d'appliquer leurs propres traitements sur les objets. Bien sûr, ils peuvent écrire leurs propres pages de présentation mais alors ils devront utiliser TclDii dont nous avons exposé les faiblesses et les défauts dans la section 3.1.4.2 : le faible pouvoir d'expression et la syntaxe complexe d'invocations des objets qui rendent cet environnement peu convivial pour les utilisateurs.

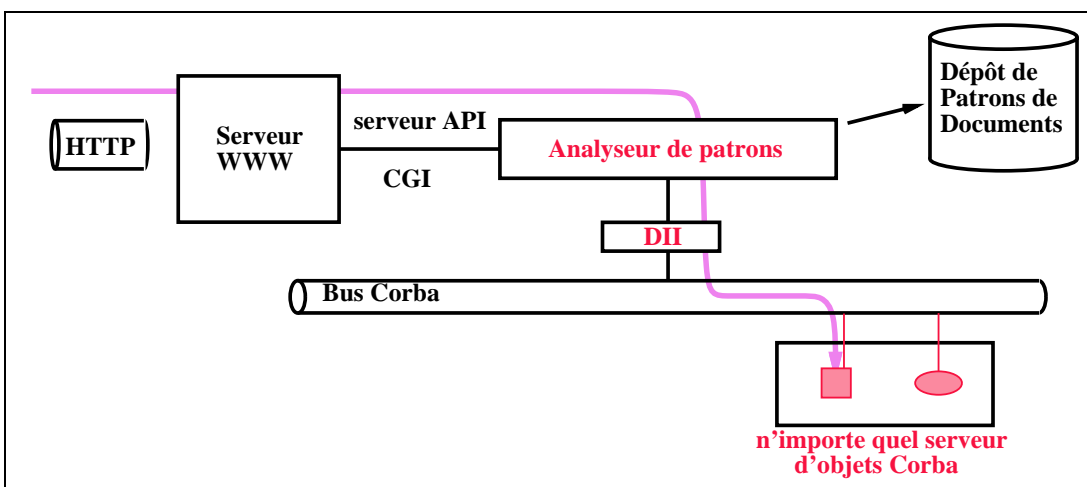


FIG. 4.8 - La génération de documents dynamiques

Cette approche permet donc de concevoir des serveurs WWW dont les documents sont générés dynamiquement en fonction de l'état du système d'informations d'un organisme. Par exemple, Digital Equipment Corporation a développé Web Broker [HK94] pour

fournir l'accès à leurs clients aux informations de l'entreprise : état des stocks, suivi des commandes, informations sur les produits. Web Broker est un environnement qui génère dynamiquement les documents en complétant des patrons (ou dans ce contexte des méta-fichiers HTML) grâce aux bases de données de DEC. Ces bases sont réparties à travers le monde dans les diverses succursales de DEC et sont accédées à travers leur propre Object Broker. De plus, cet environnement contrôle les accès des clients et génère les documents en fonction des clients : la langue maternelle de l'utilisateur et ses autorisations sur les informations.

Dans l'environnement CorbaWeb, nous avons inclus cette fonctionnalité : les patrons sont écrits en HTML et explorent dynamiquement les objets grâce au langage CorbaScript.

#### 4.2.4 Des applets Corba dans le WWW

Toutes les approches que nous venons de présenter ont tout de même un défaut majeur : elles s'exécutent toutes sur le site du serveur créant ainsi un possible goulot d'étranglement. Récemment, le modèle client-serveur du Web a acquis une nouvelle dimension à savoir la possibilité d'exécuter du code sur la station de l'utilisateur. Cela permet de décharger les serveurs et d'améliorer les modes d'interactivités avec les utilisateurs : animation graphique, contrôle de saisie, visualisation sophistiquée de données.

Ces codes mobiles peuvent être écrits dans divers langages portables comme Java, JavaScript, Python, Safe-Tcl, Caml ou Obliq. Selon le langage, le navigateur télécharge soit un source à interpréter comme dans le cas JavaScript, ou soit un pseudo-code précompilé et portable comme dans le cas de Java [LY96]. Actuellement, Java est le langage portable le plus employé dans le Web car c'est le seul dont la machine virtuelle d'exécution a été intégrée dans les navigateurs WWW les plus couramment utilisés.

Dès lors, les fournisseurs d'ORBs ont commencé à définir des règles de projection des spécifications IDL vers du code Java permettant ainsi d'invoquer tout objet Corba à partir d'une application écrite en Java. Quelques produits commencent à être distribués depuis cet été 1996, tels que OrbixWeb [ION96b] et VisiBroker for Java [Vis96]. Actuellement, chacun propose sa propre version des règles de projection, ainsi les applets Corba ne sont pas portables d'un ORB à l'autre. Néanmoins, ce problème devrait être résolu dans les mois à venir : l'étude des propositions et la définition des règles par l'OMG devraient aboutir durant le premier trimestre 1997. De plus, ces fournisseurs ont réécrit leur ORB entièrement en Java permettant ainsi de le télécharger en même temps que les applets.

Cette technologie est encore trop récente pour en saisir tous les tenants et aboutissants car trop peu d'applications complexes ont été réellement implantées. Toutefois, nous pouvons déjà imaginer quelques scénarios d'utilisation de cette nouvelle technologie (cf. figure 4.9) :

1. **L'invocation d'objets Corba distants** : Les applets peuvent invoquer les objets Corba localisés sur le serveur soit via les passerelles du serveur ou via le protocole IIOP. Ces applets sont alors des interfaces graphiques évoluées pour manipuler, invoquer, explorer les objets distants comme par exemple un gestionnaire de contexte de nommage. Dans ce cas, ces applets utilisent des souches IDL et un ORB Java qui sont dynamiquement téléchargés en même temps que l'applet.
2. **L'invocation d'objets Corba locaux** : De même les applets pourraient accéder à l'environnement local de l'utilisateur modélisé par des objets répartis spécifiés en IDL : par exemple, l'interface graphique Fresco [Fre96], les documents composites OpenDoc [Fre96], ou des objets OLE [Bro95] dans le cas d'applets écrites en ActiveX

[Mic95]. Ces applets pourront ainsi coopérer avec le bureau virtuel de l'utilisateur : ses applications et ses objets.

3. **La notification d'objets du navigateur :** Les applets peuvent instancier des objets Corba dans l'environnement de l'utilisateur soit dans l'espace du navigateur ou dans un des serveurs Corba du site de l'utilisateur. Ensuite, ces objets s'enregistrent/s'abonnent auprès des objets serveurs qui les intéressent. Ainsi, ils pourront être notifiés des changements intervenus dans ces objets Corba distants. L'utilisateur n'ira plus chercher l'information dont il a besoin comme c'est le cas actuellement avec le WWW, mais au contraire c'est l'information qui viendra à lui dès qu'elle sera disponible. Le WWW et des applications de travail coopératif pourront bénéficier de ce mode de notification asynchrone.
4. **La communication entre navigateurs :** La communication entre des applets de différents utilisateurs pourrait être supportée par des objets répartis dans chaque navigateur. Ces objets communiqueraient alors grâce au bus IIOP permettant ainsi la conception d'applications multi-utilisateurs sur Internet : des jeux multi-utilisateurs (des MUD<sup>14</sup>) ou bien des applications de travail de groupes.

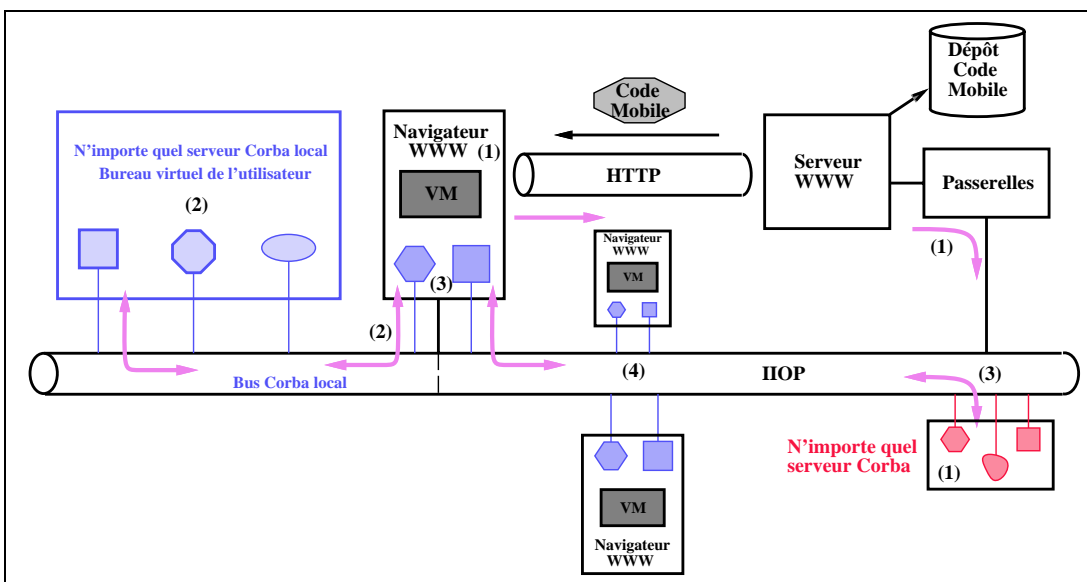


FIG. 4.9 - Des applets Corba dans le WWW

Cependant, si ces scénarios sont très intéressants et prometteurs, Java ne permet pas actuellement de les supporter complètement. Les règles de sécurité des applets interdisent la création d'une connexion réseau (i.e. socket) vers un site différent du site d'où provient le code de l'applet. Ainsi dans nos scénarios, seule la communication entre des objets du navigateur avec des objets du site du serveur WWW est autorisée. Pour pallier à cet inconvénient, il est toujours possible de faire transiter les requêtes entre les objets répartis grâce au protocole HTTP et avec un convertisseur HTTP-IIOP comme nous l'avons exposé dans la section 4.2.2. Une autre possibilité est de faire évoluer les contraintes de sécurité de Java vers plus de souplesse.

14. MUD : Multi-User Dungeon

Mais déjà, l'association de Java et Corba permet de développer localement au sein d'un organisme et de déployer globalement sur le WWW de complexes applications réellement client-serveur. Dans ce contexte, le navigateur n'est plus uniquement un terminal de consultation mais il prend part à l'exécution des services répartis. Ce processus de développement d'applications mobiles apporte de plus un avantage indéniable par rapport au modèle client-serveur classique : la maintenance et l'évolution des applications sont centralisées et donc plus aisées à administrer. De plus, le langage Java étant indépendant des architectures et des systèmes, les applications développées sont directement exécutables sur de nombreux postes de travail hétérogènes réduisant ainsi les coûts de développement. Néanmoins, ces applications seront tout de même complexes à réaliser car elles doivent mettre en œuvre un ORB et des bibliothèques Java telles que AWT<sup>15</sup> pour les interfaces graphiques.

#### 4.2.5 Synthèse

Dans cette section, nous avons présenté l'état de l'art des différentes possibilités d'intégration des objets Corba dans les trois composantes du WWW :

- **Au sein de l'infrastructure réseau** : les protocoles IIOP et HTTP coopéreront dans l'avenir pour permettre d'accéder à la toile mondiale de ressources distribuées depuis n'importe quel site du réseau. IIOP offre une couche générique de transport de requêtes structurées et spécifiées grâce au langage IDL. Comme les requêtes HTTP sont faiblement typées, IIOP pourrait, à plus ou moins long terme, remplacer HTTP pour l'accès à des services autres que des documents.
- **Du côté des serveurs** : les objets Corba permettent de concevoir des ressources WWW modulaires et structurées. Ces services peuvent être répartis sur les différents sites du domaine d'un serveur WWW réduisant ainsi le syndrome du serveur monolithique. De plus, tous les objets Corba sont accessibles par le développement de passerelles «ad hoc» ou génériques. Les documents dynamiques permettent de générer automatiquement des tableaux de bord représentant l'état d'un système orienté objet d'informations et de services. L'approche orientée objet de Corba apporte ainsi un modèle uniforme de structuration des ressources WWW.
- **Du côté des navigateurs** : les codes mobiles Corba permettent d'envisager de nouveaux types d'interactions entre des objets Corba qu'ils soient distants ou locaux et d'imaginer de nouveaux scénarios de coopération entre les utilisateurs du WWW : le téléchargement d'interfaces graphiques d'exploitation des objets Corba, la coopération entre navigateurs pour des services de type collectif ou la notification d'événements et d'informations. Le téléchargement des clients Corba décharge ainsi les serveurs WWW et favorise une exécution complètement distribuée des services. Cependant, les contraintes de sécurité du langage Java limitent fortement les types d'interactions réalisables : seules les communications entre des objets du navigateur et des objets du site d'où provient l'applet sont actuellement possibles.

Ces techniques devront être utilisées conjointement pour répondre aux critiques que nous avons formulées dans la section 4.1.5 au sujet du WWW : le **syndrome du serveur monolithique**, des **documents statiques**, le **développement «ad hoc» de ressources dynamiques**, l'**accès à des applications réparties ou distantes** et finalement l'**absence d'un modèle uniforme de structuration des ressources**.

---

15. AWT : Abstract Window Toolkit

Ainsi, l'intégration du WWW, de Corba et des codes mobiles constituera une infrastructure globale de conception, de déploiement et d'accès en ligne à des services répartis et modulaires. Elle associera le meilleur des mondes WWW et Corba : le nommage globale des ressources distribuées (i.e. URL et référence d'objet IOR), la structuration et la modularité des ressources (i.e. OMA, IDL et CORBA), l'accès aux ressources à travers des protocoles résolvant les problèmes d'hétérogénéité et d'interopérabilité (i.e. HTTP et IIOP) et finalement une interface d'accès, conviviale et hypermédia pour les utilisateurs finaux (i.e. HTML, MIME et les applets pour plus d'interactivités).

Cependant, de nombreuses expérimentations sont encore nécessaires pour bien maîtriser cette nouvelle infrastructure. L'enjeu principal sera de rendre disponible une multitude de types et d'instances de services dans un contexte fortement évolutif. Les utilisateurs auront des rôles distincts et des besoins spécifiques évoluant au fil du temps et des changements dans les systèmes d'informations et de services. De nouveaux objets seront créés dynamiquement et devront être rapidement mis à la disposition des utilisateurs. Les objets patrimoines (c'est à dire non prévus pour le Web) pourraient être accessibles sans nécessiter des changements ou trop de développements spécifiques.

Actuellement, il n'existe que des solutions technologiques pour intégrer les objets Corba dans le WWW (nous en avons fait le tour d'horizon). Ces solutions doivent être mises en œuvre pour chaque objet ou type d'objets à intégrer : le développement de passerelles pour chaque service à intégrer et le développement d'applets pour chaque type d'objets à accéder. Dans un contexte multi-rôles, chaque rôle implique de développer une (ou des) passerelle et/ou applet spécifique. En cas d'évolution des objets comme des changements dans les descriptions IDL ou en cas d'ajout de nouveaux types d'objets, ces passerelles et applets devront peut-être être modifiées. Ces développements Corba sont longs, complexes et pénibles, le lecteur pourra se rapporter aux difficultés liées à la complexité de Corba exposées dans le chapitre 2.

Ainsi, nous avons développé l'environnement CorbaWeb d'intégration d'objets Corba dans le WWW afin de proposer une solution générique et dynamique à ces problèmes.

### 4.3 L'environnement CorbaWeb

Notre objectif est de concevoir une infrastructure globale pour la conception, le déploiement et l'accès à des services orientés objet alliant le meilleur des mondes Web et Corba [MGG96b] : la navigation hypermédia du WWW et l'approche orientée objet de Corba. CorbaWeb est notre environnement d'expérimentation de cette infrastructure basée sur une passerelle générique et dynamique entre le WWW et un bus Corba. Il permet d'offrir l'accès depuis le WWW à n'importe quel type d'objets Corba. Grâce à CorbaWeb, les utilisateurs peuvent naviguer dans des graphes d'objets aussi simplement que dans des graphes de documents et invoquer toute opération sur ces objets. CorbaScript est le noyau d'exécution de cette passerelle offrant la souplesse et la flexibilité pour l'accès à tout objet Corba tel que nous l'avons présenté dans le chapitre précédent.

Dans cette section, nous présentons l'architecture et les fonctionnalités d'intégration offertes par CorbaWeb. Ensuite, nous illustrons chacune de ces fonctionnalités sur les exemples d'objets Corba que nous avons présentés tout au long de ce mémoire : l'objet *grille*, le service Corba de nommage et le service bancaire. Finalement, nous exposons les éléments clés de l'implantation de CorbaWeb.

### 4.3.1 Les principes de CorbaWeb

CorbaWeb permet de rapidement et aisément intégrer des objets Corba dans le WWW pour les rendre accessibles depuis les navigateurs. Ainsi, CorbaWeb offre à la fois un moyen d'accès via le Web à tout service Corba réparti et un moyen d'extension du Web à de nouveaux services, et cela de manière générique contrairement aux solutions fermées proposées jusqu'ici. Cette intégration ne se limite pas à une catégorie d'objets spécifiques mais permet d'accéder à tous les types d'objets Corba à travers CorbaScript.

#### 4.3.1.1 L'architecture CorbaWeb

L'environnement CorbaWeb vise à supporter une infrastructure globale pour la conception, le déploiement et l'accès en ligne à des services orientés objet. Il se décompose de la manière suivante : le bus Corba pour la conception et le support des services, l'infrastructure WWW pour l'accès à ces services et finalement la passerelle CorbaWeb pour déployer ces services sur le WWW. Notre infrastructure bénéficie ainsi du meilleur de chacun de ces deux mondes et les fait interopérer à travers la passerelle CorbaWeb (cf. figure 4.10).

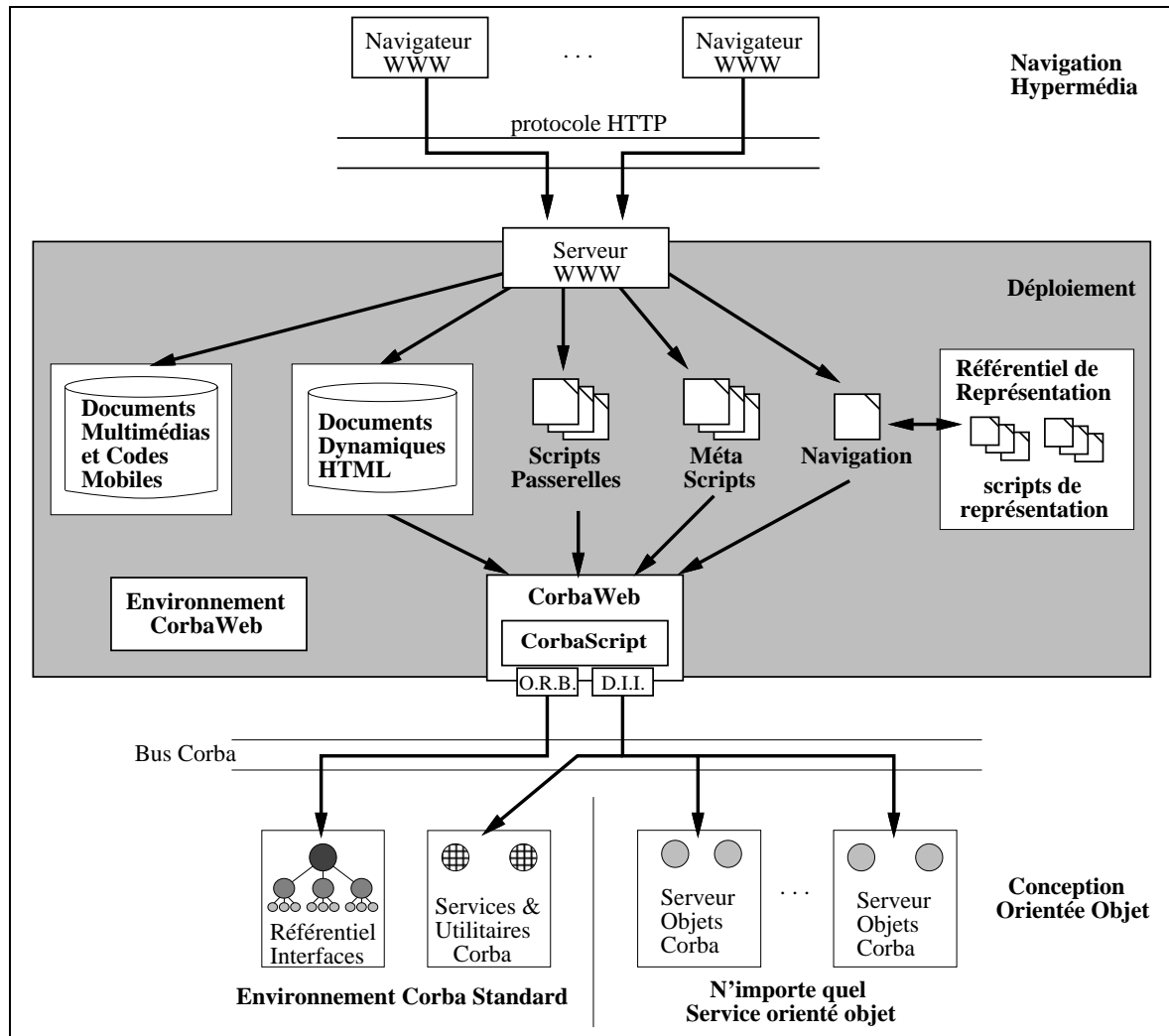


FIG. 4.10 - L'environnement de CorbaWeb

**Le bus Corba** Dans notre infrastructure, les services sont modélisés par des objets Corba qui interopèrent à travers le bus Corba orienté objet et réparti. Le langage IDL permet de spécifier ces services indépendamment de leur implantation et des divers contextes de leur utilisation (les programmes clients). Ces objets peuvent représenter de nouveaux services spécifiquement conçus pour le WWW mais également des services patrimoines d'un système d'informations. Nous considérons sous le terme de services patrimoines tous les objets Corba pour lesquels les concepteurs n'avaient pas imaginé leur utilisation à travers le WWW, comme par exemple le service standard de nommage ou notre service bancaire (c.f. chapitre 2).

L'approche orientée objet de Corba nous offre les qualités requises pour la conception de ces services : la modularité, l'abstraction, l'encapsulation et la réutilisation. Le modèle OMA de l'OMG (c'est à dire l'ORB, les services et utilitaires communs) nous apporte les éléments techniques pour supporter l'hétérogénéité, l'interopérabilité et la communication entre les composants logiciels constituant nos services.

**L'infrastructure WWW standard** A travers les navigateurs standards et le protocole HTTP du WWW<sup>16</sup>, les utilisateurs accèdent aux services du système d'informations localisés sur le bus Corba. Les objets de ces services sont représentés par des documents HTML et MIME standards. Les références d'objet et les relations entre ceux-ci sont représentées par des liens hypertextes standards (ancres HTML et URL). Ainsi, les utilisateurs naviguent à travers ces objets Corba aussi simplement qu'ils naviguent sur le WWW. Les documents actifs (formulaires HTML et langages d'applets) permettent d'invoquer les opérations sur ces objets à travers la passerelle CorbaWeb.

Un navigateur standard (Mosaic, NetScape, HotJava) fournit ainsi l'interface universelle et conviviale permettant à tout utilisateur d'accéder à tous les services orientés objet disponibles sur le bus Corba.

**La passerelle CorbaWeb** CorbaWeb fait la liaison entre les deux environnements présentés ci-dessus. Du point de vue des utilisateurs, CorbaWeb se comporte comme un «classique» serveur WWW. Les navigateurs lui envoient des requêtes HTTP et en retour, CorbaWeb renvoie les documents multimédia et/ou actifs (formulaires et applets) demandés. Du point de vue du bus Corba, CorbaWeb est un ensemble de scripts écrits en CorbaScript permettant d'invoquer n'importe quelle opération de n'importe quel objet via le mécanisme d'invocation dynamique (DII) et le référentiel des interfaces (IR). Nous désignons ces scripts sous le vocable de «scripts CorbaWeb».

Les utilisateurs désignent et accèdent à ces scripts par des URL du WWW. Lorsque une requête HTTP référence un tel script, le serveur CorbaWeb exécute le script référencé. Ce script génère à la volée un document WWW en affichant les résultats de l'invocation des opérations et/ou de la consultation des attributs des objets disponibles sur le bus Corba. Trois types de scripts CorbaWeb sont disponibles pour connecter le monde WWW et celui de Corba : les documents dynamiques, les scripts passerelles et les méta scripts.

1. **Les documents dynamiques** permettent de créer des tableaux de bord HTML synthétisant l'état d'un ou plusieurs objets, la partie statique est exprimée en HTML et celle dynamique en CorbaScript.

---

16. Nous voulons réutiliser et ne pas modifier la technologie WWW actuelle pour pouvoir bénéficier de toutes les améliorations futures : protocole sécurisé de communication, nouveaux langages d'applets et nouvelles extensions des navigateurs.

2. **Les scripts passerelles** permettent d'offrir un accès spécifique à un ou des objets pour consulter leur état et aussi pour appliquer des opérations sur ceux-ci.
3. **Les méta scripts** fournissent un moyen de concevoir des traitements génériques applicables sur tout type d'objets comme par exemple l'exécution d'opérations, la génération automatique d'interfaces HTML et la navigation dans des graphes d'objets en fonction du profil des utilisateurs.

Ainsi, CorbaWeb offre une claire séparation entre les aspects fonctionnels d'un service orienté objet et sa représentation/utilisation depuis le WWW. Des services non conçus pour le WWW peuvent y être aisément intégrés et ils ne nécessitent aucun changement dans leur spécification IDL ou dans leur implantation (répartition, langages, OS, ...). De nouveaux services spécifiques au WWW peuvent être développés et rendus accessibles grâce à la souplesse et la flexibilité du langage CorbaScript.

**L'interpréteur CorbaWeb** L'interpréteur de scripts CorbaWeb est une extension de l'interpréteur CorbaScript dont nous avons présenté l'implantation dans la section 3.3. Lorsque les requêtes HTTP parviennent au serveur WWW, elles sont transformées en un ensemble de variables du langage CorbaScript et sont alors utilisables dans les scripts CorbaWeb. Comme les scripts génèrent des documents pour le WWW, nous avons défini une table de correspondance entre les éléments du langage IDL et leur représentation en HTML : les types complexes IDL (structures, unions, tableaux) sont représentés par des listes HTML, les références d'objet par des ancres, les opérations et attributs par des formulaires.

Avant d'illustrer les trois types de scripts CorbaWeb, il est nécessaire de préciser les éléments nouveaux, introduits dans CorbaWeb, par rapport au langage CorbaScript tel que nous l'avons décrit au chapitre 3 : les règles de correspondance entre le langage IDL et le langage HTML, les modules CorbaScript pour simplifier l'écriture des scripts CorbaWeb et les variables CorbaScript associées aux requêtes HTTP. Ces précisions permettent ensuite de lire plus facilement les exemples de scripts CorbaWeb présentés à partir de la section 4.3.2.

#### 4.3.1.2 La correspondance entre le langage IDL et le langage HTML

Dans l'environnement CorbaWeb, nous utilisons le langage HTML comme langage de description des interfaces graphiques. Les scripts CorbaWeb génèrent de tels documents pour permettre aux utilisateurs de voir l'état des objets Corba et pour leur appliquer des traitements tels que l'invocation d'opérations, la consultation ou la modification d'attributs sur les objets Corba. Pour cela, nous avons défini des règles associant à chaque concept du langage IDL une représentation en HTML. Ces règles sont alors utilisées par l'interpréteur CorbaWeb et aussi par les scripts pour représenter chaque type de valeurs IDL via un texte HTML correspondant et pour appliquer des traitements via des formulaires HTML. Le tableau 4.2 résume ces différentes règles de correspondance.

Les scripts génèrent les documents HTML en utilisant l'instruction *print* du langage CorbaScript. Lorsqu'un script affiche une valeur d'un type simple IDL, l'instruction *print* génère un simple texte HTML comme dans le cas des chaînes et des caractères. Les valeurs des différents types d'entiers et de réels sont représentées par leur valeur numérique. Une valeur d'un type énuméré est affichée symboliquement, *true* ou *false* pour les booléens, *masculin* ou *feminin* pour le type énuméré *sexe*.

Pour les types composés, nous utilisons les tags de structuration du langage HTML. Les valeurs de type structure et union sont représentées par des listes simples du langage HTML (i.e. `<UL>..</UL>`). Les éléments de la liste sont alors composés de leurs champs parcourus et affichés récursivement. Un élément de liste est composé du tag `<LI>` suivi du nom IDL du champ et de sa valeur.

Les tableaux et les séquences sont quant à eux représentés par des listes ordonnées HTML (i.e. `<OL>..</OL>`). Comme l'affichage est réalisé récursivement, nous pouvons obtenir de multiples listes imbriquées pour des déclarations de types IDL composées et complexes, comme un tableau de structures contenant elles-mêmes des séquences de tableaux ... Les tableaux et séquences d'octets sont les seuls cas dérogeant à cette règle.

Pour ceux-ci, aucune interprétation n'est réalisée et leur contenu est simplement imprimé sur le flux de sortie. De toute façon, ces données n'étant pas typées, nous ne pouvons faire aucune correspondance avec le langage HTML. Par contre, nous avons utilisé ce cas pour créer des objets Corba stockant des données binaires comme par exemple des images ou des sons (i.e. des Binary Large Objects ou Blobs[PHS96]).

Concept Corba	Traduction HTML	Représentation HTML
<b>Types IDL de base</b>	simples textes	
entier et réel	valeur numérique	<i>10</i>
chaîne et caractère	valeur texte	<i>un texte</i>
booléen et énumération	valeur texte	<i>true, masculin</i>
<b>Types IDL composés</b>	tags de structuration	
structure et union chaque champ	liste simple entrée de liste	<code>&lt;UL&gt;...&lt;/UL&gt;</code> <code>&lt;LI&gt;&lt;B&gt; champ&lt;/B&gt; valeur</code>
tableau et séquence chaque élément	liste ordonnée entrée de liste	<code>&lt;OL&gt;...&lt;/OL&gt;</code> <code>&lt;LI&gt; valeur</code>
tableau et séquence d'octets	contenu représenté sans interprétation	
<b>référence d'objet</b>	ancres vers l'URL du méta script de visualisation	<code>&lt;A href="/cw/view.cs?reference"&gt;</code> <i>référence objet</i> <code>&lt;/A&gt;</code>
<b>interface d'un objet</b>	des formulaires pour chaque opération et attribut	
<b>opération</b>  chaque paramètre si énumération si séquence de chaîne si autres types invocation	formulaire référençant le méta script d'exécution un élément graphique un menu déroulant un éditeur de texte un champ texte un bouton de soumission	<code>&lt;FORM action="cw/exec.cs"</code> <code>method=post&gt;...&lt;/FORM&gt;</code>  <code>&lt;SELECT&gt;...&lt;/SELECT&gt;</code> <code>&lt;TEXTAREA ...&gt;...&lt;/TEXTAREA&gt;</code> <code>&lt;INPUT type=text ...&gt;</code> <code>&lt;INPUT type=submit ...&gt;</code>
<b>attribut</b>  paramètre et invocation	un formulaire pour consulter un pour modifier l'attribut idem que pour les opérations	<code>&lt;FORM ...&gt;...&lt;/FORM&gt;</code>

TAB. 4.2 - Les règles de correspondance entre le langage IDL et le langage HTML

Les références d'objets sont, par défaut, automatiquement représentées par des ancres HTML (i.e. `<A href = ..>..</A>`). L'ancre affiche la référence au format CorbaScript c'est à dire `interfaceIDL("nom:serveur:machine")` et elle référence le méta script de représentation des objets Corba `/cw/view.cs`. Si l'utilisateur active l'ancre alors le méta script

est exécuté et il génère une représentation visuelle de l'objet. Cette représentation peut être paramétrée selon le type de l'objet ou être générée automatiquement à partir de la description IDL de l'objet. Nous reviendrons plus en détail sur ce méta script dans la section 4.3.3.4. Néanmoins si ce format d'ancre n'est pas adapté, le concepteur de scripts CorbaWeb peut générer manuellement l'ancre qu'il désire.

Pour l'application de traitements sur les objets, nous utilisons les formulaires HTML (i.e. `<FORM ..>..</FORM>`). Nous verrons dans la sous-section suivante que l'environnement CorbaWeb fournit des primitives pour générer manuellement ou automatiquement ces formulaires. Pour la génération automatique, CorbaWeb consulte le référentiel des interfaces afin de découvrir le type des paramètres pour l'invocation des opérations. A partir de ces informations, il génère automatiquement les champs pour saisir les paramètres : les énumérations sont représentées par des menus déroulants (i.e. `<SELECT><OPTION>..</SELECT>`), les séquences de chaînes par des zones d'édition de texte (i.e. `<TEXTAREA ..>..</TEXTAREA>`). Les formulaires, automatiquement générés, référencent le méta script d'exécution d'opérations sur lequel nous reviendrons dans la section 4.3.3.2. Lorsque l'utilisateur a rempli les zones d'un formulaire, il peut alors sélectionner le bouton de soumission (i.e. `<INPUT type=submit ..>`).

Dans l'avenir, nous intégrerons les applets à la place de certains formulaires. Mais alors cela nécessitera des développements d'applets spécifiques et non génériques car il nous paraît actuellement difficile de générer automatiquement des applets à l'exécution comme nous pouvons le faire pour les formulaires.

Les règles pour les valeurs IDL sont automatiquement appliquées par l'interpréteur lorsqu'un script les affiche par l'intermédiaire de l'instruction *print*. Dans le cas des formulaires, le script doit appeler des primitives effectuant la génération. Ces primitives sont regroupées dans deux modules écrits en CorbaScript que nous allons présenter maintenant.

#### 4.3.1.3 Les modules et variables utilitaires de CorbaWeb

Pour simplifier et aider à la génération de documents HTML, nous avons implanté deux modules CorbaScript qui peuvent être importés par les scripts CorbaWeb. Le module HTML rassemble de nombreuses procédures pour générer des tags balisant un document HTML. Le module CW contient quant à lui des procédures en relation avec les objets Corba : les ancres et des formulaires automatiquement générés.

Le tableau 4.3 énumère quelques unes des procédures du module HTML. Ces procédures permettent de générer les tags pour structurer le document HTML : l'entête MIME (i.e. *mime\_header()*), les tags balisant le début et la fin des composantes d'un document (i.e. le document lui-même, son entête et son corps), le titre et les sections, la mise en page (comme *P()* et *BR()*), la typographie (comme *PRE()*, *B()* et *I()*), les ancres hypertextes, les listes, les formulaires et bien d'autres pour les tableaux, les images et les applets.

La procédure *iterate\_list* génère une liste HTML en exécutant une fonction sur chacun des éléments d'un tableau illustrant le polymorphisme des arguments d'une procédure CorbaScript (nous avons déjà montré ceci sur la procédure de tri d'un tableau).

Un script peut construire des formulaires grâce à quelques procédures simples (comme *form()*, *input..()*). Les formulaires HTML ont deux modes de soumission *GET* et *POST*. Dans le premier mode, les données saisies par l'utilisateur sont encodées dans l'URL qui activera le script associé au formulaire. Le second mode transmet les données dans le corps de la requête HTTP, cela permet de passer une quantité plus importante de données entre l'utilisateur et le script. Lorsque l'utilisateur soumet un formulaire (sélection du bouton de soumission), le script associé au formulaire est exécuté et CorbaWeb crée automatiquement

des variables CorbaScript contenant les données saisies par l'utilisateur. Chacune de ces variables est désignée par le nom associé à la zone du formulaire.

Procédures	Tags HTML	Génération HTML
mime_header ()	entête MIME	<i>Content-type: text/html</i>
begin () end ()	début document fin document	<HTML> </HTML>
begin_head() end_head()	début d'entête fin d'entête	<HEAD> </HEAD>
begin_body () end_body ()	début du corps fin du corps	<BODY> </BODY>
TITLE (message) H (num,texte)	titre du document début d'une section	<TITLE>message</TITLE> <Hnum>texte</Hnum>
P () BR ()	nouveau paragraphe retour à la ligne	<P>  
PRE(texte) B (texte) I (texte)	texte préformaté texte en gras texte en italique	<PRE>texte</PRE> <B>texte</B> <I>texte</I>
a_href (URL,ancre)	ancre hypertexte	<A href="URL"> ancre</A>
iterate_list (liste, fonction)	liste d'entrée fonction sur chacune	<UL>..</UL> <LI>
form (script, query_string,method) form_get (script,query) form_post(script,query) end_form ()	début d'un formulaire    fin de formulaire	<FORM ACTION="script?query_string" METHOD=method> method=GET method=POST </FORM>
input_submit (valeur) input_reset (valeur) input_text(nom,valeur, taille) textarea (nom,cols,ligs, valeur)	bouton de soumission de réinitialisation champ de saisie texte  un éditeur de texte	<INPUT TYPE="submit" VALUE="valeur"> <INPUT TYPE="reset" VALUE="valeur"> <INPUT TYPE="text" NAME="nom" VALUE="valeur" SIZE=taille> <TEXTAREA NAME="nom" COLS=cols ROWS=ligs> valeur</TEXTAREA>
Autres procédures	pour les autres tags	listes, tableaux, images, applets

TAB. 4.3 - *Le module HTML*

Le second module de l'environnement CorbaWeb est résumé dans le tableau 4.4. Ce module contient quelques procédures utilitaires à notre environnement : il permet entre autres de générer le copyright de l'environnement, des ancres sur des objets ou sur l'exécution d'opérations et de générer automatiquement les formulaires sur l'interface IDL d'un objet.

Nous avons défini une procédure pour générer manuellement dans un script des ancres (i.e. *object2URL(objet,ancre,script)*). Cette procédure permet ainsi de spécifier l'objet Corba cible du lien hypertexte, le texte visible par l'utilisateur (i.e. *ancre*) et le script CorbaWeb qui est exécuté lorsque l'utilisateur sélectionne l'ancre. Cette procédure permet ainsi de personnaliser la génération automatique. De plus, comme le format des URL interdit de stocker directement les caractères spéciaux tels que les guillemets et les parenthèses dans des URL, il est nécessaire d'encoder les références d'objet pour permettre leur transport dans des URL car elles utilisent ces caractères spéciaux. L'exemple ci-dessous

Procédures	Fonction
copyright (contexte)	génération du copyright de CorbaWeb
object2URL (objet, ancre, script)	génération d'une ancre HTML pour l'objet cette ancre référence un script CorbaScript
method2URL (objet, operation, ancre, script)	génération d'une ancre HTML permettant d'exécuter une opération sur l'objet
generate_interface (objet)	génération automatique de l'ensemble des formulaires représentant l'interface IDL
Autres procédures	génération des opérations et des attributs

TAB. 4.4 - *Le module CW*

illustre ce recodage :

```
refGrille = grille("UneGrille:ServeurDeGrilles:duff")
CW.object2URL (refGrille, "Référence sur la grille", "/cw/view.cs")
# l'instruction précédente dans un script CorbaWeb génère le texte HTML suivant
<A href="/cw/view.cs?grille%28%22UneGrille:ServeurDeGrilles:duff%22%29">Référence sur la grille</A>
```

Cette procédure cache ainsi au développeur de scripts et encapsule complètement les limitations et les difficultés d'utilisation des URL pour transporter des références d'objet CorbaScript. Elle réalise automatiquement cet encodage. Le décodage est réalisé par le noyau CorbaWeb. Lorsqu'il reçoit une requête HTTP, il transforme automatiquement les *%nn* en caractères ASCII.

La procédure *method2URL* réalise le même type de traitement que *object2URL* (encodage au format URL) mais en plus de la référence d'objet, elle permet de transporter un fragment de script à appliquer sur l'objet lorsque l'ancre sera activée par l'utilisateur comme par exemple pour invoquer une opération, consulter ou modifier un attribut.

La procédure *generate\_interface* permet quant à elle de générer automatiquement les formulaires HTML représentant l'interface fonctionnelle d'un objet Corba. Elle applique les règles de correspondance pour les opérations et les attributs que nous avons présentés précédemment. Chaque formulaire est composé d'un bouton de soumission et d'un ensemble de zones graphiques pour saisir des paramètres (menu déroulant, champ texte et éditeur de texte). L'action associée à ces formulaires applique l'opération, modifie ou consulte l'attribut correspondant.

Nous sommes actuellement en train de compléter ces deux modules HTML et CW afin d'améliorer et de simplifier leur utilisation. Ces améliorations passeront en partie par la définition d'un ensemble de classes d'objets CorbaScript.

Cette énumération peut paraître un peu confuse et longue mais elle permettra une lecture plus aisée des nombreux exemples de scripts CorbaWeb qui vont suivre jusqu'à la fin de ce chapitre.

Mais avant ces illustrations, il est nécessaire de préciser un dernier élément de l'environnement CorbaWeb à savoir la traduction des requêtes HTTP en variables CorbaScript. Lorsque l'environnement CorbaWeb reçoit une requête d'accès à un script CorbaWeb, il transforme cette requête en un ensemble de variables CorbaScript et lance l'exécution du script demandé par la requête. Nous avons repris les noms de variables définies par le protocole CGI qui permet la communication entre un serveur WWW et ces programmes d'extensions. Ce choix provient du fait que CorbaWeb peut être interfacé avec un serveur WWW classique via ce protocole. Comme actuellement ce protocole est le plus utilisé pour étendre les serveurs et que ces variables sont bien connues des développeurs de programmes

CGI, nous avons estimé que cela ne servirait à rien d'introduire notre propre notation. Le tableau 4.5 donne un aperçu de quelques unes des variables ainsi créées.

Variables	Fonction
SCRIPT_NAME	nom du script courant
QUERY_STRING	paramètre d'invocation du script
REMOTE_USER	identification de l'utilisateur
HTTP_REFERER	URL de la source de cette ancre
Variables script	chacun des champs du formulaire
Autres variables	pour le protocole HTTP

TAB. 4.5 - *Les variables HTTP*

La variable *SCRIPT\_NAME* permet au script CorbaWeb de connaître le nom qui sert à le référencer dans le WWW. La chaîne de paramètre du script est obtenue grâce à la variable *QUERY\_STRING*. L'identification de l'utilisateur est contenue dans la variable *REMOTE\_USER* permettant de réaliser des traitements différents en fonction de l'utilisateur. Nous illustrons cette possibilité dans le méta script de représentation qui génère la représentation des objets Corba en fonction de l'utilisateur qui y accède. *HTTP\_REFERER* permet de connaître l'adresse du document WWW référençant le script courant. Cette variable permet ainsi de pouvoir générer des ancres de retour en arrière. Par exemple, lorsqu'un utilisateur invoque une opération à partir d'un formulaire, le script exécutant l'opération peut générer une ancre pour permettre de retourner sur le document contenant le formulaire.

De nombreuses autres variables sont aussi disponibles pour obtenir des informations sur le serveur WWW (comme nom et adresse IP), le navigateur (nom, liste des formats MIME acceptés, adresse IP) et sur la requête HTTP (version du protocole, mode d'authentification). De plus, toutes les variables des formulaires sont automatiquement transformées en variables CorbaScript. Actuellement, toutes ces variables contiennent des chaînes de caractères et nécessitent d'être converties vers d'autres types de données CorbaScript (entier, réel, booléen, tableau, référence d'objet ...) si nécessaire. Pour réaliser ces conversions, nous utilisons l'opérateur CorbaScript d'évaluation de chaînes (*eval("10")* retourne un objet CorbaScript de type *CS\_Long* et contenant la valeur 10). A l'avenir, nous allons réfléchir sur une convention de nommage des variables des formulaires pour les typer, ainsi l'interpréteur CorbaWeb créera directement des valeurs typées plutôt que des chaînes.

Nous avons présenté les règles de correspondance entre le langage IDL et le langage HTML ainsi que les nouvelles fonctionnalités introduites dans CorbaWeb. Celles-ci nous permettent d'écrire les scripts CorbaWeb interfaçant les objets Corba avec les navigateurs du WWW.

Nous allons pouvoir enfin illustrer réellement les scripts CorbaWeb en mettant en œuvre toutes ces fonctionnalités. Nous passons en revue les différents modes d'utilisations de CorbaWeb, c'est à dire les documents dynamiques, les scripts passerelles, les méta scripts et finalement la navigation. Nous illustrons ces modes à travers les exemples d'objets Corba que nous avons présentés au cours de cette thèse : la *grille*, le *service de nommage* et le *service bancaire*. Il est intéressant de noter qu'aucun de ces services n'a été conçu à l'origine pour être exploité depuis le World-Wide Web : l'exemple le plus marquant est le service Corba de nommage défini par l'OMG et que nous réutilisons tel quel sans le modifier. Ainsi, ces exemples montrent que l'environnement CorbaWeb est réellement une passerelle générique et dynamique pour accéder depuis le WWW à tout objet Corba.

### 4.3.2 Des ressources WWW dynamiques

Cette présentation par l'exemple des divers modes de scripts de CorbaWeb se déroule en trois temps. Dans cette sous-section, nous discutons des documents dynamiques et des scripts passerelles. La suivante discute de l'intérêt des méta scripts en présentant quatre exemples génériques. Puis, nous illustrons la navigation dans le service de nommage et le service bancaire. Finalement, nous concluons cette présentation de l'environnement CorbaWeb par quelques précisions sur l'implantation et nos perspectives de travail.

#### 4.3.2.1 Des documents HTML dynamiques

Le premier mode d'utilisation de CorbaWeb permet de définir des documents dynamiques. Ils permettent de générer aisément des tableaux de bord plus ou moins complexes représentant l'état d'un système d'informations modélisé par des objets Corba. A l'inverse des classiques documents statiques du WWW, ces documents sont générés à la volée chaque fois que les utilisateurs les consultent. Le principal avantage pour les utilisateurs est alors de toujours obtenir des informations à jour. Du côté des fournisseurs d'informations, cette approche leur permet d'automatiser la génération de tels tableaux de bord et de pouvoir concevoir rapidement et aisément une multitude de nouveaux tableaux en fonction des besoins des utilisateurs.

Ces documents sont stockés dans des fichiers patrons HTML. La partie statique est écrite avec le langage HTML et la partie dynamique est exprimée par l'intermédiaire de scripts CorbaWeb. Les scripts accèdent aux objets du système d'informations à travers les mécanismes offerts par le langage CorbaScript (invocation d'opérations et accès aux attributs). Plusieurs fragments de scripts peuvent être contenus dans un même patron. Les variables et les procédures sont partagées entre ces fragments. Ces fragments sont encadrés entre des crochets (*[ print ville ]*) pour les isoler de la partie statique du patron HTML.

L'interpréteur CorbaWeb charge les patrons et les interprète. Les parties statiques sont envoyées directement au navigateur sans aucune interprétation. Lorsque l'interpréteur rencontre le marqueur de début '[' de scripts, il exécute alors le code CorbaScript jusqu'au marqueur de fin ']'. Lorsque ces scripts affichent des résultats par l'instruction *print*, ces impressions sont redirigées vers le navigateur. L'interpréteur utilise les règles de correspondances des données IDL en tags HTML que nous avons précédemment exposées : affichage récursif des valeurs composées et génération d'ancres pour les références d'objets.

La figure 4.11 illustre l'exemple d'un tel patron. Ce patron permet de générer dynamiquement la liste des comptes débiteurs d'une agence de notre système bancaire. A partir de leur navigateur WWW, les gestionnaires d'agence obtiennent ainsi un tableau récapitulatif et toujours à jour des «mauvais» clients. Ce tableau est désigné dans le WWW par une URL classique (*http://duff:8080/cw/liste\_des\_comptes\_debiteurs.html*). Cette URL peut être contenue dans tout autre document WWW de l'agence mais aussi dans des annuaires d'URL («signets» d'un navigateur, pages personnelles, et moteurs de recherche). L'utilisateur n'a plus qu'à sélectionner cette URL et le document dynamiquement généré lui sera retourné par le serveur WWW, ici, le serveur fonctionnant sur le port 8080 de la machine *duff*.

```

[ # code CorbaScript d'initialisation
  NS = CosNaming::NamingContext ("root:NS")
  agence = NS.resolve ( [ ["bancaire",""], ["agence",""] ] )
  ville = agence.adresse.ville
]
<HTML> <HEAD>
<TITLE>Liste des comptes débiteurs de l'agence de
[ # affichage du nom de la ville de l'agence
  print ville
]
</TITLE> </HEAD> <BODY>
<TABLE ALIGN=ABSCENTER BORDER=5 CELLSPACING=5 CELLPADDING=5>
<CAPTION>Liste des comptes débiteurs de l'agence de
[ print ville ]
</CAPTION>
<TR> <TH>Nom du client<TH>Numéro Compte<TH>Solde débiteur </TR>
[ # affichage de la liste des comptes débiteurs
  clients = agence.liste_des_clients ()
  for client in clients {
    nom_client = client.nom
    comptes = client.liste_des_comptes ()
    for compte in comptes {
      solde = compte.solde
      if ( solde < 0 ) {
        print "<TR>"
        print "<TD> ", nom_client, "</TD>"
        print "<TD> ", compte.numero, "</TD>"
        print "<TD ALIGN=RIGHT> ", solde, "</TD>"
        print "</TR>\n"
      } } }
]
</TABLE> </BODY> </HTML>

```

FIG. 4.11 - La liste des comptes débiteurs d'une agence

Lorsque le document est généré à partir de ce patron, le premier fragment de code consulte le service de nommage pour retrouver la référence de l'agence. La référence du service Orbix de nommage est ici créée manuellement car par défaut, ce service a toujours la même référence. Ensuite, en appliquant l'opération *resolve*, nous recherchons la référence de l'agence locale au système d'informations. Cette référence est stockée sous le nom *agence* dans le contexte de nommage *bancaire* accessible depuis la racine du service de nommage pointée par *NS*.

Il est important de noter que ce script s'exécute sur la machine *duff* mais que l'agence et les contextes de nommage peuvent être répartis sur n'importe quelle machine du réseau. L'objet *agence* peut être déplacé sans pour autant devoir réécrire ce script. Cette indépendance est la conséquence de l'utilisation d'un service de nommage qui permet d'obtenir les références des objets à l'exécution.

L'objet *agence* est interrogé pour obtenir le nom de la ville. De manière transparente, le script accède à l'attribut *adresse* puis au champ *ville* de la structure de type IDL *adresse* retournée. Ensuite, le patron est constitué d'un ensemble de tags standards HTML pour structurer le document : structuration, titre et description de l'entête d'un tableau constitué de trois colonnes. Nous avons placé le fragment *[ print ville ]* pour afficher la ville dans le titre du document et dans la légende du tableau HTML.

La partie principale du patron va parcourir la liste des comptes de chaque client de l'agence afin de déterminer quels sont les comptes dont le solde est négatif. Lorsqu'un compte débiteur est rencontré, le script génère une ligne du tableau HTML avec le nom du client du compte, le numéro du compte ainsi que le solde courant. Les boucles *for* permettent d'itérer sur des séquences IDL. L'extraction des informations se fait simplement en invoquant les opérations des objets parcourus (i.e. *liste\_des\_clients()* et *liste\_des\_comptes()*)

ou en consultant les attributs (*client.nom*, *compte.nom* et *compte.solde*). Comme la version d'Orbix 2.0 que nous utilisons n'intègre pas de service de transactions, notre script peut générer des comptes qui seraient en train d'être crédités ou bien détruits. Mais cela est plus lié à l'implantation non transactionnelle de notre service bancaire qu'au script lui-même.

Finalement, le patron se termine par quelques tags HTML indiquant la fin du tableau, du corps et du document. La figure 4.12 montre une capture d'écran réalisée suite à l'accès à ce document dynamique.

The screenshot shows a Netscape browser window titled "Netscape: Liste des comptes débiteurs de l'agence de LILLE". The address bar contains "http://duff:8080/cw/agence\_comptes\_debiteurs.html". The main content area displays a table with the following data:

Nom du client	Numéro Compte	Solde débiteur
Gransart Christophe	C_1	-1200
Merle Philippe	C_3	-500
Montagnon Valérie	C_4	-1000

FIG. 4.12 - La visualisation de la liste des comptes débiteurs d'une agence

Ainsi, le document visualisé par les utilisateurs est toujours à jour par rapport à l'état du système. La simplicité d'écriture des patrons permet de rapidement répondre à de nouveaux besoins : en dupliquant ce patron et en lui apportant de légères modifications, un utilisateur peut en quelques instants créer un nouveau tableau de bord pour voir les comptes créditeurs, la liste des comptes de clients masculins ou habitant telle ville.

Nous constatons que pour concevoir ces tableaux de bord, il n'est pas nécessaire de modifier les objets Corba ou de savoir programmer avec Corba. Il suffit de connaître le langage HTML, le langage CorbaScript et les spécifications des services à visualiser. La conception d'un nouveau patron peut se faire interactivement grâce à un éditeur de texte et un navigateur. Nous considérons que cela est à la portée des utilisateurs.

L'avantage majeur de cette approche est donc de séparer clairement la conception des objets fonctionnels d'un service de leur représentation dans le WWW. Mais l'inconvénient est que cette approche est dédiée uniquement à la consultation des objets, l'utilisateur ne peut pas agir sur ces objets. Pour la partie suivante, nous allons illustrer une autre possibilité de CorbaWeb pour réaliser des scripts d'accès et d'invocations des objets.

#### 4.3.2.2 Des scripts d'accès à des objets Corba

Nous allons maintenant présenter la manière d'appliquer des traitements sur les objets Corba. La figure 4.13 représente un document HTML fournissant une interface graphique sur l'objet *grille* que nous avons présenté au chapitre 3. Ce document est référencé dans le

WWW par l'URL suivante *http://duff:8080/cw/grille.cs*. A la vue de cette capture d'écran, rien n'indique aux utilisateurs que ce document est dynamique et fournit une passerelle d'accès sur un objet Corba, illustrant ainsi la totale transparence qu'offre CorbaWeb.

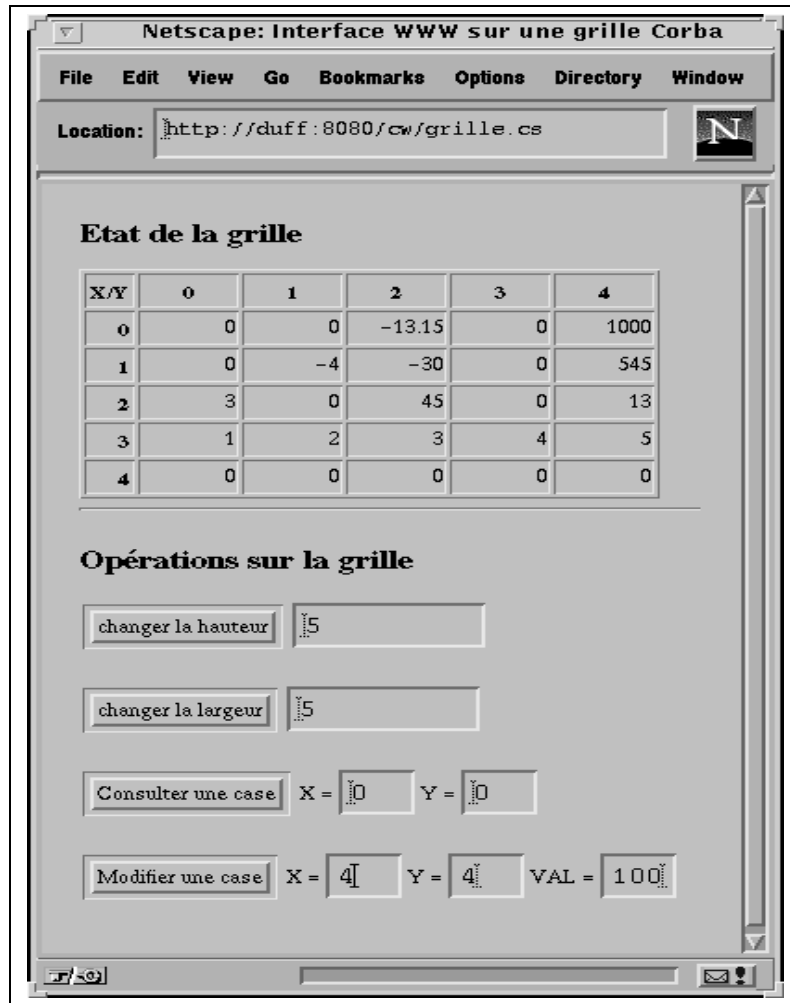


FIG. 4.13 - L'accès à un objet grille à travers le WWW

La partie haute du document représente sous la forme d'un tableau HTML l'état de la grille. L'instance représentée ici a une largeur et une hauteur égales à 5. Chaque case du tableau HTML contient la valeur correspondante dans la grille à la génération de ce document. Si la grille change d'état, le document n'est pas automatiquement réactualisé. Mais il suffit de charger le document pour voir les dernières modifications de la grille.

La partie basse du document contient quant à elle des formulaires HTML pour modifier la grille. Les deux premiers formulaires permettent de modifier respectivement la hauteur et la largeur de la grille. Ils sont composés d'un bouton pour soumettre le traitement (i.e. boutons *changer la hauteur* et *changer la largeur*) et un champ texte à saisir pour indiquer la nouvelle hauteur ou largeur. Chaque champ contient déjà la valeur courante associée à cette information. Les deux formulaires suivants permettent respectivement de consulter et de modifier l'état d'une case de la grille (i.e. boutons *Consulter une case* et *Modifier une case*). Les coordonnées et la nouvelle valeur sont saisies par l'intermédiaire de champs de texte.

Par l'intermédiaire des formulaires, l'utilisateur peut appliquer des traitements sur cette grille : en remplissant le dernier formulaire, la case de coordonnées (4,4) prendra la

nouvelle valeur 100. Lorsque l'utilisateur sélectionne le bouton de soumission, le navigateur WWW lui affiche le prochain document représenté par la figure 4.14. Ce document est généré dynamiquement suite à la soumission du formulaire. L'action du formulaire référence l'URL suivante `http://duff:8080/cw/grille_actions?modifier`. Ce document contient une ancre pour retourner sur la page d'interface de la grille. Si l'utilisateur sélectionne cette ancre plutôt que d'utiliser le bouton «arrière» de son navigateur alors il obtiendra un nouveau document présentant le nouvel état de la grille.



FIG. 4.14 - L'invocation d'une opération sur une grille

Ces deux illustrations précédentes montrent que nous pouvons développer des ressources WWW fournissant l'accès à un service orienté objet réparti. Bien sûr, le service fourni par l'objet grille est assez primitif mais il permet d'illustrer concrètement et simplement ce que nous pouvons réaliser avec l'environnement CorbaWeb. Cet exemple fait apparaître l'existence de deux ressources WWW sur le serveur `http://duff:8080`. Ces deux ressources sont en fait deux scripts CorbaWeb : `/cw/grille.cs` et `/cw/grille_actions.cs`.

Le premier script (cf. figure 4.15) se connecte à un objet grille dont la référence CorbaScript est `grille("UneGrille:ServeurDeGrilles")`<sup>17</sup>. Cependant, cette référence aurait très bien pu être obtenue via le service de nommage comme nous l'avons illustré pour le document dynamique sur l'agence. Ensuite, il consulte la *largeur* et la *hauteur* de cette grille.

Le script doit générer explicitement les entêtes MIME et HTML. Ces entêtes ne sont pas générés automatiquement car nous pouvons aussi utiliser CorbaWeb pour générer d'autres types de documents multimédia tels que des images ou des sons. Le script génère les tags de structuration du document : le titre, la première section et la déclaration du tableau. Le tableau généré est constitué de *largeur+1* colonnes et de *hauteur+1* lignes. Chaque ligne est balisée par les tags `<TR>` et `</TR>` tandis que les valeurs des cases sont entourées par les tags `<TD>` et `</TD>` (le tag `<TH>` permet de mettre une case en gras).

La première ligne du tableau est générée par la première boucle illustrant l'itération en CorbaScript sur un intervalle de valeurs entières. Les deux boucles imbriquées suivantes permettent de générer chacune des cases du tableau en consultant les cases correspondantes de la grille (i.e. `refGrille.consulter(x,y)`). Cette implantation n'est pas performante car elle réalise *largeur\*hauteur* invocations sur la grille à travers le bus Corba. Pour l'optimiser, il suffirait de modifier l'interface IDL grille en lui ajoutant une opération renvoyant l'état

17. Si une référence ne précise pas le nom de la machine alors elle désigne un objet sur la machine où s'exécute l'interpréteur.

global de la grille (*sequence* < *sequence* <*float*> > *etat\_global* ());) et d'implanter cette opération. Mais alors, nous ne ferions plus de l'intégration d'objets dans le WWW mais de la conception de services spécifiquement pensés pour le WWW<sup>18</sup>.

```

refGrille = grille ("UneGrille:ServeurDeGrilles")
largeur = refGrille.largeur
hauteur = refGrille.hauteur
import HTML
HTML.mime_header ()
HTML.TITLE ("Interface WWW sur une grille Corba")
HTML.H (2, "Etat de la grille")
print "<TABLE ALIGN=ABSCEENTER BORDER=1 CELLSPACING=2 CELLPADDING=2>\n"
print "<TR> <TH>X/Y"
for x in range (0,largeur-1) {
    print "<TH>", x
}
print "</TR>"
for y in range (0,hauteur-1) {
    print "<TR ALIGN=RIGHT> <TH>", y
    for x in range (0,largeur-1) {
        print "<TD WIDTH=50>", refGrille.consulter(x,y), "</TD>"
    }
    print "</TR>"
}
print "</TABLE>"
HTML.HR ()
HTML.H (2, "Opérations sur la grille")
HTML.form_post ("/cw/grille_actions.cs", "changer_hauteur")
HTML.input_submit ("changer la hauteur")
HTML.input_text ("nouvelleHauteur", hauteur, 10)
HTML.end_form ()
# code similaire pour le formulaire "changer la largeur"
# et formulaire "Consulter une case" similiaire au code suivant
HTML.form_post ("/cw/grille_actions.cs", "modifier")
HTML.input_submit ("Modifier une case")
print "X = "
HTML.input_text ("posX", 0, 3)
print "Y = "
HTML.input_text ("posY", 0, 3)
print "VAL = "
HTML.input_text ("nouvelleValeur", 0, 5)
HTML.end_form ()

```

FIG. 4.15 - Le script d'interfaçage d'une grille dans le WWW

La fin du script génère la partie du document contenant les formulaires. Dans le script de la figure 4.15, nous n'avons mis que le code de génération de deux de ces formulaires : celui pour *changer la hauteur* et celui pour *Modifier la valeur*. Le module HTML nous fournit des procédures pour automatiser la génération des formulaires (i.e. *form\_post()*, *end\_form* et *input\_..()*). L'action de chaque formulaire référence le script CorbaWeb */cw/grille\_actions* avec comme paramètre une valeur de *QUERY\_STRING* différente : *changer\_hauteur*, *changer\_largeur*, *consulter* et *modifier*. Ce paramètre permettra à ce script de différencier le traitement à appliquer sur la grille. Chaque formulaire est composé d'un bouton pour soumettre les données saisies par l'utilisateur dans les zones textes (i.e. *input\_text*). Chacune de ces zones a un nom différent (*nouvelleHauteur*, *nouvelleLargeur*, *posX*, *posY* et *nouvelleValeur*). Lorsque le contenu saisi du formulaire est envoyé au script d'actions, celui-ci pourra accéder au contenu de chaque champ du formulaire par l'intermédiaire de variables CorbaScript automatiquement créées. Ces variables ont pour nom les noms des champs correspondants.

18. De toute façon, l'environnement CorbaWeb n'interdit aucune de ces deux approches, au contraire, il permet de réutiliser des composants préexistants et il favorise l'accès à de nouveaux services optimisés.

Le second script *cw/grille\_actions* (cf. figure 4.16) est plus simple car il a moins de textes HTML à générer. Il se connecte à la grille et selon la valeur de *QUERY\_STRING*, il applique le traitement adéquat : changer la hauteur, changer la largeur, consulter ou modifier une case. Pour chaque traitement, le script consulte les variables provenant des formulaires et affiche un message descriptif du traitement réalisé. Actuellement, les variables du formulaire ne contiennent que des chaînes de caractères et il est donc nécessaire de les convertir via l'opérateur CorbaScript d'évaluation de chaîne (i.e. *eval*). Dans l'avenir, nous prévoyons d'automatiser ces conversions lors de la création de ces variables par l'interpréteur CorbaWeb. La fin du script génère une ancre pour retourner sur le script précédent.

```

refGrille = grille ("UneGrille:ServeurDeGrilles")
import HTML
HTML.mime_header ()
if ( QUERY_STRING == "changer_hauteur" ) {
    refGrille.hauteur = eval(nouvelleHauteur)
    print "La nouvelle hauteur de la grille est ", nouvelleHauteur
} else if ( QUERY_STRING == "changer_largeur" ) {
    refGrille.largeur = eval(nouvelleLargeur)
    print "La nouvelle largeur de la grille est ", nouvelleLargeur
} else if ( QUERY_STRING == "consulter" ) {
    v = refGrille.consulter ( eval(posX), eval(posY))
    print "La case (", posX, ', ', posY, ") a pour valeur ", v
} else if ( QUERY_STRING == "modifier" ) {
    refGrille.modifier ( eval(posX), eval(posY), eval(nouvelleValeur))
    print "La case (", posX, ', ', posY, ") a pris la nouvelle valeur "
        , nouvelleValeur
}
HTML.P()
HTML.a_href ("/cw/grille.cs","Retour à la page précédente")

```

FIG. 4.16 - *Le script d'invocation des opérations sur une grille*

#### 4.3.2.3 Les limites de cette approche

Les deux modes que nous venons de présenter permettent d'intégrer rapidement et simplement n'importe quel objet Corba dans le WWW. Cependant, l'inconvénient de ces deux modes est qu'il est nécessaire de développer des scripts spécifiques pour chaque objet. Si un système d'informations est composé d'une grande quantité d'objets alors cela implique le développement d'une multitude de scripts. Il est tout de même possible d'écrire un seul script par type IDL d'objets. Ces scripts recevront en paramètre la référence de l'objet sur lequel ils doivent opérer, plutôt que de se connecter à une instance spécifique.

Mais, si l'on veut interfacier des milliers de types IDL d'objets alors il faudra développer des milliers de scripts CorbaWeb pour visualiser les objets et leur appliquer des traitements. Pour lutter contre cette inflation de scripts, nous allons introduire, dans la section suivante, les méta scripts: le troisième et dernier mode d'utilisation de CorbaWeb.

#### 4.3.3 Des méta scripts

Les méta scripts sont des scripts CorbaWeb qui reçoivent en paramètres des références d'objet, des types ou des scripts. Ensuite, ils appliquent des méta traitements sur ces données. Dans cette section, nous illustrons quatre méta scripts différents mais nous pouvons en imaginer de nombreux autres.

Le méta script d'**interfaces** génère automatiquement une interface graphique HTML pour consulter et appliquer des traitements sur un objet. Celui d'**exécution d'opérations**

peut exécuter n'importe quel traitement unitaire sur un objet Corba (une invocation, une consultation ou une modification). Le méta script d'**exécution de scripts** permet à l'utilisateur d'envoyer ses propres scripts s'exécuter sur les objets Corba. Finalement, nous présentons le méta script de **représentation pour la navigation** dans les graphes d'objets.

#### 4.3.3.1 Le méta script d'interface

Lorsque nous avons de nombreux types et objets Corba à interfacier dans le WWW, il est nécessaire d'écrire une multitude de scripts. L'objectif du méta script de génération d'interfaces est d'éviter cette inflation. Grâce à un unique script, nous pouvons interfacier n'importe quel objet dans le WWW.

Ce méta script reçoit en paramètre une référence d'objet CorbaScript. Puis, il génère automatiquement une interface graphique en HTML représentant l'interface fonctionnelle de cet objet. Pour cela, il va consulter le référentiel des interfaces et construit un formulaire pour chaque opération et attribut de l'interface IDL de l'objet passé en paramètre.

Une opération est représentée par un formulaire contenant un bouton pour appliquer le traitement et un ensemble d'éléments graphiques pour saisir les paramètres. L'action de ce formulaire référence l'URL du méta script d'exécution d'opérations que nous présentons dans la sous-section suivante. Selon le type de chaque paramètre, le méta script génère un élément graphique adapté.

Par défaut, les paramètres sont représentés par des champs textes à saisir c'est à dire pour les chaînes et les valeurs numériques. Un paramètre de type énumération est représenté par un menu déroulant permettant à l'utilisateur de choisir la valeur. Chaque entrée de ce menu est la représentation de la valeur correspondante dans l'énumération. Par exemple, l'utilisateur peut fixer un paramètre booléen à l'aide d'un menu contenant les deux entrées suivantes *false* et *true*.

Un paramètre d'un type IDL structuré est représenté par une succession d'éléments graphiques pour chacun des champs de la structure. Ce processus est lui-même récursif : si une structure a un champ de type structuré alors ce champ est décomposé en un ensemble d'éléments graphiques. Pour les types tableaux et séquences, l'utilisateur saisit les éléments dans un champ texte en les séparant par des virgules.

Les références d'objets sont actuellement saisies manuellement par l'utilisateur dans un champ texte. Il utilise pour cela les règles de syntaxe du langage CorbaScript (i.e. *TYPE("nom:serveur:machine")*). Dans l'avenir, nous remplacerons ces champs par des applets Java pour supporter des modes d'interactions plus conviviaux comme le copier/coller et le glisser/coller d'une référence.

Un attribut est accessible par un ou deux formulaires selon sa définition en consultation seule (i.e. *readonly* d'IDL) ou en consultation/modification (i.e. mode IDL par défaut). Le formulaire pour la consultation n'est composé que d'un unique bouton pour obtenir la valeur de l'attribut. L'action de ce formulaire référence le méta script d'exécution d'opérations. Le formulaire de modification permet de saisir la nouvelle valeur de l'attribut. La génération de ce paramètre suit les mêmes règles que pour les paramètres d'une opération : champ texte par défaut, menu déroulant pour les énumérations, ensemble d'éléments graphiques pour les structures.

Lorsqu'une interface IDL hérite d'autres interfaces IDL, le méta script parcourt récursivement le graphe d'héritage pour représenter chacune des interfaces IDL suivant les règles précédentes.

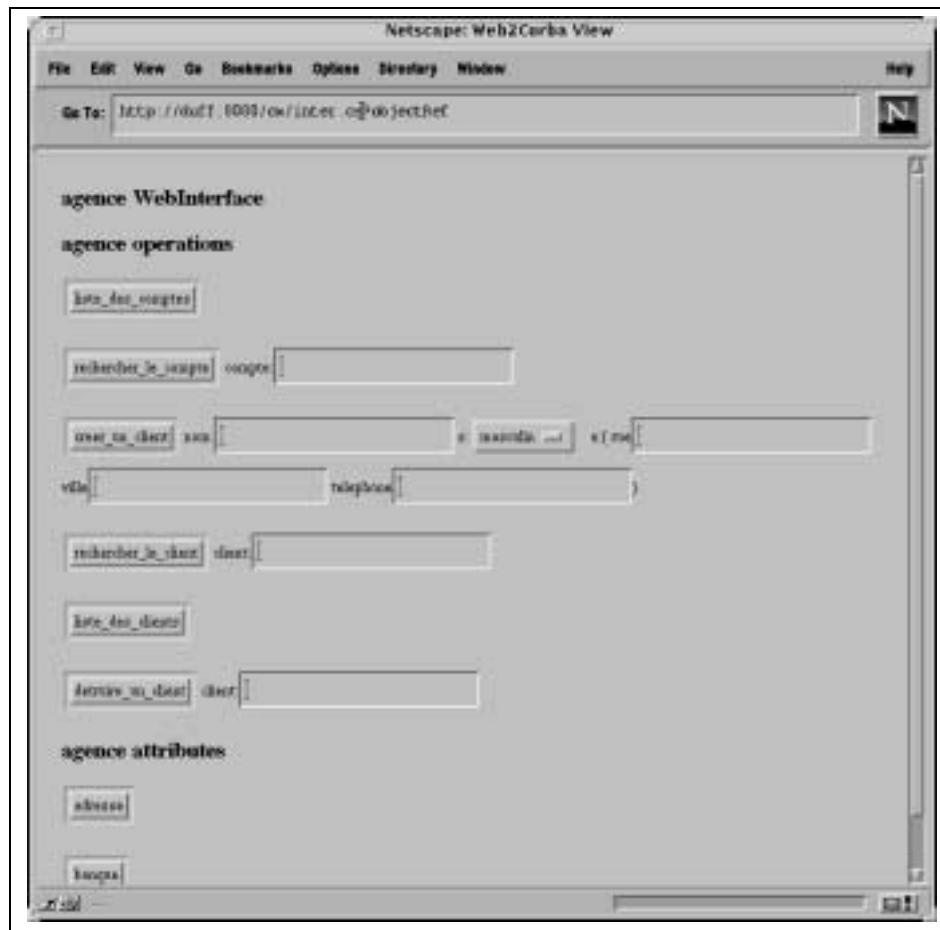


FIG. 4.17 - Une interface HTML sur une Agence

La figure 4.17 est une capture d'écran de l'exécution de ce méta script sur un objet de type Agence (voir l'interface IDL correspondante en section 2.3.3). L'utilisateur peut appliquer chacune des opérations par l'intermédiaire d'un formulaire (i.e. *liste\_des\_comptes*, *rechercher\_le\_compte*, *creer\_un\_client*, *rechercher\_le\_client*, *liste\_des\_clients* et *destruire\_le\_client*). Les attributs en consultation sont accessibles par les formulaires *adresse* et *banque*.

Le formulaire *creer\_un\_client* illustre la génération des divers éléments graphiques pour saisir les paramètres. Le premier paramètre *nom* de type chaîne est saisi via un champ texte. Le deuxième est du type énuméré *sexe*, il est donc matérialisé par un menu déroulant avec les choix *masculin* et *feminin*. Le dernier est décomposé en trois champs textes pour chacun des champs de la structure *adresse*.

La figure 4.18 contient le code du méta script de génération d'interfaces. Ce code est court car il inclut seulement la partie principale du méta script. Le code de génération des formulaires est encapsulé dans la procédure *generate\_interface* du module *CW*.

Ce code génère la structure classique d'un document HTML puis il traite deux cas de figure: la variable *QUERY\_STRING* contient ou non une chaîne.

Lorsque l'utilisateur invoque le méta script par l'URL */cw/inter.cs*, cette variable contient une chaîne vide. Le méta script génère une interface d'accueil composée d'un formulaire pour que l'utilisateur saisisse l'objet qu'il désire utiliser. L'entête du formulaire est généré par la procédure *form\_post*. Elle prend deux paramètres: l'URL de l'action associée au formulaire (i.e. ici, le nom du script courant *SCRIPT\_NAME*) et la valeur qui sera

affectée à la variable `QUERY_STRING` lors de l'exécution de ce script (i.e. «generate»). Le champ pour saisir la référence a le nom `objectRef`. La page d'accueil se termine par le «copyright» de notre environnement.

```
import CW, HTML
HTML.mime_header ()
HTML.begin ()
if ( QUERY_STRING == "" ) {
    HTML.TITLE ("Interface2HTML")
    HTML.form_post (SCRIPT_NAME, "generate")
    HTML.input_text ("objectRef", "", 60)
    HTML.input_submit ("Interface")
    HTML.input_reset ("Reset")
    HTML.end_form ()
} else {
    HTML.TITLE ( "Interface2HTML for " + objectRef )
    try {
        object = eval (objectRef)
        CW.generate_interface (object)
    } catchany (v) {
        print "ERROR : ", object, " isn't a Corba object\n"
    }
}
CW.copyright ("INTERFACE")
HTML.end ()
```

FIG. 4.18 - *Le méta script de génération automatique d'interfaces HTML*

Lorsque l'utilisateur soumet le formulaire, le méta script d'interfaces est exécuté car l'action du formulaire le référence. Les variables `QUERY_STRING` et `objectRef` contiennent respectivement la valeur «generate» et une chaîne contenant la référence saisie par l'utilisateur. Ainsi, le méta script exécute le second cas de figure. Il génère un titre HTML et évalue la variable `objectRef`. Ensuite, il applique la procédure `CW.generate_interface` sur la référence de l'objet. En cas de problème, cette procédure provoque une exception qui est signalée à l'utilisateur (i.e. bloc `try-catchany`).

Nous ne présentons pas le code de la procédure `generate_interface` car il fait plusieurs centaines de lignes CorbaScript. Il est relativement long car il doit, à partir du type IDL de l'objet passé en paramètre, explorer le référentiel des interfaces afin de construire les formulaires en suivant la politique de génération que nous avons exposée ci-dessus. Les formulaires des opérations et des attributs sont générés en consultant les métadonnées de l'IR : nom de l'opération/attribut et liste des paramètres. Cette procédure génère, en fonction du type IDL de chaque paramètre, l'élément graphique correspondant : champ, menu et ensemble de champs. Chaque formulaire référence le méta script d'exécution d'opérations.

La figure 4.19 montre l'interface HTML d'accueil de ce méta script. L'utilisateur saisit la référence de l'objet qu'il désire utiliser, dans notre exemple, la référence CorbaScript `bancaire::agence("agence:agenceLille")`. Ensuite, il sélectionne le bouton de soumission et en retour, il reçoit le document HTML généré automatiquement pour exploiter l'objet.

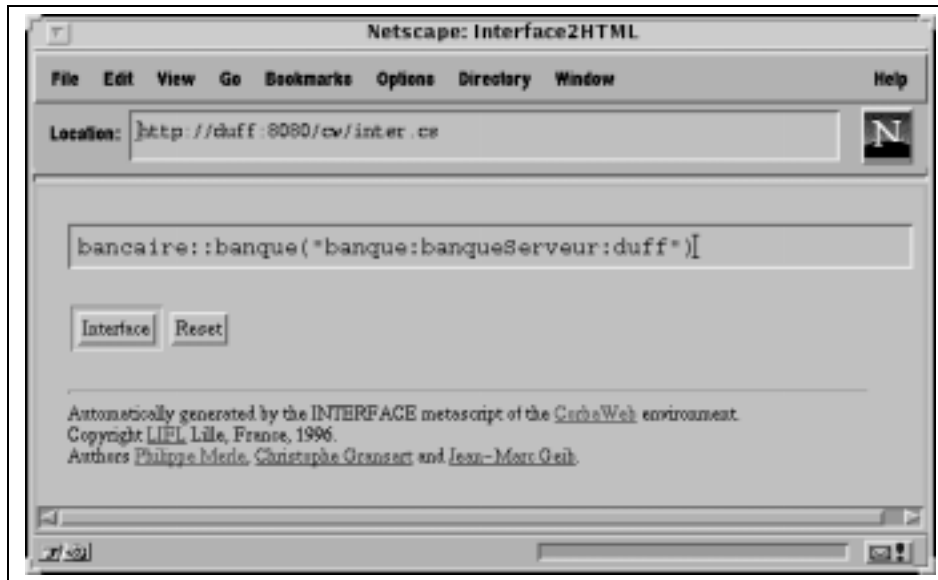


FIG. 4.19 - L'interface WWW d'accès à tout objet Corba

Ainsi, toutes les opérations et attributs définis dans l'interface IDL d'un objet sont directement et instantanément accessibles. Le méta script génère automatiquement tous les formulaires nécessaires. De ce fait, tout objet Corba est interfaçable dans le WWW sans nécessiter l'écriture d'une seule ligne de programme. Néanmoins, la génération automatique est toujours réalisée suivant les mêmes règles. Vu que nous avons écrit ce méta script et la procédure *generate\_interface* en CorbaScript, il est toujours possible de les adapter à un contexte spécifique. La personnalisation de cette génération peut par exemple changer l'ordre des formulaires, le parcours du graphe d'héritage IDL ou utiliser des applets pour des types spécifiques de paramètres.

#### 4.3.3.2 Le méta script d'exécution d'opération

L'action des formulaires générés par le méta script précédent référence le méta script d'exécution d'opération. Ce méta script permet d'exécuter un traitement sur un objet. Ce traitement peut être une invocation d'opération, une consultation ou une modification d'un attribut.

Ce méta script reçoit en paramètre un fragment de script contenant l'objet et le traitement à lui appliquer. Le méta script exécute ce fragment et génère en sortie un document contenant le résultat de cette exécution. L'impression des valeurs IDL suit les règles que nous avons définies dans la section 4.3.1.2 : parcours des structures et tableaux pour générer des listes HTML et la génération automatique d'ancres HTML pour les références d'objet. Finalement, le document contient une ancre pour retourner sur le document précédent.

Ce méta script n'est jamais directement invoqué par les utilisateurs. Il est désigné par l'URL d'action d'un formulaire. Il est exécuté lorsque l'utilisateur sélectionne le bouton de soumission d'un formulaire généré par le méta script d'interface. Le fragment est contenu dans la partie *QUERY\_STRING* de l'URL d'action. Il peut faire référence à des variables du formulaire pour obtenir les paramètres du traitement.

La figure 4.20 représente l'exécution de ce méta script sur l'invocation de l'opération *liste\_des\_agences* sur un objet Banque (i.e. la référence *bancaire::banque("banque:banqueServeur")*). Cette opération retourne une séquence d'éléments structurés comportant les champs *ville* et *agence*. Le résultat de l'invocation est automatiquement affiché en HTML :

une liste pour la séquence, une liste pour chaque élément structuré et une entrée pour chaque champ de la structure. Chaque entrée des listes est composée du nom et de la valeur du champ correspondant. Les références d'objet sont matérialisées par des ancres HTML. L'URL de ces ancres référence le méta script de représentation que nous présentons dans la section 4.3.3.4.

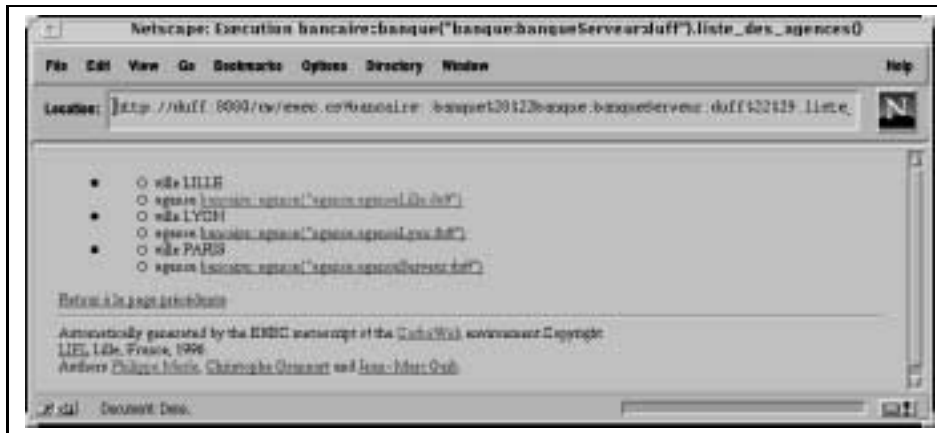


FIG. 4.20 - L'exécution d'une opération sur la Banque

```
import CW, HTML
HTML.mime_header ()
HTML.begin ()
HTML.TITLE ("Execution " + QUERY_STRING)
try {
  print eval (QUERY_STRING)
} catchany (e) {
  print "Exception : ", e
}
HTML.P ()
HTML.a_href (HTTP_REFERER,"Retour à la page précédente")
CW.copyright ("EXEC")
HTML.end ()
```

FIG. 4.21 - Le méta script d'exécution d'opérations

Le titre du document généré par ce méta script (cf. figure 4.21) indique le traitement qui va être appliqué sur l'objet. Puis, le script affiche le résultat de l'évaluation de la variable *QUERY\_STRING*. L'évaluation de cette variable va alors exécuter le fragment de code, qu'elle contient, et ainsi appliquer le traitement unitaire. Si ce traitement provoque une exception, elle est signalée à l'utilisateur. Finalement, une ancre est générée pour retourner sur le document d'où vient l'utilisateur (i.e. la variable *HTTP\_REFERER*). L'utilisation de cette ancre permet ainsi au navigateur de recharger le document précédent pour le mettre à jour. Si le document affiche l'état de l'objet, il est nécessaire de le régénérer car le traitement a pu modifier cet état.

Ce méta script complète donc le méta script de génération d'interfaces HTML. Grâce à ces deux méta scripts, les utilisateurs peuvent accéder à n'importe quel objet Corba et leur appliquer tous les traitements définis dans leur interface IDL.

#### 4.3.3.3 Le méta script d'exécution de script

Cependant, l'approche précédente ne permet que d'exécuter des traitements unitaires. Lorsque l'utilisateur veut réaliser une tâche complexe, il doit utiliser une succession de

formulaires pour chaque traitement unitaire. Ainsi, nous avons défini un nouveau méta script qui permet à l'utilisateur d'émettre un script CorbaScript pour réaliser ces tâches complexes.

Le méta script d'exécution de script offre une interface WWW pour exprimer des scripts complexes. L'utilisateur saisit son script dans un éditeur de texte contenu dans une page HTML. Ce script peut utiliser toute la puissance d'expression du langage CorbaScript. Nous considérons ce script comme un agent mobile : il est rédigé sur le site de l'utilisateur et il sera exécuté sur le site du serveur WWW. Le méta script reçoit le script utilisateur puis il l'exécute. En retour, l'utilisateur reçoit un document contenant les résultats de l'exécution de son script. En fait, ces résultats sont la conséquence de l'utilisation de l'instruction d'affichage (i.e. *print*).

Ce méta script 4.22 est construit sur le même moule que le méta script d'interface. Il génère une page d'accueil contenant un formulaire. Celui-ci est composé d'un éditeur de texte (i.e. *HTML.textarea("userScript")*) et des boutons pour exécuter le script et effacer le texte en cours. Lorsque l'utilisateur émet son script, celui-ci est affiché par le code *HTML.PRE(userScript)* puis exécuté par l'évaluation *eval(userScript)*.

```
import CW, HTML
HTML.mime_header ()
HTML.begin ()
HTML.TITLE ("Script Execution")
if ( QUERY_STRING == "" ) {
    print "Type your script:"
    HTML.P ()
    HTML.form_post (SCRIPT_NAME, "with_a_script")
        HTML.textarea ("userScript", 60, 15)
        HTML.input_submit ("Execute")
        HTML.input_reset ("Reset")
    HTML.end_form ()
} else {
    HTML.H (2, "Your script")
    HTML.PRE (userScript)
    HTML.HR ()
    HTML.H (2, "Result")
    print eval (userScript)
}
CW.copyright ("SCRIPT")
HTML.end ()
```

FIG. 4.22 - *Le méta script d'exécution de script*

Les figures 4.23 et 4.24 illustrent l'utilisation de ces scripts mobiles. Dans la figure 4.23, l'utilisateur a saisi un script calculant la somme totale déposée sur les comptes de chacun des clients de l'agence. Il obtient la référence de l'agence à travers le service de nommage. La première boucle itère sur la liste des clients de l'agence. La seconde parcourt les comptes de chaque client. Pour chacun des clients, le script imprime le solde total de ces comptes et génère une ancre pour accéder à ce client. La figure 4.24 contient une capture du document généré en retour par le méta script.



FIG. 4.23 - L'exécution d'un script à travers le WWW

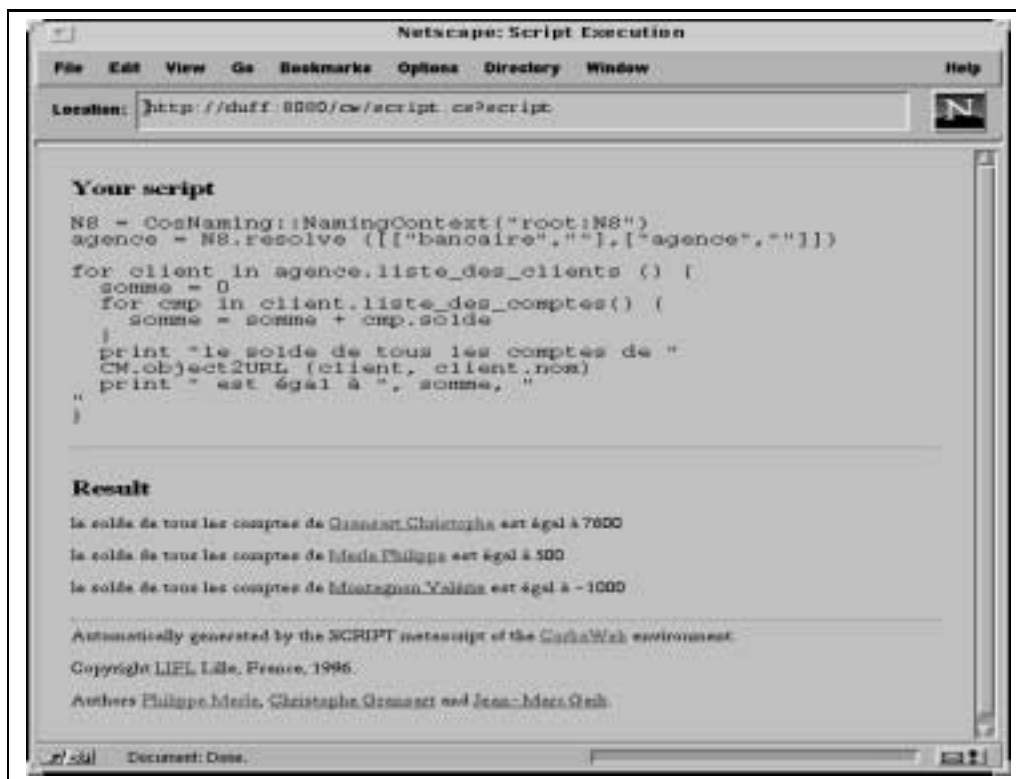


FIG. 4.24 - Le résultat de l'exécution d'un script

Ce méta script permet ainsi aux utilisateurs d'émettre des scripts afin de répondre à leurs besoins spécifiques : des recherches d'informations ou des traitements complexes pouvant modifier le système d'informations composé par les objets. A travers ce type de méta script, CorbaWeb devient aussi un environnement pour le support de scripts mobiles. Néanmoins, ce méta script pourrait être amélioré pour charger le script utilisateur depuis

son système de fichiers. Ainsi, les utilisateurs pourraient stocker les scripts qu'ils utilisent fréquemment plutôt que les saisir à chaque fois.

#### 4.3.3.4 Le méta script de représentation

Nous allons aborder le dernier méta script présenté dans ce mémoire. L'objectif de ce méta script est de permettre la navigation dans des graphes d'objets en fonction du profil des utilisateurs.

Comme nous l'avons vu, la génération automatique d'interfaces HTML est très intéressante car elle permet d'accéder à n'importe quel objet. Mais cet accès n'est pas personnalisable en fonction du type de l'objet. D'un autre côté, nous avons vu que nous pouvions, avec CorbaWeb, créer des scripts spécifiques à chaque type d'objets (l'exemple de la grille). Cependant, si nous voulons interfacier des milliers de types IDL d'objets alors il faut développer des milliers de scripts spécifiques. De plus, ces scripts doivent se référencer mutuellement pour permettre la navigation dans le graphe des relations que peuvent avoir ces objets. Lorsqu'un script doit générer une ancre sur une référence d'objet, ce script doit connaître le script qui interface le type IDL de l'objet. Prenons l'exemple suivant pour illustrer notre propos.

```
interface Employeur;
interface Employe {
    attribute string nom;
    attribute Employeur employeur;
};
// définition suivante nécessaire car IDL interdit les séquences anonymes
typedef sequence<Employe> DesEmployes;
interface Employeur {
    attribute DesEmployes les_employes;
};
```

Cette spécification IDL décrit deux types d'objets en relation. L'interface *Employe* possède un attribut qui permet de connaître l'*employeur* et un pour le *nom*. L'interface *Employeur* possède un attribut pour obtenir la liste des employés. L'intégration de ces objets dans le WWW peut être réalisée par un script pour chacun de ces types, par exemple, *employe.cs* et *employeur.cs*.

```
# fragment du script 'employe.cs'
# l_employe = la référence sur l'employé à représenter
print "Nom : ", l_employe.nom
HTML.object2URL (l_employe.employeur,"mon employeur", "/employeur.cs")

# fragment du script 'employeur.cs'
# l_employeur = la référence sur l'employeur à représenter
for e in l_employeur.liste_des_employes {
    HTML.object2URL (e, e.nom, "/employe.cs")
}
```

Les fragments de code ci-dessus illustrent une écriture possible de ces deux scripts de représentation. La procédure *HTML.object2URL* génère une ancre pour accéder à un objet. Elle prend trois paramètres : une référence, le texte de l'ancre et le script à exécuter. Lorsque le script pour les employés est exécuté, il génère une ancre représentant la relation *employeur* entre l'employé et son employeur. Cette ancre référence le script *employeur.cs*. Ainsi lorsque l'utilisateur visualise un employé, il peut suivre l'ancre pour naviguer vers l'objet employeur. De même, le script *employeur.cs* génère un ensemble d'ancres pour représenter la liste des employés d'un employeur. Chaque ancre référence le script de représentation *employe.cs*.

Ces fragments de scripts illustrent l'interdépendance qui existe entre les scripts de représentation. Ces scripts forment alors un graphe de dépendances. Dans un graphe d'objets plus complexe, cela impliquera que tout script doit potentiellement connaître tous les autres scripts du graphe.

Lorsqu'un nouveau type d'objets est ajouté, le développeur écrit un nouveau script de représentation. Mais il doit aussi modifier tous les scripts qui potentiellement peuvent générer une ancre vers des objets de ce nouveau type. Pour illustrer cela, créons un nouveau type d'objets :

```
// une nouvelle interface IDL
interface Cadre : Employe {
    attribute string responsabilite;
};

# fragment du script 'cadre.cs'
# le_cadre = la référence de l'objet Cadre à représenter
print "Nom : ", le_cadre.nom
HTML.object2URL (le_cadre.employeur, "mon employeur", "/employeur.cs")
print "Responsabilité : ", le_cadre
```

Nous avons donc créé un sous-type d'*Employe* appelé *Cadre* qui possède l'attribut supplémentaire *responsabilite*. Pour visualiser ce nouveau type, nous devons écrire le nouveau script spécifique *cadre.cs*. Mais alors le script *employeur.cs* doit être modifié pour permettre à un utilisateur de visualiser aussi les informations spécifiques à un cadre.

```
# fragment du script 'employeur.cs' modifié
# l_employeur = la référence sur l'employeur à représenter
for e in l_employeur.liste_des_employes {
    if ( e._type == Cadre ) {
        HTML.object2URL (e, e.nom, "/cadre.cs")
    } else {
        HTML.object2URL (e, e.nom, "/employe.cs")
    }
}
```

Le langage CorbaScript permet de manipuler les types car ils sont eux-mêmes des objets de la machine virtuelle. Le script *employeur.cs* décide donc à l'exécution quels scripts doivent être utilisés dans les ancres d'accès à ces employés. Imaginons maintenant que nous ajoutions à notre exemple de nombreux sous-types d'*Employe* et d'*Employeur* comme *Ouvrier*, *Industriel*, *Vendeur* et *Commerçant*. Alors il faut développer un script pour chacun de ces types mais il faut revoir tous les scripts déjà existants.

Grâce à cet exemple, nous venons de soulever le problème de la navigation dans un graphe de relations fortement polymorphes. Pour résoudre ce problème, la programmation objet nous inciterait à définir une classe racine possédant une méthode *afficheToi*. Cette classe racine serait héritée par tous les types que nous venons de définir. L'implantation de chacun de ces types spécialiserait le traitement de génération de la représentation. Cela donnerait les spécifications IDL et le fragment de script suivant :

```
// nouvelle interface
interface ObjetVisible {
    string afficheToi ();
};
interface Employe : ObjetVisible { ... };
interface Employeur : ObjetVisible ( ... );

# fragment du script 'voir.cs'
# l_objet = la référence sur l'objet à représenter (Employe, Employeur ...)
print l_objet.afficheToi ()
```

Avec cette solution, il n'y a plus qu'un seul script, c'est à dire *voir.cs*. Les objets implantent eux-mêmes leur politique de représentation. Lorsqu'ils veulent générer une ancre pour représenter une relation avec un autre objet, il leur suffit d'utiliser le nom unique de script */voir.cs*.

A priori, nous pouvons considérer que le problème est résolu. Mais, il y a un petit «hic» dans cette solution : nous avons modifié les interfaces IDL des objets Corba. Cela peut être acceptable pour les objets que nous définissons et implantons nous-mêmes. Néanmoins, si nous réutilisons un service déjà spécifié et implanté alors cette solution ne peut pas être appliquée. Prenons l'exemple d'une implantation du service de nommage. Nous disposons d'une description IDL du service et d'un exécutable binaire implantant le service. Nous pouvons toujours changer la description mais il nous est impossible de modifier l'exécutable.

Cependant, la notion de script unique élimine le problème du choix dynamique du script le mieux adapté à l'objet à représenter. Pour pouvoir appliquer cette approche, il suffit alors de pouvoir stocker les traitements de représentation à l'extérieur des objets. Nous avons donc retenu cette approche pour implanter un méta script de représentation des objets favorisant la navigation.

Le méta script de représentation permet donc la navigation dans des graphes d'objets Corba dont les relations sont fortement polymorphes. Il prend aussi en charge une nouvelle dimension par rapport à la démonstration précédente : la navigation dans les objets s'adapte au profil de l'utilisateur. Ce méta script intercepte toutes les demandes de représentation des objets et recherche le script spécifiquement adapté au type de l'objet accédé et au profil de l'utilisateur. Ces scripts spécifiques sont stockés dans un référentiel de scripts. Ce référentiel implante la politique de recherche dynamique du script adapté. Ainsi, chaque script est totalement indépendant des autres scripts. Nous pouvons introduire de nouveaux types d'objets qui ne nécessitent pas de modifier les scripts existants. De cette manière, ces scripts peuvent être développés par des individus différents.

La figure 4.25 présente le scénario de fonctionnement de ce méta script :

1. Le méta script reçoit en paramètre la référence de l'objet à représenter et le profil de l'utilisateur.
2. Il contacte le référentiel de représentation et lui demande le script le mieux adapté au type de l'objet et au profil de l'utilisateur.
3. Le référentiel applique une politique de recherche de scripts en fonction du contexte (type, profil) et retourne un script.
4. Le méta script évalue le script de représentation retourné par le référentiel.
5. Ce script retourné contient des instructions CorbaScript qui vont invoquer l'objet Corba et générer une représentation HTML. Cette représentation peut contenir des formulaires pour appliquer des traitements sur l'objet.
6. Le méta script retourne finalement le document HTML au navigateur.

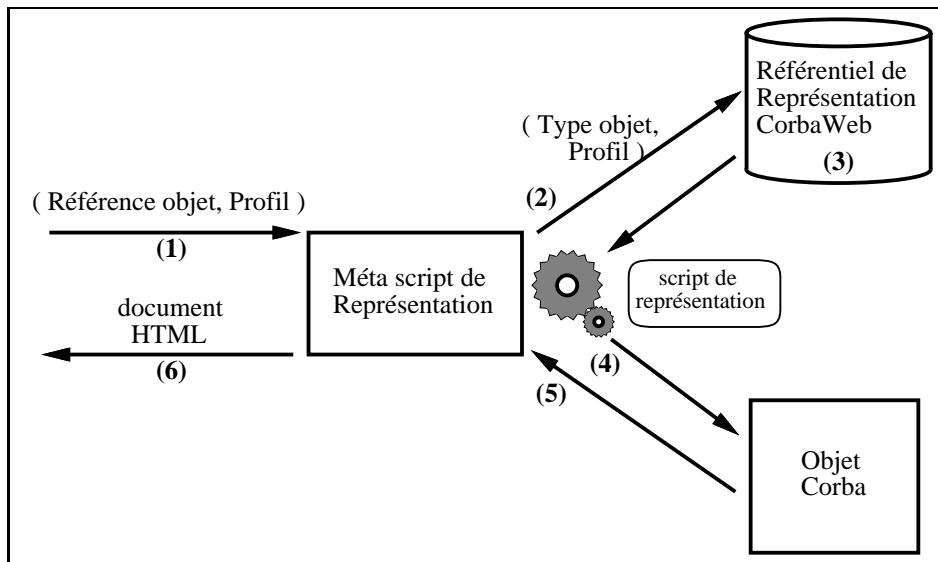


FIG. 4.25 - Le fonctionnement du méta script de représentation

Le scénario du fonctionnement du méta script de représentation laisse une grande liberté de choix pour implanter les éléments suivants :

- **La notion de profil** : l'utilisateur est caractérisé par un profil. Un profil peut être partagé par plusieurs utilisateurs. Il détermine un rôle vis-à-vis des objets du système d'informations comme par exemple un directeur de banque, un conseiller financier ou un client. Ce profil permet d'avoir une représentation adaptée des objets.

Pour l'instant, nous n'avons pas défini de modèle structurant ces profils. Nous envisageons de définir un graphe des profils permettant de les classifier. Ce futur modèle pourrait aussi servir pour le contrôle des accès aux objets.

Cette notion de profil peut être implantée en utilisant les mécanismes d'authentification fournis par le protocole HTTP. Lorsque l'utilisateur se connecte au serveur WWW, celui-ci lui demande d'entrer un nom d'utilisateur et un mot de passe. Cette identification est transmise dans les requêtes suivantes. Le serveur vérifie à chaque fois si l'utilisateur est bien enregistré. Mais d'autres solutions technologiques devront être évaluées pour diminuer la fréquence de ces vérifications.

- **Le référentiel de représentation** : le référentiel de représentation regroupe l'ensemble des scripts pour chaque couple type-profil possible. Dans l'implantation du méta script que nous allons présenter, le référentiel est simplement un répertoire contenant des fichiers de scripts associés à chaque couple type-profil.

Nous sommes en train d'expérimenter une autre solution : implanter le référentiel via un ou des objets Corba. Cela permet alors de répartir le référentiel, de l'administrer et de le mettre à jour via l'environnement CorbaWeb. Ainsi, le référentiel peut être partagé par plusieurs passerelles CorbaWeb.

- **La politique de recherche** : la politique de recherche doit retrouver le script le mieux adapté au type de l'objet et au profil de l'utilisateur. Si un tel script existe précisément pour ce couple alors la politique est simple car elle se contente de retourner ce script. Par contre, si ce script n'existe pas, elle doit rechercher un script un peu moins adapté par exemple pour un sur-type ou un sur-profil.

Nous nous retrouvons ici en face d'un problème similaire à celui de la recherche d'une méthode dans un graphe de classes à héritage multiple. Dans le domaine des langages de programmation orientés objet, ce problème n'a pas trouvé de solution unique : recherche systématique en profondeur d'abord, résolution explicite par l'utilisateur de la méthode comme en C++ ou résolution explicite par le concepteur de la classe comme en Eiffel.

Dans notre contexte, la difficulté est que cette politique doit parcourir deux graphes simultanément, celui des types et celui des profils. Conjointement à la définition d'un modèle des profils et à la répartition du référentiel, nous expérimentons les diverses stratégies existantes dans les langages orientés objet.

```

import CW, HTML
HTML.mime_header ()
HTML.begin ()
if ( QUERY_STRING == "" ) {
    HTML.TITLE ("View")
    HTML.form_post (SCRIPT_NAME, "objectRef")
    HTML.input_text ("objectRef", "", 60)
    HTML.P()
    HTML.input_submit ("View")
    HTML.input_reset ("Reset")
    HTML.end_form ()
} else {
    HTML.TITLE ("View " + objectRef)
    try {
        object = eval (objectRef)
        try {
            typeName = object._type._toString()
            exec ("ViewRepository/" + typeName + '-' + REMOTE_USER)
        } catch ( FileNotFoundException ) {
            CW.generate_interface (object)
        }
    } catchany (EXCEPTION) {
        print "ERROR : ", object, " isn't a Corba object"
        HTML.P ()
        print "Exception : ", EXCEPTION
        HTML.P ()
    }
}
CW.copyright ("VIEW")
HTML.end ()

```

FIG. 4.26 - *Le méta script de représentation des objets Corba*

Le code 4.26 est une version simplifiée du méta script de représentation. Si ce méta script est invoqué directement, il génère un formulaire pour saisir la référence de l'objet à visiter. Dans l'autre cas, il recherche le script de représentation et l'exécute.

La notion de profil est implantée par la variable *REMOTE\_USER* qui contient le nom d'identification de l'utilisateur distant. La valeur de cette variable est fixée à partir des informations extraites de la requête HTTP. Les scripts spécifiques de représentation sont stockés dans des fichiers contenus dans un répertoire (i.e. *ViewRepository*). Ces fichiers sont désignés par la chaîne *typeName-REMOTE\_USER*.

Le chargement et l'exécution sont réalisés par l'instruction *exec*. Si le fichier n'est pas trouvé alors le méta script utilise la procédure *CW.generate\_interface*. C'est la même procédure qui est utilisée dans le méta script d'interfaces pour générer automatiquement les formulaires HTML d'accès à un objet.

La variable *object* contient la référence de l'objet à représenter. Cette variable est utilisée dans les scripts de représentation pour connaître l'objet courant. Si une exception est provoquée dans le script de représentation, elle est signalée à l'utilisateur.

La figure 4.27 est une capture d'écran de la page d'accueil de ce méta script. L'utilisateur peut alors saisir la référence de l'objet qu'il désire visiter. Dans cet exemple, l'utilisateur a saisi la référence du service de nommage.

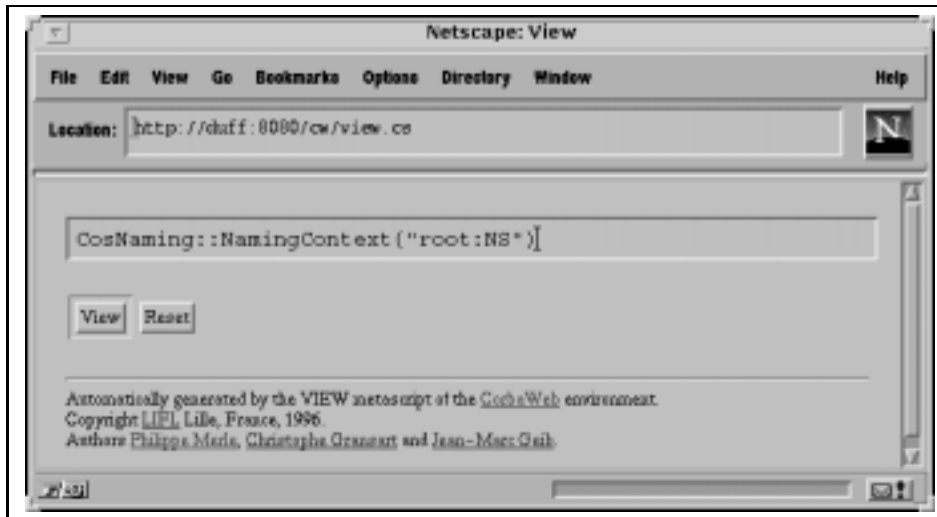


FIG. 4.27 - *Le méta script de représentation*

#### 4.3.4 L'illustration de la navigation dans le service bancaire

Nous allons illustrer étape par étape la navigation à travers les objets composant le service Bancaire. Cette navigation traverse les contextes de nommage à la recherche de l'objet banque. Une fois cet objet trouvé, nous visitons une instance d'agence, un objet client pour finalement atteindre un compte bancaire.

Pour chaque type d'objets, nous présentons le script spécifique de représentation ainsi qu'une illustration graphique de son exécution. Afin de ne pas surcharger cette présentation, nous n'utilisons qu'un seul profil au cours de la navigation.

##### 4.3.4.1 La représentation d'un contexte de nommage

Après avoir saisi la référence du contexte de nommage (cf. figure 4.27), l'utilisateur obtient un document énumérant le contenu de ce contexte (cf. figure 4.28). Chaque entrée de la liste représente une association contenue dans le contexte. Cette association est matérialisée par une ancre HTML dont le libellé est le nom de l'association. En sélectionnant cette ancre, l'utilisateur va visiter l'objet associé. Deux types d'entrées sont possibles : celle référant un objet et celle référant un contexte de nommage lié. Cette différence est représentée graphiquement par une icône représentant respectivement un document et un répertoire. Dans la figure 4.28, le contexte de nommage ne contient qu'une seule entrée : un lien nommé «*bancaire*» sur un autre contexte de nommage (l'icône permet de s'en rendre compte).



FIG. 4.28 - Le contenu d'un contexte de nommage Corba

L'utilisateur peut naviguer dans les objets en suivant les hyperliens comme il le fait dans le «classique» WWW. A la sélection de l'entrée «*bancaire*», le navigateur présente à l'utilisateur le contenu de cet autre contexte (cf. figure 4.29). Deux objets y sont référencés par les noms «*banque*» et «*agence*». La nature des icônes permet de faire la distinction entre les associations du type contexte de celles du type objet.

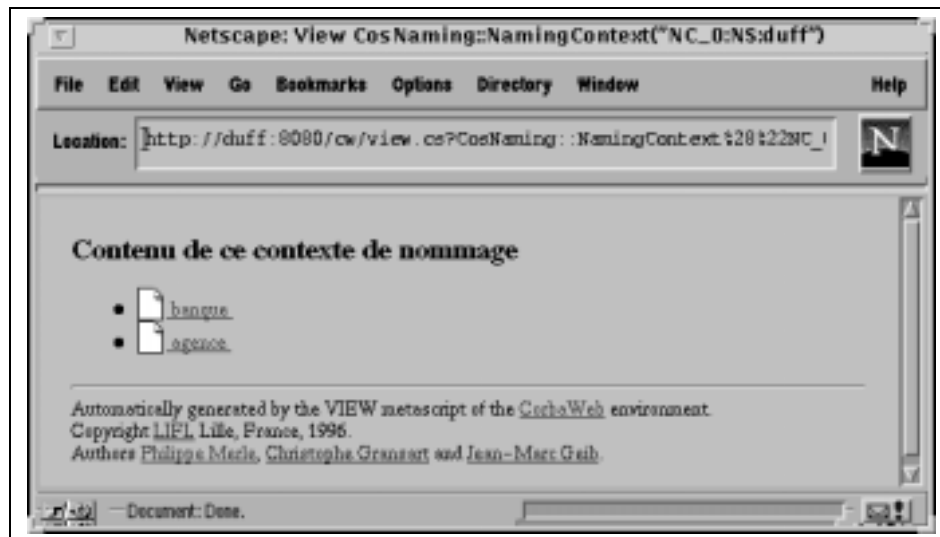


FIG. 4.29 - La navigation dans un contexte de nommage Corba

Si ce contexte avait été visité par un administrateur du service de nommage, le document contiendrait en plus des formulaires pour appliquer les opérations sur ce contexte de nommage. A la place des formulaires, nous pourrions mettre une «ORBllet»<sup>19</sup> développée spécifiquement pour fournir une interface graphique sur un contexte.

19. Une ORBllet est une application Java qui utilise un ORB. Cet ORB peut lui-même être téléchargé avec l'application.

```

HTML.H (2, "Contenu de ce contexte de nommage")
proc image (URL) {
    return "<IMG ALIGN=absbottom BORDER=0 SRC=\"\" + URL + \">"
}
proc display_entry (b) {
    bn = b.binding_name[0]
    if ( b.binding_type == CosNaming::BindingType.ncontext ) {
        im = image ("internal-gopher-menu")
    } else {
        im = image ("internal-gopher-unknown")
    }
    ref = object.resolve ( [ [ bn.id, bn.kind ] ] )
    CW.object2URL ( ref, [ im, ' ', bn.id, ' ', bn.kind ] )
}
object.list (100, bl, bi)
HTML.iterate_list (bl,display_entry)

```

FIG. 4.30 - Le script de représentation d'un contexte de nommage Corba

Le code 4.30 est le script spécifique de représentation de tout objet du type *CosNaming::NamingContext*. Ce script se contente de générer la liste du contenu. Il invoque l'opération *list* sur le contexte courant (i.e. *object*). Cette opération prend deux paramètres en mode *out*. La variable *bl* contient au retour de l'opération l'information décrivant le contenu du contexte dans une séquence de structures IDL. Si le contexte contient plus de 100 associations, l'objet Corba référencé par la variable *bi* permet d'itérer sur les associations suivantes (cet objet est du type *CosNaming::BindingIterator*)<sup>20</sup>. Ensuite le script appelle la procédure *HTML.iterate\_list*. Cette procédure génère une liste HTML et exécute la procédure *display\_entry* sur chacun des éléments contenus dans *bl*. La procédure *display\_entry* sélectionne l'icône à afficher en fonction de la nature de l'association. Ces icônes sont des images contenues en standard dans les navigateurs WWW (i.e. *internal-gopher-menu* et *internal-gopher-unknown* sont des URNs). Les ancres sont finalement générées par la procédure *CW.object2URL*. Le troisième paramètre n'est pas indiqué car par défaut, il a la valeur */cw/view.cs* (i.e. l'URL du méta script de représentation).

#### 4.3.4.2 La représentation d'une banque

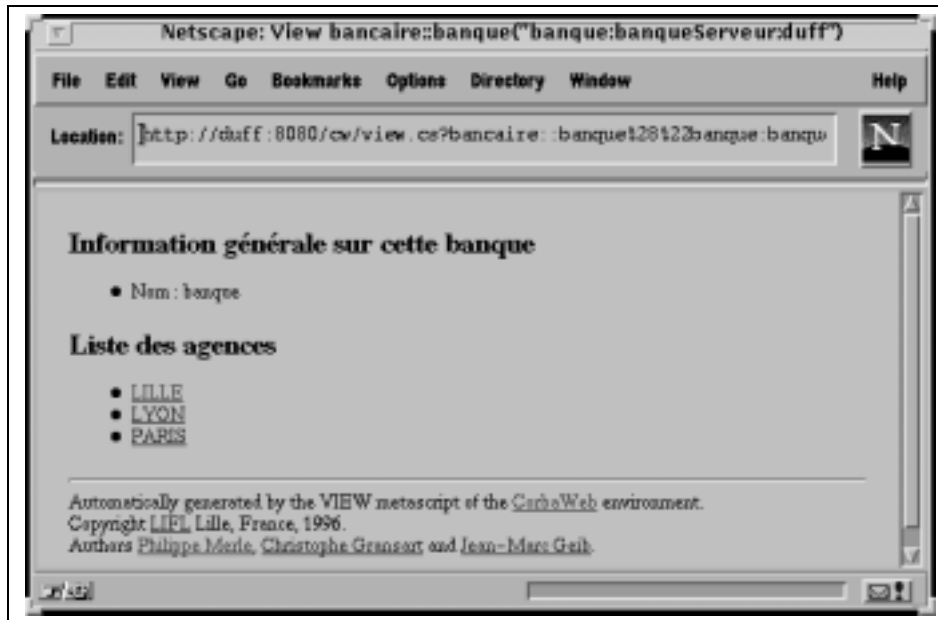
Un des bénéfices majeurs du méta script de représentation est qu'il permet de séparer conceptuellement les deux activités suivantes : la réalisation fonctionnelle d'un service et la conception du script d'intégration dans le WWW. Les concepteurs d'objets sont des spécialistes de Corba, ils savent concevoir des spécifications IDL et implanter des objets. D'un autre côté, les concepteurs de scripts ne sont pas forcément des spécialistes en programmation. Ils sont plutôt des spécialistes en Interfaces Homme-Machine. Ils doivent tout de même connaître le langage CorbaScript et les interfaces IDL<sup>21</sup>.

De plus, comme les scripts de représentation n'ont aucune connexion entre eux, leur développement peut être réalisé par plusieurs personnes, en même temps et sans nécessiter une coopération forte entre elles.

Accédons maintenant à l'objet «*banque*» contenu dans le contexte de nommage. Cet objet est alors représenté par un document contenant le nom de la banque et la liste de ces agences (cf. figure 4.31).

20. Dans cet exemple, nous n'affichons que les 100 premières entrées du contexte. Quelques instructions supplémentaires seraient nécessaires pour utiliser l'itérateur *bi*.

21. Cependant, ils peuvent découvrir ces interfaces IDL via la fonction d'aide en ligne de l'interpréteur CorbaScript.

FIG. 4.31 - *La navigation dans un objet Banque*

La figure 4.32 contient le script de représentation d'un objet de type banque. Il affiche le nom de la banque et la liste des agences. Cette liste est imprimée via la même technique : celle employée dans le script pour les contextes : itération sur une liste et impression des ancres vers le méta script de représentation.

```
HTML.H (2, "Information générale sur cette banque")
  print "<UL><LI> Nom : ", object.nom, "</UL>\n"

HTML.H (2, "Liste des agences de cette banque")
  proc display_item_agence (a) {
    CW.object2URL ( a.agence, a.ville )
  }
  HTML.iterate_list (object.liste_des_agences (), display_item_agence)
```

FIG. 4.32 - *Le script de représentation d'un objet Banque*

En général, l'utilisateur tient à avoir une représentation uniforme des informations permettant d'acquérir des habitudes de navigation. Cette uniformisation se traduit par une forte similitude dans les scripts de représentation. Ainsi, nous pouvons envisager de les générer automatiquement en fonction de règles d'ergonomie. Une autre possibilité serait d'avoir un atelier de conception visuel pour produire ces scripts. Des squelettes de scripts de représentation pourrait être générés en fonction des profils. Le rédacteur de scripts sélectionnerait les attributs de l'objet et les formulaires qu'il désire présenter à l'utilisateur final. L'atelier générerait automatiquement le script CorbaWeb approprié. L'utilisateur aurait toujours la possibilité d'écrire des fragments de scripts pour des opérations de consultation non triviales.

De même, les scripts pour des profils différents ont des éléments communs qui peuvent être stockés dans des modules de représentation. Dans notre exemple de service bancaire, tous les scripts de représentation d'une banque affiche la liste des agences. Mais pour le directeur de la banque, le script contient en plus des formulaires pour ajouter et retirer des agences.

### 4.3.4.3 La représentation d'une agence

La figure 4.33 est la représentation de l'agence de LILLE. Elle contient des informations générales : une ancre sur l'objet banque dont nous venons de parler, l'adresse et le téléphone. Les deux listes suivantes représentent les clients et les comptes de cette agence.



FIG. 4.33 - La navigation dans un objet Agence

```
HTML.H (2,"Information sur cette agence")
print "<UL>\n"
print "<LI> "
CW.object2URL (object.banque, "Ici pour aller au siège social")
adr = object.adresse
print "<LI> Ville : ", adr.ville
print "<LI> Rue : ", adr.rue
print "<LI> Téléphone : ", adr.telephone
print "</UL>\n"

HTML.H (2,"Liste des clients" )
proc display_item_client (client) {
  CW.object2URL (client, client.nom )
}
HTML.iterate_list (object.liste_des_clients(), display_item_client)

HTML.H (2,"Liste des comptes" )
proc display_item_compte (compte) {
  print compte._type._name, ' '
  CW.object2URL (compte, [compte.client.nom, ' ', compte.numero])
}
HTML.iterate_list (object.liste_des_comptes(), display_item_compte)
```

FIG. 4.34 - Le script de représentation d'un objet Agence

Pour afficher le texte des ancres vers les sous-objets, le script 4.34 consulte l'état de ceux-ci (i.e. *client.nom*, *compte.client.nom* et *compte.numero*). L'utilisation de l'expression *compte.\_type.\_name* permet d'avoir le nom du type dynamique d'un compte sous la forme d'une chaîne de caractères. Dans l'exemple, cela permet d'indiquer aux utilisateurs la nature des objets qu'ils vont rencontrer s'ils suivent une ancre (i.e. affichage du texte *compte* ou *livret* en fonction du type dynamique de l'objet).

#### 4.3.4.4 La représentation d'un client

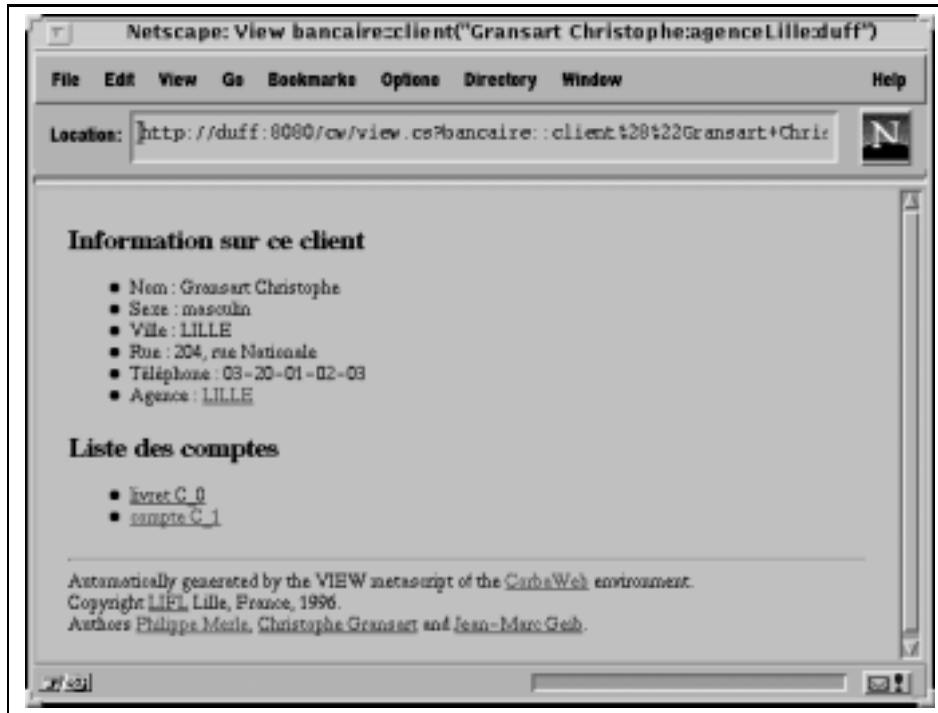


FIG. 4.35 - La navigation dans un objet Client

La représentation 4.35 et le script 4.36 montrent la poursuite de la navigation depuis l'agence précédente vers un objet client : des ancres sur l'agence et les comptes.

```
HTML.H (2, "Information sur ce client")
print "<UL>\n"
  print "<LI> Nom : ", object.nom
  print "<LI> Sexe : ", object.sexe
  adr = object.adresse
  print "<LI> Ville : ", adr.ville
  print "<LI> Rue : ", adr.rue
  print "<LI> Téléphone : ", adr.telephone
  print "<LI> Agence : "
  agence = object.agence
  CW.object2URL (agence, agence.adresse.ville )
print "</UL>\n"

HTML.H (2, "Liste des comptes")
proc display_item_compte (compte) {
  CW.object2URL (compte, compte._type._name + ' ' + compte.numero)
}
HTML.iterate_list (object.liste_des_comptes (), display_item_compte)
```

FIG. 4.36 - Le script de représentation d'un objet Client

#### 4.3.4.5 La représentation d'un compte

Cette dernière étape de la navigation dans notre service bancaire illustre la possibilité de mettre des formulaires pour opérer des traitements (cf. figure 4.37).



FIG. 4.37 - La navigation dans un objet Compte

```
HTML.H (2, "Information sur ce compte")
  print "<UL>\n"
  print "<LI> Agence : "
  CW.object2URL ( object.agence, object.agence.adresse.ville )
  print "<LI> Client : "
  CW.object2URL ( object.client, object.client.nom )
  print "<LI> Numéro : ", object.numero
  print "<LI> Solde : ", object.solde
  print "</UL>\n"

HTML.HR()
HTML.H (2, "Opérations sur le compte")
methode = object._toString() + ".credit(eval(montant))"
HTML.form_post (CW.META_SCRIPT_EXEC,string2URL(methode))
  HTML.input_submit ("crédit")
  HTML.input_text ("montant", 0, 5)
HTML.end_form ()

methode = object._toString() + ".debit(eval(montant))"
HTML.form_post (CW.META_SCRIPT_EXEC,string2URL(methode))
  HTML.input_submit ("débit")
  HTML.input_text ("montant", 0, 5)
HTML.end_form ()
```

FIG. 4.38 - Le script de représentation d'un objet Compte

Le script 4.38 illustre comment un concepteur de scripts peut intégrer des formulaires pour agir sur les objets. Actuellement, il définit le fragment de script associé à l'action du formulaire (i.e. la construction du script contenu dans la variable *methode*). Il réutilise le méta script d'exécution de traitement unitaire. Dans l'avenir, nous étendrons les modules HTML et CW pour offrir plus de souplesse et de simplicité dans l'intégration des formulaires pour les objets.

Une autre piste que nous étudions est la possibilité de définir un graphe des profils et un mécanisme d'héritage des scripts d'un sur-profil. Cet héritage pourrait permettre d'écrire un script pour le profil «directeur» en réutilisant le script pour le profil «visiteur». Ainsi, si le script visiteur est modifié, les scripts des profils plus spécialisés bénéficient instantanément de ces modifications.

### 4.3.5 Implantation et perspectives

Après ces nombreuses illustrations (écrans et scripts) des fonctionnalités de l'environnement CorbaWeb, nous discutons dans cette section de l'implantation de CorbaWeb et de quelques perspectives de travail. Dans la conclusion, nous synthétisons toutes les fonctionnalités offertes par CorbaWeb.

#### 4.3.5.1 L'adaptation de l'interpréteur CorbaScript

L'environnement CorbaWeb est principalement écrit avec le langage CorbaScript : modules HTML-CW, documents dynamiques, scripts d'accès, méta scripts et scripts de représentation. Néanmoins, nous avons dû implanter une version spécialisée de l'interpréteur CorbaScript pour prendre en charge les spécificités de CorbaWeb suivantes (cf. figure 4.39) :

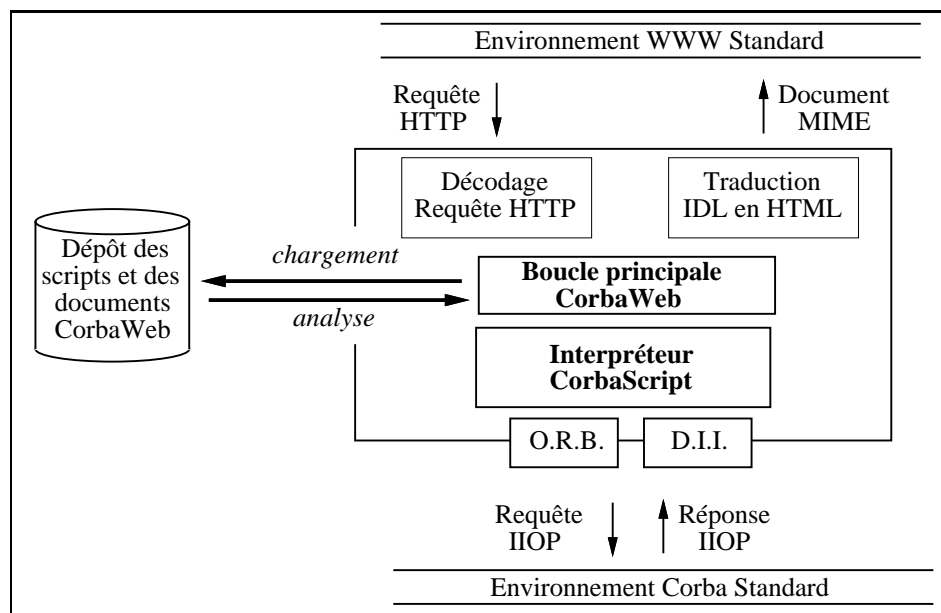


FIG. 4.39 - CorbaWeb : l'adaptation de CorbaScript

1. **Le décodage des requêtes HTTP** : lorsque l'interpréteur CorbaWeb reçoit une requête HTTP, il la transforme en un ensemble de variables CorbaScripts (*SCRIPT\_NAME*, *QUERY\_STRING* et *REMOTE\_USER*). Ces variables permettent alors aux scripts de disposer des informations sur les requêtes (voir section 4.3.1.3).

2. **La boucle principale de CorbaWeb** : lorsque la requête est décodée, l'interpréteur CorbaWeb charge le fichier document ou le script depuis un dépôt de fichiers (i.e. le répertoire courant d'exécution de l'interpréteur). Le nom du fichier est indiqué par la variable *SCRIPT\_NAME*. Si le fichier contient un document HTML, celui-ci est analysé : la partie HTML est renvoyée telle quelle, la partie script est exécutée par l'interpréteur CorbaScript. Dans le cas d'un script, il est directement exécuté.
3. **La traduction des valeurs IDL en HTML** : l'interpréteur de CorbaScript contient une classe spécifique (i.e. *CS\_CorbaIO*) pour faire les impressions des instructions *print*. CorbaWeb hérite de cette classe et la spécialise pour générer la correspondance entre des valeurs IDL et le langage HTML (voir section 4.3.1.2). Les références sont matérialisées par des ancres sur le méta script de représentation. Les types composés (structures, unions, séquences, tableaux) sont représentés par des listes HTML et leurs éléments sont récursivement parcourus.
4. **Le codage des scripts dans des URL** : comme nous transportons des références et des fragments de scripts dans les URL, il nous est nécessaire de recoder les caractères spéciaux; guillemets en *%22*, parenthèses en *%28* et *%29*, caractères d'espace en *+* et etc. Nous avons aussi ajouté la procédure *string2URL* écrite en C++. Elle prend une chaîne en paramètre et retourne une chaîne recodée dans le format URL. Elle doit être appelée explicitement dans les scripts lors de la génération manuelle d'ancres. Par exemple, *CW.object2URL()* l'utilise en interne pour coder la référence de l'objet passée en paramètre.

Ces ajouts spécifiques pour le contexte du WWW ont nécessité uniquement le développement de quelques centaines de lignes de code C++. Ce n'est rien par rapport aux dizaines de milliers de lignes de l'interpréteur CorbaScript (voir section 3.3). Dans la sous-section suivante, nous discutons de la réception des requêtes HTTP.

#### 4.3.5.2 Les différentes solutions de liaison avec le WWW

La connexion entre la passerelle CorbaWeb et un bus HTTP peut se faire de plusieurs manières. Nous présentons les diverses solutions que nous avons étudiées et celles que nous avons implantées. Il existe principalement quatre manières d'intégrer CorbaWeb sur le bus WWW :

- **Le protocole CGI** : cette solution est celle qui est la plus couramment utilisée pour étendre un serveur WWW. Nous utilisons un serveur WWW standard qui peut déclencher des exécutions de programmes CGI [McC95, Rob96]. Chaque requête HTTP vers un objet Corba va lancer l'exécution de la passerelle CorbaWeb. Cette première solution n'est pas très efficace car le serveur HTTP crée un nouveau processus et exécute notre passerelle à chaque demande d'exécution d'un script CorbaWeb. De plus, les mécanismes de cache des informations de l'IR et des modules implantés dans CorbaScript ne sont pas utilisés car le protocole CGI impose que le processus CorbaWeb se termine après le traitement de chaque requête.
- **Une API serveur** : dans cette solution, le code de CorbaWeb est inclus dans le serveur HTTP. Ce code est associé à un groupe d'URL gérées par le serveur. Chaque fois qu'une requête est adressée à l'une de ces URL, le serveur appelle une fonction de traitement de requête fournie par CorbaWeb (i.e. adaptateur API). L'avantage de cette solution par rapport à la précédente est qu'elle supprime le surcoût introduit

par la création d'un processus CorbaWeb à chaque requête. Le second avantage est que les fonctionnalités de cache de CorbaScript sont utilisées. Actuellement, il n'y a pas de standardisation de l'API de liaison entre le serveur HTTP et le service que l'on veut y inclure. Les différentes API disponibles ne sont pas compatibles et impliquent un développement «ad hoc» pour chaque serveur [Cor96b, Apa96b]. De plus, cette solution nous amène à concevoir des serveurs monolithiques dans lesquels toutes les fonctionnalités sont incluses : l'interpréteur CorbaWeb mais aussi d'autres extensions du serveur. De plus, cette solution impose d'adapter notre passerelle au serveur HTTP : gestion de la mémoire et conflits avec d'autres extensions.

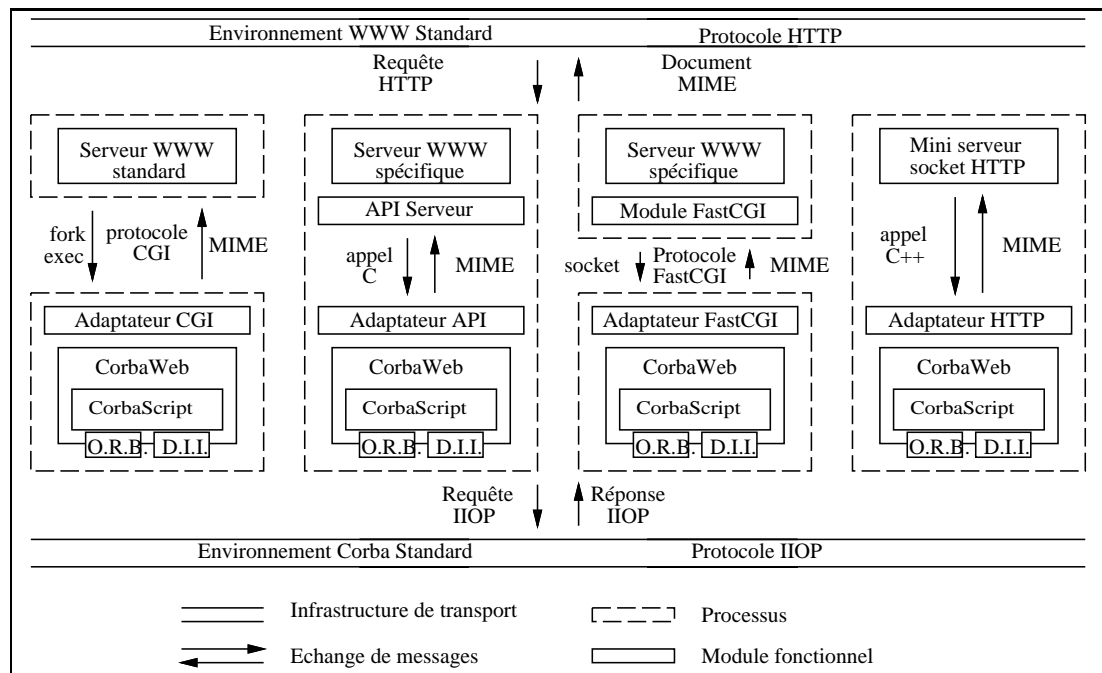


FIG. 4.40 - Diverses implantations de CorbaWeb

- **Le protocole FastCGI :** le protocole FastCGI [Mar96] est un compromis entre les deux solutions précédentes alliant les avantages des CGI avec les avantages des API. Le serveur et la passerelle CorbaWeb sont isolés dans deux processus distincts. La passerelle n'est pas lancée à chaque requête mais c'est un processus en tâche de fond. La communication entre le serveur et CorbaWeb est réalisée par l'intermédiaire du protocole FastCGI utilisant les sockets Unix. Cela permet même de placer le serveur et la passerelle sur des machines distinctes. Néanmoins, cette solution implique de revoir l'interpréteur de CorbaScript pour qu'il puisse être utilisé avec le protocole FastCGI, notamment, pour toutes les opérations d'entrées/sorties.
- **Un mini serveur HTTP :** la dernière solution que nous avons envisagée est d'intégrer un mini serveur HTTP dans la passerelle CorbaWeb. Les gains de cette solution sont la suppression des intermédiaires (création de processus avec le protocole CGI, audit des requêtes par le serveur ou protocole FastCGI) et une amélioration notable de l'efficacité. Dans cette solution, l'interpréteur CorbaWeb reçoit directement les requêtes HTTP et les transforme en variables CorbaScript. De plus, les caches installés dans CorbaScript jouent parfaitement leurs rôles. L'inconvénient de cette solution est que nous perdons toutes les fonctionnalités offertes par les serveurs HTTP

standards. Cependant, le fonctionnement de ce mini serveur en même temps qu'un serveur HTTP classique est tout à fait envisageable.

Parmi les diverses solutions présentées ci-dessus, nous avons implanté la première (CGI standard) et la dernière (mini serveur HTTP). Les deux autres solutions sont en cours d'étude pour mesurer le coût en développement par rapport aux gains fonctionnels que nous obtiendrions.

#### 4.3.5.3 Perspectives et travaux en cours

L'environnement CorbaWeb tel qu'il a été décrit dans ce chapitre est à ce jour opérationnel. Néanmoins, un certain nombre d'extensions et de réflexions doivent être menées. Nous travaillons actuellement sur trois points importants : le contrôle d'accès par profils, la notification et la répartition entre passerelles.

**Le contrôle d'accès et les profils** Dans une application client/serveur classique, c'est à dire conçue à l'aide de souches, les utilisateurs ne peuvent manipuler les objets qu'au travers des applications qui sont mises à leur disposition. Avec CorbaScript et CorbaWeb, tous les objets du système d'informations peuvent être accédés et toutes les opérations définies sur ceux-ci peuvent être appliquées. L'utilisateur a donc une vision globale du système d'informations. Dans un contexte multi-utilisateurs, cette totale liberté peut devenir dangereuse : l'intégrité du système peut être mise en péril par un acte malveillant. Toutes les informations ne doivent pas non plus être accessibles à tout le monde. Certains objets ne doivent être manipulés que par des personnes autorisées (en fonction de leur profil).

Pour ces deux raisons, nous travaillons actuellement au développement d'un mécanisme de contrôle d'accès sur les objets. L'idée générale de notre proposition de contrôle d'accès sur les objets est basée sur les contraintes suivantes :

1. conserver le principe de navigation tel que nous l'avons présenté précédemment,
2. intégrer de manière transparente les applications du patrimoine,
3. autoriser/interdire l'application d'opérations sur des objets en fonction du profil de l'utilisateur
4. développer un mécanisme indépendant des services pour obtenir une séparation claire entre le contrôle et l'exécution.

Notre choix s'est porté sur le développement d'un modèle de portes d'accès [Hor96]. Ces portes sont des objets Corba parre-feux protégeant les objets réels du système d'informations. Nous faisons l'hypothèse pour l'instant que les utilisateurs ne connaissent que les références des portes et pas celles des réels objets. Ils invoquent leurs opérations sur ces portes (cf. figure 4.41). La porte d'accès reçoit l'appel à une opération de l'objet qu'elle contrôle. Elle vérifie les droits d'accès pour son profil d'utilisateur en interrogeant la politique d'accès associée à ce profil. Elle effectue l'appel de l'opération sur l'objet réel si les droits du profil sont suffisants.

Les portes, telles que définies dans notre modèle, ont les caractéristiques suivantes: 1) elles sont **Génériques**, parce qu'elles peuvent se greffer sur n'importe quel type d'objets : ce sont les représentants de ces objets, 2) elles sont **Transparentes** : on accède à la porte d'un objet de la même façon que l'on accède à l'objet, 3) elles sont **Dynamiques** : les portes sont créées «à la volée».

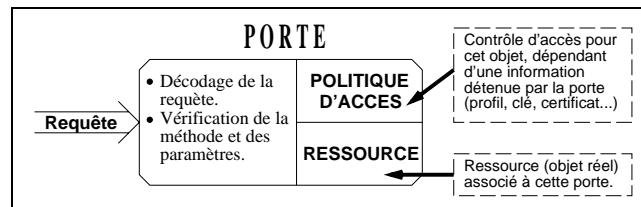


FIG. 4.41 - Schéma d'une porte d'accès

Ce travail a démarré récemment et il est trop tôt pour l'exposer totalement dans cette thèse. Il nous faut encore travailler sur la définition du concept de profils et sur le langage de description des politiques d'accès (ADL : Access Definition Language). Néanmoins, un prototype a été réalisé, il met en œuvre le DSI pour recevoir les requêtes, le DII pour invoquer le réel objet et l'IR pour les métadonnées de typage. Il obtient les informations sur la politique d'accès depuis un référentiel de politique d'accès (i.e. un ensemble d'objets Corba).

**La notification** Le principe de fonctionnement actuel de CorbaWeb est basé sur un modèle question/réponse. L'initiative du dialogue est à la charge du client. Nous avons commencé des travaux sur la notification. C'est à dire, lorsque les objets serveurs sont modifiés, ceux-ci transmettent les modifications vers les navigateurs qui les consultent. Cette notification peut se faire de plusieurs manières :

- **Par scrutation** : le navigateur interroge à intervalle régulier un objet Corba qui lui sert de centralisateur de modification pour un ensemble d'objets auxquels l'utilisateur s'est abonné.
- **A l'aide d'ORBlets** : en supposant que le bus Corba existe partout, des objets pourraient être créés dans l'environnement local de l'utilisateur : soit dans le navigateur soit dans l'espace de travail de l'utilisateur (voir section 4.2.4). Ces objets pourraient répondre à des requêtes provenant des objets serveurs Corba. Les notifications seraient alors transportées par le bus Corba IIOP.

Actuellement, nous avons testé la première solution rapidement réalisable. Le principe de scrutation ne nous satisfait pas car il introduit un surplus de trafic inutile. Nous comptons tester l'autre solution dans les mois à venir.

**La répartition et redirection entre des passerelles** Toutes les illustrations que nous avons présentées utilisaient toujours la même passerelle CorbaWeb. Cependant, il est tout à fait envisageable d'utiliser plusieurs passerelles comme point d'entrée sur un système d'informations. La figure 4.42 illustre ce scénario.

Nous pouvons imaginer un service composé d'objets répartis entre un site en France et un site aux USA (les serveurs  $s1$  et  $s2$ ). Les objets de ces serveurs dialoguent à travers le bus IIOP de Corba et donc ils se référencent. L'utilisateur navigue dans ce graphe d'objets : par exemple, il interroge un objet de  $s1$ , cet objet lui renvoie une référence sur un objet de  $s2$ . Sur chacun des sites, nous plaçons une passerelle CorbaWeb respectivement  $cw1$  et  $cw2$ . La passerelle  $cw1$  génère des documents WWW pour les utilisateurs francophones tandis que  $cw2$  est prévue pour des utilisateurs anglophones. Ainsi, un utilisateur  $n1$  en France se connecte à la passerelle  $cw1$ . Tandis qu'un utilisateur aux E.U. ( $n2$ ) passe par  $cw2$ . Dans ce scénario, lorsque l'utilisateur  $n1$  explore les objets de  $s2$ , plusieurs chemins pour les requêtes sont possibles :

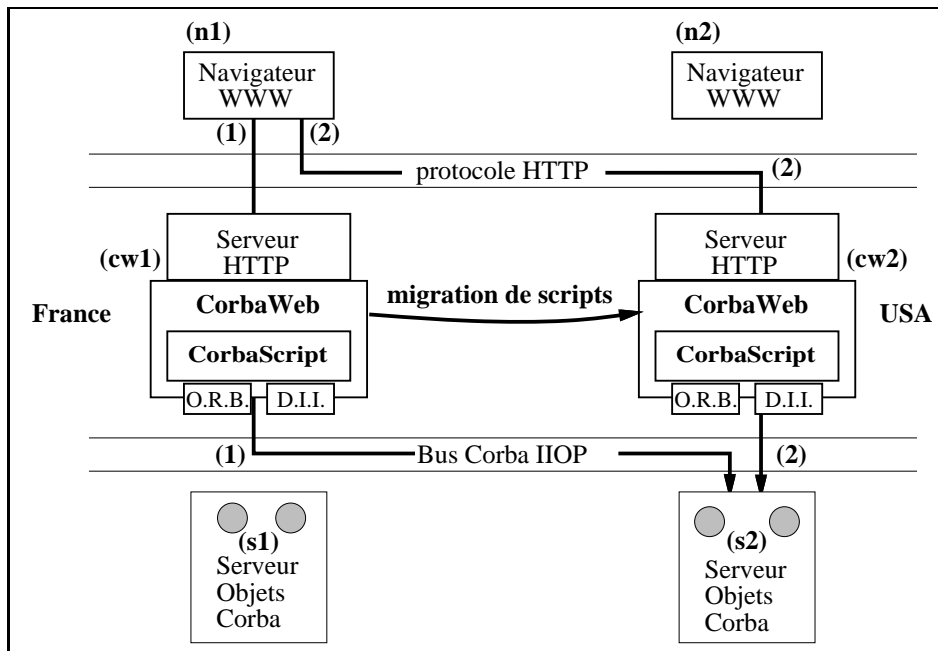


FIG. 4.42 - La coopération de deux passerelles CorbaWeb réparties

- **Par la passerelle locale :** l'utilisateur accède à la passerelle locale par des requêtes HTTP courtes distances. Le script de représentation invoque l'objet  $s_2$  pour générer une représentation en français. Ce script va invoquer de nombreuses opérations de l'objet distant créant ainsi un trafic important de requêtes longues distances sur le bus IIOP. La solution pour réduire ce trafic pourrait consister à passer par la passerelle  $cw_2$ . Pour cela, il faut adapter le module  $CW$  pour que lorsqu'il génère une ancre, il évalue quelle est la passerelle la mieux appropriée pour minimiser le trafic IIOP. Nous pourrions réaliser cette redirection entre passerelles en associant à chacune un domaine d'objets. Lorsqu'une passerelle génère une ancre sur un objet d'un autre domaine, elle indique dans l'URL l'adresse de la passerelle gérant le domaine de cet objet.
- **Par la passerelle distante :** en passant par la passerelle  $s_2$ , l'utilisateur envoie des requêtes HTTP longues distances tandis que la passerelle génère un important trafic IIOP local. Les performances devraient être améliorées car une requête HTTP génère souvent plusieurs requêtes IIOP. Par exemple, visualiser un objet *agence* implique une requête HTTP du navigateur  $n_1$  vers la passerelle  $cw_2$ . Puis, le script de représentation va consulter l'état de l'agence et de plus, il consulte chacun des sous-objets pour générer le libellé des ancres de navigation. Ainsi, il est préférable que nos requêtes IIOP soient transportées sur un réseau local plutôt que sur un réseau longue distance. Néanmoins, il est alors nécessaire que la passerelle  $cw_2$  sache générer des documents en français. Pour cela, nous devons mettre en place des mécanismes de réplication des scripts de  $cw_1$  vers  $cw_2$ . Car si les scripts français sont modifiés pour répondre à de nouveaux besoins alors les scripts sur  $cw_2$  doivent être mis à jour.
- **Par la migration du script de représentation :** dans cette dernière solution, l'utilisateur  $n_1$  accède toujours aux objets Corba par la passerelle locale  $cw_1$ . Le méta script de celle-ci détecte si l'objet à représenter est dans son domaine ou non. Lorsque l'objet est dans son domaine, la passerelle exécute le script de représentation

comme actuellement. S'il est hors de son domaine, la passerelle *cwl* envoie le script de représentation vers la passerelle proche de l'objet. Cette passerelle exécute le script localement, ainsi le trafic IOP est toujours aussi important mais sur de courtes distances. Une fois le document produit, il est renvoyé à la passerelle *cwl* qui n'a plus qu'à rediriger ce document vers l'utilisateur. La migration du script pourrait aussi bien utiliser le bus WWW-HTTP que le bus Corba-IOP. Le méta script d'exécution de scripts peut être un élément de réponse pour le transport par le bus HTTP. Pour la seconde éventualité, nous pouvons créer dans chaque passerelle un objet Corba qui possède une opération *execute*. Cette opération reçoit en paramètre le script sous la forme d'une chaîne de caractères et retourne le résultat d'exécution via une chaîne. Cette solution réduit ainsi le trafic IOP et permet de garder les passerelles indépendantes : la cohérence des scripts n'est plus nécessaire.

Ainsi, la passerelle CorbaWeb peut être répliquée pour offrir de multiples points d'accès aux services orientés objet. Mais, de nombreuses expérimentations sont encore nécessaires pour évaluer concrètement sur de réelles applications les mécanismes que nous venons d'exposer : la redirection entre passerelles, les domaines, la réplication des scripts de représentation et finalement la migration de scripts. L'objectif sera de réduire globalement le trafic afin d'améliorer les temps de réponse pour l'accès à des services et objets répartis.

## 4.4 Conclusion

Au fil de ce long chapitre, nous avons étudié et présenté de nombreux éléments : l'infrastructure «classique» du World-Wide Web, l'intégration des objets Corba dans le WWW et finalement notre environnement CorbaWeb pour l'intégration générique et dynamique de tout objet Corba dans le WWW. CorbaWeb favorise alors l'accès et la navigation dans de vastes espaces de services et d'objets répartis.

Nous concluons ce chapitre sur l'intérêt de fusionner les mondes WWW et Corba, et nous résumons les fonctionnalités de CorbaWeb.

### 4.4.1 Une infrastructure globale de services en ligne

Dans la première partie de ce chapitre, nous avons rappelé les éléments caractérisant le WWW : la **navigation hypermédia** dans des **ressources réparties** via une **interface universelle d'accès**. Néanmoins, nous avons formulé des critiques au sujet du WWW : le **syndrome du serveur monolithique**, des **documents statiques**, le **développement «ad hoc» de ressources dynamiques**, le **manque d'accès à des applications réparties ou distantes** et finalement l'**absence d'un modèle uniforme de structuration des ressources**.

Pour répondre à ces critiques, nous pensons que le WWW et Corba doivent fusionner pour fournir une infrastructure globale pour la conception, le déploiement et l'accès à une grande variété de services en ligne. Cette infrastructure aura les caractéristiques suivantes :

- **La modularité des services** : Corba apporte au WWW un modèle uniforme de structuration des ressources ou services accessibles en ligne. Les propriétés de l'approche orientée objet de Corba permettent de concevoir des ressources modulaires, structurées, encapsulées et réutilisables. Le langage IDL fournit la séparation entre l'implantation et l'interface d'un service permettant ainsi la coopération de services hétérogènes (langages, OS, machines).

- **Le développement centralisé/déploiement global** : l'intérêt du WWW est de pouvoir développer des services localement et de les rendre accessibles de n'importe quel site du réseau Internet. Pour cela, il offre deux solutions techniques : les passerelles et les codes mobiles. Les passerelles permettent d'intégrer des services Corba au sein d'un serveur WWW pour offrir de nouvelles ressources dynamiques. Les codes mobiles permettent de télécharger sur le poste du client une application pour converser avec un service Corba distant (ces codes sont aussi désignés par le terme ORBlets).
- **L'infrastructure à grande échelle** : Corba et le WWW fournissent les mécanismes pour la distribution, la communication et le nommage global des services à travers les bus logiciels IIOP et HTTP. Ce dernier est plutôt dédié à l'accès de documents tandis que le premier permet d'appliquer des traitements sur n'importe quel objet du réseau.
- **L'accès universel et convivial** : grâce à un navigateur, l'utilisateur peut accéder à tous les services disponibles sur le réseau de manière conviviale et hypermédia. Il peut dialoguer avec les services Internet de base comme la messagerie électronique, les forums de discussions ou les serveurs de documents (Ftp, Gopher et Web). Et il peut aussi accéder à des services disponibles sur le bus Corba par l'intermédiaire des applications Corba mobiles. Celles-ci permettent d'envisager de nouveaux modes d'interactions entre les utilisateurs et les ressources et aussi entre utilisateurs : les ressources peuvent notifier les utilisateurs en cas de changement et les navigateurs peuvent communiquer pour supporter des applications collectives.

Cependant, de nombreuses expérimentations sont encore nécessaires pour bien maîtriser cette nouvelle infrastructure. L'enjeu principal sera de rendre disponible une multitude de types et d'instances de services dans un contexte fortement évolutif. Les utilisateurs auront des rôles distincts et des besoins spécifiques évoluant au fil du temps et des changements dans les systèmes d'informations et de services. De nouveaux objets seront créés dynamiquement et devront être rapidement mis à la disposition des utilisateurs. Les objets patrimoines (c'est à dire non prévu pour le Web) pourraient être accessibles sans nécessiter des changements ou trop de développements spécifiques.

#### 4.4.2 La proposition CorbaWeb

L'environnement CorbaWeb fournit une telle infrastructure globale pour la conception, le déploiement et l'accès en ligne à des services orientés objet. CorbaWeb offre à la fois un moyen d'accès via le Web à tout service Corba réparti et un moyen d'extension du Web à de nouveaux services, et cela de manière générique contrairement aux solutions fermées proposées jusqu'ici. Cette intégration ne se limite pas à une catégorie d'objets spécifiques mais permet d'accéder à tous les types d'objets Corba à travers le langage CorbaScript. Les utilisateurs peuvent alors naviguer dans des graphes d'objets aussi simplement que dans des graphes de documents et invoquer toute opération sur ces objets.

Notre infrastructure se décompose de la manière suivante : le bus Corba pour la conception et le support des services, l'infrastructure WWW pour l'accès à ces services et finalement la passerelle CorbaWeb pour déployer ces services sur le WWW. Notre proposition bénéficie ainsi du meilleur de chacun de ces deux mondes : la navigation hypermédia du WWW et l'approche orientée objet de Corba. L'avantage majeur de notre approche est de clairement séparer la conception des objets fonctionnels d'un service de leur représentation dans le WWW. La connexion des deux mondes est alors réalisée dynamiquement

via la passerelle CorbaWeb. Notre langage de script CorbaScript est le noyau d'exécution de cette passerelle offrant la souplesse et la flexibilité pour l'accès à tout objet Corba tel que nous l'avons présenté dans le chapitre 3.

CorbaWeb permet d'implanter des ressources WWW dynamiquement générées par des scripts. Des tableaux de bord ou des interfaces graphiques peuvent être simplement et rapidement conçus en exploitant dynamiquement les objets Corba. Pour cela, il n'est pas nécessaire de modifier les objets Corba ou de savoir programmer le complexe environnement Corba. Il suffit de connaître le langage HTML, le langage CorbaScript et les spécifications des services à intégrer. La conception d'un nouveau patron ou d'un script d'accès peut se faire interactivement en utilisant conjointement un éditeur de texte et un navigateur. Nous considérons que cela est à la portée de tous les utilisateurs.

Les méta scripts apportent une solution générique pour intégrer rapidement des millions d'instances et des milliers de types. Nous avons présenté quatre méta scripts mais il est tout à fait possible d'en développer d'autres :

- **Le méta script d'interfaces** permet de générer automatiquement une interface HTML sur tout objet Corba. Cette interface est composée de formulaires pour invoquer les opérations, consulter et modifier les attributs définis dans l'interface de l'objet.
- **Le méta script d'exécution d'opérations** peut exécuter n'importe quel traitement unitaire sur un objet. Il est utilisé conjointement avec le méta script précédent pour réaliser un navigateur générique dans tout graphe d'objets Corba.
- **Le méta script d'exécution de scripts** permet aux utilisateurs d'exprimer des traitements complexes sur les objets Corba. Ces traitements sont exprimés en CorbaScript. Ces scripts sont mobiles car ils sont rédigés sur le site de l'utilisateur et ils s'exécutent sur le site de la passerelle CorbaWeb.
- **Le méta script de représentation** permet de personnaliser la navigation dans des graphes d'objets en fonction du type des objets et du profil de l'utilisateur. Cette personnalisation est réalisée à l'aide de scripts de représentation associés à chaque couple type-profil. Ils sont stockés dans un référentiel de scripts de représentation.

Actuellement, nous avons pris comme langage d'interface graphique le langage HTML mais notre approche est applicable avec d'autres langages de description d'interfaces tels que VRML<sup>22</sup> [Pes96]. Nous utilisons actuellement les formulaires HTML pour agir sur les objets. Néanmoins, notre approche n'est pas du tout incompatible avec l'utilisation de langages de code mobile tels que le langage Java. Il suffit alors de générer des documents HTML qui incluent ces applets.

Finalement, cette plate-forme nous ouvre de nombreuses nouvelles perspectives de travail : les diverses stratégies d'implantation du référentiel des scripts de représentation, la classification des profils des utilisateurs, le contrôle d'accès à granularité fine (les portes), la notification des navigateurs, la coopération entre passerelles réparties ou encore la migration de scripts pour réduire le trafic sur le bus Corba.

---

22. VRML : Virtual Reality Modeling Language



# Conclusion

## Contexte

L'objectif de cette thèse est de favoriser l'accès à des services à grande échelle. Nous avons choisi de concevoir ces services à l'aide du bus à objets répartis Corba. Actuellement, Corba propose une infrastructure complexe pour la conception et le développement de services : elle ne peut être mise en œuvre que par des développeurs « experts ». De plus, la chaîne de production Corba couramment utilisée ne favorise pas le déploiement et l'utilisation des futurs services à grande échelle. Lorsqu'un utilisateur veut accéder à un service, il doit disposer d'un outil spécifiquement développé pour ce service et pour la tâche à accomplir.

## Propositions

Partant de ce constat, notre projet GOODE (Generic Object-Oriented Dynamic Environment) vise à définir et fournir un ensemble de nouveaux outils flexibles pour favoriser le déploiement et l'accès à grande échelle à des services composés d'objets répartis. Ces outils mettent en œuvre les mécanismes dynamiques de Corba : l'interface d'invocation dynamique (DII) pour construire les requêtes, l'interface de squelettes dynamiques (DSI) pour recevoir des requêtes et finalement le référentiel des interfaces pour découvrir les métadonnées sur les interfaces IDL des objets répartis. Dans cette thèse, nous avons présenté les deux premiers outils de cet environnement générique et dynamique : CorbaScript et CorbaWeb.

CorbaScript est un langage de script interprété et orienté objet. Il masque la complexité de l'environnement Corba et il permet d'invoquer n'importe quelle opération, de consulter ou de modifier n'importe quel attribut de n'importe quel objet Corba. La production de scripts est grandement facilitée par l'ensemble des fonctionnalités du langage : typage dynamique, modularité et concepts orientés objet. Contrairement à l'approche statique de Corba, CorbaScript construit dynamiquement les souches IDL pour invoquer les objets en consultant le référentiel des interfaces. Ensuite, les invocations sont réalisées via l'interface DII et les objets CorbaScript reçoivent les invocations via l'interface DSI.

Ce langage peut être mis à profit dans toutes les étapes du cycle de vie des objets. CorbaScript permet d'évaluer et de prototyper rapidement des choix de conception d'interfaces IDL et de répartition des objets. Les développeurs peuvent réaliser des tests unitaires de leurs composants logiciels. Ces tests peuvent être faits en interactif ou à l'aide de fichiers de scripts réutilisables. Les administrateurs de services configurent et paramètrent les objets sans devoir développer des outils spécifiques et complexes. La création, la destruction et la composition d'objets peuvent être effectuées en quelques lignes de scripts. De nouveaux composants peuvent être construits par assemblage/agrégation de composants préexistants

et ils deviennent eux-mêmes des objets Corba. Finalement, les utilisateurs peuvent naviguer dans tout graphe d'objets Corba et ainsi, ils peuvent réaliser leurs tâches spécifiques à travers cet outil générique.

A partir de ce langage, nous avons développé l'environnement CorbaWeb pour l'intégration des objets dans le World Wide Web. Cet environnement allie les bénéfices des mondes Corba et Web : Corba pour la conception de services modulaires et le Web pour la navigation hypermédia. Grâce à CorbaWeb, les objets Corba peuvent implanter de nouveaux services pour le Web et celui-ci est vu comme une interface Homme-Machine pour accéder à tout objet Corba. Ainsi, les utilisateurs naviguent dans des graphes d'objets aussi simplement et naturellement que dans des graphes de documents.

La génération de la présentation des objets d'un service est effectuée par des scripts écrits dans notre langage CorbaScript. Ces scripts génèrent des documents WWW (MIME et HTML) représentant l'état des objets et offrant des formulaires (plus tard des applets) pour agir sur ceux-ci. L'état des objets est obtenu dynamiquement en invoquant les opérations ou en consultant les attributs des objets. Les relations entre les objets sont représentées par des ancres HTML. Les URL de ces ancres contiennent les références des objets Corba et permettent ainsi de naviguer à travers ceux-ci.

CorbaWeb offre trois approches pour intégrer les objets dans le WWW. Les documents dynamiques et les scripts d'accès permettent de concevoir une multitude de tableaux de synthèse et des interfaces sur les objets. Les méta scripts quant à eux permettent d'appliquer des traitements génériques sur tout type d'objets tels que la génération automatique d'interfaces HTML, l'exécution d'opérations unitaires, l'exécution de scripts utilisateurs ou la navigation en fonction du profil de l'utilisateur. Pour ce dernier, nous avons défini un référentiel de scripts de représentation ainsi qu'une politique de consultation de celui-ci.

L'apport majeur de CorbaWeb par rapport aux solutions existantes est de clairement séparer la partie fonctionnelle des services de leur présentation dans le WWW. Nous pouvons ainsi intégrer des services aussi bien spécifiquement développés pour que non prévus à l'origine pour le WWW. De plus, CorbaWeb est un environnement extensible dans lequel il est relativement aisé d'ajouter de nouveaux ou d'adapter des méta scripts pour des besoins plus spécifiques : la génération de VRML, l'utilisation d'applets ou les diverses stratégies d'implantation du référentiel de scripts de représentation.

## Perspectives

A travers ces deux réalisations, nous avons posé les bases de l'environnement GOODE. De nombreuses perspectives sont à explorer aussi bien au niveau du langage CorbaScript, des applications que de nouvelles pistes de recherche.

### Le langage CorbaScript

Le langage CorbaScript est actuellement opérationnel au dessus du bus à objets répartis Orbix 2.0. Jusqu'à présent, nous nous sommes principalement concentrés sur les fonctionnalités du langage et sur le développement d'un interpréteur extensible. Un premier travail concerne l'optimisation de l'interpréteur pour améliorer ses temps d'exécution.

Dans la version actuelle, CorbaScript utilise uniquement l'appel synchrone de l'invocation dynamique. Il serait intéressant d'introduire les deux autres modes : l'appel asynchrone et l'appel avec retour de résultat différé. Pour ce dernier, nous envisageons d'introduire des variables « futures ». L'attente de la réponse d'une requête n'interviendrait que lorsque le

script utiliserait la variable «future». Ces deux autres modes d'appel permettront d'introduire des notions de parallélisme dans l'interpréteur. Nous comptons évaluer l'utilisation de ces modes d'appels dans des applications coopératives.

Actuellement, CorbaScript peut uniquement accéder à un service Corba distant via le DII. Mais certains services Corba sont en partie implantés localement au processus client : les communications de groupe (Orbix+ISIS), le service d'événements asynchrones (OrbixTalk), le service de «multi-threading», la sécurité ou bien les transactions. Nous avons l'intention de connecter CorbaScript avec ce type de services Corba au fur et à mesure de leur disponibilité. Cela ne devrait pas poser de gros problèmes techniques car nous pouvons facilement intégrer dans l'interpréteur de nouveaux types de données et de nouvelles procédures écrites en C++.

Dans les mois à venir, nous comptons commencer la diffusion sur Internet du langage CorbaScript et de l'environnement CorbaWeb. Comme Orbix est l'Orb leader du marché, nous espérons obtenir de nombreux retours de la part des utilisateurs. Nous avons déjà été contactés par de nombreux utilisateurs potentiels qui sont fortement intéressés par l'utilisation d'un langage de script dynamique dans leur processus de production d'applications distribuées. Les retours nous permettront de corriger et affiner nos choix sur le langage mais aussi sur son implantation.

Pour valider nos choix d'implantation, il serait aussi intéressant de porter CorbaScript sur d'autres Orbs à la norme Corba 2.0. Des contacts ont été pris avec divers fournisseurs : DAIS de ICL, Orbeline de Visigenic et NEO de Sun.

## Des applications

Le travail présenté dans ce mémoire nous a permis d'acquérir un savoir-faire important dans le domaine de la réalisation de services distribués orientés objet et de présenter une nouvelle approche pour l'accès à ceux-ci. A présent, nous allons retransmettre ce savoir aux autres membres du programme régional Ganymède auquel nous participons.

CorbaWeb peut être qualifié de serveur HTTP générique réalisant une passerelle entre le WWW et Corba. Quant à CorbaScript, il peut être considéré comme un serveur Corba générique pouvant supporter n'importe quel type d'objets Corba. Nous envisageons de l'utiliser pour relier le monde Corba avec d'autres environnements informatiques. Actuellement, nous étudions deux contextes : les cartes à microprocesseur et l'administration d'équipement réseau.

Dans le cadre du projet OSMOSE (Operating System and MOBILE Services [Van97]) conjointement mené par le laboratoire RD2P<sup>23</sup> et la société Gemplus<sup>24</sup>, un serveur générique de cartes à microprocesseur a été développé [Cha95]. Ce serveur transforme des requêtes Corba en instructions au format APDU<sup>25</sup>. Ces travaux se focalisent sur la réalisation d'un Card Object Adapter (COA) qui permettra de faire dialoguer n'importe quelle application Corba avec n'importe quel type de carte à microprocesseur. Cet adaptateur sera composé d'objets Corba implantés avec le langage CorbaScript. Le code de ces objets sera stocké dans un référentiel d'implantation. Ce référentiel sera dynamiquement consulté comme nous le faisons pour le référentiel des scripts de représentation dans l'environnement CorbaWeb.

---

23. RD2P : Recherche et développement dossier portable

24. Gemplus est une des sociétés leader sur le marché de la carte à microprocesseur.

25. *Application Protocol Data Unit* est un standard de transport de commandes carte entre le système hôte et le lecteur de cartes [ISO94].

Nous comptons également utiliser notre principe de passerelle générique dans le domaine de l'administration de réseau. En nous basant sur les travaux précédents, nous espérons développer un serveur qui pourra contrôler les équipements réseau au travers des MIB<sup>26</sup> de ceux-ci. Ces équipements seront représentés par des objets Corba implantés avec le langage CorbaScript. Ces objets pourront ainsi être administrés par l'intermédiaire de scripts CorbaScript ou bien via l'environnement CorbaWeb.

Le projet OSACA<sup>27</sup> vise à la conception d'une infrastructure logicielle ouverte pour les applications coopératives (applications dédiées à des groupes d'utilisateurs). L'environnement CorbaWeb peut être une base intéressante pour ce type d'applications. L'environnement Corba nous permet de concevoir les services de coopération. Le Web nous offre une interface Homme-Machine hypermédia et portable (MIME, HTML et applets). Les différents modes de communication entre les applets et les objets (voir section 4.2.4) peuvent supporter les interactions et la notification des navigateurs.

Notre environnement permet déjà de prendre en compte la diversité des types de ressources et des rôles des utilisateurs (via le méta script de représentation) que nous rencontrerons dans les applications coopératives. Pour évaluer cette proposition, nous avons déjà mené une expérimentation en relation avec le laboratoire CERIM et le CHR de Lille pour l'accès au système d'information PLACO qui supporte la planification coopérative dans des unités de soins intensifs [You96, Fil96]<sup>28</sup>.

## Des pistes de recherche

Nos travaux pour favoriser l'accès aux services nous ont permis de dégager de nouvelles pistes de recherche nous paraissant prometteuses : le contrôle d'accès à granularité fine, la mobilité des scripts et l'assemblage de composants logiciels.

Nous avons commencé des travaux sur le contrôle d'accès à granularité fine (voir section 4.3.5.3 et [Hor96]). L'idée générale est de séparer les aspects fonctionnels des services des aspects contrôle d'accès. Ceci nous permet d'évaluer et proposer diverses politiques de contrôle d'accès pour un service. De plus, il est ainsi possible de plaquer un système de contrôle d'accès sur des applications déjà existantes. Nous basons notre approche sur des objets portes qui interceptent et contrôlent les invocations sur les services. Lors de la navigation dans un graphe d'objets, les portes sont créées à la volée en fonction du profil de l'utilisateur (du navigateur). Nous projetons d'harmoniser cette notion de profil avec celle définie pour le méta script de représentation de CorbaWeb.

Nous avons vu qu'à l'aide du méta script d'exécution de script (c.f. section 4.3.3.3) nous pouvions envoyer du code CorbaScript d'un navigateur Web vers l'environnement CorbaWeb. Nous comptons poursuivre dans cette voie et étudier une approche générique pour déplacer du code mobile CorbaScript mais aussi des instances d'objets CorbaScript (i.e. des agents systèmes scriptés). Dans cet environnement, nous devons mettre en place un ensemble de serveurs d'accueil des scripts. Ces serveurs seront écrits en CorbaScript pour pouvoir accepter n'importe quel script. Cette approche permet de limiter les communications longue distance : au lieu d'invoquer de nombreuses opérations d'un objet très distant, le script se déplacera vers cet objet.

---

26. MIB : Management Information Base

27. OSACA : Open Software Architecture for Cooperative Applications

28. Ce travail devrait se poursuivre dans le cadre de la thèse de Jean-Marie Renard.

Les deux éléments importants à prendre en considération seront alors : le contrôle de l'exécution des scripts et la localisation des serveurs d'accueil. Pour le premier élément, nous comptons réutiliser nos travaux sur les portes de contrôle d'accès : les scripts pourront seulement exécuter des opérations sur les objets autorisés. Pour le second point, nous comptons mettre en place un système équivalent au DNS pour localiser les serveurs d'accueil. Un script pourra se rapprocher d'un objet distant en invoquant une primitive de migration par exemple *migrerVersObjet(une\_référence\_d\_objet)*. Le noyau d'exécution recherchera le serveur d'accueil le plus proche de cet objet et lui transférera le script ou l'objet CorbaScript.

CORBA reste un modèle de bas niveau permettant la mise en place de composants, mais n'apportant aucune aide (aucune automatisation) pour l'installation d'un ensemble complexe de composants interconnectés. C'est à ce niveau que doivent intervenir les langages d'assemblage de composants (par exemple OLAN [BABR96]). Ils cachent à l'utilisateur – celui qui assemble des composants – les détails de l'infrastructure sous-jacente, le laissant se concentrer sur ses choix d'assemblage.

Nous débutons des travaux sur la spécification d'un langage de description de composants. Nous comptons utiliser CorbaScript comme «colle logicielle» pour faciliter l'interconnexion des composants : il servira de langage d'implantation du langage de description d'assemblage des composants. Les spécifications d'assemblage seront stockées dans un référentiel d'assemblage. CorbaScript consultera ce référentiel pour créer, à la volée, les souches d'assemblage (nous reprenons le principe utilisé pour le contrôle d'accès).

Ainsi, le travail présenté dans ce mémoire nous ouvre de nombreuses perspectives aussi bien sur le plan technologique, sur la réalisation d'applications qu'au niveau de la recherche sur les futures infrastructures d'objets répartis pour les services à grande échelle.



# Annexe A

## Exemples

Dans cette annexe, nous présentons quelques exemples complémentaires : le service de nommage de Corba implémenté en CorbaScript, un service de calcul sur les nombres premiers implémenté en C++ et finalement un système d'informations hospitalier multimédia.

### A.1 Un service de nommage

Dans le chapitre 2, nous avons décrit le service de nommage des objets Corba : le module IDL *CosNaming*. Nous détaillons ici une implantation de celui-ci en CorbaScript. La classe *NAMING\_IMPL* implante l'interface IDL *CosNaming::NamingContext* (d'où le lien d'héritage). L'état d'un contexte de nommage est stocké dans une table d'association implantée par la classe héritée *AssociateTable.AssociationTable*. L'implantation des opérations IDL fait appel aux primitives disponibles sur la table. Notre classe redéfinit la méthode héritée de comparaison des clés *compareKeys* utilisée par exemple pour les recherches.

La plupart des opérations reçoivent en paramètre un nom *n* de type *CosNaming::Name*. Si ce nom ne contient qu'un élément alors l'objet exécute l'opération sinon il la transmet au sous-contexte adéquat. Ainsi, une demande de résolution d'association (i.e. *resolve*) peut parcourir le graphe des contextes afin de trouver l'association demandée.

```
#####
#      Implantation du service CosNaming
import AssociateTable

# création d'un tableau à partir d'une tranche d'un tableau
#
proc sub_array (arr,n1,n2) {
  result = []
  i = n1
  while (i < n2) {
    result.append(arr[i])
    i = i + 1
  }
  return result
}

class NAMING_IMPL (AssociateTable.AssociationTable,CosNaming::NamingContext) {
  proc __NAMING_IMPL__ (self,name) {
    self.__AssociationTable__ ()
  }
}
```

```

    self.__NamingContext__ (name)
}
# spécialisation de la procédure de comparaison de clés
proc compareKeys (self,key1,key2) {
    return ( key1.id == key2.id ) && ( key1.kind == key2.kind )
}

proc is_ncontext (self,name) {
    val = self.find_key (name[0])
    if ( val[0] != CosNaming::BindingType.ncontext ) {
        throw CosNaming::NamingContext::NotFound (
            CosNaming::NamingContext::not_context,name)
    }
    return val[1]
}
proc test_valid_name (name) {
    if ( name.length == 0 || name[0].id == "" ) {
        throw CosNaming::NamingContext::InvalidName ()
    }
}

# implantations des opérations IDL de CosNaming::NamingContext
proc bind (self,n,obj) {
    test_valid_name (n)
    if ( n.length != 1 ) {
        subcontext = self.is_ncontext (n)
        return subcontext.bind (sub_array (n,1,n.length),obj)
    }
    inserted = self.insert ( n[0], [CosNaming::BindingType.nobject,obj] )
    if ( not inserted ) { throw CosNaming::NamingContext::AlreadyBound () }
}
proc rebind (self,n,obj) {
    test_valid_name (n)
    if ( n.length != 1 ) {
        subcontext = self.is_ncontext (n)
        return subcontext.rebind (sub_array (n,1,n.length),obj)
    }
    replaced = self.replace ( n[0], [CosNaming::BindingType.nobject,obj] )
    if ( not replaced ) {
        throw CosNaming::NamingContext::NotFound (
            CosNaming::NamingContext::NotFoundReason::not_context,name)
    }
}
proc bind_context (self,n,ctx) {
    test_valid_name (n)
    if ( n.length != 1 ) {
        subcontext = self.is_ncontext (n)
        return subcontext.bind_context (sub_array (n,1,n.length),ctx)
    }
    inserted = self.insert ( n[0], [CosNaming::BindingType.ncontext,ctx] )
    if ( not inserted ) { throw CosNaming::NamingContext::AlreadyBound () }
}
proc rebind_context (self,n,ctx) {
    test_valid_name (n)
    if ( n.length != 1 ) {
        subcontext = self.is_ncontext (n)

```

```

        return subcontext.rebind_context (sub_array (n,1,n.length), ctx)
    }
    self.is_ncontext (n)
    self.replace (n[0], [CosNaming::BindingType.ncontext,ctx] )
}
proc resolve (self,n) {
    test_valid_name (n)
    if ( n.length != 1 ) {
        subcontext = self.is_ncontext (n)
        return subcontext.resolve (sub_array (n,1,n.length))
    }
    return self.find_key(n[0])[1]
}
proc unbind (self,n) {
    test_valid_name (n)
    if ( n.length != 1 ) {
        subcontext = self.is_ncontext (n)
        return subcontext.unbind (sub_array (n,1,n.length))
    }
    removed = self.remove (n[0])
    if ( not removed ) {
        throw CosNaming::NamingContext::NotFound (
            CosNaming::NamingContext::NotFoundReason::not_context,name)
    }
}
proc new_context (self) {
    return NAMING_IMPL("")
}
proc bind_new_context (self,n) {
    test_valid_name (n)
    if ( n.length != 1 ) {
        subcontext = self.is_ncontext (n)
        return subcontext.bind_new_context (sub_array (n,1,n.length))
    }
    name = n[0]
    pos = self.find_position (name)
    if ( pos != Void ) {
        throw CosNaming::NamingContext::AlreadyBound ()
    }
    result = self.new_context ()
    self.append (name, [CosNaming::BindingType.ncontext,result])
    return result
}
proc destroy (self) {
    if ( self.nb_items() != 0 ) {
        throw CosNaming::NamingContext::NotEmpty ()
    }
    CORBA.BOA.dispose (self)
}
proc list (self,how_many,bl,bi) {
    bl = []
    for i in range (0, self.nb_items() - 1) {
        bl.append ( [ [ self.key(i) ], self.value(i)[0] ] )
    }
}
}

```

L'implantation du service de nommage est simple car elle délègue totalement la gestion de la structure de données à la classe *AssociationTable* contenue dans le module *AssociationTable*. Cette classe implante une table d'associations constituée de deux tableaux : l'un pour les clés et l'autre pour les valeurs. L'accès à ces deux tableaux est encapsulé par un jeu de primitives pour insérer, supprimer, modifier et rechercher des associations.

```
#####
# Ce module implante une table d'association
#
class AssociationTable {
  proc __AssociationTable__ (self){
    self.keys = []
    self.values = []
  }
  # A SPECIALISER dans les sous classes
  proc compareKeys (self,key1,key2) {
    return key1 == key2
  }
  proc nb_items (self) {
    return self.keys.length
  }
  proc key (self,pos) {
    return self.keys[pos]
  }
  proc value (self,pos) {
    return self.values[pos]
  }
  proc find_position (self,key) {
    i = 0
    keys = self.keys
    m = keys.length
    continu = true
    while ( continu && i < m ) {
      k = keys[i]
      if self.compareKeys (key,k) {
        continu = false
      } else {
        i = i + 1
      }
    }
    if ( continu ) { return Void }
    return i
  }
  proc find_key (self,key) {
    pos = self.find_position (key)
    if ( pos == Void ) {
      throw "non trouve"
    }
    return self.values[pos]
  }
  proc append (self,key,value) {
    self.keys.append(key)
    self.values.append(value)
  }
  proc insert (self,key,value) {
    pos = self.find_position (key)
    if ( pos == Void ) {
```

```

        self.append (key,value)
        return true
    }
    return false
}
proc remove_pos (self,pos) {
    self.keys.delete (pos)
    self.values.delete (pos)
}
proc remove (self,key) {
    pos = self.find_position (key)
    if ( pos != Void ) {
        self.remove_pos (pos)
        return true
    }
    return false
}
proc replace_pos (self,pos,value) {
    self.values[pos] = value
}
proc replace (self,key,value) {
    pos = self.find_position (key)
    if ( pos != Void ) {
        self.replace_pos (pos,value)
        return true
    }
    return false
}
}
}

```

L'exemple ci-dessous illustre comment un administrateur peut installer notre implantation du service de nommage. Il lance l'interpréteur en précisant un nom de serveur Orbix (e.g. *UnNouveauServiceDeNommage*). Il importe le module contenant l'implantation du service puis il crée une instance de la classe *NAMING\_IMPL*. Les utilisateurs peuvent alors se connecter à ce service en indiquant simplement sa référence CorbaScript (e.g. *CosNaming::NamingContext("root:UnNouveauServiceDeNommage")*).

```

prompt_unix > cssh -s UnNouveauServiceDeNommage
CorbaScript 1.0 (August 1 1996)
Copyright 1996 LIFL, France
>>> import NAMING
>>> NS = NAMING.NAMING_IMPL ("root")
>>>

```

```

prompt_unix > cssh
CorbaScript 1.0 (August 1 1996)
Copyright 1996 LIFL, France
>>> NS = CosNaming::NamingContext ("root:UnNouveauServiceDeNommage")
>>> un_objet = NS.resolve ( ... )

```

Cette implantation du service de nommage n'est pas complète car elle ne gère pas la persistance des contextes mais elle illustre la simplicité de développement d'un service Corba à l'aide de CorbaScript. Dans cet exemple, nous avons repris (sans les modifier) les spécifications officielles de l'OMG. De plus, le polymorphisme et le typage dynamique de CorbaScript permettent de créer des classes génériques et réutilisables.

## A.2 Un service de calcul sur les nombres premiers

Dans cet exemple, nous illustrons un service de calcul de nombres premiers. Ce service est décrit par l'interface ci-dessous et il offre trois opérations: *is\_prime\_number* évalue si un nombre est premier ou non, *prime\_numbers* calcule la suite des nombres premiers inférieurs à une borne donnée et *prime\_factors* décompose un nombre en la suite de ses facteurs premiers.

```
interface computer {
    typedef unsigned long Positive;
    typedef sequence < Positive > seqPositive;
    boolean is_prime_number (in Positive number);
    seqPositive prime_numbers (in Positive number);
    seqPositive prime_factors (in Positive number);
};
```

Le code ci-dessous illustre une implantation de ce service avec Orbix 2.0 et le langage C++. La classe *COMPUTER* implante l'interface IDL *computer* et hérite donc de la classe squelette *computerBOAImpl* générée par le compilateur IDL. L'implantation des opérations a été volontairement simplifiée, nous n'avons pas jugé nécessaire d'implanter des algorithmes performantes et complexes. Finalement, la boucle principale *main* crée une instance de *COMPUTER* et se met en attente des requêtes Corba. Cette instance aurait très bien pu être enregistrée auprès du service de nommage comme nous l'avons illustré au chapitre 2.

```
//=====
// implantation du service de calcul sur les nombres premiers
// avec Orbix 2.0
//
#include <iostream.h>
#include <computer.hh>
#define ARRAY_SIZE 1024

class COMPUTER : public virtual computerBOAImpl {
public:
    COMPUTER (const char* name) : computerBOAImpl (name) {};
    ~COMPUTER () {};
public: // IDL operations
    CORBA::Boolean is_prime_number (computer::Positive number, CORBA::Environment&);
    computer::seqPositive* prime_numbers (computer::Positive number, CORBA::Environment&);
    computer::seqPositive* prime_factors (computer::Positive number, CORBA::Environment&);
};

CORBA::Boolean COMPUTER::is_prime_number (
    computer::Positive number,
    CORBA::Environment&
) {
    if ( number < 0 ) return 0;
    if ( number == 1 ) return 1;
    if ( number % 2 == 0 ) return number == 2;
    long middle = number / 2;
    for (long i=3; i<middle; i=i+2) {
        if ( number % i == 0 ) return 0;
    }
}
```

```
    return 1;
}
computer::seqPositive* COMPUTER::prime_numbers (
    computer::Positive number,
    CORBA::Environment&
) {
    computer::Positive* buffer = computer::seqPositive::allocbuf (ARRAY_SIZE);
    int length = 0;
    buffer[0] = 1; length++;
    for (computer::Positive i=2; i<=number; i=i+1) {
        if ( is_prime_number (i) ) { buffer[length] = i; length ++; }
    }
    return new computer::seqPositive (ARRAY_SIZE,length,buffer,1);
}
computer::seqPositive* COMPUTER::prime_factors (
    computer::Positive number,
    CORBA::Environment&
) {
    computer::Positive* buffer = computer::seqPositive::allocbuf (ARRAY_SIZE);
    int length = 0;
    computer::Positive max = number / 2;
    computer::Positive rest = number;
    for (computer::Positive i=2; i<max+1; i=i+1) {
        if ( is_prime_number (i) ) {
            if ( rest % i == 0 ) {
                buffer[length] = i; length ++;
                while ( rest % i == 0 ) { rest = rest / i; }
            }
        }
        if ( rest == 1 ) break;
    }
    if (length == 0 ) { buffer[0] = number; length ++; }
    return new computer::seqPositive (ARRAY_SIZE,length,buffer,1);
}

int main (int argc, char** argv) {
    COMPUTER object ("prime");
    CORBA::Orbix.impl_is_ready ( "computerSvr" );
    return 0;
}
```

La figure A.1 illustre l'intégration de ce service dans le World Wide Web. La référence de l'objet est `computer("prime:computerSvr:duff")`. Chaque opération peut être invoquée par l'utilisateur via un formulaire.

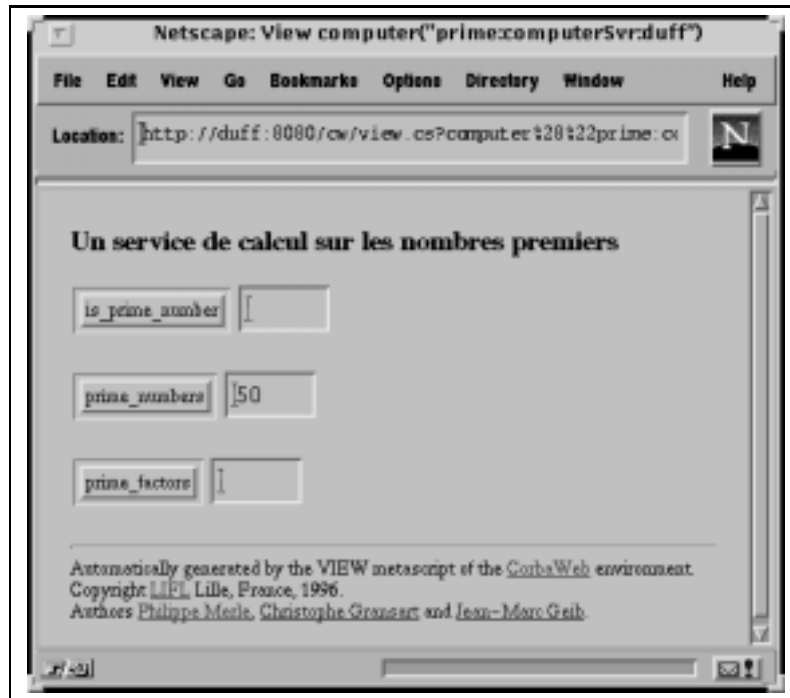


FIG. A.1 - La représentation d'un objet computer

Dans la figure A.2, l'utilisateur a demandé la liste des nombres premiers inférieurs à 50. Le résultat est automatiquement affiché par le méta scrip d'exécution d'opérations.



FIG. A.2 - L'exécution d'une opération d'un objet computer

### A.3 Un système d'informations hospitalier multimédia

Ce dernier exemple présente un mini système d'informations hospitalier multimédia (cf. figure A.3). Ce système est composé de deux services : le service d'informations hospitalier et le service de documents MIME. Le premier service contient un graphe d'objets Corba modélisant l'hôpital, les services, les radiologues, les patients, les examens et les radiographies. Le second stocke dans des objets Corba les documents multimédias au format MIME (image et son).

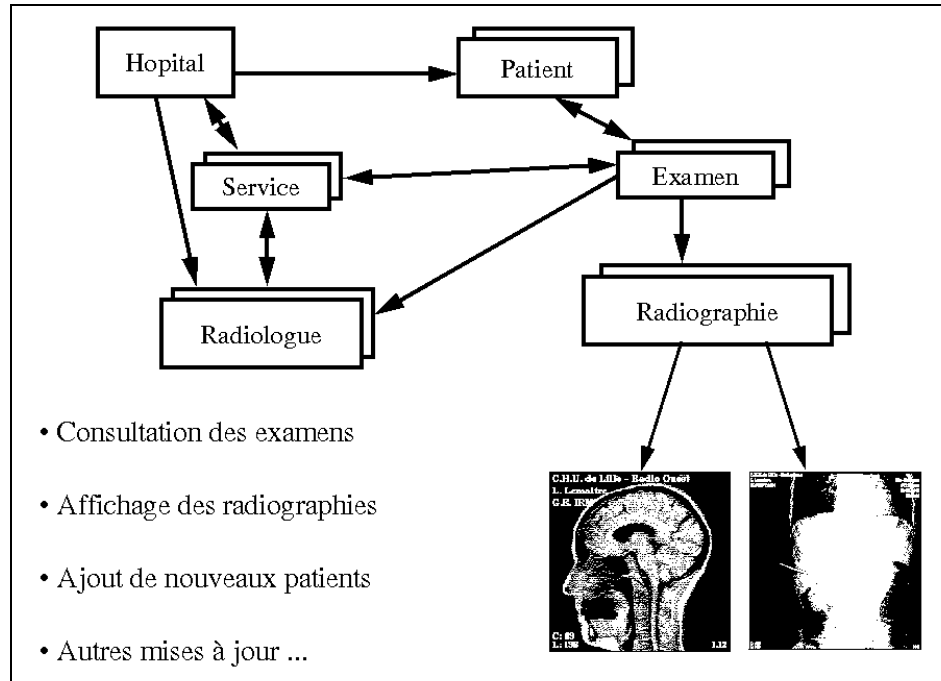


FIG. A.3 - *Le système d'informations hospitalier multimédia*

Le source IDL suivant décrit les interfaces des objets d'informations. Chaque interface possède des attributs pour interroger l'état des instances et obtenir les relations entre les objets. L'hôpital maintient quelques informations administratives (nom, adresse, téléphone) ainsi que les listes du personnel, des services et des patients.

L'interface *personne* factorise les propriétés communes aux patients et aux membres du personnel : nom, prénom, date de naissance, âge et sexe. Chaque patient référence la liste des examens qu'il a subis dans cet hôpital. Chaque membre du personnel possède une relation avec son service.

Chaque service est relié à un hôpital et maintient les listes des examens et du personnel attachées à ce service. Un examen possède des attributs décrivant l'acte médical tels que le patient, la date, la modalité, le descriptif, les commentaires, les motifs cliniques, le compte rendu, le radiologue et les radiographies. Chaque radiographie est caractérisée par des informations techniques et les références sur deux clichés radiographiques (l'image complète et une petite vignette).

Le stockage des documents multimédias dans des objets Corba nous permet d'utiliser un unique paradigme pour modéliser tout le système hospitalier. Les objets MIME possèdent des opérations pour obtenir les données multimédias. De plus, le service MIME fournit un objet «usine» pour créer les objets MIME.

```

//=====
// Description IDL d'un service MIME
//
module MIME {
    typedef string fileName;
    typedef sequence<octet> DATA;
    struct description {
        string mime_type;
        string mime_subtype;
        DATA mime_content;
    };
    interface document {
        readonly attribute string mime_type;
        readonly attribute string mime_subtype;
        readonly attribute DATA mime_content;
        description mime_document ();
    };
    interface text : document {};
    interface html : text {};
    interface image : document {};
    interface gif : image {};
    interface audio : document {};
    interface audioSun : audio {};
    interface video : document {};
//=====
    enum documentType {
        text_document, html_document,
        gif_document, audiosun_document
    };
    exception bad_file_name { string filename; };
    interface factory {
        document create_document_from_file (in fileName filename)
            raises (bad_file_name) ;
        document create_document (in documentType type, in DATA content);
    };
};

//=====
// Système d'information HOSPITALIER
//
#include <MIME.idl> // utilise le service MIME
//=====
enum type_sexe { masculin, feminin };
enum type_mois {
    janvier, fevrier, mars, avril, mai, juin, juillet,
    aout, septembre, octobre, novembre, decembre
};
struct type_date {
    unsigned short jour;
    type_mois mois;
    unsigned short annee;
};
typedef sequence<string> texte;
//=====
struct dimension_image { unsigned short x,y; };
struct dimension_fenetrage { unsigned long centre, largeur; };

```

```

struct dimension_coupe { double niveau, epaisseur; };
struct dimension_pixel { float x,y; };
//=====
module HOSPITALIER {
    interface hopital;
    interface personne;
    interface patient;
    interface personnel;
    interface radiologue;
    interface service;
    interface examen;
    interface radiographie;
    typedef sequence<patient> seqPatients;
    typedef sequence<personnel> seqPersonnels;
    typedef sequence<service> seqServices;
    typedef sequence<examen> seqExamens;
    typedef sequence<radiographie> seqRadiographies;
    //===== PERSONNE =====
    interface personne {
        attribute string nom;
        attribute string prenom;
        attribute string nom_jeune_fille;
        attribute type_date date_naissance;
        readonly attribute unsigned short age;
        attribute type_sexe sexe;
    };
    //===== PATIENT =====
    interface patient : personne {
        readonly attribute seqExamens examens;
        void nouvel_examen (in examen e);
    };
    //===== PERSONNEL =====
    interface personnel : personne {
        readonly attribute service service;
    };
    //===== RADIOLOGUE =====
    interface radiologue : personnel {};
    //===== RADIOGRAPHIE =====
    interface radiographie {
        attribute dimension_image format;
        attribute dimension_fenetrage fenetrage;
        attribute dimension_coupe coupe;
        attribute dimension_pixel taille_pixel;
        attribute MIME::gif image;
        attribute MIME::gif vignette;
    };
    //===== EXAMEN =====
    interface examen {
        readonly attribute patient patient;
        attribute type_date date;
        attribute string modalite;
        attribute string descriptif;
        attribute string commentaires;
        readonly attribute texte motifs_clinique;
        readonly attribute texte compte_rendu;
        readonly attribute radiologue radiologue;
    };
};

```

```

readonly attribute service service;
readonly attribute unsigned long nombre_images;
readonly attribute seqRadiographies images;
readonly attribute MIME::audio commentaire_sonore;
radiographie ajouter_radiographie ( in dimension_image format,
                                     in dimension_fenetrage fenetrage, in dimension_coupe coupe,
                                     in dimension_pixel taille_pixel,
                                     in MIME::image image, in MIME::image vignette );
};
//===== SERVICE =====
interface service {
    attribute string nom;
    readonly attribute hopital hopital;
    readonly attribute seqExamens examens;
    readonly attribute seqPersonnels personnels;
    examen nouvel_examen ( in patient patient, in type_date date,
                           in string modalite, in string descriptif,
                           in string commentaires, in texte motifs_clinique,
                           in texte compte_rendu, in radiologue radiologue,
                           in MIME::audio commentaire_sonore );
    void nouveau_personnel (in personnel p);
};
//===== HOPITAL =====
interface hopital {
    attribute string nom;
    attribute string adresse;
    attribute string telephone;
    readonly attribute seqPatients patients;
    readonly attribute seqServices services;
    readonly attribute seqPersonnels personnels;
    patient nouveau_patient ( in string nom, in string prenom,
                              in string nom_jeune_fille, in type_date date_naissance,
                              in type_sexe sexe );
    radiologue nouveau_radiologue ( in string nom, in string prenom,
                                    in string nom_jeune_fille, in type_date date_naissance,
                                    in type_sexe sexe, in service service );
    service nouveau_service (in string nom);
};
};

```

Les pages suivantes illustrent la navigation à travers un hôpital, un patient, un examen et finalement une radiographie. Pour chacune d'elles, nous présentons le script de représentation spécifique et une capture d'écran générée par le méta script de représentation. Ces scripts sont très semblables à ceux présentés dans le chapitre 4 : consultation de l'état de l'objet et génération d'un document HTML. La principale nouveauté se situe sur la présentation des objets MIME. La représentation de ces objets est générée par le script spécifique ci-dessous (son nom est */cw/mime.cs*). Ce script génère un document MIME en fonction de l'objet MIME désiré.

```

object = eval ( QUERY_STRING )
desc = object.mime_document ()
print "Content-type: " , desc.mime_type , '/' , desc.mime_subtype , "\n"
print "Content-Length: " , desc.mime_content.length , "\n\n"
print desc.mime_content

```

FIG. A.4 - La représentation d'un objet *HOSPITALIER::hopital*

Dans les scripts de représentation suivants, la variable *object* contient la référence sur l'objet courant.

```
HTML.H (2, "Bienvenue sur la page d'accueil de l'hopital " + object.nom)
print "<ul>\n"
  print "<li>Adresse : " , object.adresse
  print "<li>Telephone : " , object.telephone
print "</ul>\n"
HTML.H (2, "Liste des Patients")
proc display_item_patient (patient) {
  CW.object2URL (patient, patient.nom + ' ' + patient.prenom)
}
CW.generate_list (object.patients, display_item_patient)
HTML.H (2, "Liste des Services")
proc display_item_service (service) {
  CW.object2URL (service, service.nom)
}
CW.generate_list (object.services, display_item_service)
HTML.H (2, "Liste du Personnel")
proc display_item_personnel (personnel) {
  CW.object2URL (personnel, personnel.nom + ' ' + personnel.prenom)
}
CW.generate_list (object.personnels, display_item_personnel)
HTML.HR()
CW.object2URL (object, "Interface fonctionnelle de cet hopital", "/cw/inter.cs")
```

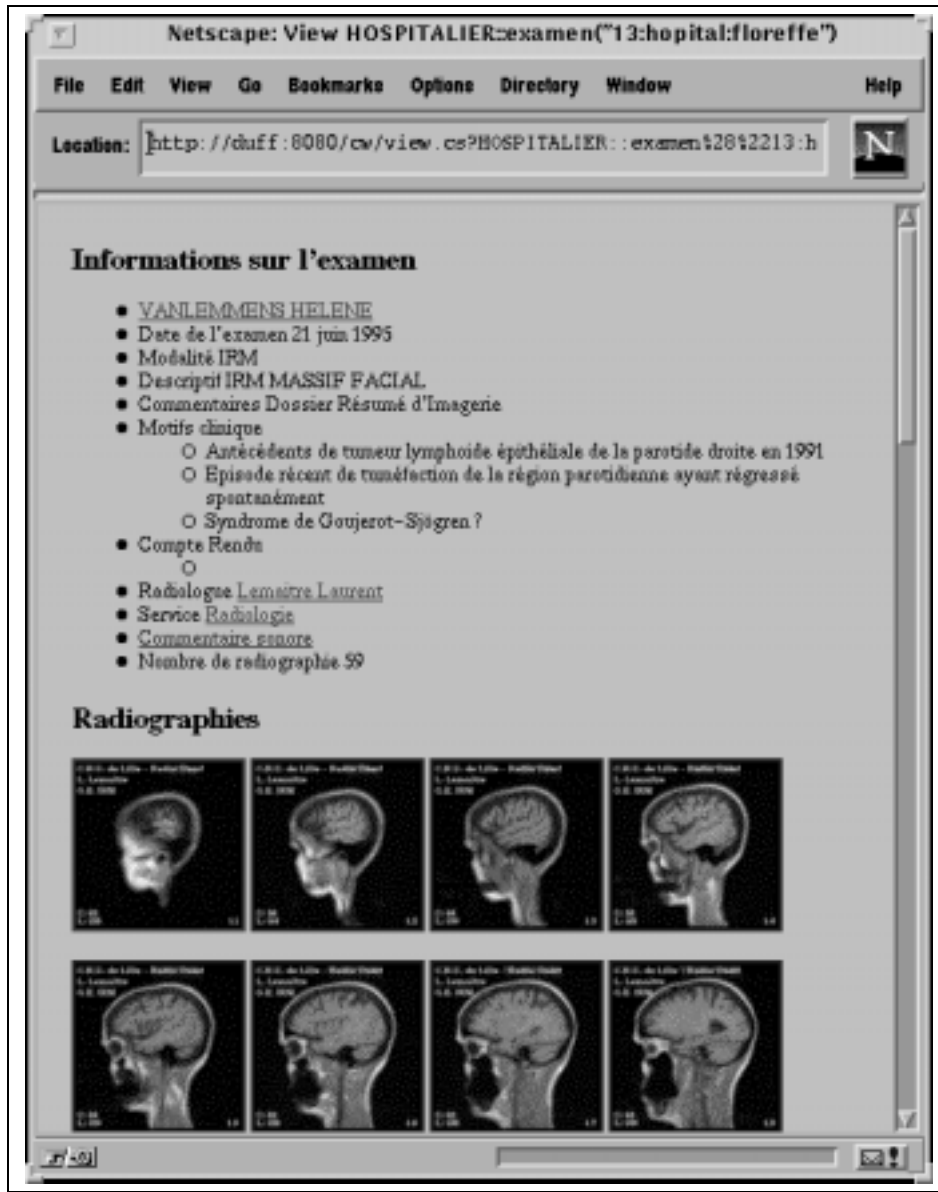


FIG. A.5 - La représentation d'un objet *HOSPITALIER::patient*

```

HTML.H (2, "Patient " + object.nom + ' ' + object.prenom )
print "<ul>\n"
  print "<li>Nom de jeune fille " , object.nom_jeune_fille
  date = object.date_naissance
  print "<li>Date de naissance ", date.jour , ' ' , date.mois , ' ' , date.annee
  print "<li>Age " , object.age
  print "<li>Sexe " , object.sexe
print "</ul>\n"
HTML.H (2, "Liste des Examens")
proc display_item_examen (examen) {
  date = examen.date
  CW.object2URL (examen, ["le ", date.jour, ' ', date.mois, ' ', date.annee] )
}
CW.generate_list (object.examens, display_item_examen)
HTML.HR()
CW.object2URL (object, "Interface fonctionnelle de ce patient", "/cw/inter.cs")

```

FIG. A.6 - La représentation d'un objet `HOSPITALIER::examen`

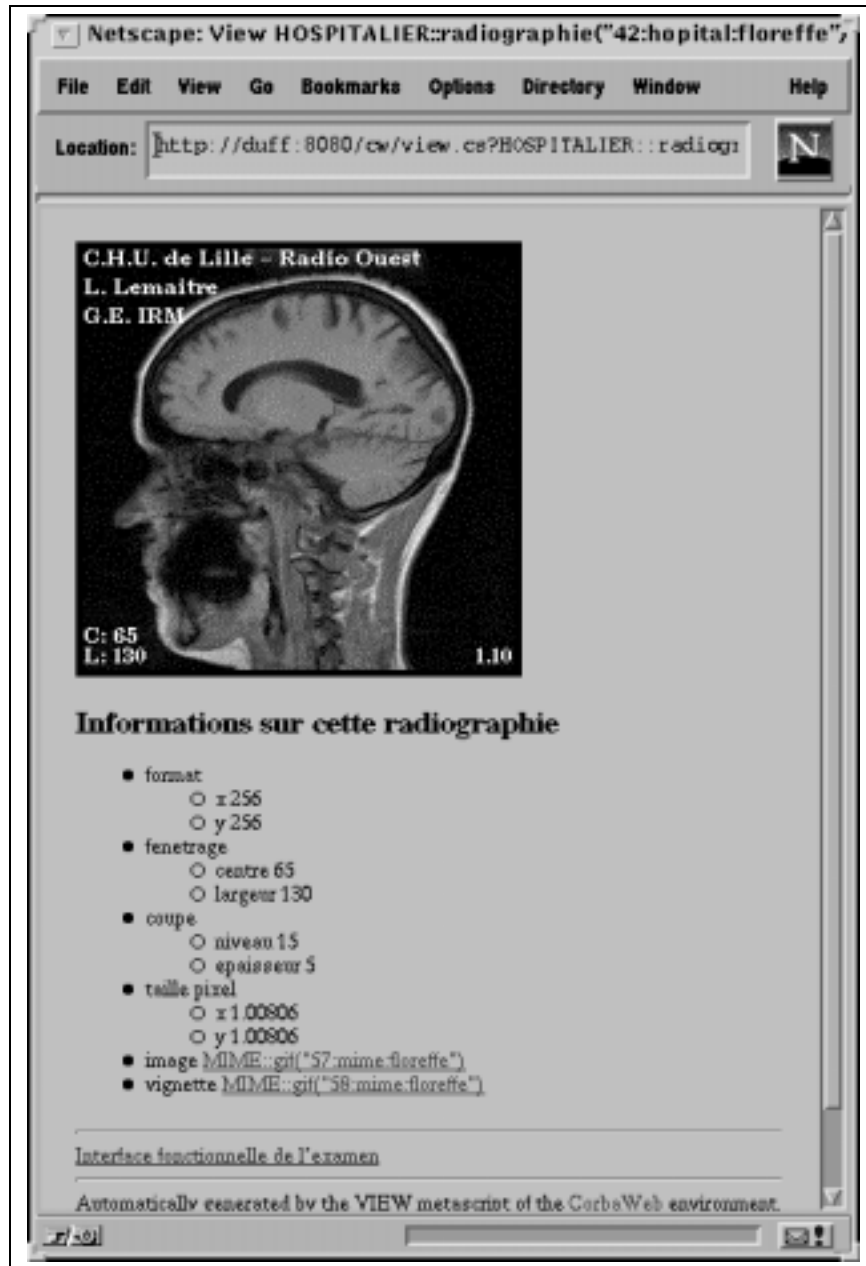
```
HTML.H (2, "Informations sur l'examen")
print "<ul>\n"
patient = object.patient
print "<li>"
CW.object2URL ( patient, [patient.nom, ' ', patient.prenom] )
date = object.date
print "<li>Date de l'examen "
print date.jour, ' ', date.mois, ' ', date.annee
print "<li>Modalit&eacute;" , object.modalite
print "<li>Descriptif " , object.descriptif
print "<li>Commentaires " , object.commentaires
print "<li>Motifs clinique " , object.motifs_clinique
print "<li>Compte Rendu " , object.compte_rendu
print "<li>Radiologue "
```

```

radiologue = object.radiologue
CW.object2URL (radiologue, [radiologue.nom, ' ', radiologue.prenom])
print "<li>Service "
service = object.service
CW.object2URL (service, service.nom)
print "<li>"
CW.object2URL (object.commentaire_sonore, "Commentaire sonore", "/cw/mime.cs")
print "<li>Nombre de radiographie " , object.nombre_images
print "</ul>\n"
HTML.H (2, "Radiographies")
images = object.images
len = images.length
i = 0
while i < len {
  j = 0
  while j < 4 and i < len {
    r = images[i]
    CW.object2URL (r, ["<IMG src=\""/cw/mime.cs?\"",
                      string2URL(r.information().vignette._toString()), "\">"] )
    print "\n"
    j = j + 1
    i = i + 1
  }
  print "<P>\n"
}
print "<HR>"
CW.object2URL (object, "Interface fonctionnelle de l'examen", "/cw/inter.cs")

```

Cet exemple illustre l'intégration d'objets d'information et d'objets multimédia dans le WWW. Cette intégration est alors totalement transparente pour l'utilisateur final et elle n'est pas très complexe pour les concepteurs de scripts. De plus, le service MIME peut être amélioré et réutilisé pour d'autres applications multimédia.

FIG. A.7 - La représentation d'un objet `HOSPITALIER::radiographie`

```

print "<IMG src=\""/cw/mime.cs?\", string2URL(object.image._toString()) , \">\"
HTML.H(2,"Informations sur cette radiographie")
print "<ul>\n"
  print "<li> format ", object.format
  print "<li> fenetrage ", object.fenetrage
  print "<li> coupe ", object.coupe
  print "<li> taille pixel ", object.taille_pixel
  print "<li> image "
  CW.object2URL (desc.image, desc.image._toString(), "/cw/mime.cs")
  print "<li> vignette "
  CW.object2URL (desc.vignette, desc.vignette._toString(), "/cw/mime.cs")
print "</ul>\n"
HTML.HR()
CW.object2URL (object, "Interface fonctionnelle de l'examen", "/cw/inter.cs")

```



# Bibliographie

- [ABLN85] G. T. Almes, Andrew P. Black, E.D. Lazowska, et J.D. Noe. The Eden System: A Technical Review. *IEEE Transactions on Software Engineering*, 11(1):43–59, Janvier 1985.
- [AML<sup>+</sup>93] Farhad Anklesaria, Mark P. McCahill, Paul Lindner, David Johnson, Daniel Torrey, et Bob Alberti. The Internet Gopher Protocol (a distributed document search and retrieval protocol). Internet Network Working Group RFC 1436, Microcomputer and Workstation Networks Center / Computer and Information Systems University of Minnesota, Mars 1993.  
<URL:<http://www.internic.net/rfc/rfc1436.txt>>.
- [Apa96a] The Apache Home Page, 1996.  
<URL:<http://www.apache.org/>>.
- [Apa96b] Apache server API. Rapport technique, The Apache Server Group, 1996.  
<URL:<http://www.apache.org/docs/API.html>>.
- [ASG<sup>+</sup>94] George Almasi, Anca Suvaiala, Cristian Goina, Calin Cascaval, et V. «Juggy» Jagannathan. TclDii: A TCL Interface to the Orbix Dynamic Invocation Interface. Rapport technique, Concurrent Engineering Research Center (CERC), 1994.  
<URL:<http://www.cerc.wvu.edu/dice/iss/TclDii/TclDiiDoc.ps>>.
- [ASM<sup>+</sup>95] Georges Almasi, Anca Suvaiala, Ion Muslea, Calin Cascaval, Tad Davis, et V. «Juggy» Jagannathan. Web\* - A Technology to Make Information Available on the Web. Dans *Proceedings of the Fourth IEEE Workshop on Enabling Technology: Infrastructure for Collaborative Enterprises: WETICE'95*. Concurrent Engineering Research Center (CERC), 95.  
<URL:<http://webstar.cerc.wvn.edu/>>.
- [BABR96] Luc Bellissard, Slim Ben Atallah, F. Boyer, et Michel Riveill. Distributed Application Configuration. Dans *Proceedings of the 16th International Conference on Distributed Computing Systems*, pages 579–585, Hong-Kong, Mai 1996. IEEE Computer Society.  
<URL:<http://sirac.inrialpes.fr/Biblio/rapports.html>>.
- [BAKR96] Luc Bellissard, Slim Ben Atallah, Alain Kerbrat, et Michel Riveill. Component-based Programming and Application Management with Olan. Dans Jean-Pierre Briot, Jean-Marc Geib, et Akinori Yonezawa, éditeurs, *Object-Based Parallel and Distributed Computation, France-Japan Workshop, OBPDC'95, Tokyo, Japan, June 1995*, volume 1107 of *Lecture Notes in Computer Science*, pages 290–309. Springer-Verlag, 1996.

- [Bar96] Laurent Barme. *Le Son dans les Systèmes Informatiques pour le Travail Coopératif*. Thèse de doctorat en informatique, Laboratoire TRIGONE, Université des Sciences et Technologies de Lille (USTL), France, 1996.
- [BBC<sup>+</sup>71] A. Bhushan, R. Braden, W. Crowther, E. Harslem, et J. Heafner. File Transfer Protocol. Internet Network Working Group RFC 172, Juin 1971.
- [BBD<sup>+</sup>91] R. Balter, J. Bernadat, D. Decouchant, A. Duda, A. Freyssinet, S. Krakowiak, M. Meysembourg, P. Le Dot, H. Nguyen Van, E. Paire, M. Riveill, C. Roisin, X. Rousset de Pina, R. Scioville, et Vandôme. Architecture and Implementation of Guide, an Object-Oriented Distributed System. *Computing Systems*, 4(1):31–67, Hiver 1991.
- [Ben87] John K. Bennet. The Design and Implementation of Distributed Smalltalk. Dans *Proceedings of Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '87), October 4–8, 1987*, pages 318–330, New York, NY 10036, U.S.A., Octobre 1987. ACM Press.
- [Ber96] Philip A. Bernstein. Middleware: A Model for Distributed System Services. *Communications of the Association for Computing Machinery (CACM)*, 39(2):86–98, Février 1996.
- [BF93] N. Borenstein et N. Freed. MIME (Multipurpose Internet Mail Extensions) Part One: Mechanisms for Specifying and Describing the Format of Internet Message Bodies. Internet Network Working Group RFC 1521, Bellcore, InnoSoft, Septembre 1993.  
<URL:<http://www.internic.net/rfc/rfc1521.txt>>.
- [BHJ<sup>+</sup>87] Andrew P. Black, Norman C. Hutchinson, Eric Jul, Henry Levy, et Larry Carter. Distribution and Abstract Types in Emerald. *IEEE Transaction on Software Engineering*, 13(1):65–76, Janvier 1987.
- [BL94] Tim Berners-Lee. Universal Resource Identifiers in WWW: A Unifying Syntax for the Expression of Names and Addresses of Objects on the Network as used in the World-Wide Web. Internet Network Working Group RFC 1630, CERN, Juin 1994.  
<URL:<http://www.internic.net/rfc/rfc1630.txt>>.
- [BLC90] Tim Berners-Lee et Robert Cailliau. World Wide Web for a HyperText project. Rapport technique, CERN European Laboratory for Particle Physics, Genève, Suisse, 1990.  
<URL:<http://www.w3.org/hypertext/WWW/Proposal.html>>.
- [BLC95] Tim Berners-Lee et Daniel W. Connolly. Hypertext Markup Language - 2.0. Internet Network Working Group RFC 1866, MIT/W3C, Novembre 1995.  
<URL:<http://www.internic.net/rfc/rfc1866.txt>>.
- [BLCL<sup>+</sup>94] Tim Berners-Lee, Robert Cailliau, A. Luotonen, Henry Frystyk Nielsen, et A. Secret. The World-Wide Web. *Communications of the Association for Computing Machinery (CACM)*, 37(8), Août 1994.
- [BLFF96] Tim Berners-Lee, Roy T. Fielding, et Henry Frystyk Nielsen. Hypertext Transfer Protocol – HTTP/1.0. Internet Network Working Group RFC 1945,

- MIT/LCS, UC Irvine, Mai 1996.  
<URL:<http://www.internic.net/rfc/rfc1945.txt>>.
- [BLMM94] Tim Berners-Lee, Larry Masinter, et Mark McCahill. Uniform Resource Locators (URL). Internet Network Working Group RFC 1738, CERN, Xerox Corporation, University of Minnesota, Décembre 1994.  
<URL:<http://www.internic.net/rfc/rfc1738.txt>>.
- [BLR94] R. Balter, S. Lacourte, et M. Riveill. The Guide Language: Design and Experience. *Computer Journal*, 37(6):519–530, Décembre 1994.
- [BN84] Andrew D. Birrell et Bruce J. Nelson. Implementing Remote Procedure Calls. *ACM Transactions on Programming Languages and Systems*, 2(1), Février 1984.
- [BN96] Marc H. Brown et Marc A. Najork. Distributed active objects. Dans *WWW5, 5th International World-Wide Web Conference, Computer Networks and ISDN System*, volume 28 Number 7-11, pages 1037–1052, CNIT – Paris La Défense, France, 6–10 Mai 1996. Elsevier.
- [BNOW94] Andrew Birrell, Greg Nelson, Susan Owicki, et Edward Wobber. Network Objects. Rapport technique, Digital Equipment Corporation Systems Research Center, 1994.
- [Bro93] Michael L. Brodie. The promise of distributed computing and the challenges of legacy information systems. Dans *Proceedings of the IFIP WG2.6 Database Semantics Conference on Interoperable Database Systems (DS-5), Lorne, Victoria, Australia, 16-20 November, 1992*, pages 1–31, 1993.
- [Bro95] Kraig Brockschmidt. *Inside OLE2*. Microsoft Press, second edition edition, 1995.
- [BW94] Andrew P. Black et Jonathan Walpole. Objects to the rescue! or *httpd*: the next generation operating system. Dans *Proceedings of the 6th ACM-SIGOPS European Workshop, Dagstuhl Castle, Warden, Germany, September, 1994*, pages 100–104, New York, NY 10036, U.S.A., Septembre 1994. ACM Press.
- [Car94] Andy Carmichael, éditeur. *Object Development Methods*. Numéro ISBN 0-9627477-9-3 in Advances in Object Technology. SIGS Books Inc., 588 Broadway, Suite 604, New York, NY 10012, U.S.A., Mai 1994.
- [CD95] A. Conta et S. Deering. Internet Control Message Protocol (ICMPv6) for the Internet Protocol Version 6 (IPv6). Internet Network Working Group RFC 1885, Digital Equipment Corporation, Xerox PARC, Décembre 1995.  
<URL:<http://www.internic.net/rfc/rfc1885.txt>>.
- [CDK94] George Coulouris, Jean Dollimore, et Tim Kindberg. *Distributed Systems: Concepts and Design*. Addison-Wesley Publishing Company Inc., second edition, 1994.  
<URL:<http://www.dcs.qmw.ac.uk/research/distrib/book.html#guide>>.
- [Cha92] D. Chappell. The OSF Distributed Computing Environment. *ConneXions*, 6(10):10–15, Octobre 1992.

- [Cha95] Thomas Chamussy. Intégration de la carte à objet multi-applicative dans CORBA. Mémoire de dea, Laboratoire d'Informatique Fondamentale de Lille (LIFL), Université des Sciences et Technologies de Lille (USTL), France, Septembre 1995.
- [CK94] R. Cartell et M. Kaufmann. *The Object Databas Standard: ODMG-93*. San Francisco, CA, U.S.A., 1994.
- [Cla92] W.J. Clark. Multipoint Multimedia Conferencing. *IEEE Communications Magazine*, Mai 1992.
- [Cor96a] Netscape Communications Corporation. Netscape Navigator 3.0 Software Development Kit. Rapport technique, Netscape Communications Corporation, U.S.A., 1996.  
<URL:[http://home.netscape.com/comprod/development\\_partners/plugin\\_api/index.html](http://home.netscape.com/comprod/development_partners/plugin_api/index.html)>.
- [Cor96b] Netscape Communications Corporation. Netscape Server API (NSAPI). Rapport technique, Netscape Communications Corporation, U.S.A., 1996.  
<URL:[http://home.netscape.com/newsref/std/server\\_api.html](http://home.netscape.com/newsref/std/server_api.html)>.
- [Cro82] David M. Crocker. Standard for the Format of ARPA Internet Text Messages. Internet Network Working Group RFC 822, University of Delaware, Dept. of Electrical Engineering, Newark, DE 19711, Août 1982.  
<URL:<http://www.internic.net/rfc/rfc822.txt>>.
- [Cro95] Pascal Croisy. *Collecticiel temps réel et apprentissage coopératif: des aspects sociaux et pédagogiques jusqu'au modèle multi-agent de l'interface de groupe*. Thèse de doctorat en informatique, Laboratoire TRIGONE, Université des Sciences et Technologies de Lille (USTL), France, 1995.
- [Dec86] D. Decouchant. A distributed object manager for the Smalltalk-80 system. Dans *Proceedings of Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '86)*, Novembre 1986.
- [DQV92] D. Decouchant, V. Quint, et I. Vatton. L'édition coopérative de documents avec Griffon. Dans *Proceedings of IHM'92 Workshop*, pages 137–141, Telecom Paris, Décembre 1992.
- [DZ83] J.D. Day et H. Zimmerman. The OSI Reference Model. Dans *Proceedings of IEEE*, volume 71, pages 1334–1340, Décembre 1983.
- [Ede92] Daniel R. Edelson. Smart Pointers: They're Smart, But They're Not Pointers. Dans *Proceedings of the C++ Conference*, pages 1–19, Portland, OR, U.S.A., Août 1992. Usenix Association.
- [Edw95a] Nigel Edwards. Object Wrapping (for WWW) – The Key to Integrated Services? The AnsaWeb Project, Document Identifier APM.1464, Architecture Projects Management Ltd, Cambridge, U.K., 1995.  
<URL:<http://www.ansa.co.uk/ANSA/ISF/1464/1464prt1.html>>.
- [Edw95b] Nigel Edwards. A Stub Compiler for CGI: The Programmer's Guide. The AnsaWeb Project, Document Identifier APM. 1465, Architecture Projects Management Ltd, Cambridge, U.K., Avril 1995.

- [EKL<sup>+</sup>96] Joseph W. Erkes, Kevin B. Kenny, John W. Lewis, Brion D. Sarachan, Michael W. Sobolewski, et Robert N. Sum. Implementing Shared Manufacturing Services on the World-Wide Web. *Communications of the Association for Computing Machinery (CACM)*, 39(2):34–45, Février 1996.
- [ER94] Nigel Edwards et Owen Rees. Distributed Objects and The World Wide Web. The AnsaWeb Project, Document Identifier APM.1283.00.08, Architecture Projects Management Ltd, Cambridge, U.K., 1994.  
<URL:<http://www.ansa.co.uk/phase3-doc-root/sponsors/APM.1283.00.08.html>>.
- [EW94] Clarence Skip Ellis et Jacques Wainer. A Conceptual Model for Groupware. Dans *Proceedings of ACM Int. Conf. on Computer-Supported Cooperative Work (CSCW'94)*, 10/94 Chapel Hill, NC, U.S.A., 1994.
- [Fil96] Jérôme Fillière. Système d'information CORBA pour la prescription dans une unité de soins intensifs. Mémoire de dea, Laboratoire d'Informatique Fondamentale de Lille (LIFL), Université des Sciences et Technologies de Lille (USTL), France, Juillet 1996.
- [Fre96] Fresco - A Fresh Approach to User Interface Systems, 1996.  
<URL:<http://www.faslab.com/fresco/HomePage.html>>.
- [Ful96] Jim Fulton. The Python Object Publisher, A Gateway Between the World-Wide Web and Python-Based Object Systems. Dans *OOSPLA96 Workshop: Toward the Integration of WWW and Distributed Object Technology, Octobre 6 1996, U.S.A.* Digital Creations, L.C., 1996.
- [GldOA92] Daniel Gilly et Inc. l'équipe de O'Reilly & Associates. *UNIX in a Nutshell*. O'Reilly, Sebastapol, Californie, U.S.A., second edition, revised and expanded for SVR4 and solaris 2.0 edition, Juin 1992.
- [GM95] James Gosling et Henry McGilton. The Java Language Environment: a White Paper. Technical report, Sun Microsystems Inc., 7550 Garcia Avenue, Mountain View, CA 94043 U.S.A., Octobre 1995.  
<URL:<http://java.sun.com/whitePaper/java-whitepaper-1.html>>.
- [GR83] Adele Goldberg et David Robson. *SMALLTALK-80. The Language and Its Implementation*. Addison-Wesley Publishing Company Inc., Massachusetts, U.S.A., 1983.
- [Gra95] Christophe Gransart. *BOX: Un Modèle et un Langage à Objets pour la Programmation Parallèle et Distribuée*. Thèse de doctorat en informatique, Laboratoire d'Informatique Fondamentale de Lille (LIFL), Université des Sciences et Technologies de Lille (USTL), France, Janvier 1995.
- [GX92] Object Management Group et X/Open. Object Management Architecture Guide. OMG TC Document 92-11-1, Object Management Group, Headquarters, 492 Old Connecticut Path, Framingham, MA 01701, U.S.A., Septembre 1992.
- [GX93] Object Management Group et X/Open. The Common Object Request Broker: Architecture and Specification. OMG TC Document 93-12-43, Object Management Group, Headquarters, 492 Old Connecticut Path, Framingham, MA 01701, U.S.A., Décembre 1993.

- [HK94] Edwin E. Hastings et Dilip H. Kumar. Providing Customers Information Using the WEB and CORBA. Dans *The Second World-Wide Web Conference '94: Mosaic and the Web*, Octobre 1994.
- [Hoo95] Frédéric Hoogstoel. *Une approche organisationnelle du travail coopératif assisté par ordinateur. Application au projet CO-LEARN*. Thèse de doctorat en informatique, Laboratoire TRIGONE, Université des Sciences et Technologies de Lille (USTL), France, 1995.
- [Hor96] Emmanuel Horckmans. Un modèle de contrôle d'accès pour les espaces d'information orientés objets et hétérogènes du type Web-Corba-Carte. Mémoire de dea, Laboratoire d'Informatique Fondamentale de Lille (LIFL), Université des Sciences et Technologies de Lille (USTL), France, Juillet 1996.
- [HvST96] Philip Homburg, Maarten van Steen, et Andrew S. Tanenbau. An Architecture for A Scalable Wide Area Distributed System. Dans *Proceedings of the Seventh SIGOPS European Workshop*, Connemara, Ireland, Septembre 1996. ACM.
- [II94] IONA et Isis. An Introduction to Orbix+Isis. Rapport technique, IONA Technologies Ltd. and Isis Distributed Systems, 1994.
- [ION95a] IONA. *Orbix 2: Programming Guide*. IONA Technologies Ltd., 8-34 Percy Place, Dublin 4, Ireland, 2.0 edition, Novembre 1995.
- [ION95b] IONA. *Orbix 2: Reference Guide*. IONA Technologies Ltd., 8-34 Percy Place, Dublin 4, Ireland, 2.0 edition, Novembre 1995.
- [ION95c] IONA. The Orbix Architecture. Rapport technique, IONA Technologies Ltd., 1995.  
<URL:<http://www.iona.com/>>.
- [ION96a] IONA. OrbixNames: The Orbix 2.0 Naming Service. Rapport technique, IONA Technologies Ltd., 8-34 Percy Place, Dublin 4, Ireland, Août 1996.
- [ION96b] IONA. *OrbixWeb: Programmer's Guide*. IONA Technologies Ltd., 8-34 Percy Place, Dublin 4, Ireland, 1996.  
<URL:[http://www.iona.com](http://www.iona.com/)>.
- [ISO86] ISO International Organization for Standardization, Genève, Suisse. *International Standard ISO/IEC 8879: Information Processing – Text and Office Systems - Standard Generalized Markup Language (SGML)*, 1986.
- [ISO92] ISO International Organization for Standardization, Genève, Suisse. *International Standard ISO/IEC 9075: Information Technology — Database — languages — SQL*, third edition, Novembre 1992.
- [ISO94] ISO International Organization for Standardization, Genève, Suisse. *International Standard ISO/IEC 7816: Integrated circuit(s) cards with contacts – Part 3: electronic signals and transmission protocols*, Novembre 1994.
- [Jan95] Bill Janssen. *Using ILU with Python: A Tutorial*. Xerox Palo Alto Research Center (Xerox PARC), Mai 1995.

- [Jan96] Bill Janssen. The Inter-Language Unification – ILU, 1996.  
<URL:ftp://ftp.parc.xerox.com/pub/ilu/ilu.html>.
- [JLHB88] Eric Jul, Henry Levy, Norman C. Hutchinson, et Andrew P. Black. Fine-Grained Mobility in the Emerald System. *ACM Transactions on Computer Systems*, 6(1):109–133, Février 1988.
- [Kha94] Raman Khanna, éditeur. *Distributed computing: Implementation and Management Strategies*. PTR Prentice Hall, 1994.
- [KL86] Brian Kantor et Phil Lapsley. Network News Transfer Protocol: A Proposed Standard for the Stream-Based Transmission of News. Internet Network Working Group RFC 977, U.C. San Diego, U.C. Berkeley, Février 1986.  
<URL:http://www.internic.net/rfc/rfc977.txt>.
- [Kun95] John A. Kunze. Functional Requirements for Internet Resource Locators. Internet Network Working Group RFC 1736, IS&T, UC Berkeley, Février 1995.  
<URL:http://www.internic.net/rfc/rfc1736.txt>.
- [Lin94] P.F. Linington. Introduction to the Open Distributed Processing Reference Model. Dans J. De Meer, V. Heymer, et R. Roth, éditeurs, *Proceedings of IFIP TC6/WF6.4 International Workshop on Open Distributed Processing, Berlin, Allemagne*, pages 3–13. Elsevier Science Publishers B.V. (North Holland), 8–11 Octobre 1994.
- [Lis85] Barbara H. Liskov. The Argus language and system. Dans *Lecture Notes in Computer Science (Distributed Systems: methods and tools for specification)*. Springer-Verlag, 1985.
- [Lut96] Mark Lutz. *Programming Python: Object-Oriented Scripting*. Numéro ISBN: 1-56592-197-6. O'Reilly & Associates, Octobre 1996.
- [LY96] Tim Lindholm et Frank Yellin. *The Java Virtual Machine Specification*. The Java Series. Addison-Wesley Publishing Company Inc., Massachusetts, U.S.A., Juillet 1996.
- [Ma94] James G. Mitchell et al. An Overview of the Spring System. Dans *Proceedings of Comcon Spring 1994*, Février 1994.  
<URL:http://www.sun.com/tech/projects/spring/papers/overview.ps>.
- [Mar96] Open Market. FastCGI, 1996.  
<URL:http://www.fastcgi.com>.
- [McC95] R. McCool. *The Common Gateway Interface*. National Center for Supercomputing Applications, University of Illinois at Urbana-Champaign, U.S.A., 1995.  
<URL:http://hoohoo.ncsa.uiuc.edu/cgi/overview.html>.
- [Mer96] Philippe Merle. How to Make Corba Objects User-Friendly with a Generic Object-Oriented Dynamic Environment? Dans *Travail Coopératif et Communication Avancée, Journée Jeunes Chercheurs Ganymède*, Lille, France, Mai 1996.

- [Mey92] Bertrand Meyer. *Eiffel: The Language*. Prentice-Hall, Englewood Cliffs, NJ 07632, U.S.A., 1992.
- [MGG96a] Philippe Merle, Christophe Gransart, et Jean-Marc Geib. CorbaScript and CorbaWeb: A Generic Object-Oriented Dynamic Environment upon CORBA. Dans *Proceedings of TOOLS Europe'96, Palais des Congrès, Paris, France, February 26-29, 1996*. Prentice-Hall, Février 1996.
- [MGG96b] Philippe Merle, Christophe Gransart, et Jean-Marc Geib. CorbaWeb: A Generic Object Navigator. Dans *The Fifth International World Wide Web Conference (WWW5), Paris, France, May 6-10, 1996*, volume 28 (7-11), pages 1269-1281, Paris, France, Mai 1996. Elsevier, Computer Networks and ISDN System.
- [MGG96c] Philippe Merle, Christophe Gransart, et Jean-Marc Geib. CorbaWeb: A WWW and Corba Worlds Integration. Dans *The 2th COOTS, Workshop on Distributed Object Computing on the Internet*, Toronto, Canada, Juin 1996.
- [MGG96d] Philippe Merle, Christophe Gransart, et Jean-Marc Geib. Future Trends for a Global Object Web. Dans *Joint W3C/OMG Workshop on Distributed Objects and Mobile Code, Boston, Massachusetts, June 24-25, 1996*. OMG/W3C, Juin 1996.
- [MGG96e] Philippe Merle, Christophe Gransart, et Jean-Marc Geib. How to Make Corba Objects User-Friendly with a Generic Object-Oriented Dynamic Environment? Dans *European Conference on Object-Oriented Programming, Workshop on Putting Distributed Objects to Work*, Linz, Austria, Juillet 1996.
- [Mic95] Microsoft. The ActiveX Resources Area, 1995.  
<URL:<http://www.microsoft.com/activex/>>.
- [MNC<sup>+</sup>91] Gerald Masini, Amedeo Napoli, Dominique Colnet, Daniel Léonard, et Karl Tombre. *Les Langages à objets*. InterEditions, France, 1991.
- [Moc87a] P. Mockapetris. Domain Names - Concepts and Facilities. Internet Network Working Group RFC 1034, USC/Information Sciences Institute, Novembre 1987.  
<URL:<http://www.internic.net/rfc/rfc1034.txt>>.
- [Moc87b] P. Mockapetris. Domain Names - Implementation and Specification. Internet Network Working Group RFC 1035, USC/Information Sciences Institute, Novembre 1987.  
<URL:<http://www.internic.net/rfc/rfc1035.txt>>.
- [Moo93] K. Moore. MIME (Multipurpose Internet Mail Extensions) Part Two: Message Header Extensions for Non-ASCII Text. Internet Network Working Group RFC 1522, University of Tennessee, Septembre 1993.  
<URL:<http://www.internic.net/rfc/rfc1522.txt>>.
- [Mot96] Motorola. The IRIDIUM Project Home Page, 1996.  
<URL:<http://www.iridium.com/>>.

- [MRTS90] Sape J. Mullender, G. Van Rossum, Andrew S. Tanenbam, et H. Van Staveren. Amoeba: A Distributed Operating System for the 1990s. *IEEE Computer*, 23(5):44–53, Mai 1990.
- [MZ95] Thomas J. Mowbray et Ron Zahavi. *The Essential CORBA: Systems Integration Using Distributed Objects*. Numéro ISBN 0-471-10611-9. Object Management Group and John Wiley & Sons, Inc., 1995.
- [Nel81] Bruce J. Nelson. *Remote Procedure Call*. PhD thesis, Carnegie-Mellon University, U.S.A., Mai 1981.
- [Nel91] Greg Nelson. *Systems Programming with Modula-3*. Prentice-Hall, Englewood Cliffs, NJ 07632, U.S.A., 1991.
- [Net96] Welcome to Netscape, 1996.  
<URL:<http://home.netscape.com/>>.
- [NT95] Oscar Nierstrasz et Dennis Tsichritzis, éditeurs. *Object-Oriented Software Composition*. The Object-Oriented Series. Prentice-Hall, Englewood Cliffs, NJ 07632, U.S.A., 1995.  
*ISBN 0-13-220674-9*.
- [O2W96] O2 Technology Web Server, 1996.  
<URL:<http://www.o2tech.fr>>.
- [OHE95] Robert Orfali, Dan Harkey, et Jiri Edwards. *Essential Distributed Objects Survival Guide*. John Wiley and Sons Inc., New York, U.S.A., 1995.
- [OHE96] Robert Orfali, Dan Harkey, et Jiri Edwards. *Objets Répartis Guide de Survie*. International Thomson Publishing France, Paris, France, 1996.  
*ISBN 2-84180-043-1, Traduction française de [OHE95]*.
- [OLW96] John K. Ousterhout, Jacob Y. Levy, et Brent B. Welch. The Safe-Tcl Security Model. Rapport technique, Sun Microsystems Laboratories, 7550 Garcia Avenue, Mountain View, CA 94043 U.S.A., Novembre 1996.  
<URL:<http://www.sunlabs.com/research/tcl/safeTcl.ps>>.
- [OMG91] Object Management Group. The Common Object Request Broker: Architecture and Specification. OMG TC Document 91-12-1, Object Management Group, Headquarters, 492 Old Connecticut Path, Framingham, MA 01701, U.S.A., Décembre 1991.
- [OMG93] Object Management Group. Object Services Architecture. OMG TC Document 93-2-1, Object Management Group, Headquarters, 492 Old Connecticut Path, Framingham, MA 01701, U.S.A., Février 1993.
- [OMG94a] Object Management Group. Common Facilities Architecture. OMG TC Document 94-9-40, Object Management Group, Headquarters, 492 Old Connecticut Path, Framingham, MA 01701, U.S.A., Septembre 26, 1994.
- [OMG94b] Object Management Group. *Common Object Services Specifications*, volume 1. John Wiley and Sons Inc., New York, U.S.A., 1994.
- [OMG94c] Object Management Group. *Common Object Services Specifications*, volume 2. John Wiley and Sons Inc., New York, U.S.A., 1994.

- [OMG94d] Object Management Group. IDL C++ Language Mapping Specification. OMG TC Document 94-9-4, Object Management Group, Headquarters, 492 Old Connecticut Path, Framingham, MA 01701, U.S.A., Septembre 1994.
- [OMG95a] Object Management Group. Common Facilities Architecture. OMG TC Document 95-1-2, Object Management Group, Headquarters, 492 Old Connecticut Path, Framingham, MA 01701, U.S.A., Janvier 3, 1995.
- [OMG95b] Object Management Group. The Common Object Request Broker: Architecture and Specification. Rapport technique, Object Management Group, Headquarters, 492 Old Connecticut Path, Framingham, MA 01701, U.S.A., Juillet 1995.
- [OMG95c] Object Management Group. *CORBA services: Common Object Service Specification*. Object Management Group, Headquarters, 492 Old Connecticut Path, Framingham, MA 01701, U.S.A., Mars 31, 1995.
- [OMG95d] Object Management Group. IDL - Ada Language Mapping Specification (Version 1.0). OMG TC Document 95-5-16, Object Management Group, Headquarters, 492 Old Connecticut Path, Framingham, MA 01701, U.S.A., Mai 22, 1995.
- [Ous94] John K. Ousterhout. *TCL and the TK Toolkit*. Numéro ISBN: 0-201-63337-X. Addison-Wesley Publishing Company Inc., 1994.
- [OWS96] Oracle Corporation Home Page, 1996.  
<URL:<http://www.oracle.com>>.
- [Pax95] Vern Paxson. *Flex, version 2.5*, Mars 1995.
- [Pes96] Mark D. Pesce. VRML architecture group, 1996.  
<URL:<http://vag.vrml.org>>.
- [PHS96] Irfan Pyarali, Timothy H. Harrison, et Douglas C. Schmidt. Design and Performance of an Object-Oriented Framework for High-Performance Electronic Medical Imaging. Dans *Proceedings of the Second Conference on Object-Oriented Technologies and Systems (COOTS), Toronto, Canada, June 17-21, 1996*, pages 191–208. USENIX Association, Juin 1996.
- [Pos80] Jonathan B. Postel. User Datagram Protocol. Internet Network Working Group RFC 768, Information Sciences Institute, University of Southern California, 4676 Admiralty Way, Marina del Rey, California 90291, Août 1980.  
<URL:<http://www.internic.net/rfc/rfc768.txt>>.
- [Pos81] Jonathan B. Postel. Internet Protocol. Internet Network Working Group RFC 791, Information Sciences Institute, University of Southern California, 4676 Admiralty Way, Marina del Rey, California 90291, Septembre 1981.  
<URL:<http://www.internic.net/rfc/rfc791.txt>>.
- [Pos82] Jonathan B. Postel. Simple Mail Transfer Protocol. Internet Network Working Group RFC 821, Information Sciences Institute, University of Southern California, 4676 Admiralty Way, Marina del Rey, California 90291, Août 1982.  
<URL:<http://www.internic.net/rfc/rfc821.txt>>.

- [Pos94] Jonathan B. Postel. Domain Name System Structure and Delegation. Internet Network Working Group RFC 1591, Information Sciences Institute, University of Southern California, 4676 Admiralty Way, Marina del Rey, California 90291, Mars 1994.  
<URL:<http://www.internic.net/rfc/rfc1591.txt>>.
- [PR83] Jonathan B. Postel et Joyce Reynolds. Telnet Protocol Specification. Internet Network Working Group RFC 854, Information Sciences Institute, University of Southern California, 4676 Admiralty Way, Marina del Rey, California 90291, Mai 1983.  
<URL:<http://www.internic.net/rfc/rfc854.txt>>.
- [PR85] Jonathan B. Postel et Joyce Reynolds. File Transfer Protocol (FTP). Internet Network Working Group RFC 959, USC/Information Sciences Institute, University of Southern California, 4676 Admiralty Way, Marina del Rey, California 90291, Octobre 1985.  
<URL:<http://www.internic.net/rfc/rfc959.txt>>.
- [Pro97] CoManDOS Project. CoManDOS – Construction and Management of Distributed Office Systems. Final Report on the Global Architecture Esprit project 834, Septembre 1997.
- [PSS94] F. Pacull, A. Sandoz, et A. Schiper. Duplex: A distributed collaborative editing environment in large scale. Dans *Proceedings of ACM Int. Conf. on Computer-Supported Cooperative Work (CSCW'94)*, 10/94 Chapel Hill, NC, U.S.A., 1994.
- [REM<sup>+</sup>95] Owen Rees, Nigel Edwards, Mark Madsen, Mike Beasley, et Ashley McCleaghnan. A Web of Distributed Objects. Dans *WWW4, 4th International World-Wide Web Conference*, Boston, MA, U.S.A., Décembre 1995.  
<URL:<http://www.w3.org/pub/Conferences/WWW4/Papers/85/>>.
- [RHKP95] Sanjay R. Radia, Graham Hamilton, Peter B. Kessler, et Michael L. Powell. The Spring Object Model. Dans *Proceedings of USENIX Conference on Object-Oriented Technologies*, Monterey CA, U.S.A., Juin 1995.
- [RKF92] W. Rosenberry, D. Kenney, et G. Fisher. *Understanding DCE*. O'Reilly, Sebastapol, Californie, U.S.A., 1992.
- [RNP93] Sanjay R. Radia, Michael N. Nelson, et Michael L. Powell. The Spring Name Service. Rapport Technique SMLI-93-16, Sun Microsystems Laboratories, Octobre 1993.  
<URL:<http://www.sun.com/tech/projects/spring/papers/>>.
- [Rob96] David Robinson. The WWW Common Gateway Interface Version 1.1. IETF draft – Work in progress (expires July 8, 1996), University of Cambridge, Janvier 8, 1996.  
<URL:<http://www.ast.cam.ac.uk/drtr/cgi-spec.html>>.
- [Rol95] Pierre Rolin. *Réseaux haut débit*, volume ISBN 2-86601-491-X. Editions HERMES, Paris, France, Octobre 1995.
- [Ros93] M.T. Rose. *The Internet Message: Closing the Book with Electronic Mail*. Prentice-Hall, Englewood Cliffs, NJ 07632, U.S.A., 1993.

- [Rou96] François Rouaix. A Web navigator with applets in Caml. Dans *WWW5, 5th International World-Wide Web Conference, Computer Networks and ISDN System*, volume 28 Number 7-11, pages 1365–1371, CNIT – Paris La Défense, France, 6–10 Mai 1996. Elsevier.
- [Rud93] Alan W. Rudge. I'll be seeing you. *IEEE Review*, pages 235–238, Novembre 1993.
- [SGH<sup>+</sup>89] Marc Shapiro, Yvon Gourhant, Sabine Habert, Laurence Mosseri, Michel Ruffin, et Céline Valot. SOS: An Object-Oriented Operating System — Assessment and Perspectives. *Computing Systems*, 2(4):287–338, Décembre 1989.
- [Sha86] Marc Shapiro. Structure and Encapsulation in Distributed Systems: The Proxy Principe. Dans *Proceedings of the 6th International Conference on Distributed Computer Systems*, pages 198–204, Cambridge, Mass., U.S.A., Mai 1986. IEEE.
- [Sie96a] Jon M. Siegel. *CORBA Fundamentals and Programming*. Numéro ISBN: 0-471-12148-7. OMG and John Wiley and Sons, Inc., New York, U.S.A., 1996.
- [Sie96b] Jon M. Siegel. OMG: Building the Object Technology Infrastructure. Dans *Tutoriel à IFIP/IEEE International Conference on Distributed Platforms (ICDP'96)*, Dresden, Germany, 27 Février - 1 Mars 1996.
- [Sil95] Maffei Silvano. *Run-Time Support for Object-Oriented Distributed Programming*. PhD thesis, Department of Computer Science, Université de Zurich, Suisse, 1995.  
<URL:<http://www.olsen.ch/maffei/electra.html>>.
- [SM94] Karen Sollins et Larry Masinter. Functional Requirements for Uniform Resource Names. Internet Network Working Group RFC 1737, MIT/LCS, Xerox Corporation, Décembre 1994.  
<URL:<http://www.internic.net/rfc/rfc1737.txt>>.
- [SMI96] Sun Microsystems Inc. Jeeves Alpha2 Documentation, 1996.  
<URL:<http://java.sun.com/products/jeeves/alpha2/doc/index.html>>.
- [Sol94a] Richard Mark Soley. Object Management Architecture Guide. OMG TC Document 94-11-1, Object Management Group, Headquarters, 492 Old Connecticut Path, Framingham, MA 01701, U.S.A., Novembre 1994.
- [Sol94b] Richard Mark Soley. *Role of object technology in distributed systems*, pages 469–478. In [Kha94], 1994.
- [Str87] Bjarne Stroustrup. The Evolution of C++ 1985 to 1987. Dans *Usenix C++ Workshop Proceedings*, pages 1–22. Usenix Association, Novembre 1987.
- [Sun89] Sun Microsystems, Inc. NFS: Network File System Protocol specification. Internet Network Working Group RFC 1094, Network Information Center, SRI International, Mars 1989.  
<URL:<http://www.internic.net/rfc/rfc1094.txt>>.

- [Tan90] Andrew Tanenbaum. *RESEAUX: Architectures, protocoles, applications*. InterEditions, 1990.
- [Tho95] D. Thompson. NCSA Mosaic Common Client Interface. Rapport technique, National Center for Supercomputing Applications, University of Illinois at Urbana-Champaign, U.S.A., 1995.  
<URL:<http://yahoo.ncsa.uiuc.edu/mosaic/ci-spec.html>>.
- [Van97] Jean-Jacques Vandewalle. *Le projet OSMOSE (Operating System and MOBILE Services): Modélisation, implantation et interopérabilité de services carte à microprocesseur par l'approche orientée objet*. Thèse de doctorat en informatique, Laboratoire d'Informatique Fondamentale de Lille (LIFL), Université des Sciences et Technologies de Lille (USTL), France, 1997.
- [Vin93] Steve Vinoski. Distributed Object Computing with CORBA. *C++ Report*, 5(6):32–38, Août 1993.
- [Vis96] Visigenic. VisiBroker for Java: Reference Manuals, 1996.  
<URL:[http://www.visigenic.com/BW/reference\\_manuals.html](http://www.visigenic.com/BW/reference_manuals.html)>.
- [vR96] Guido van Rossum. Python Language Home Page, 1996.  
<URL:<http://www.python.org/>>.
- [vSHT96] Maarten van Steen, Franz J. Hauck, et Andrew S. Tanenbaum. A Model for Worldwide Tracking of Distributed Objects. Dans *Proceedings of TINA '96 Conference*, Heidelberg, Germany, Septembre 1996. Eurescom.
- [vSHvD<sup>+</sup>95] Maarten van Steen, Philip Homburg, L. van Doorn, Andrew S. Tanenbaum, et W. de Jonge. Towards Object-based Wide Area Distributed Systems. Dans L.-F. Cabrera et M. Theimer, éditeurs, *Proceedings of the Fourth International Workshop on Object Orientation in Operating Systems*, pages 224–227, Lund, Sweden, Août 1995. IEEE.
- [W3C96a] W3C. Jigsaw Overview, 1996.  
<URL:<http://www.w3.org/pub/WWW/Jigsaw>>.
- [W3C96b] W3C. The World Wide Web Consortium, 1996.  
<URL:<http://www.w3.org/pub/WWW>>.
- [WRW96] Ann Wollrath, Roger Riggs, et Jim Waldo. A Distributed Object Model for the JAVA System. Dans *Proceedings of the Second Conference on Object-Oriented Technologies and Systems (COOTS), Toronto, Canada, June 17–21, 1996*, pages 219–231. USENIX Association, 1996.
- [WS91] Larry Wall et Randal L. Schwartz. *Programming Perl*. O'Reilly, Sebastapol, Californie, U.S.A., 1991.
- [WvRA96] Aaron Watters, Guido van Rossum, et James Ahlstrom. *Internet Programming with Python*. Numéro ISBN: 1-55851-484-8. MIS Press/Henry Holt publishers, Octobre 1996.
- [YD96] Zhonghua Yang et Keith Duddy. CORBA: A Platform for Distributed Object Computing. *ACM Operating Systems Review*, 30(2):4–31, Avril 1996.

- [You96] Fouad Yousfi. *PLACO: Modélisation par Workflow et Conception d'un Système de Planification Coopérative, Application aux Unités de Soins*. Thèse de doctorat en informatique, Laboratoire d'Informatique Fondamentale de Lille (LIFL), Université des Sciences et Technologies de Lille (USTL), France, Mars 1996.