

Intergiciel

et Construction d'Applications Réparties

19 janvier 2007

Distribué sous licence Creative Commons :
<http://creativecommons.org/licenses/by-nc-nd/2.0/fr/deed.fr>

1. Introduction à l'intergiciel	S. Krakowiak ¹
2. Patrons et canevas pour l'intergiciel	S. Krakowiak
3. Le modèle de composants Fractal	T. Coupaye ² , V. Quéma ⁸ , L. Seinturier ³ , J.-B. Stefani ⁷
4. Les services Web	M. Dumas ⁴ , M.-C. Fauvet ¹
5. La plate-forme J2EE	P. Déchamboux ²
6. La plate-forme .NET	M. Riveill ⁵ , A. Beugnard ⁶ , D. Emsellem ⁵
7. La plate-forme dynamique de services OSGi	D. Donsez ¹

¹Université Joseph Fourier, Grenoble

²France Telecom R&D

³Université Pierre et Marie Curie et INRIA

⁴Queensland University of Technology, Brisbane (Australie)

⁵Université de Nice - Sophia Antipolis

⁶École Nationale Supérieure des Télécommunications de Bretagne

⁷INRIA

⁸CNRS et projet Sardes (LIG-INRIA)

Table des matières

Préface	ix
1 Introduction à l'intergiciel	1
1.1 Pourquoi l'intergiciel ?	1
1.2 Quelques classes d'intergiciel	6
1.3 Un exemple simple d'intergiciel : RPC	8
1.3.1 Motivations et besoins	8
1.3.2 Principes de réalisation	9
1.3.3 Développement d'applications avec le RPC	12
1.3.4 Résumé et conclusions	14
1.4 Problèmes et défis de la conception de l'intergiciel	15
1.4.1 Problèmes de conception	15
1.4.2 Quelques principes d'architecture	16
1.4.3 Défis de l'intergiciel	18
1.4.4 Plan de l'ouvrage	19
1.5 Note historique	19
2 Patrons et canevas pour l'intergiciel	21
2.1 Services et interfaces	21
2.1.1 Mécanismes d'interaction de base	22
2.1.2 Interfaces	24
2.1.3 Contrats et conformité	25
2.2 Patrons architecturaux	29
2.2.1 Architectures multiniveaux	29
2.2.2 Objets répartis	33
2.3 Patrons pour l'intergiciel à objets répartis	35
2.3.1 <i>Proxy</i>	35
2.3.2 <i>Factory</i>	37
2.3.3 <i>Pool</i>	38
2.3.4 <i>Adapter</i>	39
2.3.5 <i>Interceptor</i>	40
2.3.6 Comparaison et combinaison des patrons	41
2.4 Adaptabilité et séparation des préoccupations	42
2.4.1 Protocoles à méta-objets	42
2.4.2 Programmation par aspects	44

2.4.3	Approches pragmatiques	46
2.4.4	Comparaison des approches	47
2.5	Note historique	47
3	Le système de composants Fractal	49
3.1	Le modèle Fractal : historique, définition et principes	49
3.1.1	Composants Fractal	51
3.1.2	Contrôleurs	55
3.2	Plates-formes	55
3.2.1	Julia	56
3.2.2	AOKell	59
3.2.3	Autres plates-formes	63
3.3	Fractal ADL	63
3.3.1	Le langage extensible	64
3.3.2	L'usine extensible	64
3.3.3	Extension de Fractal ADL	66
3.4	Bibliothèques de composants	67
3.4.1	Dream	67
3.4.2	Autres bibliothèques	71
3.5	Comparaison	72
3.6	Conclusion	73
4	Les Services Web	77
4.1	Services Web : historique, définition, et principes	77
4.2	Modélisation de services Web	79
4.2.1	Points de vue dans la modélisation de services	79
4.2.2	Relations entre points de vue	83
4.3	Description de services Web	84
4.3.1	Description fonctionnelle et structurelle : WSDL	85
4.3.2	Description comportementale	87
4.3.3	Description d'aspects non-fonctionnels : « WS-Policy »	89
4.4	Implantation de services Web	90
4.4.1	SOAP (<i>Simple Object Access Protocol</i>)	91
4.4.2	Extensions de SOAP : spécifications WS-*	92
4.4.3	Le débat SOAP vs. REST	94
4.4.4	Exemples d'implantations de SOAP	96
4.4.5	Implantation à base d'un langage spécifique : BPEL	97
4.5	Perspectives	99
4.5.1	Fouille d'interface, test de conformité et adaptation	99
4.5.2	Transactions	100
4.5.3	Sélection dynamique	101
5	La plate-forme J2EE	103
5.1	Introduction	103
5.1.1	Historique	104
5.1.2	J2EE, pour quoi faire ?	104

5.1.3	Principes d'architecture	105
5.1.4	Modèles de programmation et conteneurs	107
5.1.5	Les acteurs dans l'environnement J2EE : organisation des rôles	109
5.1.6	Références utiles	111
5.2	Approche à composants	111
5.2.1	Modèle de composants	111
5.2.2	Empaquetage de composants et d'applications	112
5.2.3	Principes d'assemblage	114
5.2.4	Déploiement d'application	116
5.3	Applications réparties et intégration	118
5.3.1	Appel de procédure à distance	118
5.3.2	Communication asynchrone : JMS	119
5.3.3	Connecteurs au standard JCA	119
5.3.4	Connecteurs de base	120
5.4	Gestion des transactions	121
5.4.1	Notion de transaction	121
5.4.2	Manipulation des transactions dans un serveur J2EE	122
5.4.3	Mise en relation des composants J2EE avec le système transactionnel	123
5.5	Programmation de l'étage de présentation Web	123
5.5.1	Principes d'architecture des composants de présentation Web	124
5.5.2	<i>Servlets</i> : organisation de la pile protocolaire	124
5.5.3	<i>Servlets</i> HTTP	128
5.5.4	JSP : construction de pages HTML dynamiques	130
5.6	Programmation de l'étage métier	132
5.6.1	Principes d'architecture des composants métier	132
5.6.2	Composants de session	134
5.6.3	Composants réactifs	137
5.6.4	Composants entité	138
5.6.5	Gestion des événements relatifs aux composants métier	139
5.7	Conclusion et perspectives	142
6	La plate-forme .NET	145
6.1	Historique, définitions et principes	145
6.1.1	Généralité	145
6.1.2	Implantations	147
6.2	Architecture générale et concepts	147
6.2.1	CLR	147
6.2.2	MSIL et JIT	148
6.2.3	Sécurité et déploiement des composants	150
6.2.4	Composants .NET	151
6.2.5	Assemblage de composants .NET	153
6.2.6	Le langage C#	155
6.3	Applications distribuées	156
6.3.1	Le canevas .NET Remoting	156
6.3.2	Les services Web en .NET	163

6.3.3	L'Asynchronisme	167
6.4	Composants et services techniques	168
6.4.1	Accès aux données	169
6.4.2	Sécurité	170
6.4.3	Transaction	173
6.5	Exemple : Mise en œuvre d'une application complète	175
6.5.1	Architecture générale	175
6.5.2	Implémentation et déploiement des services	176
6.5.3	Client du service .NET Remoting (3)	180
6.5.4	Implementation du Service WEB (6)	181
6.5.5	Ecriture du client du Service web	182
6.6	Evaluation	183
6.6.1	.Net et les langages à objets	183
6.6.2	Compositions multi-langages en .Net	185
6.6.3	Quelques éléments de performances	187
6.6.4	La plate-forme .NET 2.0 et Indigo : nouveautés à venir	190
6.7	Conclusion	192
7	La plate-forme dynamique de services OSGi	195
7.1	Introduction	195
7.2	Motivations	196
7.3	La plate-forme de déploiement et d'exécution de services Java	197
7.3.1	Conditionnement et déploiement des applications OSGi	198
7.3.2	Service	201
7.4	Programmation des services OSGi	203
7.4.1	Activation	204
7.4.2	Enregistrement de services	205
7.4.3	Courtage et liaison de services	205
7.4.4	Interception des demandes de liaison	207
7.4.5	Événements et observateurs	210
7.4.6	Modèles de composants	212
7.5	Services Standard	215
7.5.1	Http Service	215
7.5.2	Wire Admin	217
7.5.3	UPnP Device Driver	218
7.6	Conclusion et Perspectives	222
8	Conclusion	225
A	Annexes	227
A	Les Services Web en pratique	229
A.1	Préambule	229
A.2	Beehive et Axis	230
A.3	Déploiement d'un service	230
A.4	Déploiement d'un client	231

A.5	Un service avec une opération paramétrée, et un client	232
A.6	Perfect Glass Inc. et l'entrepôt	233
A.6.1	Version 1 : paramètres simples	233
A.6.2	Version 2 : paramètres complexes	233
A.7	Composition de services	234
A.7.1	Composition à base d'opérations bi-directionnelles	234
A.7.2	Composition à base d'opérations uni-directionnelles	234
A.8	Ressources	235
A.8.1	Arborescence de <code>webserv_prat_1/</code>	235
A.8.2	Code du service <code>TimeNow</code>	236
A.8.3	Code du client <code>ClientTimeNow</code>	237
A.8.4	Code du service <code>Hello</code>	237
A.8.5	Code du client <code>ClientHelloInParam</code>	238
A.8.6	Code du programme <code>qDemandeeLocal</code>	239
A.8.7	Code du programme <code>EntrepotLocal</code>	239
A.8.8	Code des procédures de <code>Saisie</code>	240
A.8.9	Code du client <code>ClientHelloObjet</code>	241
A.8.10	Code de la classe <code>Personne</code>	242
B	.NET en pratique	245
B.1	Les outils à utiliser	245
B.2	Architecture de l'application “Cartes de Visite”	246
B.3	Les objets métiers	247
B.4	La couche d'accès aux données	247
B.5	La couche métier	248
B.6	Premières manipulations en C#	249
B.6.1	Construire le diagramme UML	249
B.6.2	Générer les fichiers de documentation	249
B.6.3	Compilation du code	251
B.6.4	Utilisation de la réflexivité	251
B.6.5	Sérialisation binaire et XML	252
B.7	La couche de stockage physique des données	254
B.8	Accès direct à la base de données par un client léger	254
B.9	La couche d'accès aux données	256
B.10	La logique métier : <code>AnnuaireService.dll</code>	257
B.11	Utilisation depuis un client local	257
B.12	Publications des services	258
B.12.1	Publication .NET Remoting	258
B.12.2	Publication WEB Service	259
B.13	Utilisation de ces services	260
B.13.1	Par un client lourd	260
B.13.2	Par un client léger	261
B.14	Pour poursuivre le TP	262
C	OSGI en pratique	265

D AOKell : une réalisation réflexive du modèle Fractal	267
D.1 Préliminaires	267
D.1.1 Logiciels nécessaires	267
D.1.2 AOKell 2.0	268
D.1.3 Test de l'installation	268
D.2 Contrôleur de journalisation orienté objet	269
D.2.1 Contrôleur objet faiblement couplé	270
D.2.2 Contrôleur objet fortement couplé	274
D.3 Contrôleur de journalisation orienté composant	278
E Joram : un intergiciel de communication asynchrone	283
E.1 Introduction	283
E.1.1 Les intergiciels à messages (MOM)	283
E.1.2 JMS (<i>Java Messaging Service</i>)	284
E.1.3 Ce que JMS ne dit pas et ne fait pas	287
E.1.4 JORAM : une mise en œuvre d'un service de messagerie JMS	287
E.2 Le modèle de programmation JMS	288
E.2.1 Les abstractions de JMS	289
E.2.2 Principe de fonctionnement	289
E.2.3 Messages	290
E.2.4 Objets JMS	291
E.2.5 Communication en mode Point à point	293
E.2.6 Communication en mode <i>Publish/Subscribe</i>	293
E.2.7 JMS par l'exemple	293
E.3 Architecture de JORAM	297
E.3.1 Principes directeurs et choix de conception	297
E.3.2 Architecture logique	297
E.3.3 Architecture centralisée	299
E.3.4 Architecture distribuée	301
E.3.5 JORAM : une plate-forme JMS configurable	301
E.4 JORAM : fonctions avancées	303
E.4.1 Répartition de charge	303
E.4.2 Fiabilité et haute disponibilité	305
E.4.3 Connectivité élargie	305
E.4.4 Sécurité	308
E.5 JORAM : bases technologiques	308
E.5.1 Les agents distribués	308
E.5.2 JORAM : une plate-forme JMS à base d'agents	310
E.6 Conclusion	310
E.7 Références	314
F Speedo : un système de gestion de la persistance	317
F.1 Introduction	317
F.1.1 Qu'est-ce que Speedo ?	317
F.1.2 Principales fonctions de Speedo	318
F.1.3 Restrictions actuelles	320

F.1.4	Organisation du cours Speedo	320
F.2	Gestion d'objets persistants avec JDO : les principes	320
F.2.1	Classes et objets persistants	321
F.2.2	Déclinaison Speedo des principes JDO	322
F.3	Chaine de production de programme	323
F.3.1	Phases de la production de programme	323
F.3.2	Utilisation de Ant	324
F.4	Programmation d'objets Java persistants	326
F.4.1	Exécuter un programme utilisant Speedo	326
F.4.2	Utiliser le système de persistance JDO	329
F.4.3	Identifiants d'objet persistant	332
F.4.4	Rendre un objet persistant	335
F.4.5	Détruire un objet persistant	337
F.4.6	Relations et cohérence	338
F.4.7	Fonctions avancées	339
F.5	Requêtes ensemblistes	342
F.5.1	Introduction	342
F.5.2	Requêtes prédefinies	345
F.5.3	Filtre de requête	346
F.5.4	Cardinalité du résultat	349
F.5.5	Résultat complexe	350
F.5.6	Agrégats et <i>group by</i>	352
F.6	Projection vers une base de données relationnelle	353
F.6.1	Projection des identifiants d'objet persistant	353
F.6.2	Projection de la classe	355
F.6.3	Projection d'attributs simples	356
F.6.4	Projection de relations mono-valuées	356
F.6.5	Projection de relations multi-valuées	359
F.6.6	Projection de liens d'héritage	362
F.7	Intégration dans J2EE	366
F.7.1	Aucune intégration	366
F.7.2	Intégration JTA	366
F.7.3	Intégration JCA / JTA	367
F.8	Conclusion et perspectives	367
G	Infrastructure M2M pour la gestion de l'énergie	369
Glossaire		371
Bibliographie		373

Préface

Le domaine de l'intergiciel (*middleware*), apparu dans les années 1990, a pris une place centrale dans le développement des applications informatiques réparties. L'intergiciel joue aujourd'hui, pour celles-ci, un rôle analogue à celui d'un système d'exploitation pour les applications centralisées : il dissimule la complexité de l'infrastructure sous-jacente, il présente une interface commode aux développeurs d'applications, il fournit un ensemble de services communs.

Les acteurs de l'intergiciel sont nombreux et divers : les organismes de normalisation, les industriels du logiciel et des services, les utilisateurs d'applications. Les consortiums et organisations diverses qui développent et promeuvent l'usage du logiciel libre tiennent une place importante dans le monde de l'intergiciel. Enfin, ce domaine est l'objet d'une recherche active, dont les résultats sont rapidement intégrés. Une conférence scientifique annuelle ACM-IFIP-Usenix, *Middleware*, est consacrée à l'intergiciel, et de nombreux ateliers spécialisés sont fréquemment créés sur des sujets d'actualité.

Le domaine de l'intergiciel est en évolution permanente. Pour répondre à des besoins de plus en plus exigeants, les produits doivent s'adapter et s'améliorer. De nouveaux domaines s'ouvrent, comme l'informatique ubiquitaire, les systèmes embarqués, les systèmes mobiles. Le cycle de cette évolution est rapide : chaque année apporte une transformation significative du paysage des normes et des produits de l'intergiciel.

Malgré la rapidité de ces changements, il est possible de dégager quelques principes architecturaux qui guident la conception et la construction de l'intergiciel et des applications qui l'utilisent. Ces principes sont eux-mêmes développés et enrichis grâce aux résultats de la recherche et à l'expérience acquise, mais présentent une certaine stabilité.

Cet ouvrage vise à présenter une vue d'ensemble des systèmes intergiciels, sous la forme d'études de cas des principaux standards du domaine et de leur mise en œuvre pour la construction d'application réparties. Cette présentation s'attache aux aspects architecturaux : on n'y trouvera pas les détails de la dernière version de tel ou tel produit. Pour faciliter la compréhension des aspects communs, chaque chapitre consacré à un standard ou à une plate-forme met en évidence le traitement spécifique de chacun de ces aspects : architecture d'ensemble du système, composition, programmation, conditionnement et déploiement des applications. En outre, un chapitre initial présente quelques principes généraux de construction applicables à l'intergiciel. Le domaine de la composition d'applications, qui tient une place importante, est présenté au travers de l'étude d'un modèle innovant issu de la recherche.

Pour illustrer des aspects techniques plus spécialisés, une part substantielle de l'ouvrage est constituée d'annexes. Certaines illustrent la mise en œuvre concrète des systèmes

présentés. D'autres donnent un aperçu de systèmes ou d'applications qui n'ont pas trouvé leur place dans le corps de l'ouvrage.

L'ouvrage ne couvre pas toutes les phases du cycle de vie des applications réparties. L'accent est mis sur les méthodes et les outils pour la *construction* de ces applications. Les phases antérieure (*conception*) et postérieure (*administration*) ne sont pas abordées. Chacun de ces aspects justifierait une étude de même ampleur que celle ici présentée.

L'ouvrage s'adresse à toute personne intéressée par l'intergiciel, aux concepteurs, développeurs et utilisateurs d'applications réparties, ainsi qu'aux étudiants en informatique (niveau Master ou école d'ingénieurs) et aux chercheurs désirant aborder le domaine de l'intergiciel. Nous supposons que le lecteur a des connaissances générales de base dans le domaine des applications informatiques, des systèmes d'exploitation et des réseaux. La pratique du langage Java, sans être requise, facilite la lecture des exemples.

Ce livre est issu d'une série d'écoles d'été organisées sur le thème de l'intergiciel et de la construction d'applications réparties (Saint-Malo 1996, Autrans 1998, 1999, 2003 et 2006). La forme et le contenu de ces écoles se sont naturellement adaptés pour suivre l'évolution du domaine. Depuis 2003, l'école a pris le nom d'ICAR (Intergiciel et Construction d'Applications Réparties). Ce livre reprend largement le contenu de l'école ICAR 2006. Les annexes sont notamment inspirées d'ateliers organisés au cours de cette école.

Nous tenons à remercier les organismes qui nous ont soutenus pour l'organisation d'ICAR 2006 : l'IMAG (Informatique et Mathématiques Appliquées de Grenoble), l'INRIA (Institut National de Recherche en Informatique et Automatique, unité de recherche de Rhône-Alpes), et ObjectWeb [ObjectWeb 1999], consortium développant des systèmes intergiciels en logiciel libre. Nous remercions également les organismes d'appartenance des organisateurs et intervenants d'ICAR 2006 : France Telecom R&D, INRIA, Institut National Polytechnique de Grenoble, Queensland University of Technology (Australie), Scalagent Distributed Technologies, Université Joseph Fourier, Université de Nice-Sophia Antipolis.

Sans avoir directement participé à la rédaction, Alan Schmitt et Christophe Taton sont intervenus dans ICAR 2006. Danièle Herzog a assuré avec efficacité le soutien logistique de cette école.

L'ouvrage a bénéficié d'interactions avec de nombreux collègues dont la liste serait trop longue pour n'oublier personne. Mentionnons néanmoins, parmi les intervenants et les organisateurs d'écoles précédentes, Daniel Hagimont (IRIT, Toulouse) et Philippe Merle (INRIA et LIFL, Lille).

Les organisateurs d'ICAR 2006

Noël De Palma (Institut National Polytechnique de Grenoble)

Sacha Krakowiak (Université Joseph Fourier)

Jacques Mossière (Institut National Polytechnique de Grenoble)

Chapitre 1

Introduction à l'intergiciel

Ce chapitre présente les fonctions de l'intergiciel, les besoins auxquels elles répondent, et les principales classes d'intergiciel. L'analyse d'un exemple simple, celui de l'appel de procédure à distance, permet d'introduire les notions de base relatives aux systèmes intergiciels et les problèmes que pose leur conception. Le chapitre se termine par une note historique résumant les grandes étapes de l'évolution de l'intergiciel.

1.1 Pourquoi l'intergiciel ?

Faire passer la production de logiciel à un stade industriel grâce au développement de composants réutilisables a été un des premiers objectifs identifiés du génie logiciel. Transformer l'accès à l'information et aux ressources informatiques en un service omniprésent, sur le modèle de l'accès à l'énergie électrique ou aux télécommunications, a été un rêve des créateurs de l'Internet. Bien que des progrès significatifs aient été enregistrés, ces objectifs font encore partie aujourd'hui des défis pour le long terme.

Dans leurs efforts pour relever ces défis, les concepteurs et les réalisateurs d'applications informatiques rencontrent des problèmes plus concrets dans leur pratique quotidienne. Les brèves études de cas qui suivent illustrent quelques situations typiques, et mettent en évidence les principaux problèmes et les solutions proposées.

Exemple 1 : réutiliser le logiciel patrimonial. Les entreprises et les organisations construisent maintenant des systèmes d'information globaux intégrant des applications jusqu'ici indépendantes, ainsi que des développements nouveaux. Ce processus d'intégration doit tenir compte des applications dites *patrimoniales* (en anglais : *legacy*), qui ont été développées avant l'avènement des standards ouverts actuels, utilisent des outils « propriétaires », et nécessitent des environnements spécifiques.

Une application patrimoniale ne peut être utilisée qu'à travers une interface spécifiée, et ne peut être modifiée. La plupart du temps, l'application doit être reprise telle quelle car le coût de sa réécriture serait prohibitif.

Le principe des solutions actuelles est d'adopter une norme commune, non liée à un

langage particulier, pour interconnecter différentes applications. La norme spécifie des interfaces et des protocoles d'échange pour la communication entre applications. Les protocoles sont réalisés par une couche logicielle qui fonctionne comme un bus d'échanges entre applications, également appelé courtier (en anglais *broker*). Pour intégrer une application patrimoniale, il faut développer une *enveloppe* (en anglais *wrapper*), c'est-à-dire une couche logicielle qui fait le pont entre l'interface originelle de l'application et une nouvelle interface conforme à la norme choisie.

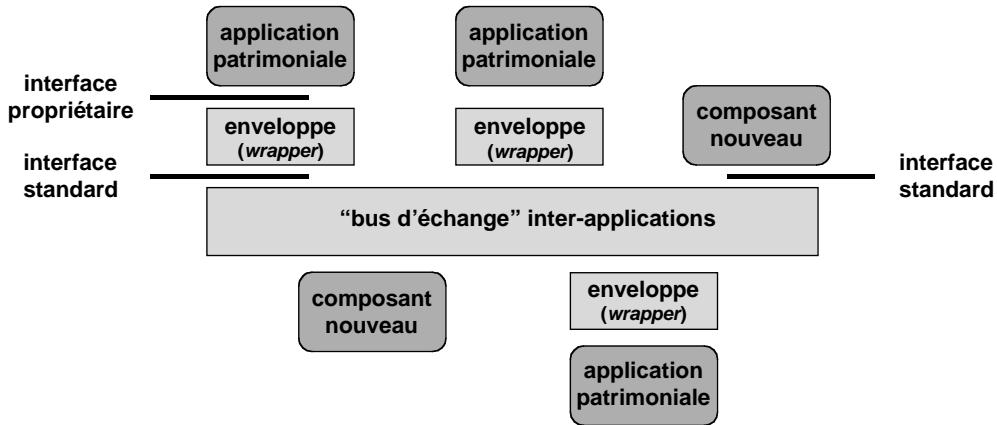


Figure 1.1 – Intégration d'applications patrimoniales

Une application patrimoniale ainsi « enveloppée » peut maintenant être intégrée avec d'autres applications du même type et avec des composants nouveaux, en utilisant les protocoles normalisés du courtier. Des exemples de courtiers sont CORBA, les files de messages, les systèmes à publication et abonnement (en anglais *publish-subscribe*). On en trouvera des exemples dans ce livre.

Exemple 2 : systèmes de médiation. Un nombre croissant de systèmes prend la forme d'une collection d'équipements divers (capteurs ou effecteurs) connectés entre eux par un réseau. Chacun de ces dispositifs remplit une fonction locale d'interaction avec le monde extérieur, et interagit à distance avec d'autres capteurs ou effecteurs. Des applications construites sur ce modèle se rencontrent dans divers domaines : réseaux d'ordinateurs, systèmes de télécommunications, équipements d'alimentation électrique permanente, systèmes décentralisés de production.

La gestion de tels systèmes comporte des tâches telles que la mesure continue des performances, l'élaboration de statistiques d'utilisation, l'enregistrement d'un journal, le traitement des signaux d'alarme, la collecte d'informations pour la facturation, la maintenance à distance, le téléchargement et l'installation de nouveaux services. L'exécution de ces tâches nécessite l'accès à distance au matériel, la collecte et l'agrégation de données, et la réaction à des événements critiques. Les systèmes réalisant ces tâches s'appellent *systèmes de médiation*¹.

¹Ce terme a en fait un sens plus général dans le domaine de la gestion de données ; nous l'utilisons ici

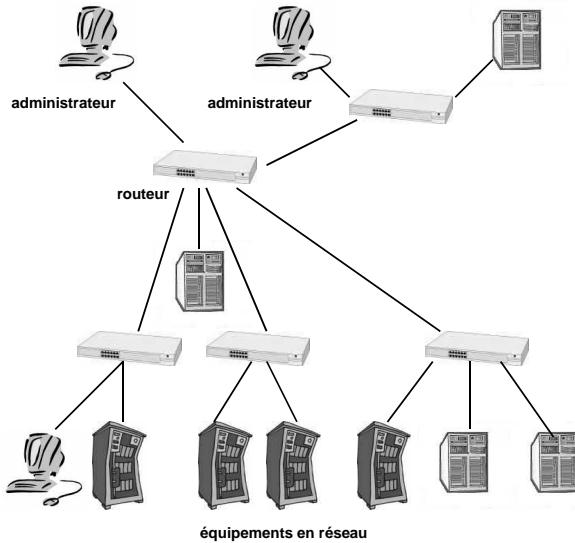


Figure 1.2 – Surveillance et commande d'équipements en réseau

L'infrastructure interne de communication d'un système de médiation doit réaliser la collecte de données et leur acheminement depuis ou vers les capteurs et effecteurs. La communication est souvent déclenchée par un événement externe, comme la détection d'un signal d'alarme ou le franchissement d'un seuil critique par une grandeur observée.

Un système de communication adapté à ces exigences est un *bus à messages*, c'est-à-dire un canal commun auquel sont raccordées les diverses entités (Figure 1.2). La communication est asynchrone et peut mettre en jeu un nombre variable de participants. Les destinataires d'un message peuvent être les membres d'un groupe prédéfini, ou les « abonnés » à un sujet spécifié.

Exemple 3 : architectures à composants. Le développement d'applications par composition de « briques » logicielles s'est révélé une tâche beaucoup plus ardue qu'on ne l'imaginait initialement [McIlroy 1968]. Les architectures actuelles à base de composants logiciels reposent sur la séparation des fonctions et sur des interfaces standard bien définies. Une organisation répandue pour les applications d'entreprise est l'architecture à trois étages (*3-tier*²), dans laquelle une application est composée de trois couches : entre l'étage de présentation, responsable de l'interface avec le client, et l'étage de gestion de bases de données, responsable de l'accès aux données permanentes, se trouve un étage de traitement (*business logic*) qui réalise les fonctions spécifiques de l'application. Dans cet étage intermédiaire, les aspects propres à l'application sont organisés comme un ensemble de « composants », unités de construction pouvant être déployées indépendamment.

Cette architecture repose sur une infrastructure fournissant un environnement de déploiement et d'exécution pour les composants ainsi qu'un ensemble de services communs tels que la gestion de transactions et la sécurité. En outre, une application pour un domaine

dans le sens restreint indiqué

²La traduction correcte de *tier* est « étage » ou « niveau ».

particulier (par exemple télécommunications, finance, avionique, etc.) peut utiliser une bibliothèque de composants développés pour ce domaine.

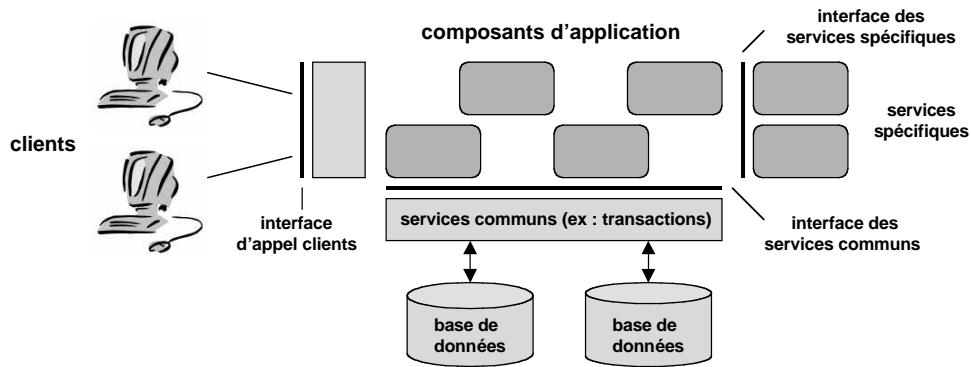


Figure 1.3 – Un environnement pour applications à base de composants

Cette organisation a les avantages suivants :

- elle permet aux équipes de développement de se concentrer sur les problèmes propres à l'application, les aspects génériques et l'intendance étant pris en charge par les services communs ;
- elle améliore la portabilité et l'interopérabilité en définissant des interfaces standard ; ainsi on peut réutiliser un composant sur tout système fournissant les interfaces appropriées, et on peut intégrer du code patrimonial dans des « enveloppes » qui exportent ces interfaces ;
- elle facilite le passage à grande échelle, en séparant la couche d'application de la couche de gestion de bases de données ; les capacités de traitement de ces deux couches peuvent être séparément augmentées pour faire face à un accroissement de la charge.

Des exemples de spécifications pour de tels environnements sont les *Enterprise JavaBeans* (EJB) et la plate-forme .NET, examinés dans la suite de ce livre.

Exemple 4 : adaptation de clients par des mandataires. Les utilisateurs accèdent aux applications sur l'Internet via des équipements dont les caractéristiques et les performances couvrent un spectre de plus en plus large. Entre un PC de haut de gamme, un téléphone portable et un assistant personnel, les écarts de bande passante, de capacités locales de traitement, de capacités de visualisation, sont très importants. On ne peut pas attendre d'un serveur qu'il adapte les services qu'il fournit aux capacités locales de chaque point d'accès. On ne peut pas non plus imposer un format uniforme aux clients, car cela reviendrait à les aligner sur le niveau de service le plus bas (texte seul, écran noir et blanc, etc.).

La solution préférée est d'interposer une couche d'adaptation, appelée *mandataire* (en anglais *proxy*) entre les clients et les serveurs. Un mandataire différent peut être réalisé pour chaque classe de point d'accès côté client (téléphone, assistant, etc.). La fonction du mandataire est d'adapter les caractéristiques du flot de communication depuis ou vers le client aux capacités de son point d'accès et aux conditions courantes du réseau. À

cet effet, le mandataire utilise ses propres ressources de stockage et de traitement. Les mandataires peuvent être hébergés sur des équipements dédiés (Figure 1.4), ou sur des serveurs communs.

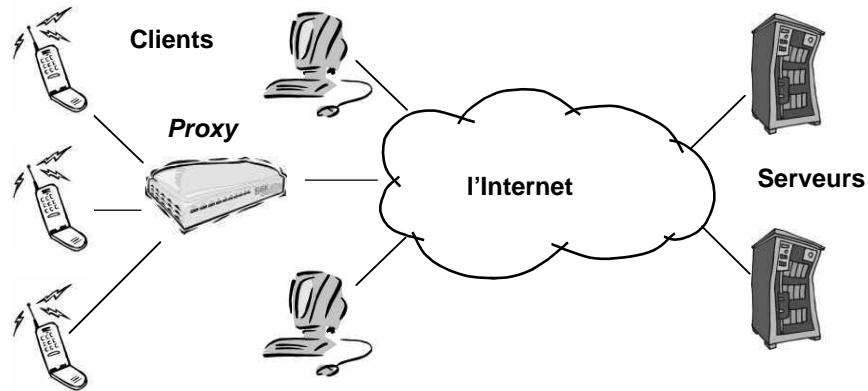


Figure 1.4 – Adaptation des communications aux ressources des clients par des mandataires

Des exemples d'adaptation sont la compression des données pour réagir à des variations de bande passante du réseau, la réduction de la qualité des images pour prendre en compte des capacités réduites d'affichage, le passage de la couleur aux niveaux de gris. Un exemple de mise en œuvre de l'adaptation par mandataires est décrit dans [Fox et al. 1998].

Les quatre études de cas qui précèdent ont une caractéristique commune : dans chacun des systèmes présentés, des applications utilisent des logiciels de niveau intermédiaire, installés au-dessus des systèmes d'exploitation et des protocoles de communication, qui réalisent les fonctions suivantes :

1. cacher la *répartition*, c'est-à-dire le fait qu'une application est constituée de parties interconnectées s'exécutant à des emplacements géographiquement répartis ;
2. cacher l'*hétérogénéité* des composants matériels, des systèmes d'exploitation et des protocoles de communication utilisés par les différentes parties d'une application ;
3. fournir des *interfaces* uniformes, normalisées, et de haut niveau aux équipes de développement et d'intégration, pour faciliter la construction, la réutilisation, le portage et l'interopérabilité des applications ;
4. fournir un ensemble de *services* communs réalisant des fonctions d'intérêt général, pour éviter la duplication des efforts et faciliter la coopération entre applications.

Cette couche intermédiaire de logiciel, schématisée sur la Figure 1.5, est désignée par le terme générique d'*intergiciel* (en anglais *middleware*). Un intergiciel peut être à usage général ou dédié à une classe particulière d'applications.

L'utilisation de l'intergiciel présente plusieurs avantages dont la plupart résultent de la capacité d'abstraction qu'il procure : cacher les détails des mécanismes de bas niveau, assurer l'indépendance vis-à-vis des langages et des plates-formes, permettre de réutiliser l'expérience et parfois le code, faciliter l'évolution des applications. En conséquence, on

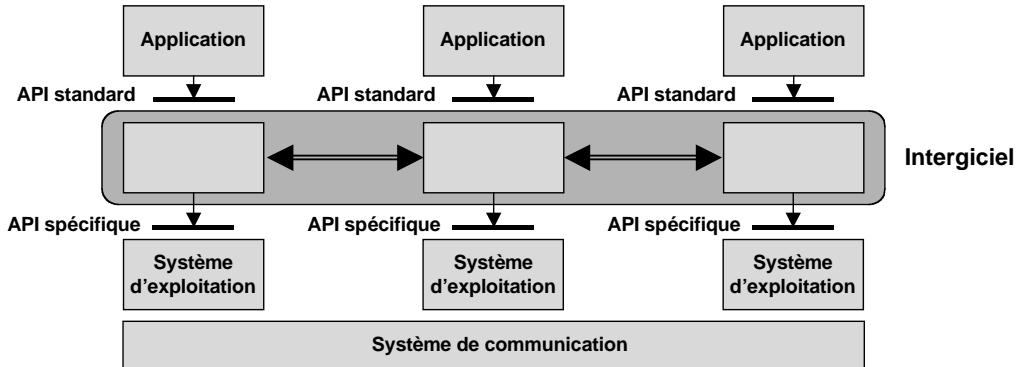


Figure 1.5 – Organisation de l'intergiciel

peut espérer réduire le coût et la durée de développement des applications (l'effort étant concentré sur les problèmes spécifiques et non sur l'intendance), et améliorer leur portabilité et leur interopérabilité.

Un inconvénient potentiel est la perte de performances liée à la traversée de couches supplémentaires de logiciel. L'utilisation de techniques intergicielles implique par ailleurs de prévoir la formation des équipes de développement.

1.2 Quelques classes d'intergiciel

Les systèmes intergiciels peuvent être classés selon différents critères, prenant en compte les propriétés de l'infrastructure de communication, l'architecture d'ensemble des applications, les interfaces fournies.

Caractéristiques de la communication. L'infrastructure de communication sous-jacente à un intergiciel est caractérisée par plusieurs propriétés qui permettent une première classification.

1. *Topologie fixe ou variable.* Dans un système de communication fixe, les entités communicantes résident à des emplacements fixes, et la configuration du réseau ne change pas (ou bien ces changements sont des opérations peu fréquentes, prévues à l'avance). Dans un système de communication mobile, tout ou partie des entités communicantes peuvent changer de place, et se connecter ou se déconnecter dynamiquement, y compris pendant le déroulement des applications.
2. *Caractéristiques prévisibles ou imprévisibles.* Dans certains systèmes de communication, on peut assurer des bornes pour des facteurs de performance tels que la gigue ou la latence. Néanmoins, dans beaucoup de situations pratiques, ces bornes ne peuvent être garanties, car les facteurs de performance dépendent de la charge de dispositifs partagés tels que les routeurs ou les voies de transmission. Un système de communication est dit *synchrone* si on peut garantir une borne supérieure pour le temps

de transmission d'un message ; si cette borne ne peut être établie, le système est dit *asynchrone*³.

Ces caractéristiques se combinent comme suit.

- Fixe, imprévisible. C'est le cas le plus courant, aussi bien pour les réseaux locaux que pour les réseaux à grande distance (comme l'Internet). Bien que l'on puisse souvent estimer une moyenne pour la durée de transmission, il n'est pas possible de lui garantir une borne supérieure.
- Fixe, prévisible. Cette combinaison s'applique aux environnements spécialement développés pour des applications ayant des contraintes particulières, comme les applications critiques en temps réel. Le protocole de communication garantit alors une borne supérieure pour le temps de transfert, en utilisant la réservation préalable de ressources.
- Variable, imprévisible. C'est le cas de systèmes de communication qui comprennent des appareils mobiles (dits aussi nomades) tels que les téléphones mobiles ou les assistants personnels. Ces appareils utilisent la communication sans fil, qui est sujette à des variations imprévisibles de performances. Les environnements dits *ubiquitaires*, ou *omniprésents* [Weiser 1993], permettant la connexion et la déconnexion dynamique de dispositifs très variés, appartiennent à cette catégorie.

Avec les techniques actuelles de communication, la classe (variable, prévisible) est vide.

L'imprévisibilité des performances du système de communication rend difficile la tâche de garantir aux applications des niveaux de qualité spécifiés. L'adaptabilité, c'est-à-dire la capacité à réagir à des variations des performances des communications, est la qualité principale requise de l'intergiciel dans de telles situations.

Architecture et interfaces. Plusieurs critères permettent de caractériser l'architecture d'ensemble d'un système intergiciel.

1. *Unités de décomposition.* Les applications gérées par les systèmes intergiciel sont habituellement décomposées en parties. Celles-ci peuvent prendre différentes formes, qui se distinguent par leur définition, leurs propriétés, et leur mode d'interaction. Des exemples, illustrés dans la suite de ce livre, en sont les objets, les composants et les agents.
2. *Rôles.* Un système intergiciel gère l'activité d'un ensemble de participants (utilisateurs ou systèmes autonomes). Ces participants peuvent avoir des rôles prédéfinis tels que *client* (demandeur de service) et *serveur* (fournisseur de service), ou *difusseur* (émetteur d'information) et *abonné* (récepteur d'information). Les participants peuvent aussi se trouver au même niveau, chacun pouvant indifféremment assumer un rôle différent, selon les besoins ; une telle organisation est dite *pair à pair* (en anglais *peer to peer*, ou P2P).
3. *Interfaces de fourniture de service.* Les primitives de communication fournies par un système intergiciel peuvent suivre un schéma synchrone or asynchrone (ces termes sont malheureusement surchargés, et ont ici un sens différent de celui vu

³Dans un système réparti, le terme *asynchrone* indique généralement en outre qu'on ne peut fixer de borne supérieure au rapport des vitesses de calcul sur les différents sites (cela est une conséquence du caractère imprévisible de la charge sur les processeurs partagés).

précédemment pour le système de communication de base). Dans le schéma synchrone, un processus client envoie un message de requête à un serveur distant et se bloque en attendant la réponse. Le serveur reçoit la requête, exécute l'opération demandée, et renvoie un message de réponse au client. À la réception de la réponse, le client reprend son exécution. Dans le schéma asynchrone, l'envoi de requête n'est pas bloquant, et il peut ou non y avoir une réponse. L'appel de procédure ou de méthode à distance est un exemple de communication synchrone ; les files de messages ou les systèmes de publication-abonnement (*publish-subscribe*) sont des exemples de communication asynchrone.

Les diverses combinaisons des propriétés ci-dessus produisent des systèmes variés, qui diffèrent selon leur structure ou leurs interfaces ; des exemples en sont présentés dans la suite. En dépit de cette diversité, nous souhaitons mettre en évidence quelques principes architecturaux communs à tous ces systèmes.

1.3 Un exemple simple d'intergiciel : l'appel de procédure à distance

Nous présentons maintenant un exemple simple d'intergiciel : l'appel de procédure à distance (en anglais, *Remote Procedure Call*, ou RPC). Nous ne visons pas à couvrir tous les détails de ce mécanisme, que l'on peut trouver dans tous les manuels traitant de systèmes répartis ; l'objectif est de mettre en évidence quelques problèmes de conception des systèmes intergiciels, et quelques constructions communes, ou patrons, visant à résoudre ces problèmes.

1.3.1 Motivations et besoins

L'abstraction procédurale est un concept fondamental de la programmation. Une procédure, dans un langage impératif, peut être vue comme une « boîte noire » qui remplit une tâche spécifiée en exécutant une séquence de code encapsulée, le corps de la procédure. L'encapsulation traduit le fait que la procédure ne peut être appelée qu'à travers une interface qui spécifie ses paramètres et ses résultats sous la forme d'un ensemble de conteneurs typés (les paramètres formels). Un processus qui appelle une procédure fournit les paramètres effectifs associés aux conteneurs, exécute l'appel, et reçoit les résultats lors du retour, c'est-à-dire à la fin de l'exécution du corps de la procédure.

L'appel de procédure à distance peut être spécifié comme suit. Soit sur un site *A* un processus *p* qui appelle une procédure locale *P* (Figure 1.6a). Il s'agit de concevoir un mécanisme permettant à *p* de faire exécuter *P* sur un site distant *B* (Figure 1.6b), en préservant la forme et la sémantique de l'appel (c'est-à-dire son effet global dans l'environnement de *p*). *A* et *B* sont respectivement appelés site client et site serveur, car l'appel de procédure à distance suit le schéma client-serveur de communication synchrone par requête et réponse.

L'invariance de la sémantique entre appel local et distant maintient l'abstraction procédurale. Une application qui utilise le RPC est facilement portable d'un environnement à un autre, car elle est indépendante des protocoles de communication sous-jacents.

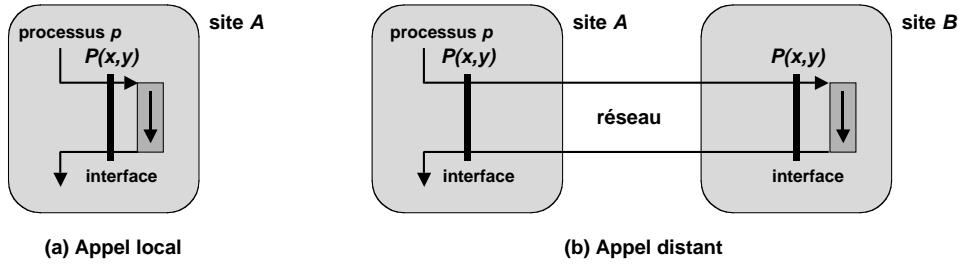


Figure 1.6 – Schéma de l'appel de procédure à distance

Elle peut également être portée aisément, sans modifications, d'un système centralisé vers un réseau.

Néanmoins, le maintien de la sémantique est une tâche délicate, pour deux raisons principales :

- les modes de défaillance sont différents dans les cas centralisé et réparti ; dans cette dernière situation, le site client, le site serveur et le réseau sont sujets à des défaillances indépendantes ;
- même en l'absence de pannes, la sémantique du passage des paramètres est différente (par exemple, on ne peut pas passer un pointeur en paramètre dans un environnement réparti car le processus appelant et la procédure appelée s'exécutent dans des espaces d'adressage différents).

Le problème du passage des paramètres est généralement résolu en utilisant le passage par valeur pour les types simples. Cette solution s'étend aux paramètres composés de taille fixe (tableaux ou structures). Dans le cas général, l'appel par référence n'est pas possible ; on peut néanmoins construire des routines spécifiques d'emballage et de déballage des paramètres pour des structures à base de pointeurs, graphes ou listes. Les aspects techniques de la transmission des paramètres sont examinés en 1.3.2.

La spécification du comportement du RPC en présence de défaillances soulève deux difficultés, lorsque le système de communication utilisé est asynchrone. D'une part, on ne connaît en général pas de borne supérieure pour le temps de transmission d'un message ; un mécanisme de détection de perte de message à base de délais de garde risque donc des fausses alarmes. D'autre part, il est difficile de différencier l'effet de la perte d'un message de celui de la panne d'un site distant. En conséquence, une action de reprise peut conduire à une décision erronée, comme de réexécuter une procédure déjà exécutée. Les aspects relatifs à la tolérance aux fautes sont examinés en 1.3.2.

1.3.2 Principes de réalisation

La réalisation standard du RPC [Birrell and Nelson 1984] repose sur deux modules logiciels (Figure 1.7), la souche, ou talon, client (*client stub*) et la souche serveur (*server stub*). La souche client agit comme un représentant local du serveur sur le site client ; la souche serveur joue un rôle symétrique. Ainsi, aussi bien le processus appelant, côté client, que la procédure appelée, côté serveur, conservent la même interface que dans le cas centralisé. Les souches client et serveur échangent des messages via le système de

communication. En outre, ils utilisent un service de désignation pour aider le client à localiser le serveur (ce point est développé en 1.3.3).

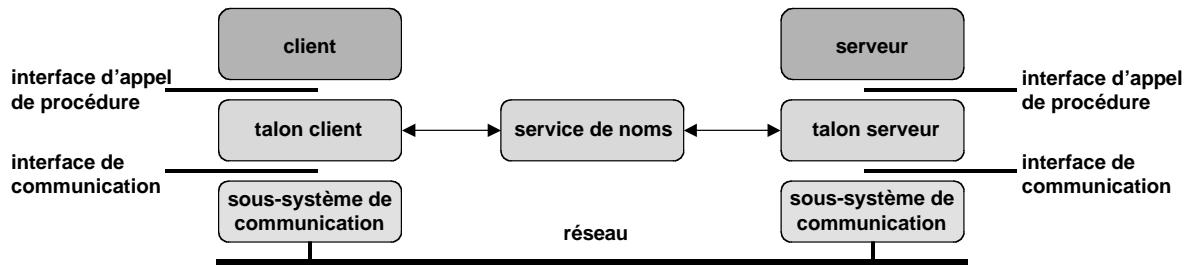


Figure 1.7 – Composants principaux de l'appel de procédure à distance

Les fonctions des souches sont résumées ci-après.

Gestion des processus et synchronisation. Du côté client, le processus⁴ appelant doit être bloqué en attendant le retour de la procédure.

Du côté serveur, le problème principal est celui de la gestion du parallélisme. L'exécution de la procédure est une opération séquentielle, mais le serveur peut être utilisé par plusieurs clients. Le multiplexage des ressources du serveur (notamment si le support du serveur est un multiprocesseur ou une grappe) nécessite d'organiser l'exécution parallèle des appels. On peut utiliser pour cela des processus ordinaires ou des processus légers (*threads*), ce dernier choix (illustré sur la figure 1.8) étant le plus fréquent. Un *thread* veilleur (en anglais *daemon*) attend les requêtes sur un port spécifié. Dans la solution séquentielle (Figure 1.8a), le veilleur lui-même exécute la procédure ; il n'y a pas d'exécution parallèle sur le serveur. Dans le schéma illustré (Figure 1.8b), un nouveau *thread* est créé pour effectuer l'appel, le veilleur revenant attendre l'appel suivant ; l'exécutant disparaît à la fin de l'appel. Pour éviter le coût de la création, une solution consiste à gérer une réserve de *threads* créés à l'avance (Figure 1.8c). Ces *threads* communiquent avec le veilleur à travers un tampon partagé, selon le schéma producteur-consommateur. Chaque *thread* attend qu'une tâche d'exécution soit proposée ; quand il a terminé, il retourne dans la réserve et se remet en attente. Un appel qui arrive alors que tous les *threads* sont occupés est retardé jusqu'à ce que l'un d'eux se libère.

Voir [Lea 1999] pour une discussion de la gestion de l'exécution côté serveur, illustrée par des *threads* Java.

Toutes ces opérations de synchronisation et de gestion des processus sont réalisées dans les souches et sont invisibles aux programmes principaux du client et du serveur.

Emballage et déballage des paramètres. Les paramètres et les résultats doivent être transmis sur le réseau. Ils doivent donc être mis sous une forme sérialisée permettant cette transmission. Pour assurer la portabilité, cette forme doit être conforme à un standard connu, et indépendante des protocoles de communication utilisés et des conventions

⁴selon l'organisation adoptée, il peut s'agir d'un processus ordinaire ou d'un processus léger (*thread*).

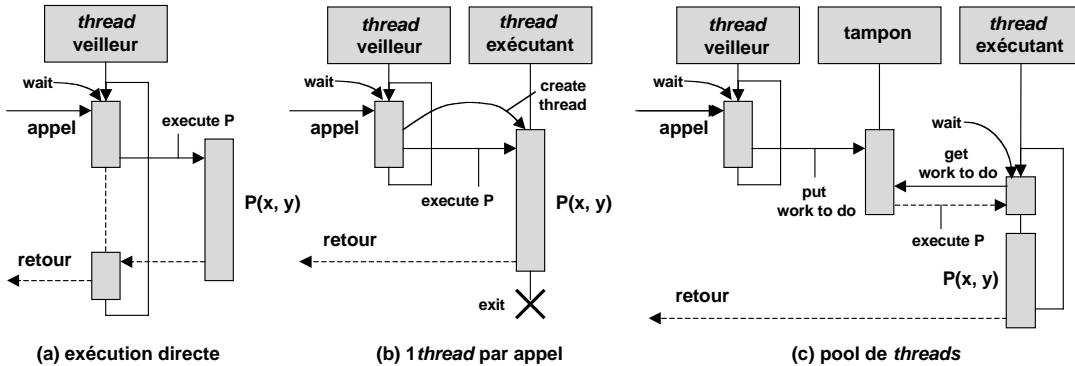


Figure 1.8 – Appel de procédure à distance : gestion de l'exécution côté serveur

locales de représentation des données sur les sites client et serveur, comme l'ordre des octets. La conversion des données depuis leur représentation locale vers une forme standard transmissible est appelée emballage (en anglais *marshalling*) ; la conversion en sens inverse est appelée déballage (en anglais *unmarshalling*).

Un emballeur (*marshaller*) est un jeu de routines, une par type de données (par exemple `writeInt`, `writeString`, etc.), qui écrivent des données du type spécifié dans un flot de données séquentiel. Un déballeur (*unmarshaller*) remplit la fonction inverse et fournit des routines (par exemple `readInt`, `readString`, etc.) qui extraient des données d'un type spécifié à partir d'un flot séquentiel. Ces routines sont appelées par les souches quand une conversion est nécessaire. L'interface et l'organisation des emballeurs et déballeurs dépend du langage utilisé, qui spécifie les types de données, et du format choisi pour la représentation standard.

Réaction aux défaillances. Comme indiqué plus haut, les défaillances (ou pannes) peuvent survenir sur le site client, sur le site serveur, ou sur le réseau. La prise en compte des défaillances nécessite de formuler des hypothèses de défaillances, de détecter les défaillances, et enfin de réagir à cette détection.

Les hypothèses habituelles sont celle de la panne franche (*fail-stop*) pour les sites (ou bien un site fonctionne correctement, ou bien il est arrêté), et celle de la perte de message pour le réseau (ou bien un message arrive sans erreur, ou bien il n'arrive pas, ce qui suppose que les erreurs de transmission sont détectées et éventuellement corrigées à un niveau plus bas du système de communication). Les mécanismes de détection de pannes reposent sur des délais de garde. À l'envoi d'un message, une horloge de garde est armée, avec une valeur estimée de la borne supérieure du délai de réception de la réponse. Le dépassement du délai de garde déclenche une action de reprise.

Les horloges de garde sont placées côté client, à l'envoi du message d'appel, et côté serveur, à l'envoi du message de réponse. Dans les deux cas, l'action de reprise consiste à renvoyer le message. Le problème vient du fait qu'il est difficile d'estimer les délais de garde : un message d'appel peut être renvoyé alors que l'appel a déjà eu lieu, et la procédure peut ainsi être exécutée plusieurs fois.

Le résultat net est qu'il est généralement impossible de garantir la sémantique d'appel

dite « exactement une fois » (après réparation de toutes les pannes, l'appel a été exécuté une fois et une seule). La plupart des systèmes assurent la sémantique « au plus une fois » (l'appel a été exécuté une fois ou pas du tout, ce qui exclut les cas d'exécution partielle ou multiple). La sémantique « au moins une fois », qui autorise les exécutions multiples, est acceptable si l'appel est idempotent, c'est-à-dire si l'effet de deux appels successifs est identique à celui d'un seul appel.

L'organisation d'ensemble du RPC, en dehors de la gestion des pannes, est schématisée sur la figure 1.9.

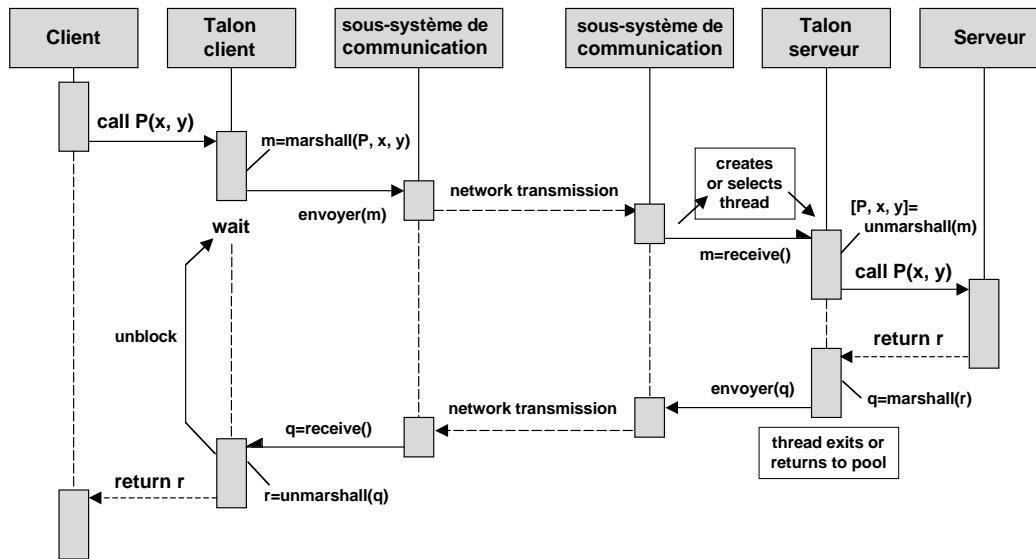


Figure 1.9 – Flot d'exécution dans un appel de procédure à distance

1.3.3 Développement d'applications avec le RPC

Pour utiliser le RPC dans des applications, il faut régler plusieurs problèmes pratiques : liaison entre client et serveur, création des souches, déploiement et démarrage de l'application. Ces aspects sont examinés ci-après.

Liaison client-serveur. Le client doit connaître l'adresse du serveur et le numéro de port de destination de l'appel. Ces informations peuvent être connues à l'avance et inscrites « en dur » dans le programme. Néanmoins, pour assurer l'indépendance logique entre client et serveur, permettre une gestion souple des ressources, et augmenter la disponibilité, il est préférable de permettre une liaison tardive du client au serveur. Le client doit localiser le serveur au plus tard au moment de l'appel.

Cette fonction est assurée par le service de désignation, qui fonctionne comme un annuaire associant les noms (et éventuellement les numéros de version) des procédures avec les adresses et numéros de port des serveurs correspondants. Un serveur enregistre

un nom de procédure, associé à son adresse IP et au numéro de port auquel le veilleur attend les appels. Un client (plus précisément, la souche client) consulte l'annuaire pour obtenir ces informations de localisation. Le service de désignation est généralement réalisé par un serveur spécialisé, dont l'adresse et le numéro de port sont connus de tous les sites participant à l'application⁵.

La figure 1.10 montre le schéma général d'utilisation du service de désignation.

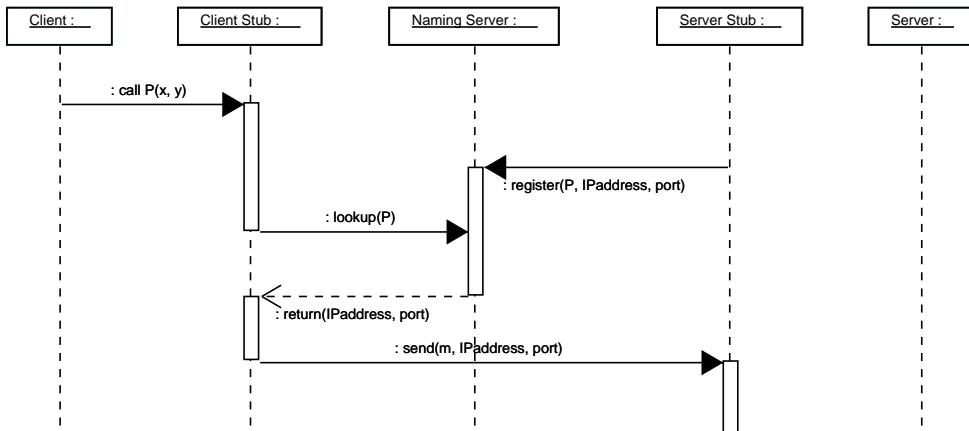


Figure 1.10 – Localisation du serveur dans l'appel de procédure à distance.

La liaison inverse (du serveur vers le client) est établie en incluant l'adresse IP et le numéro de port du client dans le message d'appel.

Génération des souches. Comme nous l'avons vu en 1.3.2, les souches remplissent des fonctions bien définies, dont une partie est générique (comme la gestion des processus) et une autre est propre à chaque appel (comme l'emballage et le déballage des paramètres). Ce schéma permet la génération automatique des souches.

Les paramètres d'un appel nécessaires pour la génération de souches sont spécifiés dans une notation appelée langage de définition d'interface (en anglais, *Interface Definition Language*, ou IDL). Une description d'interface écrite en IDL contient toute l'information qui définit l'interface pour un appel, et fonctionne comme un contrat entre l'appelant et l'appelé. Pour chaque paramètre, la description spécifie son type et son mode de passage (valeur, copie-restauration, etc.). Des informations supplémentaires telles que le numéro de version et le mode d'activation du serveur peuvent être spécifiées.

Plusieurs IDL ont été définis (par exemple Sun XDR, OSF DCE). Le générateur de souches utilise un format commun de représentation de données défini par l'IDL ; il insère les routines de conversion fournies par les emballeurs et déballeurs correspondants. Le cycle complet de développement d'une application utilisant le RPC est schématisé sur la figure 1.11 (la notation est celle de Sun RPC).

⁵Si on connaît le site du serveur, on peut utiliser sur ce site un service local d'annuaire (*portmapper*) accessible sur un port prédéfini (n°111).

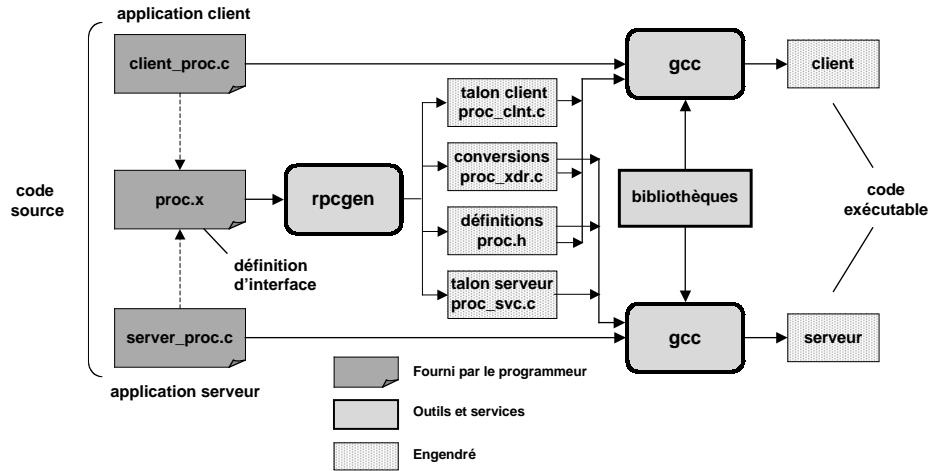


Figure 1.11 – Infrastructure de développement pour l'appel de procédure à distance.

Déploiement d'une application. Le déploiement d'une application répartie est l'installation, sur les différents sites, des parties de programme qui constituent l'application, et le lancement de leur exécution. Dans le cas du RPC, l'installation est généralement réalisée par des scripts prédéfinis qui appellent les outils représentés sur la figure 1.11, en utilisant éventuellement un système réparti de fichiers pour retrouver les programmes source. Les contraintes de l'activation sont que le serveur doit être lancé avant le premier appel d'un client et que le service de désignation doit être lancé avant le serveur pour permettre à ce dernier de s'enregistrer. Les activations peuvent être réalisées par un script lancé par l'administrateur du système ou éventuellement par un client (dans ce cas, celui-ci doit disposer des autorisations nécessaires).

1.3.4 Résumé et conclusions

Plusieurs enseignements utiles peuvent être tirés de cette étude de cas simple.

1. La transparence complète (c'est-à-dire la préservation du comportement d'une application entre environnements centralisé et réparti) ne peut pas être réalisée. Bien que cet objectif ait été initialement fixé aux premiers temps du développement de l'intergiciel, l'expérience a montré que la transparence avait des limites, et qu'il valait mieux explicitement prendre en compte le caractère réparti des applications, au moins pour les aspects où la distinction est pertinente, tels que la tolérance aux fautes et les performances. Voir [Waldo et al. 1997] pour une discussion détaillée.
2. Plusieurs schémas utiles, ou patrons, ont été mis en évidence. L'usage de représentants locaux pour gérer la communication entre entités distantes est l'un des patrons les plus courants de l'intergiciel (le patron de conception PROXY, voir chapitre 2, section 2.3.1). Un autre patron universel est la liaison client-serveur via un service de désignation (le patron d'architecture BROKER). D'autres patrons, peut-être moins immédiatement visibles, sont l'organisation de l'activité du serveur par création dynamique ou gestion d'une réserve de *threads* (voir chapitre 2, section

2.3.3), et le schéma détection-réaction pour le traitement des défaillances.

3. Le développement d'une application répartie, même avec un schéma d'exécution conceptuellement aussi simple que le RPC, met en jeu une infrastructure logistique importante : IDL, générateurs de souches, représentation commune des données, emballageurs et déballageurs, mécanisme de réaction aux défaillances, service de désignation, outils de déploiement. La conception de cette infrastructure, avec l'objectif de simplifier la tâche des développeurs d'applications, est un thème récurrent de cet ouvrage.

En ce qui concerne l'utilité du RPC comme outil pour la structuration des applications réparties, on peut noter quelques limitations.

- La structure de l'application est statique ; rien n'est prévu pour la création dynamique de serveurs ou pour la restructuration d'une application.
- La communication est restreinte à un schéma synchrone : rien n'est prévu pour la communication asynchrone, dirigée par les événements.
- Les données gérées par le client et le serveur ne sont pas persistantes, c'est-à-dire qu'elles disparaissent avec les processus qui les ont créées. Ces données peuvent bien sûr être rangées dans des fichiers, mais leur sauvegarde et leur restauration doivent être explicitement programmées ; rien n'est prévu pour assurer la persistance automatique.

Dans la suite de cet ouvrage, nous présentons d'autres infrastructures, qui échappent à ces limitations.

1.4 Problèmes et défis de la conception de l'intergiciel

Dans cette section, nous examinons les problèmes posés par la conception des systèmes intergiciels, et quelques principes d'architecture permettant d'y répondre. Nous concluons par une brève revue de quelques défis actuels posés par l'intergiciel, qui orientent les recherches dans ce domaine.

1.4.1 Problèmes de conception

La fonction de l'intergiciel est l'intermédiation entre les parties d'une application, ou entre applications. Les considérations *architecturales* tiennent donc une place centrale dans la conception de l'intergiciel. L'architecture couvre l'organisation, la structure d'ensemble, et les schémas de communication, aussi bien pour les applications que pour l'intergiciel lui-même.

Outre les aspects architecturaux, les principaux problèmes de la conception de l'intergiciel sont ceux résultant de la répartition. Nous les résumons ci-après.

La désignation et la liaison sont des notions centrales de la conception de l'intergiciel, puisque celui-ci peut être défini comme le logiciel assurant la liaison entre les différentes parties d'une application répartie, et facilitant leur interaction.

Dans tout système intergiciel, cette interaction repose sur une couche de communication. En outre, la communication est une des fonctions que fournit l'intergiciel aux applications. Les entités communicantes peuvent assumer des rôles divers tels que client-serveur ou pair à pair. À un niveau plus élevé, l'intergiciel permet différents modes d'interaction

(appel synchrone, communication asynchrone par messages, coordination via des objets partagés).

L'architecture logicielle définit l'organisation d'un système réalisé comme assemblage de parties. Les notions liées à la composition et aux composants logiciels sont maintenant un élément central de la conception des systèmes intergiciels, aussi bien pour leur propre structure que pour celle des applications qui les utilisent.

La gestion de données soulève le problème de la persistance (conservation à long terme et procédures d'accès) et des transactions (maintien de la cohérence pour l'accès concurrent aux données en présence de défaillances éventuelles).

La qualité de service comprend diverses propriétés d'une application qui ne sont pas explicitement spécifiées dans ses interfaces fonctionnelles, mais qui sont très importantes pour ses utilisateurs. Des exemples de ces propriétés sont la fiabilité et la disponibilité, les performances (spécialement pour les applications en temps réel), et la sécurité.

L'administration est une étape du cycle de vie des applications qui prend une importance croissante. Elle comprend des fonctions telles que la configuration et le déploiement, la surveillance (*monitoring*), la réaction aux événements indésirables (surcharge, défaillances) et la reconfiguration. La complexité croissante des tâches d'administration conduit à envisager de les automatiser (*autonomic computing*).

Les différents aspects ci-dessus apparaissent dans les études de cas qui constituent la suite de cet ouvrage.

1.4.2 Quelques principes d'architecture

Les principes développés ci-après ne sont pas propres aux systèmes intergiciels, mais prennent une importance particulière dans ce domaine car les qualités et défauts de l'intergiciel conditionnent ceux des applications, en raison de son double rôle de médiateur et de fournisseur de services communs.

Séparation des préoccupations

En génie logiciel, la séparation des préoccupations (*separation of concerns*) est une démarche de conception qui consiste à isoler, dans un système, des aspects indépendants ou faiblement couplés, et à traiter séparément chacun de ces aspects.

Les avantages attendus sont de permettre au concepteur et au développeur de se concentrer sur un problème à la fois, d'éliminer les interactions artificielles entre des aspects orthogonaux, et de permettre l'évolution indépendante des besoins et des contraintes associés à chaque aspect. La séparation des préoccupations a une incidence profonde aussi bien sur l'architecture de l'intergiciel que sur la définition des rôles pour la répartition des tâches de conception et de développement.

La séparation des préoccupations peut être vue comme un « méta-principe », qui peut prendre diverses formes spécifiques dont nous donnons quatre exemples ci-après.

- Le principe d'encapsulation (voir chapitre 2, section 2.1.3) dissocie les préoccupations de l'utilisateur d'un composant logiciel de celles de son réalisateur, en les plaçant de part et d'autre d'une définition commune d'interface.
- Le principe d'abstraction permet de décomposer un système complexe en niveaux (voir chapitre 2, section 2.2.1), dont chacun fournit une vue qui cache les détails non

pertinents, qui sont pris en compte aux niveaux inférieurs.

- La séparation entre politiques et mécanismes [Levin et al. 1975] est un principe largement applicable, notamment dans le domaine de la gestion et de la protection de ressources. Cette séparation donne de la souplesse au concepteur de politiques, tout en évitant de « sur-spécifier » des mécanismes. Il doit par exemple être possible de modifier une politique sans avoir à réimplémenter les mécanismes qui la mettent en œuvre.
- Le principe de la persistance orthogonale sépare la définition de la durée de vie des données d'autres aspects tels que le type des données ou les propriétés de leurs fonctions d'accès.

Ces points sont développés dans le chapitre 2.

Dans un sens plus restreint, la séparation des préoccupations vise à traiter des aspects dont la mise en œuvre (au moins dans l'état de l'art courant) est dispersée entre différentes parties d'un système logiciel, et étroitement imbriquée avec celle d'autres aspects. L'objectif est de permettre une expression séparée des aspects que l'on considère comme indépendants et, en dernier ressort, d'automatiser la tâche de production du code qui traite chacun des aspects. Des exemples d'aspects ainsi traités sont ceux liés aux propriétés « extra-fonctionnelles » (voir chapitre 2, section 2.1.2) telles que la disponibilité, la sécurité, la persistance, ainsi que ceux liés aux fonctions courantes que sont la journalisation, la mise au point, l'observation, la gestion de transactions. Tous ces aspects sont typiquement réalisés par des morceaux de code dispersés dans différentes parties d'une application.

La séparation des préoccupations facilite également l'identification des rôles spécialisés au sein des équipes de conception et de développement, tant pour l'intergiciel proprement dit que pour les applications. En se concentrant sur un aspect particulier, la personne ou le groupe qui remplit un rôle peut mieux appliquer ses compétences et travailler plus efficacement. Des exemples de rôles associés aux différents aspects des applications à base de composants sont donnés dans le chapitre 5.

Évolution et adaptation

Les systèmes logiciels doivent s'accommoder du changement. En effet, les spécifications évoluent à mesure de la prise en compte des nouveaux besoins des utilisateurs, et la diversité des systèmes et organes de communication entraîne des conditions variables d'exécution. S'y ajoutent des événements imprévisibles comme les variations importantes de la charge et les divers types de défaillances. La conception des applications comme celle de l'intergiciel doit tenir compte de ces conditions changeantes : l'évolution des programmes répond à celle des besoins, leur adaptation dynamique répond aux variations des conditions d'exécution.

Pour faciliter l'évolution d'un système, sa structure interne doit être rendue accessible. Il y a une contradiction apparente entre cette exigence et le principe d'encapsulation, qui vise à cacher les détails de la réalisation.

Ce problème peut être abordé par plusieurs voies. Des techniques pragmatiques, reposant souvent sur l'interception (voir chapitre 2, section 2.3.5), sont largement utilisées dans les intergiciels commerciaux. Une approche plus systématique utilise la réflexivité. Un système est dit *réflexif* [Smith 1982, Maes 1987] quand il fournit une représentation de lui-même permettant de l'observer et de l'adapter, c'est-à-dire de modifier son comportement.

Pour être cohérente, cette représentation doit être *causalement connectée* au système : toute modification apportée au système doit se traduire par une modification homologue de sa représentation, et vice versa. Les métas-objets fournissent une telle représentation explicite des mécanismes de base d'un système, ainsi que des protocoles pour examiner et modifier cette représentation. La programmation par aspects, une technique destinée à assurer la séparation des préoccupations, est également utile pour réaliser l'évolution dynamique d'un système. Ces techniques et leur utilisation dans les systèmes intergiciels sont examinées dans le chapitre 2.

1.4.3 Défis de l'intergiciel

Les concepteurs des futurs systèmes intergiciels sont confrontés à plusieurs défis.

- Performances. Les systèmes intergiciels reposent sur des mécanismes d'interception et d'indirection, qui induisent des pertes de performances. L'adaptabilité introduit des indirections supplémentaires, qui aggravent encore la situation. Ce défaut peut être compensé par diverses méthodes d'optimisation, qui visent à éliminer les coûts supplémentaires inutiles en utilisant l'injection⁶ directe du code de l'intergiciel dans celui des applications. La souplesse d'utilisation doit être préservée, en permettant d'annuler, en cas de besoin, l'effet de ces optimisations.
- Passage à grande échelle. À mesure que les applications deviennent de plus en plus étroitement interconnectées et interdépendantes, le nombre d'objets, d'usagers et d'appareils divers composant ces applications tend à augmenter. Cela pose le problème de la capacité de croissance (*scalability*) pour la communication et pour les algorithmes de gestion d'objets, et accroît la complexité de l'administration (ainsi, on peut s'interroger sur la possibilité d'observer l'état d'un très grand système réparti, et sur le sens même que peut avoir une telle notion). Le passage à grande échelle rend également plus complexe la préservation des différentes formes de qualité de service.
- Ubiquité. L'informatique ubiquitaire (ou omniprésente) est une vision du futur proche, dans laquelle un nombre croissant d'appareils (capteurs, processeurs, actionneurs) inclus dans divers objets physiques participent à un réseau d'information global. La mobilité et la reconfiguration dynamique seront des traits dominants de ces systèmes, imposant une adaptation permanente des applications. Les principes d'architecture applicables aux systèmes d'informatique ubiquitaire restent encore largement à élaborer.
- Administration. L'administration de grandes applications hétérogènes, largement réparties et en évolution permanente, soulève de nombreuses questions telles que l'observation cohérente, la sécurité, l'équilibre entre autonomie et interdépendance pour les différents sous-systèmes, la définition et la réalisation des politiques d'allocation de ressources, etc.

⁶Cette technique s'apparente à l'optimisation des appels de procédure par insertion de leur code à l'endroit de l'appel (*inlining*).

1.4.4 Plan de l'ouvrage

Le chapitre 2 couvre les aspects architecturaux des systèmes intergiciels. Outre une présentation des principes généraux d'organisation de ces systèmes, ce chapitre décrit un ensemble de patrons (en anglais *patterns*) et de canevas (en anglais *frameworks*) récurrents dans la conception de tous les types d'intergiciels et des applications qui les utilisent.

Les chapitres suivants sont des études de cas de plusieurs familles d'intergiciel, qui sont représentatives de l'état actuel du domaine, tant en ce qui concerne les produits utilisés que les normes proposées ou en cours d'élaboration.

Le chapitre 3 présente un modèle de composants appelé Fractal, dont les caractéristiques sont : (a) une grande généralité (indépendance par rapport au langage de programmation, définition explicite des dépendances, composition hiérarchique avec possibilité de partage) ; (b) la présence d'une interface d'administration extensible, distincte de l'interface fonctionnelle.

Le chapitre 4 est une introduction aux services Web. Ce terme recouvre un ensemble de normes et de systèmes permettant de créer, d'intégrer et de composer des applications réparties en utilisant les protocoles et les infrastructures du World Wide Web.

Le chapitre 5 décrit les principes et techniques de la plate-forme J2EE, spécifiée par Sun Microsystems, qui rassemble divers intergiciels pour la construction et l'intégration d'applications. J2EE a pour base le langage Java.

Le chapitre 6 décrit la plate-forme .NET de Microsoft, plate-forme intergicielle pour la construction d'applications réparties et de services Web.

Le chapitre 7 présente une autre démarche pour la construction modulaire de systèmes. Il s'agit d'une architecture à base de composants, définie par le consortium OSGi, et constituée de deux éléments : une plate-forme pour la fourniture de services et un environnement de déploiement, organisés autour d'un modèle simple de composants.

L'ouvrage comprend des annexes qui présentent la mise en œuvre pratique de certaines des plates-formes ci-dessus (annexes A : Les services Web en pratique, B : .NET en pratique, C : OSGI en pratique), ainsi que des compléments divers.

1.5 Note historique

Le terme *middleware* semble être apparu vers 1990⁷, mais des systèmes intergiciels existaient bien avant cette date. Des logiciels commerciaux de communication par messages étaient disponibles à la fin des années 1970. La référence classique sur l'appel de procédure à distance est [Birrell and Nelson 1984], mais des constructions analogues, liées à des langages particuliers, existaient déjà auparavant (la notion d'appel de procédure à distance apparaît dans [White 1976]⁸ et une première réalisation est proposée dans [Brinch Hansen 1978]).

Vers le milieu des années 1980, plusieurs projets développent des infrastructures intergicielles pour objets répartis, et élaborent les principaux concepts qui vont influencer les normes et les produits futurs. Les précurseurs sont Cronus [Schantz et al. 1986]

⁷Le terme français *intergiciel* est apparu plus récemment (autour de 1999-2000).

⁸version étendue du RFC Internet 707.

et Eden [Almes et al. 1985], suivis par Amoeba [Mullender et al. 1990], ANSA-aware [ANSA], Arjuna [Parrington et al. 1995], Argus [Liskov 1988], Chorus/COOL [Lea et al. 1993], Clouds [Dasgupta et al. 1989], Comandos [Cahill et al. 1994], Emerald [Jul et al. 1988], Gothic [Banâtre and Banâtre 1991], Guide [Balter et al. 1991], Network Objects [Birrell et al. 1995], SOS [Shapiro et al. 1989], et Spring [Mitchell et al. 1994].

L'*Open Software Foundation* (OSF), qui deviendra plus tard l'*Open Group* [Open Group], est créée en 1988 dans le but d'unifier les diverses versions du système d'exploitation Unix. Cet objectif ne fut jamais atteint, mais l'OSF devait spécifier une plate-forme intergicielle, le *Distributed Computing Environment* (DCE) [Lendenmann 1996], qui comportait notamment un service d'appel de procédure à distance, un système réparti de gestion de fichiers, un serveur de temps, et un service de sécurité.

L'*Object Management Group* (OMG) [OMG] est créé en 1989 pour définir des normes pour l'intergiciel à objets répartis. Sa première proposition (1991) fut la spécification de CORBA 1.0 (la version courante, en 2005, est CORBA 3). Ses propositions ultérieures sont des normes pour la modélisation (UML, MOF) et les composants (CCM). L'*Object Database Management Group* (ODMG) [ODMG] définit des normes pour les bases de données à objets, qui visent à unifier la programmation par objets et la gestion de données persistantes.

Le modèle de référence de l'*Open Distributed Processing* (RM-ODP) [ODP 1995a], [ODP 1995b] a été conjointement défini par deux organismes de normalisation, l'ISO et l'ITU-T. Il propose un ensemble de concepts définissant un cadre générique pour le calcul réparti ouvert, plutôt que des normes spécifiques.

La définition du langage Java par Sun Microsystems en 1995 ouvre la voie à plusieurs intergiciels, dont *Java Remote Method Invocation* (RMI) [Wollrath et al. 1996] et les *Enterprise JavaBeans* (EJB) [Monson-Haefel 2002]. Ces systèmes, et d'autres, sont intégrés dans une plate-forme commune, J2EE [J2EE 2005].

Microsoft développe à la même époque le *Distributed Component Object Model* (DCOM) [Grimes 1997], un intergiciel définissant des objets répartis composable, dont une version améliorée est COM+ [Platt 1999]. Son offre courante est .NET [.NET], plate-forme intergicielle pour la construction d'applications réparties et de services pour le Web.

La première conférence scientifique entièrement consacrée à l'intergiciel a lieu en 1998 [Middleware 1998]. Parmi les thèmes actuels de la recherche sur l'intergiciel, on peut noter les techniques d'adaptation (réflexivité, aspects), l'administration et notamment les travaux sur les systèmes autonomes (pouvant réagir à des surcharges ou à des défaillances), et l'intergiciel pour appareils mobiles et environnements ubiquitaires.

Chapitre 2

Patrons et canevas pour l'intergiciel

Ce chapitre présente les grands principes de conception des systèmes intergiciels, ainsi que quelques patrons élémentaires récurrents dans toutes les architectures intergicielles. Divers patrons plus élaborés peuvent être construits en étendant et en combinant ces constructions de base. Le chapitre débute par une présentation des principes architecturaux et des éléments constitutifs des systèmes intergiciels, notamment les objets répartis et les organisations multi-couches. Il continue par une discussion des patrons de base relatifs aux objets répartis. Le chapitre se termine par une présentation des patrons liés à la séparation des préoccupations, qui comprend une discussion des techniques de réalisation pour l'intergiciel réflexif.

2.1 Services et interfaces

Un système matériel et/ou logiciel est organisé comme un ensemble de parties, ou composants¹. Le système entier, et chacun de ses composants, remplit une fonction qui peut être décrite comme la fourniture d'un *service*. Selon une définition tirée de [Bieber and Carpenter 2002], « un service est un comportement défini par contrat, qui peut être réalisé et fourni par tout composant pour être utilisé par tout composant, sur la base unique du contrat ».

Pour fournir ses services, un composant repose généralement sur des services qu'il demande à d'autres composants. Par souci d'uniformité, le système entier peut être considéré comme un composant, qui interagit avec un environnement externe spécifié ; le service fourni par le système repose sur des hypothèses sur les services que ce dernier reçoit de

¹Dans ce chapitre, nous utilisons le terme de *composant* dans un sens non-technique, pour désigner une unité de décomposition d'un système.

son environnement².

La fourniture de services peut être considérée à différents niveaux d'abstraction. Un service fourni est généralement matérialisé par un ensemble d'interfaces, dont chacune représente un aspect du service. L'utilisation de ces interfaces repose sur des patrons élémentaires d'interaction entre les composants du système. Dans la section 2.1.1, nous passons brièvement en revue ces patrons d'interaction. Les interfaces sont discutées dans la section 2.1.2, et les contrats sont l'objet de la section 2.1.3.

2.1.1 Mécanismes d'interaction de base

Les composants interagissent via un système de communication sous-jacent. Nous supposons acquises les notions de base sur la communication et nous examinons quelques patrons d'interaction utilisés pour la fourniture de services.

La forme la plus simple de communication est un événement transitoire asynchrone (Figure 2.1a). Un composant *A* (plus précisément, un *thread* s'exécutant dans le composant *A*) produit un événement (c'est-à-dire envoie un message élémentaire à un ensemble spécifié de destinataires), et poursuit son exécution. Le message peut être un simple signal, ou peut porter une valeur. L'attribut « transitoire » signifie que le message est perdu s'il n'est pas attendu. La réception de l'événement par le composant *B* déclenche une réaction, c'est-à-dire lance l'exécution d'un programme (le traitant) associé à cet événement. Ce mécanisme peut être utilisé par *A* pour demander un service à *B*, lorsqu'aucun résultat n'est attendu en retour ; ou il peut être utilisé par *B* pour observer ou surveiller l'activité de *A*.

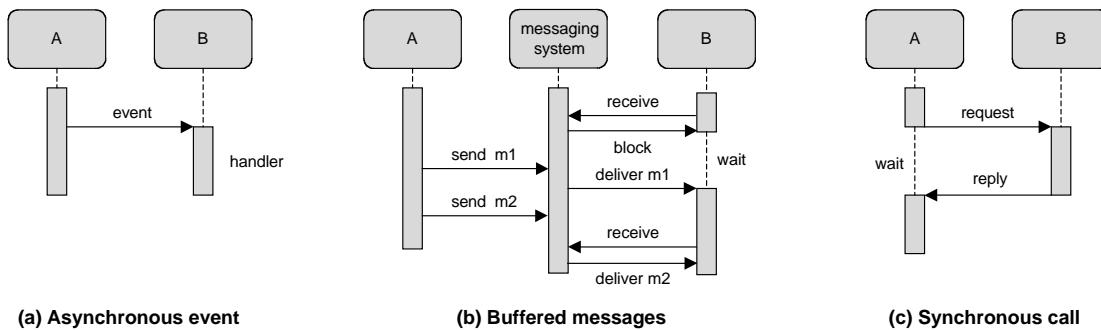


Figure 2.1 – Quelques mécanismes de base pour l'interaction

Une forme de communication plus élaborée est le passage asynchrone de messages persistants (2.1b). Un message est un bloc d'information qui est transmis d'un émetteur à un récepteur. L'attribut « persistant » signifie que le système de communication assure un rôle de tampon : si le récepteur attend le message, le système de communication le lui délivre ; sinon, le message reste disponible pour une lecture ultérieure.

Un autre mécanisme courant est l'appel synchrone (2.1c), dans lequel *A* (le client d'un service fourni par *B*) envoie un message de requête à *B* et attend une réponse. Ce patron

²Par exemple un ordinateur fournit un service spécifié, à condition de disposer d'une alimentation électrique spécifiée, et dans une plage spécifiée de conditions d'environnement, telles que température, humidité, etc.

est utilisé dans le RPC (voir chapitre 1, section 1.3).

Les interactions synchrone et asynchrone peuvent être combinées, par exemple dans diverses formes de « RPC asynchrone ». Le but est de permettre au demandeur d'un service de continuer son exécution après l'envoi de sa requête. Le problème est alors pour le demandeur de récupérer les résultats, ce qui peut être fait de plusieurs manières. Par exemple, le fournisseur peut informer le demandeur, par un événement asynchrone, que les résultats sont disponibles ; ou le demandeur peut appeler le fournisseur à un moment ultérieur pour connaître l'état de l'exécution.

Il peut arriver que la fourniture d'un service par *B* à *A* repose sur l'utilisation par *B* d'un service fourni par *A* (le contrat entre fournisseur et client du service engage les deux parties). Par exemple, dans la figure 2.2a, l'exécution de l'appel depuis *A* vers *B* repose sur un *rappel* (en anglais *callback*) depuis *B* à une fonction fournie par *A*. Sur cet exemple, le rappel est exécuté par un nouveau *thread*, tandis que le *thread* initial continue d'attendre la terminaison de son appel.

Les exceptions sont un mécanisme qui traite les conditions considérées comme sortant du cadre de l'exécution normale d'un service : pannes, valeurs de paramètres hors limites, etc. Lorsqu'une telle condition est détectée, l'exécution du service est proprement terminée (par exemple les ressources sont libérées) et le contrôle est rendu à l'appelant, avec une information sur la nature de l'exception. Une exception peut ainsi être considérée comme un « rappel à sens unique ». Le demandeur du service doit fournir un traitant pour chaque exception possible.

La notion de rappel peut encore être étendue. Le service fourni par *B* à *A* peut être demandé depuis une source extérieure, *A* fournissant toujours à *B* une ou plusieurs interfaces de rappel. Ce patron d'interaction (Figure 2.2b) est appelé *inversion du contrôle*, parce que le flot de contrôle va de *B* (le fournisseur) vers *A* (le demandeur). Ce cas se produit notamment lorsque *B* « contrôle » *A*, c'est-à-dire lui fournit des services d'administration tels que la surveillance ou la sauvegarde persistante ; dans cette situation, la demande de service a une origine externe (elle est par exemple déclenchée par un événement extérieur tel qu'un signal d'horloge).

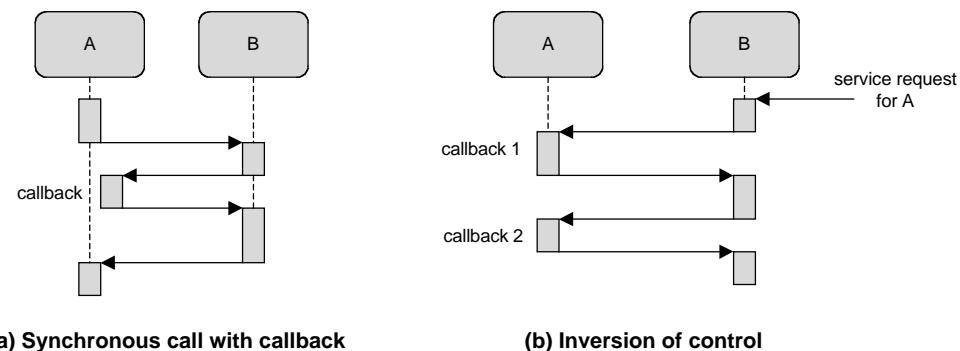


Figure 2.2 – Inversion du contrôle

Les interactions ci-dessus sont discrètes et n'impliquent pas explicitement une notion de temps autre que l'ordre des événements. Les échanges continus nécessitent une forme

de synchronisation en temps réel. Par exemple les données multimédia sont échangées via des *flots de données*, qui permettent la transmission continue d'une séquence de données soumise à des contraintes temporelles.

2.1.2 Interfaces

Un service élémentaire fourni par un composant logiciel est défini par une *interface*, qui est une description concrète de l'interaction entre le demandeur et le fournisseur du service. Un service complexe peut être défini par plusieurs interfaces, dont chacune représente un aspect particulier du service. Il y a en fait deux vues complémentaires d'une interface.

- la vue d'usage : une interface définit les opérations et structures de données utilisées pour la fourniture d'un service ;
- la vue contractuelle : une interface définit un contrat entre le demandeur et le fournisseur d'un service.

La définition effective d'une interface requiert donc une représentation concrète des deux vues, par exemple un langage de programmation pour la vue d'usage et un langage de spécification pour la vue contractuelle.

Rappelons qu'aussi bien la vue d'usage que la vue contractuelle comportent deux partenaires³. En conséquence, la fourniture d'un service implique en réalité *deux* interfaces : l'interface présentée par le composant qui fournit un service, et l'interface attendue par le client du service. L'interface fournie (ou serveur) doit être « conforme » à l'interface requise (ou client), c'est-à-dire compatible avec elle ; nous revenons plus loin sur la définition de la conformité.

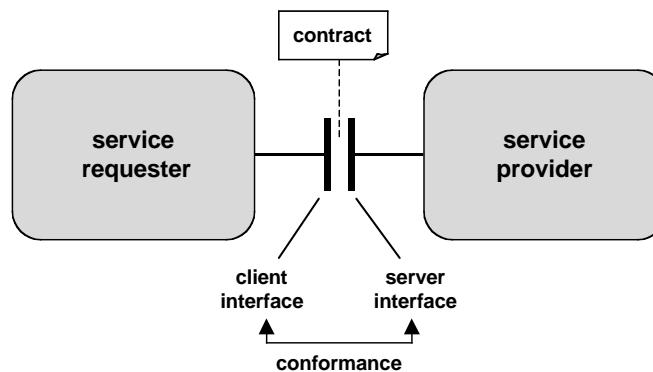


Figure 2.3 – Interfaces

La représentation concrète d'une interface, fournie ou requise, consiste en un ensemble d'opérations, qui peut prendre des formes diverses, correspondant aux patrons d'interaction décrits en 2.1.1.

³Certaines formes de service mettent en jeu plus de deux partenaires, par exemple un fournisseur avec demandeurs multiples, etc. Il est toujours possible de décrire de telles situations par des relations bilatérales entre un demandeur et un fournisseur, par exemple en définissant des interfaces virtuelles qui multiplexent des interfaces réelles, etc.

- procédure synchrone ou appel de méthode, avec paramètres et valeur de retour ; accès à un attribut, c'est-à-dire à une structure de données (cette forme peut être convertie dans la précédente au moyen de fonctions d'accès (en lecture, en anglais *getter* ou en écriture, en anglais *setter*) sur les éléments de cette structure de données) ;
- appel de procédure asynchrone ;
- source ou puits d'événements ;
- flot de données fournisseur (*output channel*) ou récepteur (*input channel*) ;

Le contrat associé à l'interface peut par exemple spécifier des contraintes sur l'ordre d'exécution des opérations de l'interface (par exemple ouvrir un fichier avant de le lire). Les diverses formes de contrats sont examinées dans la section 2.1.3.

Diverses notations, appelées Langages de Description d'Interface (IDL), ont été conçues pour décrire formellement des interfaces. Il n'y a pas actuellement de modèle unique commun pour un IDL, mais la syntaxe de la plupart des IDLs existants est inspirée par celle d'un langage de programmation procédural. Certains langages (par exemple Java, C#) comportent une notion d'interface et définissent donc leur propre IDL. Une définition d'interface typique spécifie la signature de chaque opération, c'est-à-dire son nom, son type et le mode de transmission de ses paramètres et valeurs de retour, ainsi que les exceptions qu'elle peut provoquer à l'exécution (le demandeur doit fournir des traitants pour ces exceptions).

La représentation d'une interface, avec le contrat associé, définit complètement l'interaction entre le demandeur et le fournisseur du service représenté par l'interface. En conséquence, ni le demandeur ni le fournisseur ne doit faire d'autre hypothèse sur son partenaire que celles explicitement spécifiées dans l'interface. En d'autres termes, tout ce qui est au-delà de l'interface est vu par chaque partenaire comme une « boîte noire ». C'est le *principe d'encapsulation*, qui est un cas particulier de la séparation des préoccupations. Le principe d'encapsulation assure l'indépendance entre interface et réalisation, et permet de modifier un système selon le principe « je branche et ça marche »(*plug and play*) : un composant peut être remplacé par un autre à condition que les interfaces entre le composant remplacé et le reste du système restent compatibles.

2.1.3 Contrats et conformité

Le contrat entre le fournisseur et le client d'un service peut prendre diverses formes, selon les propriétés spécifiées et selon l'expression plus ou moins formelle de la spécification. Par exemple, le terme *Service Level Agreement* (SLA) est utilisé pour un contrat légal entre le fournisseur et le client d'un service global de haut niveau (par exemple entre un fournisseur d'accès à l'Internet (en anglais *Internet Service Provider*, ou ISP) et ses clients.

D'un point de vue technique, différentes sortes de propriétés peuvent être spécifiées. D'après [Beugnard et al. 1999], on peut distinguer quatre niveaux de contrats.

- Le niveau 1 s'applique à la forme des opérations, généralement en définissant des *types* pour les opérations et les paramètres. Cette partie du contrat peut être statiquement vérifiée.
- Le niveau 2 s'applique au comportement dynamique des opérations de l'interface, en spécifiant la sémantique de chaque opération.
- Le niveau 3 s'applique aux interactions dynamiques entre les opérations d'une interface, en spécifiant des contraintes de synchronisation entre les exécutions de ces

opérations. Si le service est composé de plusieurs interfaces, il peut aussi exister des contraintes entre l'exécution d'opérations appartenant à différentes interfaces.

- Le niveau 4 s'applique aux propriétés extra-fonctionnelles, c'est-à-dire à celles qui n'apparaissent pas explicitement dans l'interface. Le terme de « Qualité de Service » (QoS) est aussi utilisé pour ces propriétés, qui comprennent performances, sécurité, disponibilité, etc.

Notons encore que le contrat s'applique dans les deux sens, à tous les niveaux : il engage donc le demandeur aussi bien que le fournisseur. Par exemple, les paramètres passés lors d'un appel de fonction sont contraints par leur type ; si l'interface comporte un rappel, la procédure qui réalise l'action correspondante côté client doit être fournie (cela revient à spécifier une procédure comme paramètre).

L'essence d'un contrat d'interface est exprimée par la notion de conformité. Une interface $I2$ est dite *conforme* à une interface $I1$ si un composant qui réalise toute méthode spécifiée dans $I2$ peut partout être utilisé à la place d'un composant qui réalise toute méthode spécifiée dans $I1$. En d'autres termes, $I2$ est conforme à $I1$ si $I2$ satisfait le contrat de $I1$.

La conformité peut être vérifiée à chacun des quatre niveaux définis ci-dessus. Nous les examinons successivement.

Contrats syntaxiques

Un contrat syntaxique est fondé sur la forme des opérations. Un tel contrat s'exprime couramment en termes de types. Un *type* définit un prédictat qui s'applique aux objets⁴ de ce type. Le type d'un objet X est noté $T(X)$. La notion de conformité est exprimée par le *sous-typage* : si $T2$ est un sous-type de $T1$ (noté $T2 \sqsubseteq T1$), tout objet de type $T2$ est aussi un objet de type $T1$ (en d'autre termes, un objet de type $T2$ peut être utilisé partout où un objet de type $T1$ est attendu). La relation de sous-typage ainsi définie est appelée *sous-typage vrai*, ou *conforme*.

Considérons des interfaces définies comme un ensemble de procédures. Pour de telles interfaces, le sous-typage conforme est défini comme suit : une interface $I2$ est un sous-type d'une interface de type $I1$ (noté $T(I2) \sqsubseteq T(I1)$) si $I2$ a au moins le même nombre de procédures que $I1$ (elle peut en avoir plus), et si pour chaque procédure définie dans $I1$ il existe une procédure conforme dans $I2$. Une procédure $Proc2$ est dite conforme à une procédure $Proc1$ lorsque les relations suivantes sont vérifiées entre les signatures de ces procédures.

- $Proc1$ et $Proc2$ ont le même nombre de paramètres et valeurs de retour (les exceptions déclarées sont considérées comme des valeurs de retour).
- pour chaque valeur de retour $R1$ de $Proc1$, il existe une valeur de retour correspondante $R2$ de $Proc2$ telle que $T(R2) \sqsubseteq T(R1)$ (relation dite *covariante*).
- pour chaque paramètre d'appel $X1$ de $Proc1$, il existe un paramètre d'appel correspondant $X2$ de $Proc2$ tel que $T(X1) \sqsubseteq T(X2)$ (relation dite *contravariante*).

Ces règles illustrent un principe général de possibilité de substitution : une entité $E2$ peut être substituée à une autre entité $E1$ si $E2$ « fournit au moins autant et requiert au

⁴Ici le terme d'*objet* désigne toute entité identifiable dans le présent contexte, par exemple une variable, une procédure, une interface, un composant.

plus autant » que $E1$. Ici les termes « fournit » et « requiert » doivent être adaptés à chaque situation spécifique (par exemple dans un appel de procédure, les paramètres d'appel sont « requis » et le résultat est « fourni »). La relation d'ordre qu'impliquent les termes « au plus autant » et « au moins autant » est la relation de sous-typage.

Notons que la relation de sous-typage définie dans la plupart des langages de programmation ne satisfait généralement pas la contravariance des types de paramètres et n'est donc pas un sous-typage vrai. Dans un tel cas (qui est par exemple celui de Java), des erreurs de conformité peuvent échapper à la détection statique et doivent être capturées par un test à l'exécution.

La notion de conformité peut être étendue aux autres formes de définitions d'interface, par exemple celles contenant des sources ou puits d'événements, ou des flots de données (*streams*).

Rappelons que la relation entre types est purement syntaxique et ne capture pas la sémantique de la conformité. La vérification de la sémantique est le but des contrats comportementaux.

Contrats comportementaux

Les contrats comportementaux sont fondés sur une méthode proposée dans [Hoare 1969] pour prouver des propriétés de programmes, en utilisant des pré- et post-conditions avec des règles de preuve fondées sur la logique du premier ordre. Soit A une action séquentielle. Alors la notation

$$\{P\} A \{Q\},$$

dans lequel P et Q sont des assertions (prédicats sur l'état de l'univers du programme), a le sens suivant : si l'exécution de A est lancée dans un état dans lequel P est vrai, et si A se termine, alors Q est vrai à la fin de cette exécution. Une condition supplémentaire peut être spécifiée sous la forme d'un prédicat invariant I qui doit être préservé par l'exécution de A . Ainsi si P et I sont initialement vrais, Q et I sont vrais à la fin de A , si A se termine. L'invariant peut être utilisé pour spécifier une contrainte de cohérence.

Ceci peut être transposé comme suit en termes de services et de contrats. Avant l'exécution d'un service,

- le demandeur doit garantir la précondition P et l'invariant I ,
- le fournisseur doit garantir que le service est effectivement livré dans un temps fini, et doit assurer la postcondition Q et l'invariant I .

Les cas possibles de terminaison anormale doivent être spécifiés dans le contrat et traités par réessay ou par la levée d'une exception. Cette méthode a été développée sous le nom de « conception par contrat » [Meyer 1992] via des extensions au langage Eiffel permettant l'expression de pré- et post-conditions et de prédicats invariants. Ces conditions sont vérifiées à l'exécution. Des outils analogues ont été développés pour Java [Kramer 1998].

La notion de sous-typage peut être étendue aux contrats comportementaux, en spécifiant les contraintes de conformité pour les assertions. Soit une procédure $Proc1$ définie dans l'interface $I1$, et la procédure correspondante (conforme) $Proc2$ définie dans l'interface $I2$, telle que $T(I2) \sqsubseteq T(I1)$. Soit $P1$ et $Q1$ (resp. $P2$ et $Q2$) les pré- et post-conditions définies pour $Proc1$ (resp. $Proc2$). Les conditions suivantes doivent être vérifiées :

$$P1 \Rightarrow P2 \text{ et } Q2 \Rightarrow Q1$$

En d'autres termes, un sous-type a des préconditions plus faibles et des postconditions plus fortes que son super-type, ce qui illustre de nouveau la condition de substitution.

Contrats de synchronisation

L'expression de la validité des programmes au moyen d'assertions peut être étendue aux programmes concurrents. Le but ici est de séparer, autant que possible, la description des contraintes de synchronisation du code des procédures. Les expressions de chemin (*path expressions*), qui spécifient des contraintes sur l'ordre et la concurrence de l'exécution des procédures, ont été proposées dans [Campbell and Habermann 1974]. Les développements ultérieurs (compteurs et politiques de synchronisation) ont essentiellement été des extensions et des raffinements de cette construction, dont la réalisation repose sur l'exécution de procédures engendrées à partir de la description statique des contraintes. Plusieurs articles décrivant des propositions dans ce domaine figurent dans [CACM 1993], mais ces techniques n'ont pas trouvé une large application.

Une forme très simple de contrat de synchronisation est la clause `synchronized` de Java, qui spécifie une exécution en exclusion mutuelle. Un autre exemple est le choix d'une politique de gestion de file d'attente (par exemple FIFO, priorité, etc.) parmi un ensemble prédéfini pour la gestion d'une resource partagée.

Les travaux plus récents (voir par exemple [Chakrabarti et al. 2002]) visent à vérifier les contraintes de synchronisation à la compilation, pour détecter assez tôt les incompatibilités.

Contrats de Qualité de Service

Les spécifications associées à l'interface d'un système ou d'une partie de système, exprimés ou non de manière formelle, sont appelés *fonctionnelles*. Un système peut en outre être l'objet de spécifications supplémentaires, qui s'appliquent à des aspects qui n'apparaissent pas explicitement dans son interface. Ces spécifications sont dites *extra-fonctionnelles*⁵.

La qualité de service (un autre nom pour ces propriétés) inclut les aspects suivants.

- *Disponibilité*. La disponibilité d'un service est une mesure statistique de la fraction du temps pendant laquelle le service est prêt à être rendu. Elle dépend à la fois du taux de défaillances du système qui fournit le service et du temps nécessaire pour restaurer le service après une défaillance.
- *Performances*. Cette qualité couvre plusieurs aspects, qui sont essentiels pour les applications en temps réel (applications dont la validité ou l'utilité repose sur des contraintes temporelles). Certains de ces aspects sont liés à la communication (bornes sur la latence, la gigue, la bande passante) ; d'autres s'appliquent à la vitesse de traitement ou à la latence d'accès aux données.
- *Securité*. La sécurité couvre des propriétés liées à l'usage correct d'un service par ses utilisateurs selon des règles d'usage spécifiées. Elle comprend la confidentialité, l'intégrité, l'authentification, et le contrôle des droits d'accès.

⁵Noter que la définition d'une spécification comme « fonctionnelle » ou « extra-fonctionnelle » n'est pas absolue, mais dépend de l'état de l'art : un aspect qui est extra-fonctionnel aujourd'hui deviendra fonctionnel lorsque des progrès techniques permettront d'intégrer ses spécifications dans celles de l'interface.

D'autres aspects extra-fonctionnels, plus difficiles à quantifier, sont la maintenabilité et la facilité d'évolution.

La plupart des aspects de qualité de service étant liés à un environnement variable, il est important que les politiques de gestion de la QoS puissent être adaptables. Les contrats de QoS comportent donc généralement la possibilité de négociation, c'est-à-dire de redéfinition des termes du contrat via des échanges, à l'exécution, entre le demandeur et le fournisseur du service.

2.2 Patrons architecturaux

Dans cette section, nous examinons quelques principes de base pour la structuration des systèmes intergiciels. La plupart des systèmes examinés dans ce livre sont organisés selon ces principes, qui fournissent essentiellement des indications pour décomposer un système complexe en parties.

2.2.1 Architectures multiniveaux

Architectures en couches

La décomposition d'un système complexe en niveaux d'abstraction est un ancien et puissant principe d'organisation. Il régit beaucoup de domaines de la conception de systèmes, via des notions largement utilisées telles que les machines virtuelles et les piles de protocoles.

L'abstraction est une démarche de conception visant à construire une vue simplifiée d'un système sous la forme d'un ensemble organisé d'interfaces, qui ne rendent visibles que les aspects jugés pertinents. La réalisation de ces interfaces en termes d'entités plus détaillées est laissée à une étape ultérieure de raffinement. Un système complexe peut ainsi être décrit à différents niveaux d'abstraction. L'organisation la plus simple (Figure 2.4a) est une hiérarchie de couches, dont chaque niveau i définit ses propres entités, qui fournissent une interface au niveau supérieur ($i+1$). Ces entités sont réalisées en utilisant l'interface fournie par le niveau inférieur ($i-1$), jusqu'à un niveau de base prédéfini (généralement réalisé par matériel). Cette architecture est décrite dans [Buschmann et al. 1995] sous le nom de patron LAYERS.

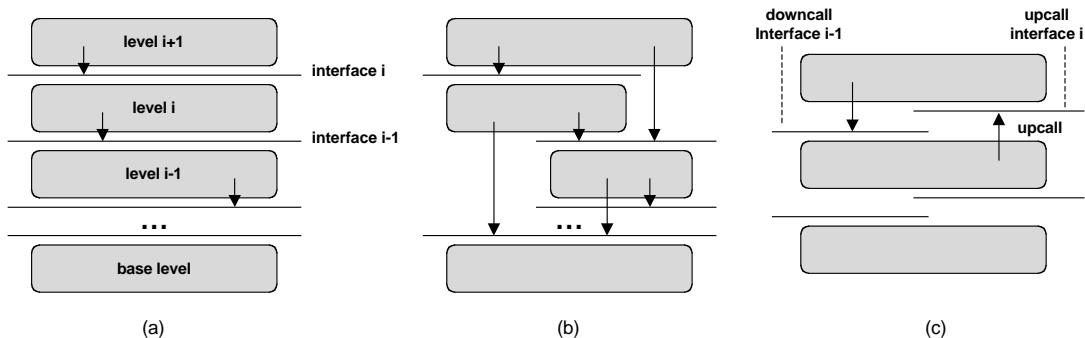


Figure 2.4 – Organisations de systèmes en couches

L'interface fournie par chaque niveau peut être vue comme un ensemble de fonctions définissant une bibliothèque, auquel cas elle est souvent appelée API (*Application Programming Interface*)⁶. Une vue alternative est de considérer chaque niveau comme une machine virtuelle, dont le « langage » (le jeu d'instructions) est défini par son interface. En vertu du principe d'encapsulation, une machine virtuelle masque les détails de réalisation de tous les niveaux inférieurs. Les machines virtuelles ont été utilisées pour émuler un ordinateur ou un système d'exploitation au-dessus d'un autre, pour émuler un nombre quelconque de ressources identiques par multiplexage d'une ressource physique, ou pour réaliser l'environnement d'exécution d'un langage de programmation (par exemple la *Java Virtual Machine* (JVM) [Lindholm and Yellin 1996]).

Ce schéma de base peut être étendu de plusieurs manières. Dans la première extension (Figure 2.4b), une couche de niveau i peut utiliser tout ou partie des interfaces fournies par les machines de niveau inférieur. Dans la seconde extension, une couche de niveau i peut rappeler la couche de niveau $i+1$, en utilisant une interface de rappel (*callback*) fournie par cette couche. Dans ce contexte, le rappel est appelé « appel ascendant » (*upcall*) (par référence à la hiérarchie « verticale » des couches).

Bien que les appels ascendants puissent être synchrones, leur utilisation la plus fréquente est la propagation d'événements asynchrones vers le haut de la hiérarchie des couches. Considérons la structure d'un noyau de système d'exploitation. La couche supérieure (application) active le noyau par appels descendants synchrones, en utilisant l'API des appels système. Le noyau active aussi les fonctions réalisées par le matériel (par exemple mettre à jour la MMU, envoyer une commande à un disque) par l'équivalent d'appels synchrones. En sens inverse, le matériel active typiquement le noyau via des interruptions asynchrones (appels ascendants), qui déclenchent l'exécution de traitants. Cette structure d'appel est souvent répétée aux niveaux plus élevés : chaque couche reçoit des appels synchrones de la couche supérieure et des appels asynchrones de la couche inférieure. Ce patron, décrit dans [Schmidt et al. 2000] sous le nom de HALF SYNC, HALF ASYNC, est largement utilisé dans les protocoles de communication.

Architectures multiétages

Le développement des systèmes répartis a promu une forme différente d'architecture multiniveaux. Considérons l'évolution historique d'une forme usuelle d'applications client-serveur, dans laquelle les demandes d'un client sont traitées en utilisant l'information stockée dans une base de données.

Dans les années 1970 (Figure 2.5a), les fonctions de gestion de données et l'application elle-même sont exécutées sur un serveur central (*mainframe*). Le poste du client est un simple terminal, qui réalise une forme primitive d'interface utilisateur.

Dans les années 1980 (Figure 2.5b), les stations de travail apparaissent comme machines clientes, et permettent de réaliser des interfaces graphique élaborées pour l'utilisateur. Les capacités de traitement de la station cliente lui permettent en outre de participer au traitement de l'application, réduisant ainsi la charge du serveur et améliorant la capacité de croissance (car l'addition d'une nouvelle station cliente ajoute de la puissance de traitement

⁶Une interface complexe peut aussi être partitionnée en plusieurs APIs, chacune étant liée à une fonction spécifique.

pour les applications).

L'inconvénient de cette architecture est que l'application est maintenant à cheval sur les machines client et serveur ; l'interface de communication est à présent interne à l'application. Une modification de cette dernière peut maintenant impliquer des changements à la fois sur les machines client et serveur, et éventuellement une modification de l'interface de communication.

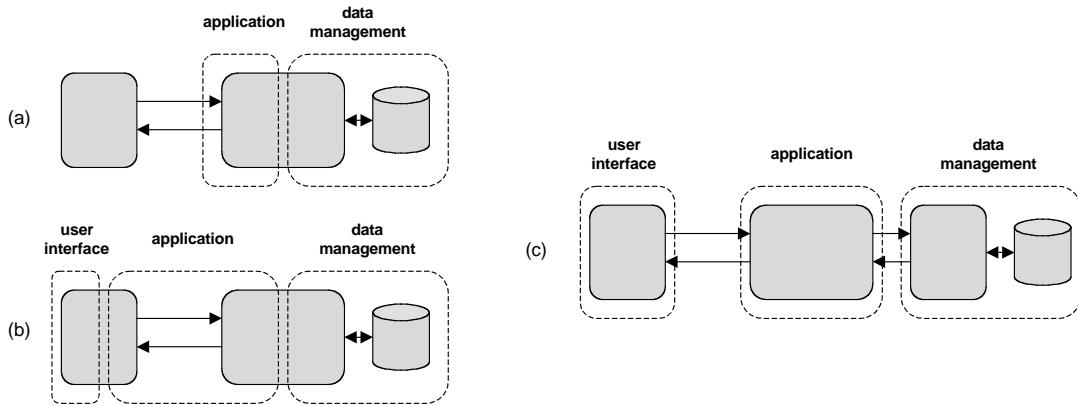


Figure 2.5 – Architectures multiétages

Ces défauts sont corrigés par l'architecture décrite sur la Figure 2.5c, introduite à la fin des années 1990. Les fonctions de l'application sont partagées entre trois machines : la station client ne réalise que l'interface graphique, l'application proprement dite réside sur un serveur dédié, et la gestion de la base de données est dévolue à une autre machine. Chacune de ces divisions « horizontales » est appelée un *étage* (en anglais *tier*). Une spécialisation plus fine des fonctions donne lieu à d'autres architectures multiétages. Noter que chaque étage peut lui-même faire l'objet d'une décomposition « verticale » en niveaux d'abstraction.

L'architecture multiétages conserve l'avantage du passage à grande échelle, à condition que les serveurs puissent être renforcés de manière incrémentale (par exemple en ajoutant des machines à une grappe). En outre les interfaces entre étages peuvent être conçues pour favoriser la séparation de préoccupations, puisque les interfaces logiques coïncident maintenant avec les interfaces de communication. Par exemple, l'interface entre l'étage d'application et l'étage de gestion de données peut être rendue générique, pour accepter facilement un nouveau type de base de données, ou pour intégrer une application patrimoniale, en utilisant un adaptateur (section 2.3.4) pour la conversion d'interface.

Des exemples d'architectures multiétages sont présentés dans le chapitre 5.

Canevas

Un canevas logiciel (en anglais *framework*) est un squelette de programme qui peut être directement réutilisé, ou adapté selon des règles bien définies, pour résoudre une famille de problèmes apparentés. Cette définition recouvre de nombreux cas d'espèce ; nous nous intéressons ici à une forme particulière de canevas composée d'une infrastructure

dans laquelle des composants logiciels peuvent être insérés en vue de fournir des services spécifiques. Ces canevas illustrent des notions relatives aux interfaces, aux rappels et à l'inversion du contrôle.

Le premier exemple (Figure 2.6a) est le micronoyau, une architecture introduite dans les années 1980 et visant à développer des systèmes d'exploitation facilement configurables. Un système d'exploitation à micronoyau se compose de deux couches :

- Le micronoyau proprement dit, qui gère les ressources matérielles (processeurs, mémoire, entrées-sorties, interface de réseau), et fournit au niveau supérieur une API abstraite de gestion de ressources.
- Le noyau, qui réalise un système d'exploitation spécifique (une « personnalité ») en utilisant l'API du micronoyau.

Un noyau de système d'exploitation construit sur un micronoyau est généralement organisé comme un ensemble de *serveurs*, dont chacun est chargé d'une fonction spécifique (gestion de processus, système de fichiers, etc.). Un appel système typique émis par une application est traité comme suit .

- Le noyau analyse l'appel et active le micronoyau en utilisant la fonction appropriée de son API.
- Le micronoyau rappelle un serveur dans le noyau. Au retour de cet appel ascendant, le micronoyau peut interagir avec le matériel ; cette séquence peut être itérée, par exemple si plusieurs serveurs sont en jeu.
- Le micronoyau rend la main au noyau, qui termine le travail et revient à l'application.

Pour ajouter une nouvelle fonction à un noyau, il faut donc développer et intégrer un nouveau serveur.

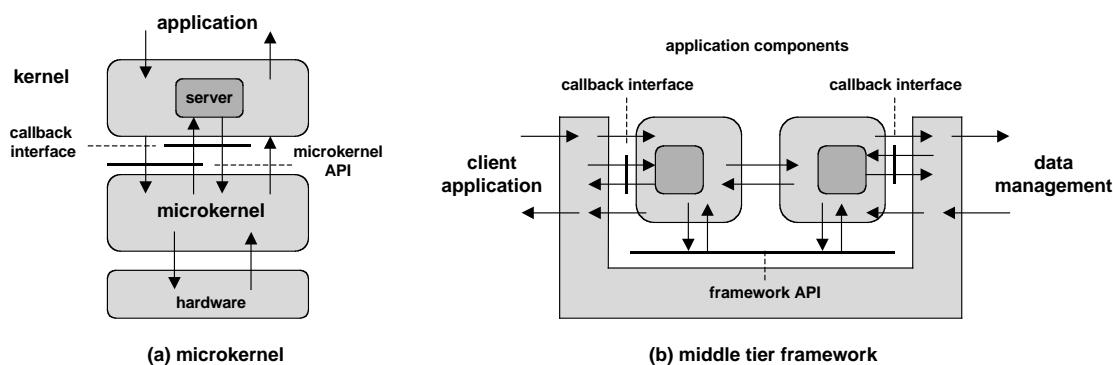


Figure 2.6 – Architectures de canevas

Le second exemple (Figure 2.6b) illustre l'organisation typique de l'étage médian d'une architecture client-serveur à 3 étages. Ce canevas interagit avec l'étage client et avec l'étage de gestion de données, et sert de médiateur pour l'interaction entre ces étages et le programme de l'application proprement dite. Ce programme est organisé comme un ensemble de composants, qui utilisent l'API fournie par le canevas et doivent fournir un ensemble d'interfaces de rappel. Ainsi une requête d'un client est traitée par le canevas, qui active les composants applicatifs appropriés, interagit avec eux en utilisant ses propres API et l'interface de rappel des composants, et retourne finalement au client.

Des exemples détaillés de cette organisation sont présentés au chapitre 5.

Les deux exemples ci-dessus illustrent l'inversion du contrôle. Pour fournir ses services, le canevas utilise des rappels vers les modules logiciels externes (serveurs dans l'exemple micronoyau, ou composants applicatifs dans l'étage médian). Ces modules doivent respecter le contrat du canevas, en fourni ssant des interfaces de rappel spécifiées et en utilisant l'API du canevas.

Les organisations en couches et en étages définissent une structure à gros grain pour un système complexe. L'organisation interne de chaque couche ou étage (ou couche dans un étage) utilise elle-même des entités de grain plus fin. Les objets, un moyen usuel de définir cette structure fine, sont présentés dans la section suivante.

2.2.2 Objets répartis

Programmation par objets

Les objets ont été introduits dans les années 1960 comme un moyen de structuration des systèmes logiciels. Il existe de nombreuses définitions des objets, mais les propriétés suivantes en capturent les concepts plus courants, particulièrement dans le contexte de la programmation répartie.

Un *objet*, dans un modèle de programmation, est une représentation logicielle d'une entité du monde réel (telle qu'une personne, un compte bancaire, un document, une voiture, etc.). Un objet est l'association d'un état et d'un ensemble de procédures (ou méthodes) qui opèrent sur cet état. Le modèle d'objets que nous considérons a les propriétés suivantes.

- *Encapsulation*. Un objet a une interface, qui comprend un ensemble de méthodes (procédures) et d'attributs (valeurs qui peuvent être lues et modifiées). La seule manière d'accéder à un objet (pour consulter ou modifier son état) est d'utiliser son interface. Les seules parties de l'état visibles depuis l'extérieur de l'objet sont celles explicitement présentes dans l'interface ; l'utilisation d'un objet ne doit reposer sur aucune hypothèse sur sa réalisation. Le type d'un objet est défini par son interface. Comme indiqué en 2.1.2, l'encapsulation assure l'indépendance entre interface et réalisation. L'interface joue le rôle d'un contrat entre l'utilisateur et le réalisateur d'un objet. Un changement dans la réalisation d'un objet est fonctionnellement invisible à ses utilisateurs, tant que l'interface est préservée.
- *Classes et instances*. Une *classe* est une description générique commune à un ensemble d'objets (les instances de la classe). Les instances d'une classe ont la même interface (donc le même type), et leur état a la même structure ; mais chaque instance a son propre exemplaire de l'état, et elle est identifiée comme une entité distincte. Les instances d'une classe sont créées dynamiquement, par une opération appelée *instanciation* ; elles peuvent aussi être dynamiquement détruites, soit explicitement soit automatiquement (par un ramasse-miettes) selon la réalisation spécifique du modèle d'objet.
- *Héritage*. Une classe peut dériver d'une autre classe par spécialisation, autrement dit par définition de méthodes et/ou d'attributs supplémentaires, ou par redéfinition (surcharge) de méthodes existantes. On dit que la classe dérivée *étend* la classe initiale (ou classe de base) ou qu'elle *hérite* de cette classe. Certains modèles permettent à une classe d'hériter de plus d'une classe (héritage multiple).
- *Polymorphisme*. Le polymorphisme est la capacité, pour une méthode, d'accepter des

paramètres de différents types et d'avoir un comportement différent pour chacun de ces types. Ainsi un objet peut être remplacé, comme paramètre d'une méthode, par un objet « compatible ». La notion de compatibilité, ou conformité (voir section 2.1.3) est exprimée par une relation entre types, qui dépend du modèle spécifique de programmation ou du langage utilisé.

Rappelons que ces définitions ne sont pas universelles, et ne sont pas applicables à tous les modèles d'objets (par exemple il y a d'autres mécanismes que les classes pour créer des instances, les objets peuvent être actifs, etc.), mais elles sont représentatives d'un vaste ensemble de modèles utilisés dans la pratique, et sont mises en œuvre dans des langages tels que Smalltalk, C++, Eiffel, Java, ou C#.

Objets distants

Les propriétés ci-dessus font que les objets sont un bon mécanisme de structuration pour les systèmes répartis.

- L'hétérogénéité est un trait dominant de ces systèmes. L'encapsulation est un outil puissant dans un environnement hétérogène : l'utilisateur d'un objet doit seulement connaître une interface pour cet objet, qui peut avoir des réalisations différentes sur différents sites.
- La création dynamique d'instances d'objets permet de construire un ensemble d'objets ayant la même interface, éventuellement sur des sites distants différents ; dans ce cas l'intergiciel doit fournir un mécanisme pour la création d'objets distants, sous la forme de fabriques (voir section 2.3.2).
- L'héritage est un mécanisme de réutilisation, car il permet de définir une nouvelle interface à partir d'une interface existante. Il est donc utile pour les développeurs d'applications réparties, qui travaillent dans un environnement changeant et doivent définir de nouvelles classes pour traiter des nouvelles situations. Pour utiliser l'héritage, on conçoit d'abord une classe (de base) générique pour capturer un ensemble de traits communs à une large gamme de situations attendues. Des classes spécifiques, plus spécialisées, sont alors définies par extension de la classe de base. Par exemple, une interface pour un flot vidéo en couleur peut être définie comme une extension de celle d'un flot vidéo générique. Une application qui utilise des flots d'objets vidéo accepte aussi des flots d'objets en couleur, puisque ces objets réalisent l'interface des flots vidéo (c'est un exemple de polymorphisme).

La manière la plus simple et la plus courante pour répartir des objets est de permettre aux objets qui constituent une application d'être situés sur un ensemble de sites répartis (autrement dit, l'objet est l'unité de répartition ; d'autres méthodes permettent de partitionner la représentation d'un objet entre plusieurs sites). Une application cliente peut utiliser un objet situé sur un site distant en appelant une méthode de l'interface de l'objet, comme si l'objet était local. Des objets utilisés de cette manière sont appelés *objets distants*, et leur mode d'interaction est l'appel d'objet distant (*Remote Method Invocation*) ; c'est la transposition du RPC au monde des objets.

Les objets distants sont un exemple d'une organisation client-serveur. Comme un client peut utiliser plusieurs objets différents situés sur un même site distant, des termes distincts sont utilisés pour désigner le site distant (le site *serveur*) et un objet individuel qui fournit un service spécifique (un objet *servant*). Pour que le système fonctionne, un intergiciel

approprié doit localiser une réalisation de l'objet servant sur un site éventuellement distant, envoyer les paramètres sur l'emplacement de l'objet, réaliser l'appel effectif, et renvoyer les résultats à l'appelant. Un intergiciel qui réalise ces fonctions est un courtier d'objets répartis (en anglais *Object Request Broker*, ou ORB).

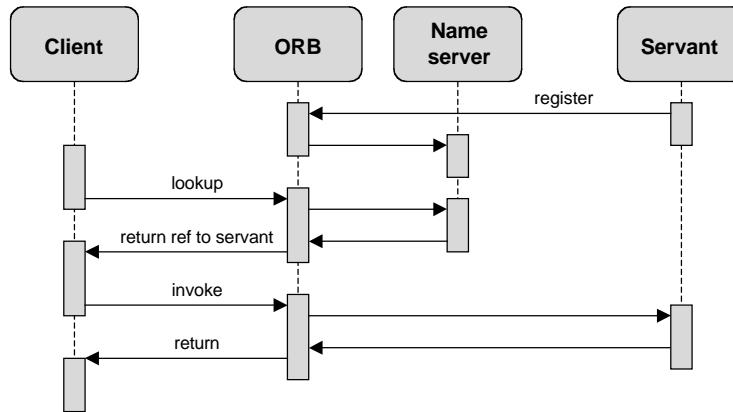


Figure 2.7 – Appel de méthode à distance

La structure d'ensemble d'un appel à un objet distant (Figure 2.7) est semblable à celle d'un RPC : l'objet distant doit d'abord être localisé, ce qui est généralement fait au moyen d'un serveur des noms ou d'un service vendeur (*trader*) ; l'appel proprement dit est ensuite réalisé. L'ORB sert de médiateur aussi bien pour la recherche que pour l'appel.

2.3 Patrons pour l'intergiciel à objets répartis

Les mécanismes d'exécution à distance reposent sur quelques patrons de conception qui ont été largement décrits dans la littérature, en particulier dans [Gamma et al. 1994], [Buschmann et al. 1995], et [Schmidt et al. 2000]. Dans cette présentation, nous mettons l'accent sur l'utilisation spécifique de ces patrons pour l'intergiciel réparti à objets, et nous examinons leurs similitudes et leurs différences. Pour une discussion plus approfondie de ces patrons, voir les références indiquées.

2.3.1 *Proxy*

PROXY (ce terme anglais est traduit par « représentant » ou « mandataire ») est un des premiers patrons de conception identifiés en programmation répartie [Shapiro 1986, Buschmann et al. 1995]. Nous n'examinons ici que son utilisation pour les objets répartis, bien que son domaine d'application ait été étendu à de nombreuses autres constructions.

1. **Contexte.** Le patron PROXY est utilisé pour des applications organisées comme un ensemble d'objets dans un environnement réparti, communiquant au moyen d'appels de méthode à distance : un client demande un service fourni par un objet éventuellement distant (le servant).

2. **Problème.** Définir un mécanisme d'accès qui n'implique pas de coder « en dur » l'emplacement du servant dans le code client, et qui ne nécessite pas une connaissance détaillée des protocoles de communication par le client.
 3. **Propriétés souhaitées.** L'accès doit être efficace à l'exécution. La programmation doit être simple pour le client ; idéalement, il ne doit pas y avoir de différence entre accès local et accès distant (cette propriété est appelée transparence d'accès).
 4. **Contraintes.** La principale contrainte résulte de l'environnement réparti : le client et le serveur sont dans des espaces d'adressage différents.
 5. **Solution.** Utiliser un représentant local du serveur sur le site du client. Ce représentant, ou mandataire, a exactement la même interface que le servant. Toute l'information relative au système de communication et à la localisation du servant est cachée dans le mandataire, et ainsi rendue invisible au client.
- L'organisation d'un mandataire est illustrée sur la figure 2.8.

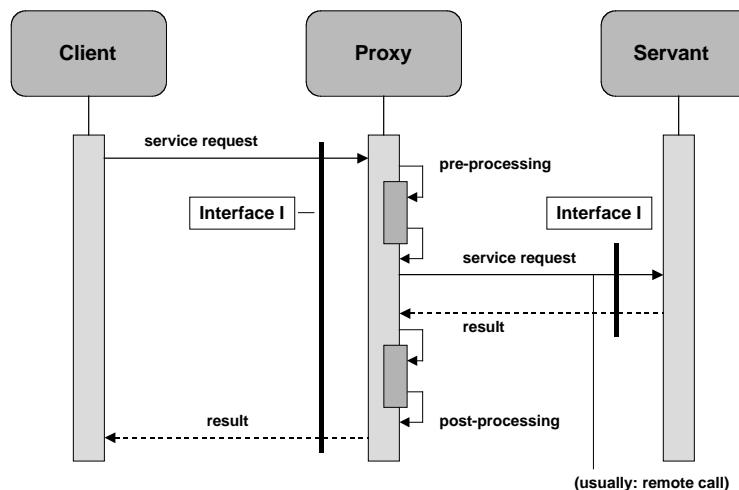


Figure 2.8 – Le patron PROXY

La structure interne d'un mandataire suit un schéma bien défini, qui facilite sa génération automatique.

- une phase de pré-traitement, qui consiste essentiellement à emballer les paramètres et à préparer le message de requête,
- l'appel effectif du servant, utilisant le protocole de communication sous-jacent pour envoyer la requête et pour recevoir la réponse,
- une phase de post-traitement, qui consiste essentiellement à déballer les valeurs de retour.

6. Usages connus.

Dans la construction de l'intergiciel, les mandataires sont utilisés comme représentants locaux pour des objets distants. Ils n'ajoutent aucune fonction. C'est le cas des souches (*stubs*) et des squelettes utilisés dans RPC ou Java-RMI.

Des variantes des *proxies* contiennent des fonctions supplémentaires. Des exemples en sont les caches et l'adaptation côté client. Dans ce dernier cas, le *proxy* peut

filtrer la sortie du serveur pour l'adapter aux capacités spécifiques d'affichage du client (couleur, résolution, etc.). De tels mandataires « intelligents » (*smart proxies*) combinent les fonctions standard d'un mandataire avec celles d'un intercepteur (voir section 2.3.5).

7. Références.

Une discussion du patron PROXY peut être trouvée dans [Gamma et al. 1994], [Buschmann et al. 1995].

2.3.2 *Factory*

1. **Contexte.** On considère des applications organisées comme un ensemble d'objets dans un environnement réparti (la notion d'objet dans ce contexte peut être très générale, et n'est pas limitée au domaine strict de la programmation par objets).
2. **Problème.** On souhaite pouvoir créer dynamiquement des familles d'objets apparentés (par exemple des instances d'une même classe), tout en permettant de reporter certaines décisions jusqu'à la phase d'exécution (par exemple le choix d'une classe concrète pour réaliser une interface donnée).
3. **Propriétés souhaitées.** Les détails de réalisation des objets créés doivent être invisibles. Le processus de création doit pouvoir être paramétré. L'évolution du mécanisme doit être facilitée (pas de décision « en dur »).
4. **Contraintes.** La principale contrainte résulte de l'environnement réparti : le client (qui demande la création de l'objet) et le serveur (qui crée effectivement l'objet) sont dans des espaces d'adressage différents.
5. **Solution.** Utiliser deux patrons corrélés : une usine abstraite ABSTRACT FACTORY définit une interface et une organisation génériques pour la création d'objets ; la création est déléguée à des usines concrètes. ABSTRACT FACTORY peut être réalisé en utilisant FACTORY METHODS (une méthode de création redéfinie dans une sous-classe).

Un autre manière d'améliorer la souplesse est d'utiliser une usine de fabrication d'usines, comme illustré sur la Figure 2.9 (le mécanisme de création est lui-même paramétré).

Un usine peut aussi être utilisée comme un gestionnaire des objets qu'elle a créés, et peut ainsi réaliser une méthode pour localiser un objet (en renvoyant une référence pour cet objet), et pour détruire un objet sur demande.

6. Usages connus.

FACTORY est un des patrons les plus largement utilisés dans l'intergiciel. Il sert à la fois dans des applications (pour créer des instances distantes d'objets applicatifs) et dans l'intergiciel lui-même (un exemple courant est l'usine à liaisons). Les usines sont aussi utilisées en liaison avec les composants (voir chapitres 5 et 7).

7. Références.

Les deux patrons ABSTRACT FACTORY et FACTORY METHOD sont décrits dans [Gamma et al. 1994].

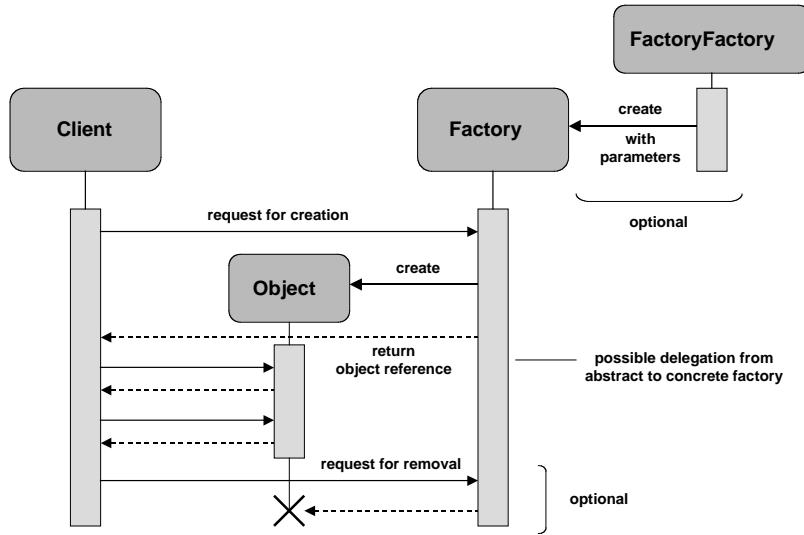


Figure 2.9 – Le patron FACTORY

2.3.3 Pool

Le patron POOL est un complément à FACTORY, qui vise à réduire le temps d'exécution de la création et de la destruction d'objets, en construisant à l'avance (lors d'une phase d'initialisation) une réserve (*pool*) d'objets. Cela se justifie si le coût de la création et de la destruction est élevé par rapport à celui des opérations sur la réserve. Les opérations de création et de destruction deviennent alors :

```

Obj create(params) {
    if (pool empty)
        obj = new Obj
        /* utilise Factory */
    else
        obj = pool.get()
        obj.init(params)
    return (obj)
}

remove(Obj obj) {
    if (pool full)
        delete(obj)
    else {
        obj.cleanup()
        pool.put(obj)
    }
}

```

Les opérations `init` et `cleanup` permettent respectivement, si nécessaire, d'initialiser l'état de l'objet créé et de remettre l'objet dans un état neutre.

On a supposé ici que la taille de la réserve était fixe. Il est possible d'ajuster la taille du pool en fonction de la demande observée. On peut encore raffiner le fonctionnement en maintenant le nombre d'objets dans la réserve au-dessus d'un certain seuil, en déclenchant les créations nécessaires si ce nombre d'objets tombe au-dessous du seuil. Cette régulation peut éventuellement se faire en travail de fond pour profiter des temps libres.

Trois cas fréquents d'usage de ce patron sont :

- La gestion de la mémoire. Dans ce cas, on peut prévoir plusieurs réserves de zones préallouées de tailles différentes, pour répondre aux demandes le plus fréquemment

observées.

- La gestion des *threads* ou des processus.
- La gestion des composants dans certains canevas (par exemple les *Entity Beans* dans la plate-forme EJB, voir chapitre 5)

Dans tous ces cas, le coût élevé de création des entités justifie largement l'usage d'une réserve.

2.3.4 Adapter

1. **Contexte.** Le contexte est celui de la fourniture de services, dans un environnement réparti : un service est défini par une interface ; les clients demandent des services ; des servants, situés sur des serveurs distants, fournissent des services.
2. **Problème.** On souhaite réutiliser un servant existant en le dotant d'une nouvelle interface conforme à celle attendue par un client (ou une classe de clients).
3. **Propriétés souhaitées.** Le mécanisme de conversion d'interface doit être efficace à l'exécution. Il doit aussi être facilement adaptable, pour répondre à des changements imprévus des besoins. Il doit être réutilisable (c'est-à-dire générique).
4. **Contraintes.** Pas de contraintes spécifiques.
5. **Solution.** Fournir un composant (l'adaptateur, ou *wrapper*) qui isole le servant en interceptant les appels de méthode à son interface. Chaque appel est précédé par un prologue et suivi par un épilogue dans l'adaptateur (Figure 2.10). Les paramètres et résultats peuvent nécessiter une conversion.

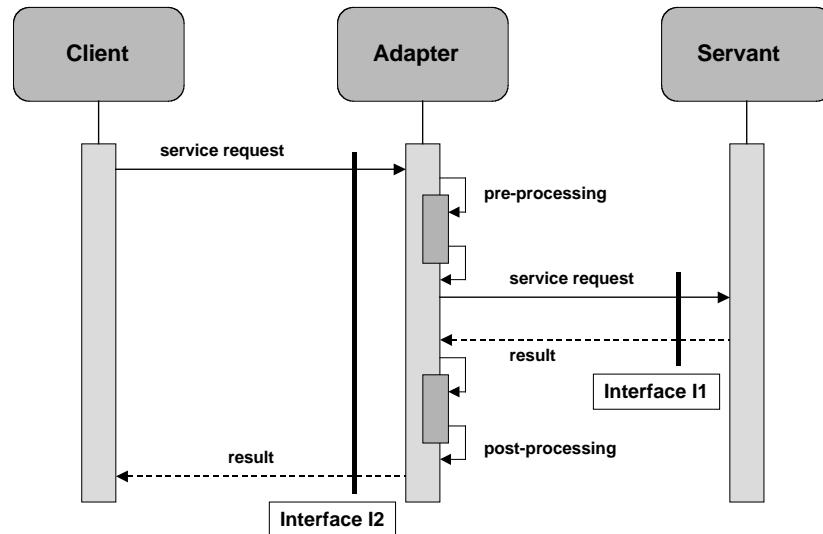


Figure 2.10 – Le patron ADAPTER

Dans des cas simples, un adaptateur peut être automatiquement engendré à partir d'une description des interfaces fournie et requise.

6. Usages connus.

Les adaptateurs sont largement utilisés dans l'intergiciel pour encapsuler des fonctions côté serveur. Des exemples sont le *Portable Object Adapter* (POA) de CORBA et les divers adaptateurs pour la réutilisation de logiciel patrimoniaux (*legacy systems*), tel que *Java Connector Architecture* (JCA).

7. Références.

ADAPTER (également appelé WRAPPER) est décrit dans [Gamma et al. 1994]. Un patron apparenté est WRAPPER FAÇADE ([Schmidt et al. 2000]), qui fournit une interface de haut niveau (par exemple sous forme d'objet) à des fonctions de bas niveau.

2.3.5 *Interceptor*

1. **Contexte.** Le contexte est celui de la fourniture de services, dans un environnement réparti : un service est défini par une interface ; les clients demandent des services ; les servants, situés sur des serveurs distants, fournissent des services. Il n'y a pas de restrictions sur la forme de la communication (uni- ou bi-directionnelle, synchrone ou asynchrone, etc.).
2. **Problème.** On veut ajouter de nouvelles capacités à un service existant, ou fournir le service par un moyen différent.
3. **Propriétés souhaitées.** Le mécanisme doit être générique (applicable à une large variété de situations). Il doit permettre de modifier un service aussi bien statiquement (à la compilation) que dynamiquement (à l'exécution).
4. **Contraintes.** Les services peuvent être ajoutés ou supprimés dynamiquement.
5. **Solution.** Créer (statiquement ou dynamiquement) des objets d'interposition, ou intercepteurs. Ces objets interceptent les appels (et/ou les retours) et insèrent un traitement spécifique, qui peut être fondé sur une analyse du contenu. Un intercepteur peut aussi rediriger un appel vers une cible différente.

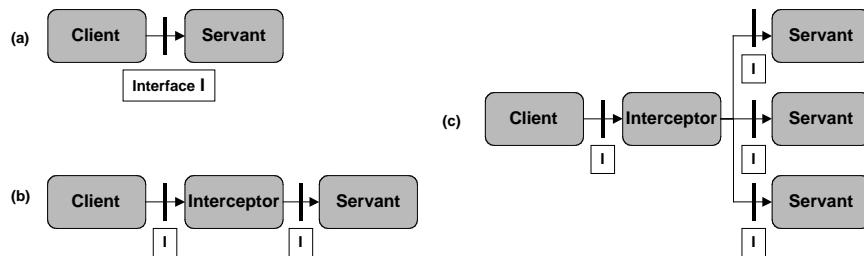


Figure 2.11 – Formes simples d'intercepteur

Ce mécanisme peut être réalisé sous différentes formes. Dans la forme la plus simple, un intercepteur est un module qui est inseré à un point spécifié dans le chemin d'appel entre le demandeur et le fournisseur d'un service (Figure 2.11a et 2.11b). Il peut aussi être utilisé comme un aiguillage entre plusieurs servants qui peuvent fournir le même service avec différentes options (Figure 2.11c), par exemple l'ajout de fonctions de tolérance aux fautes, d'équilibrage de charge ou de caches.

Sous une forme plus générale (Figure 2.12), intercepteurs et fournisseurs de service (servants) sont gérés par une infrastructure commune et créés sur demande. L'intercepteur utilise l'interface du servant et peut aussi s'appuyer sur des services fournis par l'infrastructure. Le servant peut fournir des fonctions de rappel utilisables par l'intercepteur.

6. Usages connus.

Les intercepteurs sont utilisés dans une grande variété de situations dans les systèmes intergiciels.

- pour ajouter des nouvelles capacités à des applications ou systèmes existants. Un exemple ancien est le mécanisme des « sous-contrats »[Hamilton et al. 1993]. Les *Portable Interceptors* de CORBA donnent une manière systématique d'étendre les fonctions du courtier d'objets (ORB) par insertion de modules d'interception en des points prédefinis dans le chemin d'appel. Un autre usage est l'aide aux mécanismes de tolérance aux fautes (par exemple la gestion de groupes d'objets).
- pour choisir une réalisation spécifique d'un servant à l'exécution.
- pour réaliser un canevas pour des applications à base de composants (voir chapitres 5 et 7).
- pour réaliser un intergiciel réflexif (voir 2.4.1 and 2.4.3).

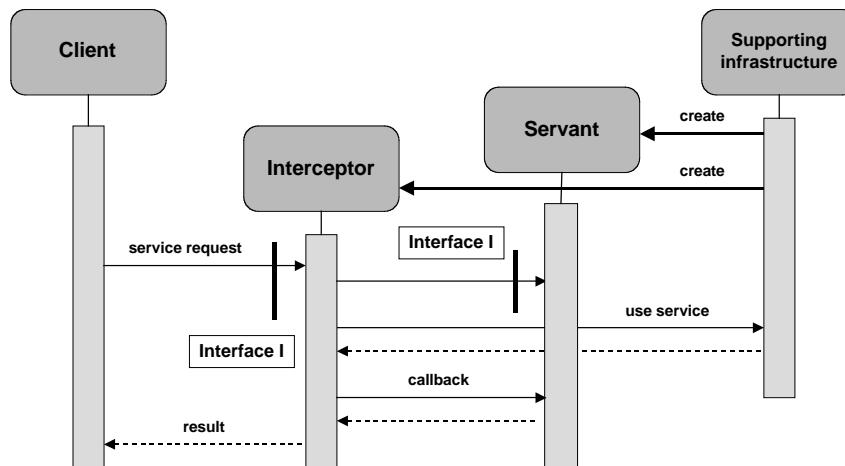


Figure 2.12 – Forme générale d'un intercepteur

7. Références.

Le patron INTERCEPTOR est décrit dans [Schmidt et al. 2000].

2.3.6 Comparaison et combinaison des patrons

Trois des patrons décrits ci-dessus (PROXY, ADAPTER, et INTERCEPTOR) ont d'étoiles relations mutuelles. Ils reposent tous trois sur un module logiciel inseré entre le demandeur et le fournisseur d'un service. Nous résumons ci-après leurs analogies et leurs différences.

- **ADAPTER vs PROXY**. ADAPTER et PROXY ont une structure semblable. PROXY préserve l'interface, alors qu'ADAPTER transforme l'interface. En outre, PROXY

implique souvent (pas toujours) un accès à distance, alors que ADAPTER est généralement local.

- **ADAPTER vs INTERCEPTOR.** ADAPTER et INTERCEPTOR ont une fonction semblable : l'un et l'autre modifient un service existant. La principale différence est que ADAPTER transforme l'interface, alors que INTERCEPTOR transforme la fonction (de fait INTERCEPTOR peut complètement masquer la cible initiale de l'appel, en la remplaçant par un servant différent).
- **PROXY vs INTERCEPTOR.** Un PROXY peut être vu comme une forme spéciale d'un INTERCEPTOR, dont la fonction se réduit à acheminer une requête vers un servant distant, en réalisant les transformations de données nécessaires à la transmission, d'une manière indépendante des protocoles de communication. En fait, comme mentionné dans 2.3.1, un *proxy* peut être combiné avec un intercepteur, devenant ainsi « intelligent » (c'est-à-dire fournissant de nouvelles fonctions en plus de la transmission des requêtes, mais laissant l'interface inchangée).

En utilisant les patrons ci-dessus, on peut tracer un premier schéma grossier et incomplet de l'organisation d'ensemble d'un ORB (Figure 2.13).

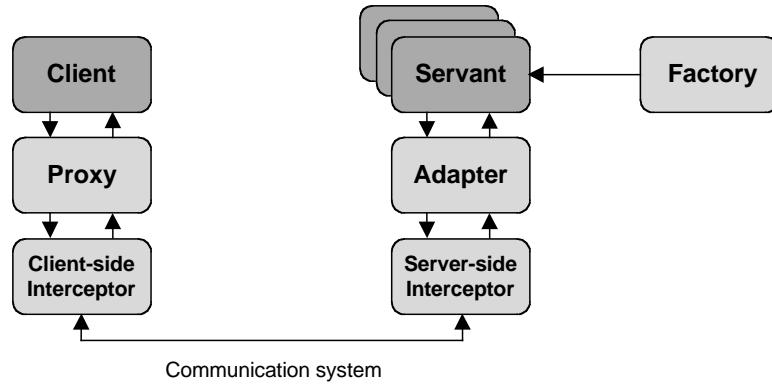


Figure 2.13 – Utilisation de patrons dans un ORB

Les principaux aspects manquants sont ceux relatifs à la liaison et à la communication.

2.4 Adaptabilité et séparation des préoccupations

Trois approches principales sont utilisées pour assurer l'adaptabilité et la séparation de préoccupations dans les systèmes intergiciels : les protocoles à méta-objets, la programmation par aspects, et les approches pragmatiques. Elles sont résumées dans les sections qui suivent.

2.4.1 Protocoles à méta-objets

La réflexion a été introduite au chapitre 1, section 1.4.2. Rappelons qu'un système réflexif est capable d'examiner et de modifier son propre comportement, en utilisant une représentation causalement connectée de lui-même.

La réflexion est une propriété intéressante pour un intericiel, parce qu'un tel système fonctionne dans un environnement qui évolue, et doit adapter son comportement à des besoins changeants. Des capacités réflexives sont présentes dans la plupart des systèmes intergiciels existants, mais sont généralement introduites localement, pour des traits isolés. Des plates-formes intergicielles dont l'architecture de base intègre la réflexion sont développées comme prototypes de recherche [RM 2000].

Une approche générale de la conception d'un système réflexif consiste à l'organiser en deux niveaux.

- Le *niveau de base*, qui fournit les fonctions définies par les spécifications du système.
- Le *méta-niveau*, qui utilise une représentation des entités du niveau de base pour observer ou modifier le comportement de ce niveau.

Cette décomposition peut être itérée en considérant le méta-niveau comme un niveau de base pour un méta-méta-niveau, et ainsi de suite, définissant ainsi une « tour réflexive ». Dans la plupart des cas pratiques, la tour est limitée à deux ou trois niveaux.

La définition d'une représentation du niveau de base, destinée à être utilisée par le méta-niveau, est un processus appelé *réification*. Il conduit à définir des méta-objets, dont chacun est une représentation, au méta-niveau, d'une structure de données ou d'une opération définie au niveau de base. Le fonctionnement des méta-objets, et leur relation aux entités du niveau de base, sont spécifiées par un *protocole à méta-objets* (MOP) [Kiczales et al. 1991].

Un exemple simple de MOP (emprunté à [Bruneton 2001]) est la réification d'un appel de méthode dans un système réflexif à objets. Au méta-niveau, un méta-objet `Méta_Obj` est associé à chaque objet `Obj`. L'exécution d'un appel de méthode `Obj.meth(params)` comporte les étapes suivantes (Figure 2.14).

1. L'appel de méthode est réifié dans un objet `m`, qui contient une représentation de `meth` et `params`. La forme précise de cette représentation est définie par le MOP. Cet objet `m` est transmis au méta-objet, qui exécute `Méta_Obj.metaMethodCall(m)`.

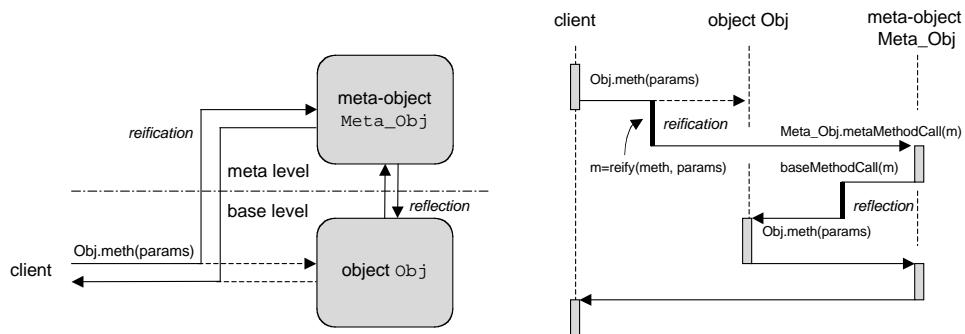


Figure 2.14 – Exécution d'un appel de méthode dans un système réflexif

2. La méthode `méta_MethodCall(m)` exécute alors les traitements spécifiés par le MOP. Pour prendre un exemple simple, il peut imprimer le nom de la méthode en vue d'une trace avant son exécution effective (en appelant une méthode telle que `m.methName.printName()`) ou il peut sauvegarder l'état de l'objet avant l'appel de

la méthode pour permettre un retour en arrière (*undo*), ou il peut vérifier la valeur des paramètres, etc.

3. Le méta-objet peut maintenant effectivement exécuter l'appel initial⁷, en appelant une méthode `baseMethodCall(m)` qui exécute essentiellement `Obj.meth(params)`⁸. Cette étape (l'inverse de la réification) est appelée *réflexion*.
4. Le méta-objet exécute alors tout post-traitement défini par le MOP, et retourne à l'appelant initial.

De même, l'opération de création d'objet peut être réifiée en appelant une usine à méta-objets (au méta-niveau). Cette usine crée un objet au niveau de base, en utilisant l'usine de ce niveau ; le nouvel objet fait alors un appel descendant (*upcall*) à l'usine à méta-objets, qui crée le méta-objet associé, et exécute toutes les opérations supplémentaires spécifiées par le MOP (Figure 2.15).

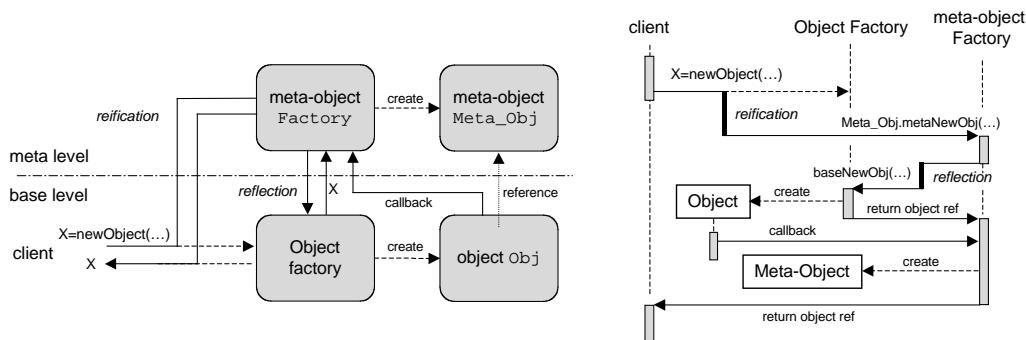


Figure 2.15 – Création d'objet dans un système réflexif

2.4.2 Programmation par aspects

La programmation par aspects (en anglais *Aspect-Oriented Programming* ou AOP) [Kiczales 1996] est motivée par les remarques suivantes.

- Beaucoup de préoccupations différentes (ou « aspects ») sont généralement présentes dans une application (des exemples usuels sont la sécurité, la persistance, la tolérance aux fautes, et d'autres aspects extra-fonctionnels).
- Le code lié à ces préoccupations est souvent étroitement imbriqué avec le code « fonctionnel » de l'application, ce qui rend les modifications et les additions difficiles et sujettes aux erreurs.

Le but de l'AOP est de définir des méthodes et outils pour mieux identifier et isoler le code relatif aux divers aspects présents dans une application. Plus précisément, une application développée avec l'AOP est construite en deux phases.

⁷Il n'exécute pas nécessairement l'appel initial ; par exemple, si le MOP est utilisé pour la protection, il peut décider que l'appel ne doit pas être exécuté, et revenir à l'appelant avec un message de violation de protection.

⁸Noter qu'il n'est pas possible d'appeler directement `Obj.meth(params)` parce que seule la forme réifiée de l'appel de méthode est accessible au méta-objet et aussi parce qu'une étape de post-traitement peut être nécessaire.

- La partie principale de l'application (le programme de base), et les parties qui traitent des différents aspects supplémentaires sont écrites indépendamment, en utilisant éventuellement des langages spécialisés pour le code des aspects.
- Toutes ces parties sont intégrées pour former l'application globale, en utilisant un outil de composition, le tisseur d'aspects (*aspect weaver*).

Un point de jonction (*join point*) est un emplacement, dans le code source du programme de base, où du code lié aux aspects peut être inséré. Le tissage d'aspects repose sur deux notions principales : le point de coupure (*point cut*), c'est-à-dire la spécification d'un ensemble de points de jonction selon un critère donné, et l'indication (*advice*), c'est-à-dire la définition de l'interaction du code inséré avec le code de base. Par exemple, si l'AOP est ajouté à un langage à objets, un point de coupure particulier peut être défini comme l'ensemble des points d'appel à une famille de méthodes (spécifiée par une expression régulière), ou l'ensemble des appels à un constructeur spécifié, etc. Une indication spécifie si le code inséré doit être exécuté avant, après, ou en remplacement des opérations situées aux points de coupure (dans le dernier cas, ces opérations peuvent toujours être appelées depuis le code inséré). La composition peut être faite statiquement (à la compilation), dynamiquement (à l'exécution), ou en combinant des techniques statiques et dynamiques.

Un problème important de l'AOP est la composition des aspects. Par exemple, si différents fragments de code liés aux aspects sont insérés au même point de jonction, l'ordre d'insertion peut être significatif si les aspects correspondants ne sont pas indépendants. Cette question ne peut généralement pas être décidée par le tisseur et nécessite une spécification supplémentaire.

Deux exemples d'outils qui réalisent l'AOP sont AspectJ [Kiczales et al. 2001] et JAC [Pawlak et al. 2001]. Ils s'appliquent à des programmes de base en Java.

AspectJ

AspectJ permet de définir les aspects en spécifiant ces derniers et le programme de base dans un code source Java, qui peut alors être compilé.

Un exemple simple donne une idée des capacités d'AspectJ. Le code présenté Figure 2.16 décrit un aspect, sous la forme de définition de points de coupure et d'indications.

```
public aspect MethodWrapping{
    /* définition de point de coupure */
    pointcut Wrappable(): call(public * MyClass.*(..));

    /* définition d'indication      */
    around(): Wrappable() {
        prelude ;    /* séquence de code devant être insérée avant un appel */
        proceed (); /* exécution d'un appel à la méthode originelle      */
        postlude   /* séquence de code devant être insérée après un appel */
    }
}
```

Figure 2.16 – Définition d'un aspect en AspectJ.

La première partie de la description définit un point de coupure (*point cut*) comme tout appel d'une méthode publique de la classe `MyClass`. La partie indication (*advice*) indique qu'un appel à une telle méthode doit être remplacé par un prologue spécifié, suivi par un appel à la méthode d'origine, suivi par un épilogue spécifié. De fait, cela revient à placer une simple enveloppe (sans modification d'interface) autour de chaque appel de méthode spécifié dans la définition du point de coupure. Cela peut être utilisé pour ajouter des capacités de journalisation à une application existante, ou pour insérer du code de test pour évaluer des pré- et post-conditions dans une conception par contrat (2.1.3).

Une autre capacité d'AspectJ est l'*introduction*, qui permet d'insérer des déclarations et méthodes supplémentaires à des endroits spécifiés dans une classe ou une interface existante. Cette possibilité doit être utilisée avec précaution, car elle peut violer le principe d'encapsulation.

JAC

JAC (*Java Aspect Components*) a des objectifs voisins de ceux d'AspectJ. Il permet d'ajouter des capacités supplémentaires (mise sous enveloppe de méthodes, introduction) à une application existante. JAC diffère d'AspectJ sur les points suivants.

- JAC n'est pas une extension de langage, mais un canevas qui peut être utilisé à l'exécution. Ainsi des aspects peuvent être dynamiquement ajoutés à une application en cours d'exécution. JAC utilise la modification du *bytecode*, et le code des classes d'application est modifié lors du chargement des classes.
- Les *point cuts* et les *advices* sont définis séparément. La liaison entre *point cuts* et *advices* est retardée jusqu'à la phase de tissage ; elle repose sur des informations fournies dans un fichier de configuration séparé. La composition d'aspects est définie par un protocole à météo-objets.

Ainsi JAC autorise une programmation souple, mais au prix d'un surcoût à l'exécution dû au tissage dynamique des aspects dans le *bytecode*.

2.4.3 Approches pragmatiques

Les approches pragmatiques de la réflexion dans l'intergiciel s'inspirent des approches systématiques ci-dessus, mais les appliquent en général de manière ad hoc, essentiellement pour des raisons d'efficacité. Ces approches sont principalement fondées sur l'interception.

Beaucoup de systèmes intergiciels définissent un chemin d'appel depuis un client vers un serveur distant, traversant plusieurs couches (application, intergiciel, système d'exploitation, protocoles de communication). Les intercepteurs peuvent être insérés en divers points de ce chemin, par exemple à l'envoi et à la réception des requêtes et des réponses.

L'insertion d'intercepteurs permet une extension non-intrusive des fonctions d'un intergiciel, sans modifier le code des applications ou l'intergiciel lui-même. Cette technique peut être considérée comme une manière ad hoc pour réaliser l'AOP : les points d'insertion sont les points de jonction et les intercepteurs réalisent directement les aspects. En spécifiant convenablement les points d'insertion pour une classe donnée d'intergiciel, conforme à une norme spécifique (par exemple CORBA, EJB), les intercepteurs peuvent être rendus génériques et peuvent être réutilisés avec différentes réalisations de cette norme. Les fonctions qui peuvent être ajoutées ou modifiées par des intercepteurs sont notamment

la surveillance (*monitoring*), la journalisation, l'enregistrement de mesures, la sécurité, la gestion de caches, l'équilibrage de charge, la duplication.

Cette technique peut aussi être combinée avec un protocole à méta-objets, l'intercepteur pouvant être inséré dans la partie réifiée du chemin d'appel (donc dans un métaniveau).

Les techniques d'interception entraînent un surcoût à l'exécution. Ce coût peut être réduit par l'usage de l'*injection de code*, c'est-à-dire par intégration directe du code de l'intercepteur dans le code du client ou du serveur (c'est l'analogue de l'insertion (*inlining*) du code des procédures dans un compilateur optimisé). Pour être efficace, cette injection doit être réalisée à bas niveau, c'est-à-dire dans le langage d'assemblage, ou (pour Java) au niveau du *bytecode*, grâce à des outils appropriés tels que BCEL [BCEL], Javassist [Tatsubori et al. 2001], ou ASM [ASM 2002]. Pour préserver la souplesse d'utilisation, il doit être possible d'annuler le processus d'injection de code en revenant au format de l'interception. Un exemple d'utilisation de l'injection de code peut être trouvé dans [Hagimont and De Palma 2002].

2.4.4 Comparaison des approches

Les principales approches de la séparation de préoccupations dans l'intergiciel peuvent être comparées comme suit.

1. Les approches fondées sur les protocoles à méta-objets (MOP) sont les plus générales et les plus systématiques. Néanmoins, elles entraînent un surcoût potentiel dû au va et vient entre métaniveau et niveau de base.
2. Les approches fondées sur les aspects (AOP) agissent à un grain plus fin que celles utilisant les MOPs et accroissent la souplesse d'utilisation, au détriment de la généralité. Les deux approches peuvent être combinées ; par exemple les aspects peuvent être utilisés pour modifier les opérations aussi bien au niveau de base qu'aux métaniveaux.
3. Les approches fondées sur l'interception ont des capacités restreintes par rapport à MOP ou AOP, mais apportent des solutions acceptables dans de nombreuses situations pratiques. Elles manquent toujours d'un modèle formel pour les outils de conception et de vérification.

Dans tous les cas, des techniques d'optimisation fondées sur la manipulation de code de bas niveau peuvent être appliquées. Ce domaine fait l'objet d'une importante activité.

2.5 Note historique

Les préoccupations architecturales dans la conception du logiciel apparaissent vers la fin des années 1960. Le système d'exploitation THE [Dijkstra 1968] est un des premiers exemples de système complexe conçu comme une hiérarchie de machines abstraites. La notion de programmation par objets est introduite dans le langage Simula-67 [Dahl et al. 1970]. La construction modulaire, une approche de la composition systématique de programmes comme un assemblage de parties, apparaît à cette période. Des règles de conception développées pour l'architecture et l'urbanisme [Alexander 1964]

sont transposées à la conception de programmes et ont une influence significative sur l'émergence des principes du génie logiciel [Naur and Randell 1969].

La notion de patron de conception vient de la même source, une dizaine d'années plus tard [Alexander et al. 1977]. Avant même que cette notion soit systématiquement utilisée, les patrons élémentaires décrits dans le présent chapitre sont identifiés. Des formes simples d'enveloppes (*wrappers*) sont développées pour convertir des données depuis un format vers un autre, par exemple dans le cadre des systèmes de bases de données, avant d'être utilisées pour transformer les méthodes d'accès. Une utilisation notoire des intercepteurs est la réalisation du premier système réparti de gestion de fichiers, Unix United [Brownbridge et al. 1982] : une couche logicielle interposée à l'interface des appels système Unix permet de rediriger de manière transparente les opérations sur les fichiers distants. Cette méthode sera plus tard étendue [Jones 1993] pour inclure du code utilisateur dans les appels système. Des intercepteurs en pile, côté client et côté serveur, sont introduits dans [Hamilton et al. 1993] sous le nom de *subcontracts*. Diverses formes de mandataires sont utilisés pour réaliser l'exécution à distance, avant que le patron soit identifié [Shapiro 1986]. Les usines semblent être d'abord apparues dans la conception d'interfaces graphiques (par exemple [Weinand et al. 1988]), dans lesquelles un grand nombre d'objets paramétrés (boutons, cadres de fenêtres, menus, etc.) sont créés dynamiquement.

L'exploration systématique des patrons de conception de logiciel commence à la fin des années 1980. Après la publication de [Gamma et al. 1994], l'activité se développe dans ce domaine, avec la création de la série des conférences PLoP [PLoP] et la publication de plusieurs livres spécialisés [Buschmann et al. 1995, Schmidt et al. 2000, Völter et al. 2002].

L'idée de la programmation réflexive est présente sous diverses formes depuis les origines (par exemple dans le mécanisme d'évaluation des langages fonctionnels tels que Lisp). Les premiers essais d'usage systématique de cette notion datent du début des années 1980 (par exemple le mécanisme des métaclasses dans Smalltalk-80) ; les bases du calcul réflexif sont posées dans [Smith 1982]. La notion de protocole à météo-objets [Kiczales et al. 1991] est introduite pour le langage CLOS, une extension objet de Lisp. L'intergiciel réflexif [Kon et al. 2002] fait l'objet de nombreux travaux depuis le milieu des années 1990, et commence à s'introduire dans les systèmes commerciaux (par exemple via le standard CORBA pour les intercepteurs portables).

Chapitre 3

Le système de composants Fractal

Les approches à base de composants apparaissent de plus en plus incontournables pour le développement de systèmes et d'applications répartis. Il s'agit de faire face à la complexité sans cesse croissante de ces logiciels et de répondre aux grands défis de l'ingénierie des systèmes : passage à grande échelle, administration, autonomie.

Après les objets dans la première moitié des années 1990, les composants se sont imposés comme le paradigme clé de l'ingénierie des intergiciels et de leurs applications dans la seconde moitié des années 1990. L'intérêt de la communauté industrielle et académique s'est d'abord porté sur les modèles de composants pour les applications comme EJB, CCM ou .NET. À partir du début des années 2000, le champ d'application des composants s'est étendu aux couches inférieures : systèmes et intergiciels. Il s'agit toujours, comme pour les applications, d'obtenir des entités logicielles composables aux interfaces spécifiées contractuellement, déployables et configurables ; mais il s'agit également d'avoir des plates-formes à composants suffisamment performantes et légères pour ne pas pénaliser les performances du système. Le modèle de composants Fractal remplit ces conditions.

Ce chapitre présente les principes de base du modèle Fractal (section 3.1). Les plates-formes mettant en œuvre ce modèle sont présentées dans la section 3.2. L'accent est mis sur deux d'entre elles, Julia (section 3.2.1) et AOKell (section 3.2.2). Les autres plates-formes existantes sont présentées brièvement en section 3.2.3. La section 3.3 présente le langage Fractal ADL qui permet de construire des assemblages de composants Fractal. La section 3.4 présente quelques bibliothèques de composants disponibles pour le développement d'applications Fractal, dont Dream (section 3.4.1) dédiée au développement d'intergiciels. La section 3.5 compare Fractal à des modèles de composants existants. Finalement, la section 3.6 conclut ce chapitre.

3.1 Le modèle Fractal : historique, définition et principes

Le modèle de composants Fractal est un modèle général dédié à la construction, au déploiement et à l'administration (e.g. observation, contrôle, reconfiguration dynamique) de systèmes logiciels complexes, tels les intergiciels ou les systèmes d'exploitation.

Le modèle de composants Fractal a été défini par France Telecom R&D et l'INRIA. Il se présente sous la forme d'une spécification et d'implémentations dans différents langages de programmation comme Java, C, C++, SmallTalk ou les langages de la plate-forme .NET. Fractal est organisé comme un projet du consortium ObjectWeb pour le *middleware open source*. Les premières discussions autour du modèle Fractal, initiées dès 2000 à France Telecom R&D dans la lignée du projet Jonathan [Dumont et al. 1998], ont abouti en juin 2002 avec la première version officielle de la spécification et la première version de Julia, qui est l'implémentation de référence de cette spécification. La spécification a évolué pour aboutir en septembre 2003 à une deuxième version comportant un certain nombre de changements au niveau de l'API. Dès le départ, un langage de description d'architecture, Fractal ADL, a été associé au modèle de composants. Basé sur une syntaxe *ad hoc* au départ, il a évolué et sa version 2, définie en janvier 2004 et implémentée en mars 2004, est basée sur une syntaxe extensible.

Les caractéristiques principales du modèle Fractal sont motivées par l'objectif de pouvoir construire, déployer et administrer des systèmes complexes tels que des intergiciels ou des systèmes d'exploitation. Le modèle est ainsi basé sur les principes suivants :

- **composants composites** (i.e. composants qui contiennent des sous-composants) pour permettre d'avoir une vue uniforme des applications à différents niveaux d'abstraction.
- **composants partagés** (i.e. sous-composants de plusieurs composites englobants) pour permettre de modéliser les ressources et leur partage, tout en préservant l'enveloppement des composants.
- **capacités d'introspection** pour permettre d'observer l'exécution d'un système.
- **capacités de (re)configuration** pour permettre de déployer et de configurer dynamiquement un système.

Par ailleurs, Fractal est un modèle *extensible* du fait qu'il permet au développeur de personnaliser les capacités de contrôle de chacun des composants de l'application. Il est ainsi possible d'obtenir un continuum dans les capacités réflexives d'un composant allant de l'absence totale de contrôle à des capacités élaborées d'introspection et d'intercession (e.g. accès et manipulation du contenu d'un composant, contrôle de son cycle de vie). Ces fonctionnalités sont définies au sens d'entités appelées contrôleur

Il est ainsi possible de distinguer différents rôles dans les activités de développement autour du modèle Fractal :

- les **développeurs de composants applicatifs** s'intéressent à la construction d'applications et de systèmes à l'aide de Fractal. Ils développent des composants et des assemblages de composants à l'aide de l'API Fractal et du langage de description d'architecture Fractal ADL. Ces composants utilisent des contrôleur et des plates-formes existants.
- les **développeurs de contrôleur** s'intéressent à la personnalisation du contrôle offerts aux composants. Ils développent de nouvelles politiques de contrôle comportant plus ou moins de fonctionnalités extra-fonctionnelles et permettant d'adapter les applications à différents contextes d'exécution (par exemple avec des ressources plus ou moins contraintes). Le développement de contrôleur est conduit à l'aide des mécanismes offerts par les plates-formes : par exemple, *mixin* pour Julia (voir section 3.2.1) ou aspect pour AOKell (voir section 3.2.2).

- les **développeurs de plates-formes** fournissent un environnement permettant d'exécuter des applications et des systèmes écrits à l'aide de composants Fractal. Un développeur de plate-forme fournit une implémentation des spécifications Fractal dans un langage de programmation et des mécanismes pour personnaliser le contrôle. On trouve ainsi des plates-formes dans différents langages comme Java ou C (voir section 3.2).

La section suivante introduit les éléments disponibles pour le développement d'applications et de systèmes avec Fractal. La section 3.3 reviendra sur le langage Fractal ADL et expliquera notamment comment il est possible de l'étendre. La section 3.1.2 introduit la notion de contrôleur qui sera reprise dans chacune des sections consacrées aux plates-formes existantes (voir respectivement les sections 3.2.1 et 3.2.2 sur Julia et AOKell).

3.1.1 Composants Fractal

La base du développement Fractal réside dans l'écriture de composants et de liaisons permettant aux composants de communiquer. Ces composants peuvent être typés. Finalement, le langage Fractal ADL constitue le vecteur privilégié pour la composition de composants.

Composants et liaisons

Un composant Fractal est une entité d'exécution qui possède une ou plusieurs interfaces. Une *interface* est un point d'accès au composant. Une interface implante un *type d'interface* qui spécifie les opérations supportées par l'interface. Il existe deux catégories d'interfaces : les interfaces *serveurs* — qui correspondent aux services fournis par le composant —, et les interfaces *clients* qui correspondent aux services requis par le composant.

Un composant Fractal est généralement composé de deux parties : une *membrane* — qui possède des interfaces fonctionnelles et des interfaces permettant l'introspection et la configuration (dynamique) du composant —, et un *contenu* qui est constitué d'un ensemble fini de *sous-composants*.

Les interfaces d'une membrane sont soit *externes*, soit *internes*. Les interfaces externes sont accessibles de l'extérieur du composant, alors que les interfaces internes sont accessibles par les sous-composants du composant. La membrane d'un composant est constituée d'un ensemble de *contrôleurs*. Les contrôleurs peuvent être considérés comme des météo-objets. Chaque contrôleur a un rôle particulier : par exemple, certains contrôleurs sont chargés de fournir une représentation causale connectée de la structure d'un composant (en termes de sous-composants). D'autres contrôleurs permettent de contrôler le comportement d'un composant et/ou de ses sous-composants. Un contrôleur peut, par exemple, permettre de suspendre/reprendre l'exécution d'un composant.

Le modèle Fractal fournit deux mécanismes permettant de définir l'architecture d'une application : l'*imbrication* (à l'aide des composants composites) et la *liaison*. La liaison est ce qui permet aux composants Fractal de communiquer. Fractal définit deux types de liaisons : primitive et composite. Les liaisons *primitives* sont établies entre une interface client et une interface serveur de deux composants résidant dans le même espace d'adres-sage. Par exemple, une liaison primitive dans le langage C (resp. Java) est implantée à

l'aide d'un pointeur (resp. référence). Les liaisons *composites* sont des chemins de communication arbitrairement complexes entre deux interfaces de composants. Les liaisons composites sont constituées d'un ensemble de composants de liaison (e.g. *stub*, *skeleton*) reliés par des liaisons primitives.

Une caractéristique originale du modèle Fractal est qu'il permet de construire des composants *partagés*. Un composant partagé est un composant qui est inclus dans plusieurs composites. De façon paradoxale, les composants partagés sont utiles pour préserver l'encapsulation. En effet, il n'est pas nécessaire à un composant de bas niveau d'exporter une interface au niveau du composite qui l'encapsule pour accéder à une interface d'un composant partagé. De fait, les composants partagés sont particulièrement adaptés à la modélisation des ressources.

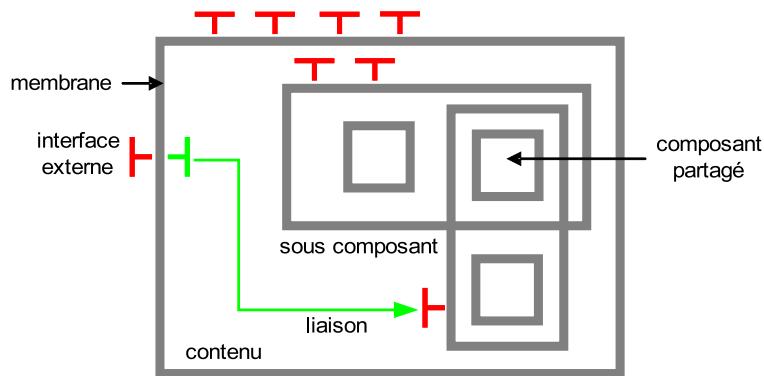


Figure 3.1 – Exemple de composant Fractal.

La figure 3.1 représente un exemple de composant Fractal. Les composants sont représentés par des rectangles. Le tour gris du carré correspond à la membrane du composant. L'intérieur du carré correspond au contenu du composant du composant. Les interfaces sont représentées par des "T" (gris clair pour les interfaces clients ; gris foncé pour les interfaces serveurs). Notons que les interfaces internes permettent à un composite de contrôler l'exposition de ses interfaces externes à ses sous-composants. Les interfaces externes apparaissant au sommet des composants sont les interfaces de contrôle du composant. Les flèches représentent les liaisons entre composants. Enfin, nous avons représenté un composant partagé entre deux composites.

Système de types

Le modèle Fractal définit un système de types optionnel. Ce système de types autorise la description des opérations supportées par les différentes interfaces d'un composant. Il permet également de préciser le rôle de chacune des interfaces (i.e. client ou serveur), ainsi que sa cardinalité et sa contingence. La contingence d'une interface indique s'il est garanti que les opérations fournies ou requises d'une interface seront présentes ou non à l'exécution :

- les opérations d'une interface *obligatoire* sont toujours présentes. Pour une interface client, cela signifie que l'interface doit être liée pour que le composant s'exécute.

- les opérations d'une interface *optionnelle* ne sont pas nécessairement disponibles. Pour un composant serveur, cela peut signifier que l'interface interne complémentaire n'est pas liée à un sous-composant implantant l'interface. Pour un composant client, cela signifie que le composant peut s'exécuter sans que son interface soit liée.

La cardinalité d'une interface de type *T* spécifie le nombre d'interfaces de type *T* que le composant peut avoir. Une cardinalité *singleton* signifie que le composant doit avoir une, et seulement une, interface de type *T*. Une cardinalité *collection* signifie que le composant peut avoir un nombre arbitraire d'interfaces du type *T*. Ces interfaces sont généralement créées de façon paresseuse à l'exécution. Fractal n'impose aucune contrainte sur la sémantique opérationnelle des interfaces de cardinalité *collection*. Il est ainsi possible de considérer les interfaces *collection* comme une collection d'interfaces dans laquelle chaque élément est traité comme une interface de cardinalité *singleton*, ou de considérer que l'invocation de méthode sur une interface *collection* provoque la diffusion du message à l'ensemble des composants liés à cette interface.

Le système de types permet également de décrire le type d'un composant comme l'ensemble des types de ses interfaces. Notons que le système de types définit une relation de sous-typage qui permet de vérifier des contraintes sur la substituabilité des composants.

Assemblage de composants

Le langage Fractal ADL permet de décrire, à l'aide d'une syntaxe XML, des assemblages de composants Fractal. Nous verrons à la section 3.3 que la DTD de ce langage n'est pas fixe, mais peut être étendue pour prendre en compte des propriétés extra-fonctionnelles.

La figure 3.3 donne un exemple de définition réalisée à l'aide de Fractal ADL. L'assemblage correspond est représenté figure 3.2. Le composant décrit est un composite dont le nom est `HelloWorld`. Ce composite possède une interface serveur, de nom `r` et de signature `java.lang.Runnable`. Par ailleurs, le composite encapsule deux composants : `Client` et `Server`. La définition du composant `Client` est intégrée à celle du composant `HelloWorld` : le composant a deux interfaces (`r` et `s`), sa classe d'implantation est `org.objectweb.julia.example.ClientImpl` et il possède une partie de contrôle de type `primitive`¹. La définition du composant `Server` suit le même principe : une interface serveur `s` est définie et `org.objectweb.julia.example.ServerImpl` correspond à la classe d'implantation. Enfin, la description ADL mentionne deux liaisons : entre les interfaces `r` du composite et du `Client` et entre les interfaces `s` du `Client` et du `Server`.

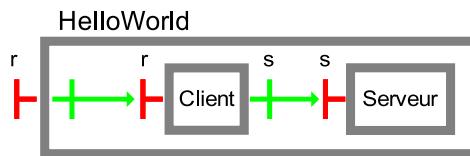


Figure 3.2 – Assemblage Fractal correspondant à l'ADL de la figure 3.3.

¹Les deux mots utilisés pour décrire les parties contrôle des composants (`primitive` et `composite`) sont définis dans un fichier de configuration qui permet de spécifier l'ensemble des contrôleurs des différents composants.

```
<definition name="HelloWorld">
  <interface name="r" role="server" signature="java.lang.Runnable"/>
  <component name="Client">
    <interface name="r" role="server" signature="java.lang.Runnable"/>
    <interface name="s" role="client" signature="Service"/>
    <content class="org.objectweb.julia.example.ClientImpl"/>
    <controller desc="primitive"/>
  </component>
  <component name="Server">
    <interface name="s" role="server" signature="Service"/>
    <content class="org.objectweb.julia.example.ServerImpl"/>
    <controller desc="primitive"/>
  </component>
  <binding client="this.r" server="Client.r"/>
  <binding client="Client.s" server="Server.s"/>
  <controller desc="composite"/>
</definition>
```

Figure 3.3 – Un exemple de définition ADL à l'aide du langage extensible Fractal ADL.

```
import org.objectweb.fractal.api.control.BindingController;

public class Client implements Runnable, BindingController {

  // Implementation de Runnable
  public void run() {
    service.print("HelloWorld!");
  }

  // Implementation de BindingController
  public String[] listFc() { return new String[] {"s"}; }
  public Object lookupFc( String cItf ) {
    if (cItf.equals("s")) { return service; }
    return null;
  }
  public void bindFc( String cItf, Object sItf ) {
    if (cItf.equals("s")) { service = (Service)sItf; }
  }
  public void unbindFc( String cItf ) {
    if (cItf.equals("s")) { service = null; }
  }
  private Service service;
}

public class ServerImpl implements Service {
  public void print( String msg ) {
    System.err.println(msg);
  }
}
```

Figure 3.4 – Implémentation des composants Client et Server.

La figure 3.4 fournit une implémentation des composants **Client** et **Server** mentionnés dans l’assemblage. La classe **ClientImpl** correspond au composant **Client** et fournit une implémentation pour l’interface **r** de type `java.lang.Runnable`. Par ailleurs, le composant **Client** manipule sa liaison avec le composant **Server** : il doit pour cela implémenter l’interface **BindingController** définie dans l’API Fractal. Le contrôleur de liaison notifiera cette implémentation de l’occurrence de toute opération concernant la gestion des liaisons de ce composant. Finalement, la classe **ServerImpl** implémente le composant **Server**.

3.1.2 Contrôleurs

Le modèle de composants Fractal n’impose la présence d’aucun contrôleur dans la membrane d’un composant. Il permet, au contraire, de créer des formes arbitrairement complexes de membranes implantant diverses sémantiques de contrôle. La spécification Fractal [Bruneton et al. 2003] définit un certain nombre de niveaux de contrôle. En l’absence de contrôle, un composant Fractal est une boîte noire qui ne permet ni introspection, ni intercession. Les composants ainsi construits sont comparables aux objets instanciés dans les langages à objets comme Java. L’intérêt de ces composants réside dans le fait qu’ils permettent d’intégrer facilement des logiciels patrimoniaux.

Au niveau de contrôle suivant, un composant Fractal fournit une interface **Component**, similaire à l’interface **IUnknown** du modèle COM [Rogerson 1997]. Cette interface donne accès aux interfaces externes (clients ou serveurs) du composant. Chaque interface a un nom qui permet de la distinguer des autres interfaces du composant.

Au niveau de contrôle supérieur, un composant Fractal possède des interfaces réifiant sa structure interne et permettant de contrôler son exécution. La spécification Fractal définit différents contrôleurs :

- le **contrôleur d’attributs** pour configurer les attributs d’un composant.
- le **contrôleur de liaisons** pour créer/rompre une liaison primitive entre deux interfaces de composants.
- le **contrôleur de contenu** pour ajouter/retrancher des sous-composants au contenu d’un composant composite.
- le **contrôleur de cycle de vie** pour contrôler les principales phases comportementales d’un composant. Par exemple, les méthodes de base fournies par un tel contrôleur permettent de démarrer et stopper l’exécution du composant.

Au delà de cet ensemble prédéfini par les spécifications, les développeurs peuvent implémenter leurs propres contrôleurs pour étendre ou spécialiser les capacités réflexives de leurs composants.

3.2 Plates-formes

Fractal est un modèle de composant indépendant des langages de programmation. Plusieurs plates-formes sont ainsi disponibles dans différents langages de programmation. Julia (voir section 3.2.1) l’implémentation de référence de Fractal, a été développée pour le langage Java. Une deuxième plate-forme Java, AOKEll, développée plus récemment est présentée en section 3.2.2. Par rapport à Julia, AOKEll apporte une mise sous forme de

composants des membranes. La section 3.2.3 présente un aperçu des autres plates-formes existantes.

3.2.1 Julia

Julia est l'implémentation de référence du modèle de composant Fractal. Sa première version remonte à juin 2002. Julia est disponible sous licence *open source* LGPL sur le site du projet Fractal².

Julia est un canevas logiciel écrit en Java qui permet de programmer les membranes des composants. Il fournit un ensemble de contrôleurs que l'utilisateur peut assembler. Par ailleurs, Julia fournit des mécanismes d'optimisation qui permettent d'obtenir un continuum allant de configurations entièrement statiques et très efficaces à des configurations dynamiquement reconfigurables et moins performantes. Le développeur d'application peut ainsi choisir l'équilibre performance/dynamicité dont il a besoin. Enfin, notons que Julia s'exécute sur toute JVM, y compris celles qui ne fournissent ni chargeur de classe, ni API de réflexivité.

Principales structures de données

Un composant Fractal est formé de plusieurs objets Java que l'on peut séparer en trois groupes (figure 3.5) :

- Les objets qui implémentent le contenu du composant. Ces objets n'ont pas été représentés sur la figure. Ils peuvent être des sous-composants (dans le cas de composants composites) ou des objets Java (pour les composants primitifs).
- Les objets qui implémentent la partie de contrôle du composant (représentés en gris). Ces objets peuvent être séparés en deux groupes : les objets implémentant les interfaces de contrôle et des intercepteurs optionnels qui interceptent les appels de méthodes entrants et sortants. Les fonctions de contrôle n'étant généralement pas indépendantes, les contrôleurs et les intercepteurs possèdent généralement des références les uns vers les autres.
- Les objets qui référencent les interfaces du composant (en blanc). Ces objets sont le seul moyen pour un composant de posséder des références vers un autre composant.

La mise en place de ces différents objets est effectuée par des fabriques de composants. Celles-ci fournissent une méthode de création qui prend en paramètres la description des parties fonctionnelle et de contrôle du composant.

Développement des contrôleurs

Fractal étant un modèle de composants extensible, il est nécessaire de pouvoir construire facilement diverses formes de contrôleurs et diverses sémantiques de contrôle. Par exemple, si l'on considère l'interface de contrôle de liaisons (`BindingController`), il est nécessaire de fournir différentes implantations de cette interface qui diffèrent par les vérifications qu'elles font lors de la création/destruction d'une liaison : interaction avec le contrôleur de cycle de vie pour vérifier qu'un composant est stoppé, vérification que les types d'interface sont compatibles quand un système de types est utilisé, vérification que

²fractal.objectweb.org

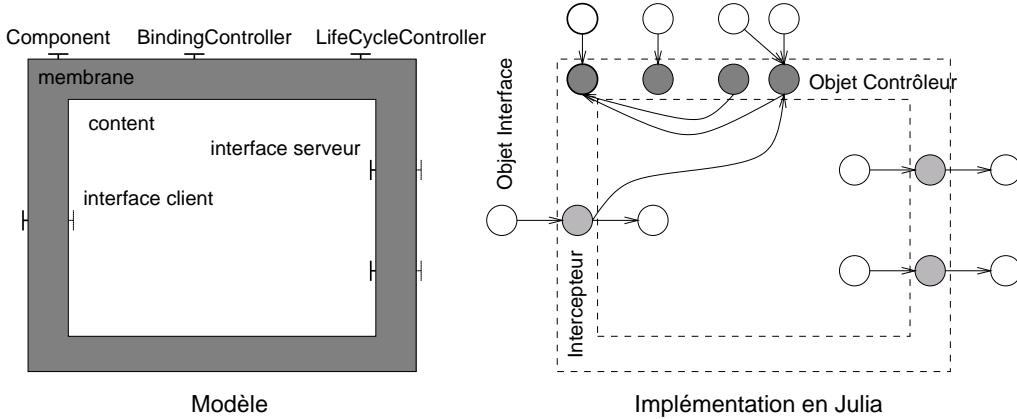


Figure 3.5 – Un composant Fractal et son implantation Julia.

les composants liés sont parents d'un même composite quand les contrôleurs de contenu sont utilisés, etc.

Il n'est pas envisageable d'utiliser l'héritage de classe pour fournir ces différentes implantations. En effet cela conduirait à une explosion combinatoire du nombre de classes nécessaires. Supposons que l'on souhaite effectuer des vérifications concernant le système de types, le cycle de vie et le contrôleur de contenu. Il existe $2^3 = 8$ combinaisons possibles de ces différentes vérifications. De fait, pour planter toutes les combinaisons possibles, il serait nécessaire de fournir huit classes, ce qui engendrerait de nombreuses duplications de code.

La solution adoptée dans Julia est l'utilisation de *classes mixin* [Bracha and Cook 1990] : une classe *mixin* est une classe dont la super-classe est spécifiée de manière abstraite en indiquant les champs et méthodes que cette super-classe doit posséder. La classe *mixin* peut s'appliquer (c'est-à-dire surcharger et ajouter des méthodes) à toute classe qui possède les caractéristiques de cette super-classe. Ces classes *mixin* sont appliquées au chargement à l'aide de l'outil ASM [ASM 2002]. Dans Julia, les classes *mixin* sont des classes abstraites développées avec certaines conventions. En l'occurrence, elles ne nécessitent pas l'utilisation d'un compilateur Java modifié ou d'un pré-processeur comme c'est le cas des classes *mixins* développées à l'aide d'extensions du langage Java. Par exemple la classe *mixin* JAM [Ancona et al. 2000] illustrée sur la partie gauche de la figure 3.6 s'écrit en Julia en pur Java (partie droite). Le mot clé *inherited* en JAM est équivalent au préfixe *_super_* utilisé dans Julia. Il permet de spécifier les membres qui doivent être présents dans la classe de base pour que le *mixin* lui soit appliqué. De façon plus précise, le préfixe *_super_* spécifie les méthodes qui sont surchargées par le *mixin*. Les méthodes qui sont requises mais pas surchargées sont spécifiées à l'aide du préfixe *_this_*.

L'application de la classe *mixin A* à la classe *Base* décrite sur la partie gauche de la figure 3.7 donne la classe *C55d992cb_0* représentée sur la partie droite de la figure 3.7.

```

 mixin  Compteur {
     inherited public void  m ();
     public int  count;
     public void  m () {
         ++count;
        super.m();
    }
}

 abstract class  Compteur {
     abstract void  _super_m ();
     public int  count;
     public void  m () {
         ++count;
        _super_m ();
    }
}

```

Figure 3.6 – Ecriture d'une classe mixin en JAM et en Julia.

```

 abstract class  Base {
     public void  m () {
        System.out.println ("m");
    }
}

 public class  C55d992cb_0  implements 
    Generated {
    // from Base
     private void  m$0 () {
        System.out.println ("m");
    }
    // from A
     public int  count;
     public void  m () {
         ++count;
        m$0();
    }
}

```

Figure 3.7 – Application d'une classe mixin.

Développement des intercepteurs

Julia donne la possibilité de développer des *intercepteurs* dont le rôle est d'intercepter les appels de méthode entrant et/ou sortant des interfaces d'un composant. Les intercepteurs doivent implémenter les interfaces interceptées. Cependant, il est inconcevable de développer, pour un aspect de contrôle donné, autant d'intercepteurs qu'il y a d'interfaces à intercepter dans l'application. En conséquence, Julia fournit un outil, appelé *générateur d'intercepteurs*, qui permet de générer dynamiquement le code de ces intercepteurs. Cette génération est effectuée à partir d'informations fournies par le développeur comme par exemple les blocs de code à exécuter avant et après l'interception.

Optimisations

Julia offre deux mécanismes d'optimisation, intra et inter composants. Le premier mécanisme permet de réduire l'empreinte mémoire d'un composant en fusionnant une partie de ses objets de contrôle. Pour ce faire, Julia fournit un outil utilisant ASM [ASM 2002] et imposant certaines contraintes sur les objets de contrôle fusionnés : par exemple, deux objets fusionnés ne peuvent pas implémenter la même interface.

Le second mécanisme d'optimisation a pour fonction d'optimiser les chaînes de liaison entre composants : il permet de court-circuiter les parties contrôle des composants qui n'ont pas d'intercepteurs. Comme nous l'avons expliqué au paragraphe 3.2.1, chaque interface serveur de composant est représentée par un objet qui contient une référence vers un

objet implantant réellement l'interface. Le principe du mécanisme de court-circuitage est représenté sur la figure 3.8 : un composant primitif est relié à un composite exportant l'interface d'un composant primitif qu'il encapsule. En conséquence, il existe deux objets de référencement d'interface (r_1 et r_2). Seuls les appels du primitif sont interceptés (objet i_1). En conséquence, Julia court-circuite l'objet r_2 et r_1 référence directement i_1 .

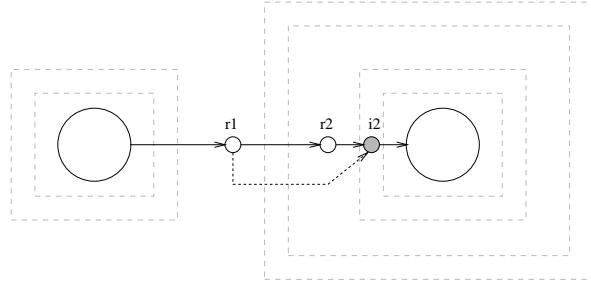


Figure 3.8 – Optimisation des chaînes de liaison.

3.2.2 AOKell

Comme Julia, le canevas logiciel AOKell [Seinturier et al. 2006] est une implémentation complète des spécifications Fractal. Le respect de l'API Fractal permet ainsi d'exécuter telle quelle, avec AOKell, des applications conçues pour Julia ou vice-versa. AOKell est disponible sous licence *open source* LGPL sur le site du projet Fractal³. Le développement de AOKell a débuté en décembre 2004.

Le canevas logiciel AOKell diffère de Julia sur deux points : l'intégration des fonctions de contrôle dans les composants est réalisée à l'aide d'aspects et les contrôleurs sont implémentés eux-mêmes sous forme de composants. Par rapport à Julia qui utilise un mécanisme de *mixin* [Bracha and Cook 1990] et de la génération de *bytecode* à la volée avec ASM, l'objectif d'AOKell est de simplifier et de réduire le temps de développement de nouveaux contrôleurs et de nouvelles membranes.

La suite de cette section présente le principe de mise sous forme de composants des membranes et la façon dont AOKell utilise les aspects.

Membranes componentisées

La membrane d'un composant Fractal est composée d'un ensemble de contrôleurs. Chaque contrôleur est dédié à une tâche précise : gestion des liaisons, du cycle de vie, etc. Loin d'être complètement autonomes, ces contrôleurs collaborent entre eux afin de remplir la fonction qui leur est assignée. Par exemple, lors du démarrage d'un composite, son contenu doit être visité afin de démarrer récursivement tous les sous-composants⁴. De ce fait, le contrôleur de cycle de vie dépend du contrôleur de contenu. Plusieurs autres dépendances de ce type peuvent être exhibées entre contrôleurs.

³fractal.objectweb.org

⁴Notons que cela ne constitue pas une obligation formelle. Il est tout à fait possible de concevoir une fonction de contrôle pour laquelle le démarrage n'implique pas cette récursion.

Jusqu'à présent ces dépendances étaient implémentées sous la forme de références stockées dans le code des contrôleurs. L'idée d'AOKell est d'appliquer à la conception de la membrane de contrôle le principe qui a été appliqué aux applications : extraire du code les schémas de dépendances et programmer celui-ci sous forme de composants. En "spécifiant contractuellement les interfaces" [Szyperski 2002] de ces composants de contrôle, AOKell espère favoriser leur réutilisation, clarifier l'architecture de la membrane et faciliter le développement de nouveaux composants de contrôle et de nouvelles membranes. Cette approche devrait également permettre d'apporter plus de flexibilité aux applications Fractal en permettant d'adapter plus facilement leur contrôle à des contextes d'exécutions variés ayant des caractéristiques différentes en terme de gestion des ressources (mémoire, activité, etc.).

Ainsi, AOKell est un canevas logiciel dans lequel les concepts de composant, d'interface et de liaison sont utilisés pour concevoir le niveau applicatif et le niveau de contrôle. Une membrane AOKell est un assemblage exportant des interfaces de contrôle et contenant un nombre quelconque de sous-composants. Chacun d'eux implémente une fonctionnalité de contrôle particulière. Comme expliqué précédemment, chaque composant de contrôle est aussi associé à un aspect qui intègre cette logique de contrôle dans les composants de niveau applicatif. La figure 3.9 présente le schéma de principe de cette solution. Par soucis de clarté, la membrane de contrôle du troisième composant applicatif a été omise.

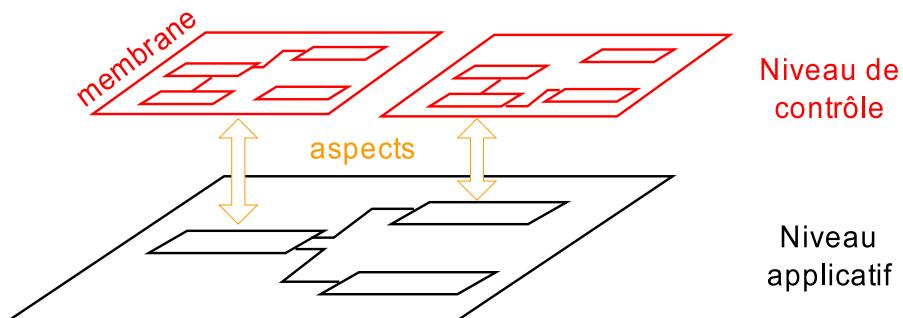


Figure 3.9 – Les niveaux de composant du canevas logiciel AOKell.

La membrane la plus courante dans les applications Fractal est celle associée aux composants primitifs. L'architecture de cette membrane est illustrée figure 3.10. Cette membrane fournit cinq contrôleurs pour gérer le cycle de vie (LC), les liaisons (BC), le nommage (NC), les références vers les composants parents (SC) et les caractéristiques communes à tout composant Fractal (Comp).

L'architecture présentée figure 3.10 illustre le fait que la fonction de contrôle des composants primitifs n'est pas réalisée simplement par cinq contrôleurs isolés, mais est le résultat de leur coopération. Comparée à une approche purement objet, l'implémentation des membranes sous la forme d'un assemblage permet de décrire explicitement les dépendances entre contrôleurs. Elle permet également d'aboutir à des architectures logicielles de contrôle plus explicites, plus évolutives et plus maintenables. De nouvelles membranes peuvent être développées en étendant les existantes, ou en en développement des nouvelles.

Le code Fractal ADL suivant fournit la description de la membrane de la figure 3.10.

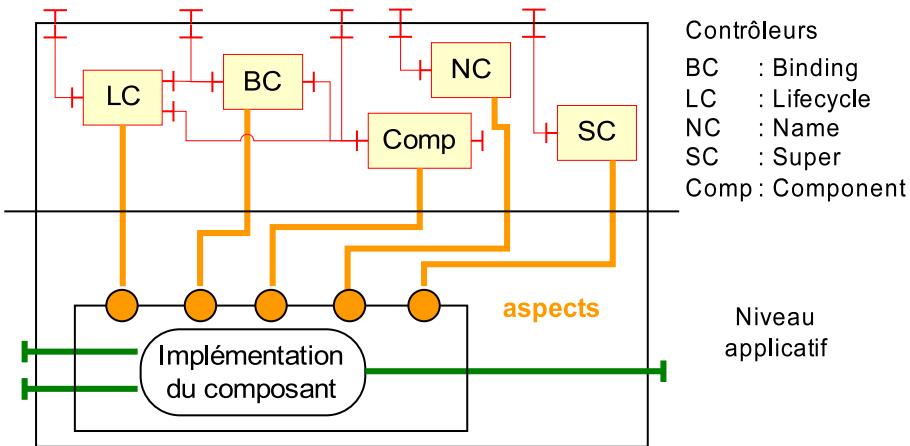


Figure 3.10 – Membrane de contrôle pour les composants primitifs.

La DTD permettant d'écrire cet assemblage et l'API Java permettant de le manipuler sont exactement les mêmes que ceux utilisés pour les composants Fractal applicatifs.

```
<definition name="org.objectweb.fractal.aokell.lib.membrane.primitive.Primitive">

    <!-- Composants de contrôle inclus dans une membrane primitive -->
    <component name="Comp" definition="ComponentController"/>
    <component name="NC" definition="NameController"/>
    <component name="LC" definition="NonCompositeLifeCycleController"/>
    <component name="BC" definition="PrimitiveBindingController"/>
    <component name="SC" definition="SuperController"/>

    <!-- Export des interfaces de contrôle -->
    <binding client="this..//component" server="Comp..//component"/>
    <binding client="this..//name-controller" server="NC..//name-controller"/>
    <binding client="this..//lifecycle-controller" server="LC..//lifecycle-controller"/>
    <binding client="this..//binding-controller" server="BC..//binding-controller"/>
    <binding client="this..//super-controller" server="SC..//super-controller"/>

    <!-- Liaisons entre composants de contrôle -->
    <binding client="BC..//component" server="Comp..//component"/>
    <binding client="LC..//binding-controller" server="//BC.binding-controller"/>
    <binding client="LC..//component" server="Comp..//component"/>

    <controller desc="mComposite"/>
</definition>
```

L'apport de l'ingénierie du contrôle sous forme de composants a été expérimenté en ré-implémentant le canevas logiciel Dream [Leclercq et al. 2005b] avec AOKell. Ce canevas logiciel est présenté à la section 3.4.1.

Intégration des contrôleurs à l'aide d'aspects

Les modèles de composants comme EJB ou CCM fournissent des environnements dans lesquels les composants sont hébergés par des conteneurs fournissant des services techniques. Par exemple, les spécifications EJB définissent des services de sécurité, persistance, transaction et de gestion de cycle de vie. La plupart du temps, cet ensemble de services est fermé et codé en dur dans les conteneurs. Une exception notable est le serveur J2EE JBoss [Fleury and Reverbel 2003] dans lequel les services peuvent être accédés via des aspects définis à l'aide du canevas logiciel JBoss AOP [Burke 2003]. De nouveaux services peuvent être définis qui seront alors accédés à l'aide de leurs aspects associés.

L'idée générale illustrée par le serveur JBoss est que les aspects, tout en fournissant un mécanisme pour modulariser les fonctionnalités transverses, permettent également d'intégrer de façon harmonieuse de nouveaux services dans les applications. Cela illustre également une des bonnes pratiques de la programmation orientée aspect : il est conseillé de ne pas implémenter directement les fonctionnalités transverses dans les aspects, mais d'y implémenter simplement la logique d'intégration (i.e. comment la fonctionnalité interagit avec le reste de l'application) et de déléguer la réalisation concrète de la fonctionnalité à un ou plusieurs objets externes. On obtient ainsi une séparation des préoccupations quasi optimale entre la logique d'intégration et celle du service à intégrer.

Cette pratique est mise en œuvre dans AOKell : chaque contrôleur est associé à un aspect chargé de l'intégration de la logique du contrôleur dans le composant. La logique d'intégration repose sur deux mécanismes : l'injection de code et la modification de comportement. Le premier mécanisme est connu dans AspectJ, sous la dénomination de déclaration inter-type (en anglais ITD pour *Inter-Type Declaration*). Avec ce mécanisme, les aspects peuvent déclarer des éléments de code (méthodes ou attributs) qui étendent la définition de classes existantes (d'où le terme ITD car les aspects sont des types qui déclarent des éléments pour le compte d'autres types, i.e. des classes). Dans le cas d'AOKell, les méthodes des interfaces de contrôle sont injectées dans les classes implémentant les composants⁵. Le code injecté est constitué d'une souche qui délègue le traitement à l'objet implémentant le contrôleur.

Le second mécanisme, la modification de comportement, correspond en AspectJ à la notion de code dit *advice*. Ainsi, la définition d'un aspect est constituée de coupes et de code advice. Les coupes sélectionnent un ensemble de points de jonction qui sont des points dans le flot d'exécution du programme autour desquels l'aspect doit être appliqué. Les blocs de code advice sont alors exécutés autour de ces points. Le code advice est utilisé dans AOKell pour intercepter les appels et les exécutions des opérations des composants. Par exemple, le contrôleur de cycle de vie peut rejeter des invocations tant qu'un composant n'a pas été démarré.

Avec les mécanismes d'injection de code et de modification de comportement, les aspects intègrent de nouvelles fonctionnalités dans les composants et contrôlent leur exécution. La réalisation concrète de la logique de contrôle est déléguée par l'aspect au contrôleur implémenté sous la forme d'un objet.

⁵Ce comportement par défaut peut être modifié afin, par exemple, de conserver des classes libres de toute injection.

3.2.3 Autres plates-formes

Au delà de Julia et de AOKell, plusieurs autres plates-formes de développement Fractal existent. On peut citer en particulier, THINK pour le langage C, ProActive pour le langage Java, FracTalk pour Smalltalk, Plasma pour C++ et FractNet pour les langages de la plate-forme .NET.

THINK [Fassino et al. 2002, Fassino 2001] est une implémentation de Fractal pour le langage C. Elle vise plus particulièrement le développement de noyaux de systèmes qu'ils soient conventionnels ou embarqués. THINK est associé à KORTEX qui est une librairie de composants système pour la gestion de la mémoire, des activités (processus et processus légers) et de leur ordonnancement, des systèmes de fichiers ou des contrôleurs matériels (IDE, Ethernet, carte graphique, etc.). Les architectures matérielles les plus courantes comme celles à base de processeurs PowerPC, ARM ou x86 sont supportées. Le développement d'applications avec THINK passe par l'utilisation d'un langage pour la définition des interfaces (IDL) des composants et un langage pour la description des assemblages de composants (ADL). Dans le cadre des travaux en cours sur THINK v3, l'ADL est en cours de convergence vers Fractal ADL (voir section 3.3).

ProActive [Baude et al. 2003] est une implémentation de Fractal pour le langage Java. Elle vise plus particulièrement le développement de composants pour les applications s'exécutant sur les grilles de calcul. ProActive est basée sur la notion d'objet/composant actif. Chaque composant est muni d'un ensemble de *threads* qui lui sont propres et qui gèrent l'activité de ce composant. Une deuxième caractéristique originale des composants Fractal/ProActive réside dans leur sémantique de communication : un mécanisme d'invocation de méthode asynchrone avec futur est utilisé. Cela permet d'obtenir de façon transparente des interactions non bloquantes dans lesquelles les composants client poursuivent leurs traitements pendant l'exécution des services invoqués.

Finalement, trois autres implémentations du modèle Fractal existent : FracTalk [Bouraqadi 2005] pour Smalltalk, Plasma [Layaïda et al. 2004] pour C++ et FractNet [Escoffier and Donsez 2005] pour les langages de la plate-forme .NET. Cette dernière est basée sur AOKell.

3.3 Fractal ADL

Fractal fournit un langage de description d'architecture (ADL) dont la caractéristique principale est d'être extensible. La motivation pour une telle extensibilité est double. D'une part, le modèle de composants étant lui-même extensible, il est possible d'associer un nombre arbitraire de contrôleurs aux composants. Supposons, par exemple, qu'un contrôleur de journalisation (*LoggerController*) soit ajouté à un composant. Il est nécessaire que l'ADL puisse être étendu facilement pour prendre en compte ce nouveau contrôleur, c'est-à-dire pour que le déployeur d'application puisse spécifier, via l'ADL, le nom du système de journalisation ainsi que son niveau (e.g. *debug*, *warning*, *error*). La seconde motivation réside dans le fait qu'il existe de multiples usages qui peuvent être faits d'une définition ADL : déploiement, vérification, analyse, etc.

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

éléments dans les deux sections suivantes. La troisième section décrit le procédé d’extension de l’ADL.

3.3.1 Le langage extensible

Le langage ADL de Fractal est basé sur le langage XML. Contrairement aux autres ADL qui fixent l’ensemble des propriétés (implantation, liaisons, attributs, localisation, etc.) qui doivent être décrives pour chaque composant, l’ADL Fractal n’impose rien. Il est constitué d’un ensemble (extensible) de modules permettant la description de divers aspects de l’application. Chaque module — à l’exception du module de base — s’applique à un ou plusieurs autres modules, c’est-à-dire rajoute un ensemble d’éléments et d’attributs XML à ces modules. Le module de base définit l’élément XML qui doit être utilisé pour démarrer la description de tout composant. Cet élément, appelé **definition**, a un attribut obligatoire, appelé **name**, qui spécifie le nom du composant décrit.

Différents types de modules peuvent être définis. Un exemple typique de module est le module **containment** qui s’applique au module de base en permettant d’exprimer des relations de contenance entre composants. Ce module définit un élément XML **component** qui peut être ajouté en sous-élément d’un élément **definition** ou de lui-même pour spécifier les sous-composants d’un composant. Notons que l’élément **component** a un attribut obligatoire **name** qui permet de spécifier le nom du sous-composant. Fractal ADL définit actuellement trois autres modules qui s’appliquent soit au module de base, soit au module **containment** pour spécifier l’architecture de l’application : le module **interface** permet de décrire les interfaces d’un composant ; le module **implementation** permet de décrire l’implantation des composants primitifs ; le module **controller** permet la description de la partie contrôle des composants.

Les modules ne servent pas uniquement à décrire les aspects architecturaux de l’application. Par exemple, Fractal ADL fournit des modules permettant d’exprimer des relations de référencement et d’héritage entre descriptions ADL. Le rôle principal de ces modules est de faciliter l’écriture de définition ADL⁶. Le module de référencement s’applique au module **containment** en ajoutant un attribut **definition** à l’élément **component** pour référencer une définition de composant. Le module d’héritage s’applique au module de base. Il permet à une définition d’en étendre une autre (via un attribut **extends**). Notons que l’héritage proposé par Fractal ADL est multiple.

3.3.2 L’usine extensible

L’usine permet de traiter les définitions écrites à l’aide du langage extensible Fractal ADL. Elle est constituée d’un ensemble de composants Fractal qui peuvent être assemblés pour traiter les différents modules de Fractal ADL décrits précédemment. Ces composants sont représentés sur la figure 3.11.

Le **composant loader** analyse les définitions ADL et construit un arbre abstrait correspondant (AST pour *Abstract Syntax Tree*). L’AST implante deux API distinctes : une API *générique* similaire à celle de DOM [W3C-DOM 2005] qui permet de naviguer dans l’arbre ; une API typée qui varie suivant les modules qui sont utilisés dans Fractal ADL. Par

⁶Notons cependant que le module de référencement est nécessaire pour la description des composants partagés.

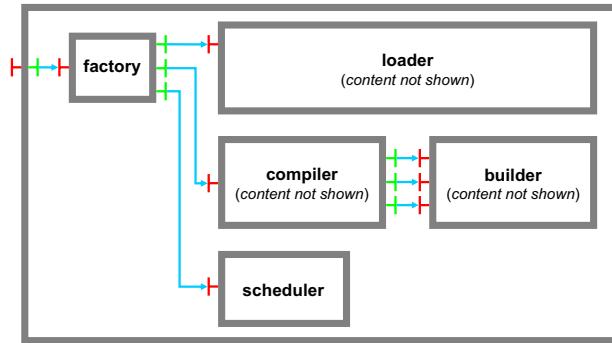


Figure 3.11 – Architecture de l’usine Fractal ADL.

exemple, si le module *interface* est utilisé, l’API typée contient des méthodes permettant de récupérer les informations sur les interfaces de composants (nom, signature, rôle, etc.). Le composant *loader* est un composite encapsulant une chaîne de composants primitifs (figure 3.12). Le composant le plus à droite dans la chaîne (*basic loader*) est responsable de la création des AST à partir de définitions ADL. Les autres composants effectuent des vérifications et des transformations sur les AST. Chaque composant correspond à un module de l’ADL. Par exemple, le composant *binding loader* vérifie les liaisons déclarées dans l’AST ; le composant *attribute loader* vérifie les attributs, etc.

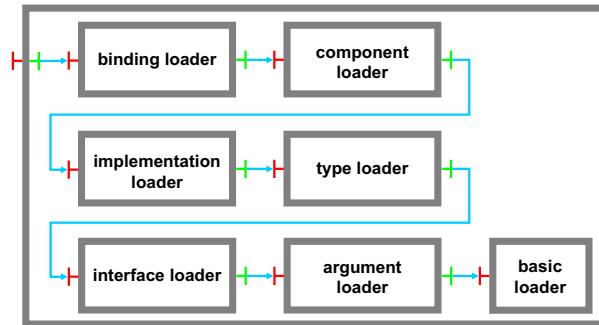


Figure 3.12 – Architecture du composant *loader*.

Le composant compiler utilise l’AST pour définir un ensemble de tâches à exécuter (e.g. création de composants, établissement de liaisons). Le composant *compiler* est un composite encapsulant un ensemble de composants *compiler* primitifs (figure 3.13). Chaque *compiler* primitif produit des tâches correspondant à un ou plusieurs modules ADL. Par exemple, le composant *binding compiler* produit des tâches de création de liaisons. Les tâches produites par un *compiler* primitif peuvent dépendre des tâches produites par un autre *compiler* primitif ce qui impose un ordre dans l’exécution des *compiler*. Par exemple, le *compiler* primitif qui produit les tâches de création des composants doit être exécuté avant les *compilers* en charge des liaisons et des attributs.

Le composant builder définit un comportement concret pour les tâches créées par le *compiler*. Par exemple, un comportement concret d’une tâche de création de composant peut être d’instancier le composant à partir d’une classe Java. Le composant *builder* est

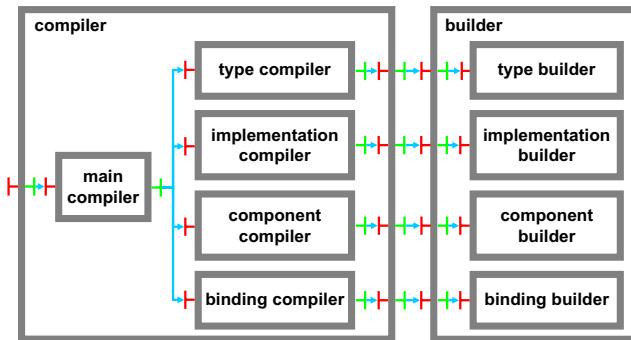


Figure 3.13 – Architecture des composants *compiler* et *builder*.

un composite qui encapsule plusieurs *builders* primitifs (figure 3.13). Chaque canevas logiciel peut définir ses composants *builder*. Par exemple, Julia fournit actuellement quatre composants *builder* qui permettent respectivement de créer des composants avec l’API Java, l’API Fractal ou de produire du code source permettant d’instancier les composants à partir de ces deux API.

3.3.3 Extension de Fractal ADL

Cette section décrit brièvement la démarche à suivre pour étendre Fractal ADL. Supposons que le développeur de l’application ait défini une interface de contrôle **LoggerController** permettant de configurer le nom et le niveau de journalisation de *loggers* associés aux composants (voir figure 3.14). Nous montrons tout d’abord comment le langage ADL peut être étendu pour autoriser la description de ces deux paramètres dans la description ADL de l’application. Nous décrivons ensuite les composants qui doivent être rajoutés à l’usine pour que les *loggers* soient configurés lors du déploiement de l’application.

```

public interface LoggerController {
    String getLoggerName ();
    void setLoggerName (String logger);
    int getLogLevel ();
    void setLogLevel (int level);
}

```

Figure 3.14 – L’interface **LoggerController**.

L’extension du langage consiste à créer un module *logger* qui s’applique aux modules de base **containment** en permettant de rajouter un élément **logger** dont la syntaxe est la suivante :

```
<logger name="logger" level="DEBUG"/>
```

Le développeur du module doit effectuer le travail suivant : (1) définition des interfaces qui seront implantées par l’arbre abstrait lorsque le fichier XML aura été analysé par le *basic loader*; (2) écriture de “fragments” de DTD spécifiant que le module **logger** permet

d'ajouter un élément XML `logger` aux éléments `definition` et `component`.

L'extension de l'usine requiert l'ajout de deux composants : un *compiler* et un *builder*⁷. Le composant *compiler* doit créer des tâches qui configureront le nom et le niveau de journalisation des *loggers* déclarés dans la description ADL. L'implantation du *compiler* est simple : il crée une tâche pour chaque *logger* à configurer et ajoute une dépendance de cette tâche vers la tâche de création du composant auquel le *logger* appartient. Concernant le composant *builder*, il est uniquement nécessaire de récupérer le logger associé au composant et de l'initialiser à l'aide des informations contenues dans l'arbre abstrait.

3.4 Bibliothèques de composants

Cette section présente quelques bibliothèques majeures existant à ce jour pour le développement d'applications à base de composants Fractal.

3.4.1 Dream

Dans cette section, nous présentons Dream [Quéma 2005], une bibliothèque de composants dédiées à la construction d'intergiciels orientés messages dynamiquement configurables plus ou moins complexes : de simples files de messages distribuées à des systèmes publication/abonnement complexes. Nous décrivons les éléments principaux du canevas Dream et illustrons son utilisation par la ré-ingénierie de Joram, une implantation *open source* de la spécification JMS. Nous montrons que la version réalisée à l'aide de Dream a des performances comparables et offre un gain de configurabilité significatif.

Motivations L'utilisation d'intergiciels orientés messages (MOM pour *Message-Oriented Middleware*) est reconnue comme un moyen efficace pour construire des applications distribuées constituées d'entités faiblement couplées [Banavar et al. 1999]. Plusieurs MOM ont été développés ces dix dernières années [Blakeley et al. 1995, msmq 2002, joram 2002, Strom et al. 1998, van Renesse et al. 2003]. Les travaux de recherche se sont concentrés sur le support de propriétés non fonctionnelles variées : ordonnancement des messages, fiabilité, sécurité, etc. En revanche, la configurabilité des MOM a fait l'objet de moins de travaux. D'un point de vue fonctionnel, les MOM existants implantent un modèle de communication figé : publication/abonnement, événement/réaction, files de messages, etc. D'un point de vue non fonctionnel, les MOM existants fournissent souvent les mêmes propriétés non fonctionnelles pour tous les échanges de messages. Cela réduit leurs performances et rend difficile leur utilisation pour des équipements aux ressources restreintes.

Pour faire face à ces limitations, la bibliothèque Dream propose de construire des architectures modulaires à base de composants pouvant être assemblés statiquement ou dynamiquement. Dream se démarque des travaux existants par le fait qu'il ne cible pas uniquement les intergiciels synchrones et qu'il intègre des primitives de gestion de ressources permettant d'améliorer les performances des intergiciels construits.

⁷Le module `logger` ne nécessitant pas de vérifications sémantiques, il n'est pas utile de modifier le composant *loader*. Notons, néanmoins, qu'il serait possible de développer un loader vérifiant que les noms et niveaux de journalisation déclarés dans la description ADL ne sont pas nuls.

Architecture d'un composant Dream Les composants Dream sont des composants Fractal ayant deux caractéristiques : la présence d'interfaces d'entrée/sortie de messages et la possibilité de manipuler les ressources rencontrées dans les intergiciels de communication : messages et activités.

- *Les interfaces d'entrée et sortie de messages.* Le canevas Dream définit deux interfaces permettant aux composants de s'échanger des messages. Une interface d'entrée (input) permet à un composant de recevoir un message. Une interface de sortie (output) permet à un composant d'émettre un message. Les messages sont toujours transmis des sorties vers les entrées (figure 3.15 (a)). On distingue néanmoins deux modes de transfert : *push* et *pull*. Dans le mode *push*, l'échange de message est initié par la sortie qui est une interface client Push liée au composant auquel le message est destiné (figure 3.15 (b)). Dans le mode *pull*, l'échange de message est initié par l'entrée qui est une interface client Pull liée au composant émetteur du message (figure 3.15 (c)).

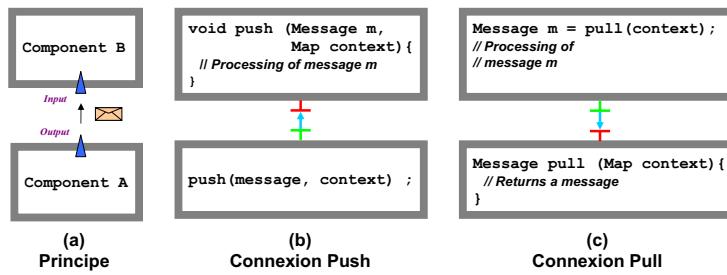


Figure 3.15 – Les interfaces d'entrée et de sortie de messages.

- *Les gestionnaires de messages.* Les messages sont gérés par des composants partagés, appelés gestionnaires de messages (*message managers*). Ces composants permettent de créer, détruire et dupliquer des messages⁸. Leur but est de gérer les ressources mémoires consommées par les MOM. Pour ce faire, ils utilisent des réserves (*pools*) de messages permettant de réduire le nombre d'allocations d'objets.
- *Les gestionnaires d'activités.* Dream distingue deux sortes de composants : les *composants actifs* et les *composants passifs*. Les composants actifs définissent des tâches à exécuter. Ces tâches permettent au composant de posséder son propre flot d'exécution. Au contraire, les composants passifs ne peuvent effectuer d'appels sur leurs interfaces clients que dans une tâche d'un composant appelant une de leurs interfaces serveurs. Les tâches d'un composant actif sont accessibles par l'intermédiaire d'un contrôleur spécifique, appelé contrôleur de tâches (*task controller*). Pour qu'une tâche soit exécutée, il faut qu'elle soit enregistrée auprès d'un gestionnaire d'activités (*activity manager*). Les gestionnaires d'activités sont des composants partagés qui encapsulent des tâches et des ordonnanceurs (*schedulers*). Les ordonnanceurs sont en charge d'associer des tâches de haut niveau à des tâches de bas niveau. Les tâches de plus haut niveau sont les tâches applicatives (i.e. enregistrées par le MOM). Les tâches de plus bas niveau encapsulent des *threads Java*. Ces concepts sont représentés

⁸Les messages sont des objets Java encapsulant des *chunks* et des sous-messages. Chaque *chunk* est également un objet Java implantant des accesseurs (*getter*) et des mutateurs (*setter*).

sur la figure 3.16. Les composants A et B ont enregistré trois tâches qui sont ordonnancées par un ordonnanceur FIFO. Celui-ci utilise pour cela deux tâches de bas niveau.

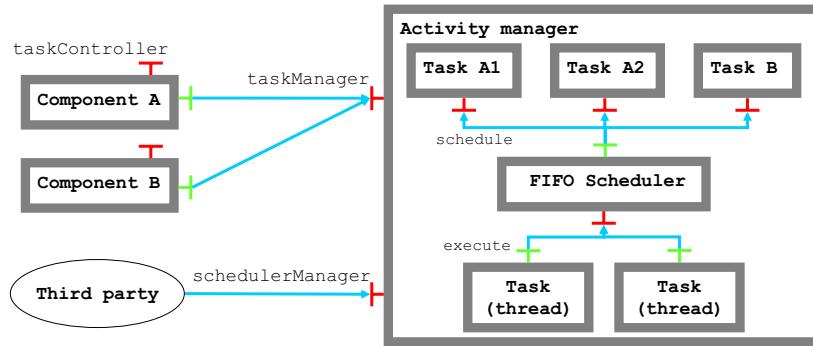


Figure 3.16 – Exemple de gestionnaire de tâches.

La bibliothèque de composants La bibliothèque de composants Dream contient des composants implantant les fonctions que l'on trouve de façon commune dans les différents MOM. Elle contient également des composants spécifiques développés pour des personnalités particulières de MOM. Par manque de place, nous nous contentons de décrire les composants formant le cœur de la bibliothèque.

- *Les files de messages* servent à stocker les messages. Elles ont une entrée — qui est utilisée par les autres composants pour stocker des messages —, et une sortie — que la file utilise pour délivrer les messages. Les files diffèrent par la manière dont les messages sont triés (FIFO, LIFO, ordre causal, etc.), leur comportement dans les différents états : file pleine (bloque vs. détruit des messages), file vide, etc.
- *Les transformateurs* sont des composants avec une entrée et une sortie. Chaque message reçu sur l'entrée est transformé, puis délivré sur la sortie. Un exemple typique de transformation consiste à rajouter une adresse IP à un message.
- *Les pompes* sont des composants avec une entrée *pull* et une sortie *push*. Les pompes ont une activité qui consiste à récupérer un message sur l'entrée et le délivrer sur la sortie.
- *Les routeurs* ont une entrée et plusieurs sorties. Le rôle d'un routeur est de router les messages reçus en entrée sur une ou plusieurs sorties. Le routage peut se faire sur le contenu du message, sa destination, etc.
- *Les agrégateurs/désagrégateurs*. Les agrégateurs sont des composants avec une ou plusieurs entrées et une sortie. Leur rôle est de délivrer sur la sortie un agrégat des messages reçus en entrée. Les désagrégateurs implémentent le comportement inverse des agrégateurs.
- *Les canaux* permettent l'échange de messages entre différents espaces d'adressages. Un canal est un composant composite distribué qui encapsule, au minimum, deux composants : un canal sortant (*channel out*) — qui permet d'envoyer des messages vers un autre espace d'adresage —, et un canal entrant (*channel in*) — qui permet de recevoir des messages en provenance d'autres espaces d'adresage.

Ré-ingénierie de JORAM Cette section présente une expérimentations qui a été réalisée à l'aide de Dream : la ré-ingénierie de la plate-forme ScalAgent qui supporte l'exécution de JORAM [joram 2002]. Nous commençons par décrire la plate-forme ScalAgent existante, puis nous présentons son implémentation à l'aide de Dream.

- La plate-forme ScalAgent [Quéma et al. 2004] permet le déploiement et l'exécution d'agents. Les agents sont des objets réactifs qui se comportent conformément au modèle “événement → réaction” [Agha 1986]. Les agents sont persistants et ont des réactions atomiques. La création, l'exécution et les communications des agents sont prises en charge par un MOM. Celui-ci est constitué d'un ensemble de serveurs d'agents organisés en bus. Comme on peut le voir sur la figure 3.17, chaque serveur d'agents est constitué de trois éléments architecturaux : l'*engine* est en charge de la création et de l'exécution des agents. Il effectue, en boucle, un ensemble d'instructions consistant à prendre un message dans le *conduit* et faire réagir l'agent destinataire. Il garantit la persistance des agents et l'atomicité de leurs réactions. Le *conduit* route les messages de l'*engine* vers les *networks*. Les *networks* assurent la transmission fiable et causalement ordonnée des messages.

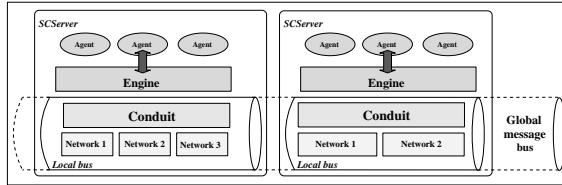


Figure 3.17 – Deux serveurs d'agents interconnectés

– Ré-ingénierie de l'intergiciel ScalAgent avec Dream

L'intergiciel ScalAgent a été ré-ingénieré à l'aide de Dream (figure 3.18). Ses principales structures (*networks*, *engine* et *conduit*) ont été préservées de manière à faciliter la comparaison. L'*engine* est un composite qui comprend deux parties. La première partie traite des messages Dream. Elle est constituée d'une file de messages — qui stocke les messages entrants — et d'un composite (*AtomicityProtocol*) qui garantit l'atomicité de la réaction des agents. La seconde partie correspond au composite *Repository*. Celui-ci est en charge de la création et de l'exécution des agents. La figure représente également deux *networks* typiques. Les deux sont des composites qui encapsulent un canal entrant (*TCPChannelIn*), un canal sortant (*TCPChannelOut*), et un transformateur (*DestinationResolver*) en charge d'ajouter l'adresse IP et le numéro de port du destinataire du message. Le *network 1* encapsule deux composants supplémentaires : le *CausalSorter* garantit l'ordonnancement causal des messages échangés. La file de messages permet de découpler les flots d'exécution de l'*engine* et du *network*. Enfin, le *conduit* est implémenté par un routeur dont l'algorithme de routage est basé sur l'identifiant de l'*engine* auquel le message est destiné.

- Gain en configurabilité. L'implémentation Dream de la plate-forme ScalAgent apporte de nombreux bénéfices en termes de configurabilité : il est aisément de changer les propriétés non fonctionnelles fournies par le MOM (e.g. atomicité, ordre causal). Par ailleurs, il est possible de modifier le nombre de composants actifs s'exécutant dans un serveur d'agents. L'architecture que nous avons présentée sur la figure 3.18 fait

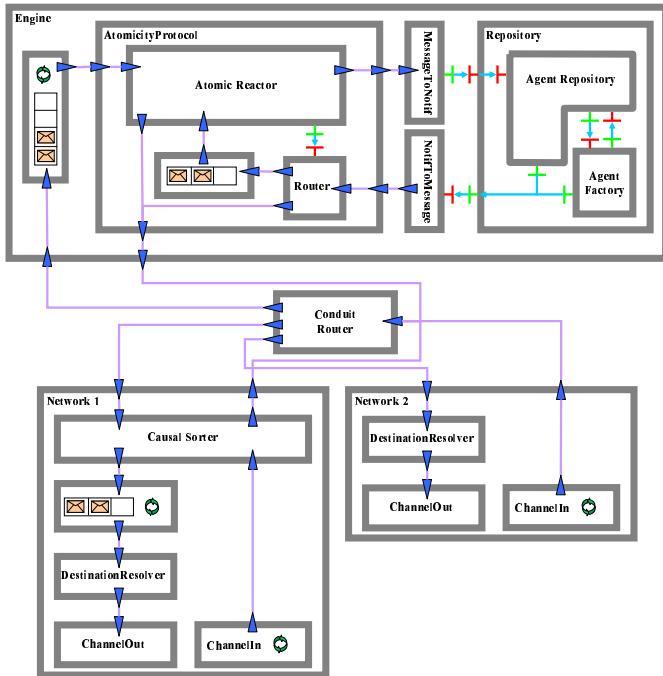


Figure 3.18 – Architecture Dream d'un serveur d'agents

intervenir trois composants actifs pour un serveur d'agents avec un seul *network*. Il est possible d'obtenir une implémentation mono-threadée en supprimant les files de messages de l'*engine* et du *network*. Enfin, il est possible d'adapter la plate-forme à des environnements contraints. Ainsi, [Leclercq et al. 2005a] présente une modification du serveur d'agents permettant son déploiement sur un équipement mobile.

- *Comparaison des performances.* Des mesures de performances ont montré que les performances obtenues par la ré-ingénierie à l'aide de Dream étaient tout à fait comparables à celles obtenues par la plate-forme ScalAgent. Par ailleurs, ces mesures ont montré qu'il était possible d'avoir des gains de performances significatifs en configurant le MOM de façon adéquat. Par exemple, la réduction du nombre de composants actifs peut permettre des améliorations de performances de l'ordre de 15% pour certaines applications.

3.4.2 Autres bibliothèques

Plusieurs autres bibliothèques de composants Fractal ont été développées. Il s'agit dans la plupart des cas de fournir des briques de base pour la construction d'intergiciels.

Parmi les bibliothèques existantes nous pouvons citer :

- Le projet Perseus [Chassande-Barrioz and Dechamboux 2003] définit une bibliothèque de composants Fractal pour la construction de services de persistance de données. Il fournit des composants de base pour la gestion de cache, de réserves (*pools*), de politiques de concurrence (mutex, lecteurs/écrivain FIFO, lecteurs/écrivain avec priorité aux écrivains, optimiste), de journalisation et de gestion

de dépendances. Les composants Perseus sont utilisés notamment pour construire Speedo [Chassande-Barrioz 2003] qui est une implémentation des spécifications JDO [JDO 2002] de Sun.

- CLIF [Dillenseger 2004] est un canevas logiciel pour l'injection de charge. Il constitue un des éléments du projet ObjectWeb JMOB pour la mesure des performances des intergiciels. CLIF permet de déployer, contrôler et superviser des injecteurs de charge. Il permet également de gérer des sondes pour mesurer l'état de ressources systèmes telles que le CPU, la mémoire ou tout autre type de ressources logicielle ou matérielle. Chaque sonde est représentée par un composant Fractal. L'implémentation de ce composant repose sur des mécanismes systèmes externes à Fractal comme par exemple l'utilisation de /proc sous Unix. Des composants sont également disponibles pour décrire les scénarios de test de charge, la collecte, le stockage et l'analyse du résultat des tests.
- Le projet GoTM [Rouvoy 2004] est une bibliothèque de composants Fractal pour la construction de moniteurs transactionnels. Il permet de construire des moniteurs transactionnels conformes à différents standards (par exemple JTS, OTS ou WS-AT) et supportant différentes formes de protocoles de validation (2PC, 2PC presume commit, 2PC presume abort, etc.).

3.5 Comparaison

De nombreux modèles de composants ont été proposés ces dix dernières années. Dans le domaine des intergiciels, les propositions peuvent être classées selon qu'ils sont issus d'initiatives industrielles, de la communauté du logiciel libre ou d'équipes de recherche académiques.

Initiatives industrielles La première catégorie comprend les modèles issus d'initiatives industrielles comme EJB (voir chapitre 5), COM+/.NET, CCM ou OSGi (voir chapitre 7). Les caractéristiques de ces modèles varient, allant de composants avec des propriétés d'introspection (COM+), à des composants avec liaisons et cycle de vie (OSGi et CCM). Le modèle Fractal est, quant à lui, entièrement réflexif et introspectable, autorise des liaisons selon différentes sémantiques de communication et fournit un modèle hiérarchique autorisant le partage. Par ailleurs, EJB, CCM et COM+/.NET sont tous les trois accompagnés d'un modèle figé de services techniques offerts par des conteneurs aux composants. Par contraste, Fractal est basé sur un modèle ouvert, dans lequel les services techniques sont entièrement programmables via la notion de contrôleur.

Plus récemment, sous l'impulsion d'IBM, l'initiative SCA [SCA 2005] a défini un modèle de composants pour des architectures orientées services. Le projet Tuscany [Tuscany 2006] fournit une implémentation en Java et en C++ de ces spécifications. SCA propose la notion de liaison pour l'assemblage et de module pour la création de hiérarchies de composants. SCA n'impose pas une forme prédéterminée de liaison, mais autorise l'utilisation de différentes technologies, comme SOAP, JMS ou IIOP pour mettre en œuvre ces liaisons. De même, Fractal autorise différents types de liaisons et n'impose pas de technologie particulière.

Initiatives du monde du logiciel libre Dans la catégorie des modèles de composants issus d'initiatives de type logiciel libre, nous pouvons notamment citer Avalon [Avalon] qui est un modèle de composants général, Kilim [ObjectWeb 2004], Pico [PicoContainer 2004] et Hivemind [Hivemind 2004] qui ciblent la configuration de logiciel, Spring [Spring 2004], Carbon [Carbon 2004] et Plexus [Plexus 2004] qui ciblent les conteneurs de composants de type EJB. De manière générale, ces modèles sont moins ouverts et extensibles que ne l'est Fractal.

Initiatives académiques Plusieurs modèles de composants ont également été proposés par des équipes de recherche académiques. Sans être exhaustif, on peut citer ArchJava [Aldrich et al. 2002], FuseJ [Suvée et al. 2005], K-Component [Dowling and Cahill 2001], OpenCOM v1 [Clarke et al. 2001] et v2 [Coulson et al. 2004].

OpenCOM est certainement le modèle le plus proche de Fractal. Il cible les systèmes devant être reconfigurés dynamiquement et en particulier les systèmes d'exploitation, les intergiciels, les PDA et les systèmes embarqués. Au niveau applicatif, les composants OpenCOM fournissent des interfaces et requièrent des réceptacles. L'architecture d'une application OpenCOM est introspectable et peut être modifiée dynamiquement.

Depuis la version 2, OpenCOM fournit les quatre notions suivantes : capsule, caplet, loader et binder. Une capsule est l'entité qui contient et gère les composants applicatifs. Une caplet est une partie d'une capsule qui contient un sous-système de l'application. Les binders et les loaders sont des entités de première classe qui offrent différentes sémantiques de chargement et de liaison pour les composants. Les caplets, les loaders et les binders sont eux-mêmes des composants.

Comparé à Fractal, les capsules et les caplets sont similaires aux composants composites. Les binders et les loaders sont quant à eux comparables aux contrôleurs. Cependant les contrôleurs Fractal ne sont pas limités à ces deux types de propriétés extra-fonctionnelles et peuvent englober d'autres services techniques. Par ailleurs, si comme dans Fractal/AOKell les binders et les loaders sont aussi des composants, Fractal/AOKell va au-delà et permet de réifier complètement l'architecture de contrôle sous la forme d'un assemblage de composants.

3.6 Conclusion

Ce chapitre a présenté le système de composants Fractal, les principales plates-formes le mettant en œuvre, le langage de description d'architecture et les bibliothèques permettant de développer des systèmes à base de composants Fractal.

La section 3.1 est consacrée à la présentation du modèle de composant Fractal. C'est un modèle hiérarchique au sens où les composants peuvent être soit primitif, soit composite et contenir des sous-composants (primitifs ou composites). Deux parties sont mises en avant dans un composant Fractal : le contenu et la membrane. Cette dernière fournit un niveau métadonnées de contrôle et de supervision du contenu. Elle est composée d'entités élémentaires, les contrôleurs, qui implémentent des interfaces dites de contrôle. Le modèle de composant Fractal est ouvert au sens où il ne présuppose pas un ensemble fini et figé de contrôleurs : de nouveaux contrôleurs peuvent être ajoutés par les développeurs en fonction des besoins.

De même, la granularité des contrôleurs est quelconque, allant d'un simple service de gestion de noms à des services plus complexes de gestion de persistance ou de transaction. Un composant Fractal est une entité logicielle qui possède des interfaces fournies (dite serveur) et/ou des interfaces requises (dites client). Le concept de liaison permet de définir des chemins de communication entre interfaces clientes et serveurs. La granularité des liaisons est également quelconque allant d'une simple référence dans l'espace d'adressage courant à des liaisons plus complexes mettant en œuvre des mécanismes de communication distante.

Le modèle Fractal est indépendant des langages de programmation. La section 3.2 a présenté deux plates-formes, Julia et AOKell, mettant en œuvre ce modèle pour le langage Java. D'autres plates-formes existent pour les langages Smalltalk, C, C++ et les langages de la plate-forme .NET. Julia est la plate-forme de référence du modèle Fractal. Le développement des contrôleurs se fait à l'aide d'un système de classes *mixin*. AOKell a été développé par la suite et apporte une approche à base de composants de contrôle pour le développement des membranes. Celles-ci sont des assemblages de composants de contrôle qui gèrent et administrent les composants du niveau de base. Les composants de contrôle sont eux-mêmes des composants Fractal qui sont contrôlés de façon ad-hoc. Finalement, AOKell utilise des techniques issues de la programmation orientée aspect pour l'intégration des niveaux de base et de contrôle.

L'API Fractal permet de construire d'assembler des composants élémentaires afin de construire des applications complètes. Le langage de description d'architecture Fractal ADL (voir section 3.3) permet de décrire ces architectures de façon plus concise qu'avec une simple API. Fractal ADL est un langage ouvert basé sur XML. Contrairement à nombre d'ADL existants, la DTD de Fractal ADL n'est pas figée et peut être étendue avec de nouvelles balises permettant d'associer des caractéristiques supplémentaires aux composants. Un mécanisme d'extension permet de définir les traitements à exécuter lorsque ces nouvelles balises sont rencontrées. Notons enfin que l'outil Fractal ADL d'analyse et d'interprétation d'un assemblage est lui-même une application Fractal (donc un assemblage de composant Fractal).

Plusieurs bibliothèques de composants sont disponibles pour faciliter la tâche des développeurs Fractal. Nous en avons présentées quelques unes dans la section 3.4, dont Dream, qui permet de développer des intergiciels. Nous aurions pu également mentionner les outils qui existent autour de Fractal, comme Fractal GUI qui permet de concevoir graphiquement une architecture de composants et de générer des squelettes de code, Fractal Explorer [Merle et al. 2004] qui est une console graphique d'administration d'applications Fractal, Fractal RMI qui permet de construire des assemblages de composants distribués communicants via un mécanisme d'invocation de méthodes à distance et Fractal JMX pour l'administration de composants à l'aide de la technologie JMX.

Après une première version stable diffusée en juillet 2002, la version 2 des spécifications Fractal parue en septembre 2003 a permis d'en consolider l'assise. Fractal est maintenant une spécification stable et mature grâce à laquelle de nombreux systèmes et applications ont pu être développés. Par ailleurs, de nombreuses activités de recherche sont conduites autour de Fractal. Sans être exhaustif, nous pouvons citer les travaux sur les approches formelles et les calculs de composants, la vérification de comportements, la sécurité et l'isolation des composants, les systèmes autonomiques, l'unification des styles de développement à

base de composants et d'aspects, la gestion de la qualité de service ou les composants pour les grilles de calcul. Au delà de leur intérêt propre, ces travaux devraient également servir de terreau pour compléter Fractal sur plusieurs points qui sont actuellement peu pris en compte comme le paquetage (*packaging*), le déploiement, la sémantique des interfaces collection ou les modèles de composants pour les architectures orientées service.

Chapitre 4

Les Services Web

L'accès aux systèmes d'information s'appuie aujourd'hui de plus en plus sur des technologies Internet. Les efforts de standardisation dans ce contexte ont accentué l'engouement des personnes et des organisations (aussi bien académiques, qu'industrielles, commerciales, ou institutionnelles) pour l'utilisation de l'Internet et ont permis l'émergence des services Web comme support de développement des applications accessibles par Internet. Ainsi, les technologies associées aux services Web sont devenues incontournables pour le développement d'applications interagissant les unes avec les autres par le biais de l'Internet. Nous proposons dans ce chapitre de faire un point sur le sujet des services Web. L'objectif de la discussion est de traiter des aspects conceptuels de la modélisation des services aussi bien que des aspects liés à leur implantation.

La section 4.1 commence par donner quelques définitions et principes nécessaires à la compréhension de la suite. Puis la section 4.2 discute de la modélisation des interactions supportées par un service Web en abordant les deux points de vue, structurel et comportemental. La section 4.3 donne un éclairage sur les principaux standards pour la description de services Web en les distinguant selon leur capacité à traiter des aspects structurels, comportementaux et non-fonctionnels des services. Dans la section 4.4 nous décrivons quelques outils existants pour la mise en œuvre de services Web. Finalement la section 4.5 donne, en forme de conclusion, quelques perspectives faisant actuellement l'objet de recherches.

4.1 Services Web : historique, définition, et principes

Plusieurs définitions des services Web ont été mises en avant par différents auteurs. Ci-dessous, nous citons une définition généralement acceptée et fournie par le consortium W3C [W3C-WSA-Group 2004] :

A Web service is a software system designed to support interoperable machine-to-machine interaction over a network. It has an interface described in a machine-processable format (specifically WSDL¹). Other systems interact with

¹ *Web Service Description Language*

the Web service in a manner prescribed by its description using SOAP² messages, typically conveyed using HTTP with an XML serialization in conjunction with other Web-related standards.

D'autres définitions similaires adoptent un point de vue plus général en ne prescrivant pas l'usage exclusif de WSDL pour la description des interfaces de service ou celui de SOAP pour le traitement des messages³. Ainsi par exemple, certains auteurs proposent l'utilisation de messages XML (sans les extensions apportées par SOAP) échangés directement sur le protocole HTTP. De façon similaire, la définition officielle du consortium W3C fait spécifiquement référence au protocole HTTP, mais en pratique, d'autres protocoles sont aussi utilisés (voir Section 4.4).

Une étude plus approfondie des points communs partagés par les différentes définitions et par les usages qui sont faits des services Web, permet de dégager au moins deux principes fondamentaux :

- Les services Web interagissent au travers d'échanges de messages encodés en XML. Dans certains cas, les messages sont plutôt transmis dans un encodage binaire, même s'ils sont généralement produits et consommés en XML ou en utilisant des interfaces de programmation basées sur les concepts d'*élément* et d'*attribut* définis par le modèle dit « XML Infoset ».
- Les interactions dans lesquelles les services Web peuvent s'engager sont décrites au sein d'interfaces. La définition W3C restreint la portée des interfaces des services Web aux aspects fonctionnels et structurels, en utilisant le langage WSDL qui, essentiellement, ne permet de décrire que des noms d'opérations et des types de messages. Cependant, d'autres auteurs proposent d'aller plus loin et de considérer aussi les aspects comportementaux et non-fonctionnels comme nous le verrons dans la section 4.3.

Par rapport à d'autres plates-formes pour le développement d'applications distribuées telles que CORBA (voir le chapitre 1) et Java RMI (voir le chapitre 5), l'une des différences primordiales est que les services Web n'imposent pas de modèles de programmation spécifiques. En d'autres termes, les services Web ne sont pas concernés par la façon dont les messages sont produits ou consommés par des programmes. Ceci permet aux vendeurs d'outils de développement d'offrir différentes méthodes et interfaces de programmation au-dessus de n'importe quel langage de programmation, sans être contraints par des standards comme c'est le cas de CORBA qui définit des ponts spécifiques entre le langage de définition IDL⁴ et différents langages de programmation. Ainsi, les fournisseurs d'outils de développement peuvent facilement différencier leurs produits avec ceux de leurs concurrents en offrant différents niveaux de sophistication. Par exemple, il est possible d'offrir des environnements pour des méthodes de programmation de services Web minimalistes au-dessus de plates-formes relativement légères comme c'est le cas de NuSOAP (pour le langage PHP) ou d'Axis (voir la section 4.4). En revanche, des plates-formes plus lourdes telles que BEA WebLogic, IBM Websphere ou .Net (voir le chapitre 6) offrent un environnement pour des méthodes de développement très sophistiquées.

²Simple Object Access Protocol

³Il faut noter que WSDL et SOAP sont des standards définis par le consortium W3C, ce qui explique que la définition ci-dessus essaye d'imposer ces standards.

⁴Interface Definition Language

Les principes à la base des services Web, bien que simples, expliquent leur adoption rapide : le nombre d'outils associés au services Web a connu un essor considérable dans une période de temps réduite, assurant ainsi que les technologies associées aux services Web seront utilisées pour de nombreuses années à venir.

La description explicite des interactions entre les services Web, est aussi un point fort des services Web. Cependant, des efforts de recherche et de développement plus approfondis sont nécessaires afin de réellement exploiter ce principe dans des outils ou des méthodes de développement. En particulier, comme nous le discuterons dans la section 4.3, l'usage des descriptions comportementales et non-fonctionnelles des services requièrent davantage d'attention de la part de la communauté de recherche.

4.2 Modélisation de services Web

4.2.1 Points de vue dans la modélisation de services

Les services Web interagissent les uns avec les autres par le biais d'envois de messages. Dans ce cadre, une interaction entre deux services, un client et un fournisseur, est décrite par un envoi de message par le client et la réception de ce message par le fournisseur. De cette interaction peuvent découler d'autres interactions, au cours desquelles les rôles de client et de fournisseurs peuvent s'inverser. Une séquence de telles interactions est appelée une *conversation*.

De ce fait, la modélisation de services Web met en jeu la description d'interactions et de leurs interdépendances, aussi bien du point de vue structurel (types de messages échangés) que du point de vue comportemental (flot de contrôle entre interactions). De plus, ces interactions peuvent être vues d'une façon globale, c'est-à-dire du point de vue d'un ensemble de services participant à une collaboration, ou de façon locale, c'est-à-dire du point de vue d'un service donné. Selon cette dimension, il est possible de distinguer trois types de modèles de services :

- *Chorégraphie*. Une chorégraphie décrit, d'une part un ensemble d'interactions qui peuvent ou doivent avoir lieu entre un ensemble de services (représentés de façon abstraite par des *rôles*), et d'autre part les dépendances entre ces interactions.
- *Interface*. Une interface décrit un ensemble d'interactions dans lesquelles un service donné peut ou doit s'engager et les dépendances entre ces interactions. Dans une interface, les interactions sont décrites du point de vue du service concerné sous forme d'*actions communicationnelles*, c'est-à-dire d'envois et de réceptions de messages.
- *Orchestration*. Une orchestration décrit un ensemble d'actions communicationnelles et d'actions internes dans lesquelles un service donné peut ou doit s'engager (afin de remplir ses fonctions) ainsi que les dépendances entre ces actions. Une orchestration peut être vue comme un raffinement d'une interface qui inclut des actions qui ne sont pas nécessairement pertinentes pour les clients du service mais qui servent à remplir les fonctions que le service fournit et doivent donc être prises en compte dans son implantation.

A titre d'exemple, la figure 4.1 présente une chorégraphie correspondant à un processus de gestion de commandes dans le domaine commercial. Afin de laisser la description de cette chorégraphie à un niveau abstrait, nous utilisons des diagrammes d'activité UML

dans lesquels les actions correspondent à des interactions entre des rôles correspondant à des types de service (client, fournisseur et entrepôt en l'occurrence). Pour simplifier, pour chaque échange de message le diagramme montre soit l'envoi de ce message, soit la réception, mais pas les deux. Par exemple, dans l'activité « Commande », nous faisons référence à l'envoi du Bon de Commande (BdC) par le client au fournisseur. Bien entendu, le fournisseur doit exécuter l'action correspondant à la réception de ce message, mais nous ne présentons pas cette action duale dans le diagramme d'activités.

Il est à remarquer aussi que les diagrammes d'activité permettent de modéliser des aspects comportementaux, et en particulier des flots de contrôle. Ils ne permettent pas cependant de modéliser des aspects structurels tels que les types de données décrivant le contenu des messages échangés. Ceux-ci peuvent être décrits au travers de diagrammes de classe mais nous omettons ces détails car ils ne contribuent pas à la compréhension du concept général de chorégraphie.

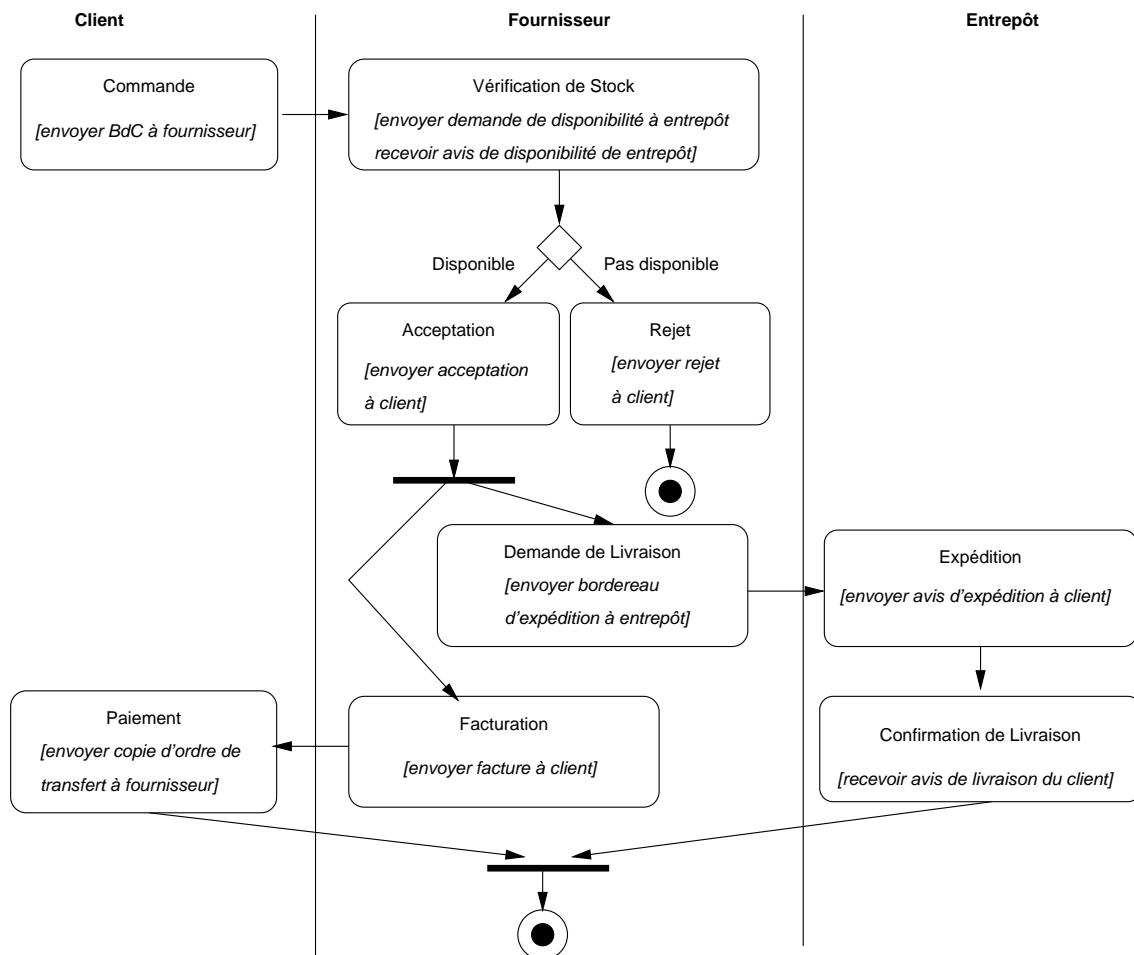


Figure 4.1 – Une chorégraphie

Il faut noter que la chorégraphie présentée dans la figure 4.1 n'adopte le point de vue d'aucun des services (ou plus précisément des rôles) concernés. En effet, cette chorégraphie

inclut des interactions entre le client et le fournisseur, entre le fournisseur et son entrepôt, et entre l'entrepôt et le client. Lorsqu'on passe à des étapes ultérieures dans le cycle de développement de services, on s'intéresse à voir des interactions du point de vue d'un service spécifique, comme par exemple un service censé jouer le rôle de fournisseur dans l'exemple en question. Dans ce cas, on utilise la notion d'interface (voir chapitre 2). Une interface de service décrit un ensemble d'interactions dans lesquelles un service donné peut ou doit s'engager. Le concept d'interface est dérivé des langages de programmation et des intergiciels à base d'appels de procédures tels que CORBA. Mais alors que traditionnellement le concept d'interface est associé surtout à une description structurelle (c'est-à-dire une description des paramètres d'une opération), dans le cadre des services Web on s'intéresse à des descriptions plus riches, incorporant des aspects comportementaux. A titre d'exemple, l'interface correspondant au rôle de fournisseur dans la chorégraphie de la figure 4.1 est représentée sur la figure 4.2. Dans cette dernière figure, chaque élément de la séquence correspond à une réception (par exemple *Recevoir bon de commande*) ou un envoi de message (par exemple *Envoyer demande de disponibilité*). L'ensemble des types de messages échangés modélisent la dimension structurelle de l'interface alors que le flot de contrôle exprimé par la séquence modélise la dimension comportementale de l'interface.

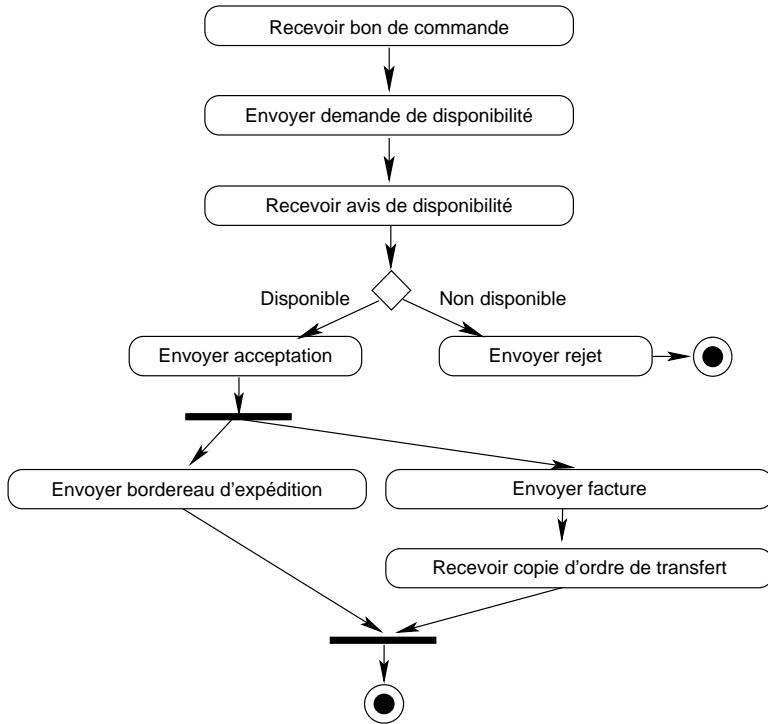


Figure 4.2 – Interface correspondant au rôle de fournisseur dans la chorégraphie de la figure 4.1.

Selon que l'on se réfère à l'interface qu'un service existant est capable de fournir, ou de l'interface qu'un service est censé fournir dans le cadre d'une chorégraphie, on parle d'*interface fournie* ou d'*interface requise* (voir chapitre 2). L'interface présentée dans la figure 4.2 correspond à l'interface requise du fournisseur, dérivée de la chorégraphie donnée dans la figure 4.1. Parfois, l'interface d'un service existant est identique ou compatible avec

l'interface requise pour participer à une chorégraphie. Cependant, ceci n'est pas toujours le cas. A titre d'exemple, considérons le cas où l'interface représentée sur la figure 4.3 est l'interface fournie par un service que l'on voudrait utiliser pour remplir le rôle de fournisseur dans la chorégraphie de la figure 4.1. Dans ce contexte, l'interface fournie diffère de l'interface requise en deux points. Premièrement, alors que dans l'interface requise le service peut répondre par une acceptation ou un rejet, dans l'interface fournie il répond avec le même type de message quel que soit le cas. Deuxièmement, dans l'interface requise, l'ordre d'expédition et la facture sont envoyés dans n'importe quel ordre, alors que dans l'interface requise l'envoi de la facture et la réception de l'ordre de paiement précèdent l'envoi de l'ordre d'expédition. Dans ce contexte, il est nécessaire de réconcilier ces deux interfaces. Ce problème est connu sous le nom d'adaptation de services et fait l'objet de plusieurs efforts de recherche (voir section 4.5.1).

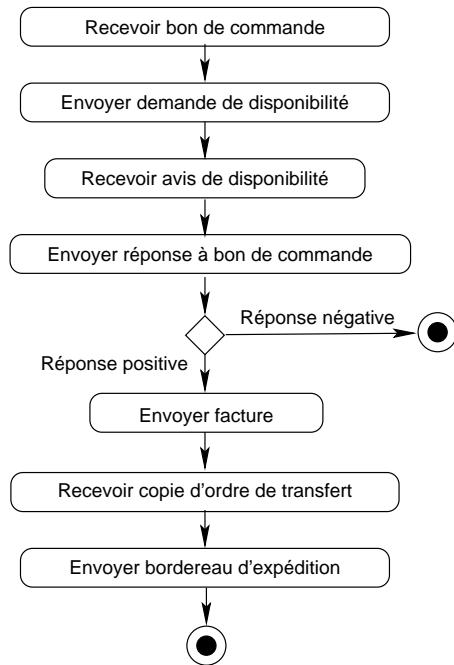


Figure 4.3 – Exemple d'interface fournie.

Enfin, la figure 4.4 présente une orchestration correspondant à l'interface de la figure 4.3. Cette orchestration rajoute une action (indiquée en lignes pointillées) qui n'est pas exposée dans l'interface. En réalité bien sûr, on peut attendre qu'une orchestration introduise davantage d'actions non-exposées, mais nous nous limitons ici à présenter une seule action non-exposée pour éviter de compliquer l'exemple. Plus généralement, une orchestration est un raffinement d'une ou plusieurs interfaces, interfaces qui à leur tour, peuvent être dérivées d'une chorégraphie. Certains auteurs considèrent ainsi qu'une orchestration est une implantation d'un service (voir par exemple [Peltz 2003]). Ceci peut faire penser qu'une orchestration est un programme exécutable, ce qui n'est pas tout à fait exact. Une orchestration peut être conçue comme un modèle décrivant le mode opératoire d'un service. Ce modèle peut manquer de quelques détails pour être exécutable, auquel cas il doit être raffiné davantage pour pouvoir être ensuite implanté dans un langage

de programmation tel que Java ou BPEL (voir Section 4.4).

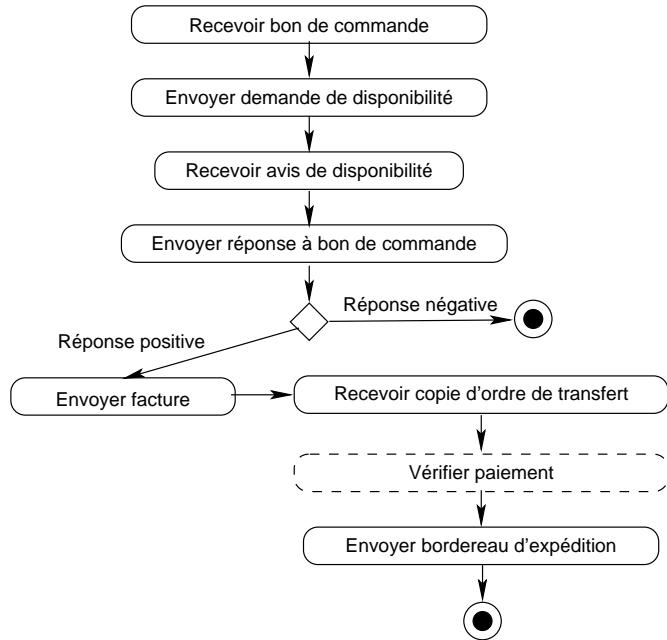


Figure 4.4 – Exemple d'orchestration raffinant l'interface de la figure 4.3.

4.2.2 Relations entre points de vue

Les points de vue discutés ci-dessus ne sont pas indépendants les uns des autres. Ces interdépendances peuvent être exploitées au cours du processus de développement pour effectuer des contrôles de cohérence entre les points de vue et pour générer du code. Par exemple, une interface requise peut constituer le point de départ de la génération d'une « ébauche d'orchestration » qui est ensuite complétée par la description détaillée des opérations internes pour enfin être raffinée jusqu'à obtenir le modèle complet d'orchestration. Ceci a pour avantage de faire coïncider l'interface fournie d'un service avec l'interface telle qu'elle est requise par les services qui l'utilisent. De plus, à condition de masquer ses opérations qui ne doivent pas être exposées, un modèle d'orchestration existant peut être utilisé pour générer son interface fournie. La cohérence de l'interface fournie résultante avec l'interface requise, peut ainsi être contrôlée. De cette manière, il est possible de détecter des situations où l'ordre des envois effectués par un service ne correspond pas à l'ordre attendu par les services destinataires avec lesquels il est censé collaborer. La résolution de ces incompatibilités s'appuie, soit sur la modification du modèle d'orchestration, soit sur la fourniture d'une enveloppe devant effectuer la médiation entre l'interface fournie et celle requise.

En résumé, les utilisations d'un modèle d'orchestration sont :

- La génération de l'interface requise pour chacun des services amenés à collaborer. L'intérêt de cette interface requise, au cours de la phase de développement des services a été souligné plus haut. Par exemple, étant donnée la chorégraphie décrite

figure 4.1, il est possible de dériver les interfaces requises des services fournisseur, client et entrepôt.

- Le contrôle, au moment de la conception, de la cohérence de l'interface existante d'un service avec le modèle de chorégraphie auquel il participe. Ainsi, la capacité du service à assumer le rôle qu'il est sensé jouer dans cette chorégraphie est vérifiée.
- La génération d'une ébauche d'un modèle d'orchestration pour chaque participant. Cette ébauche peut ensuite être détaillée dans la perspective, pour chaque participant, de remplir son rôle.

Pour une définition formelle des concepts de chorégraphie, d'interface et d'orchestration, le lecteur peut se référer à [Dijkman and Dumas 2004]. Il faut noter que ces concepts apparaissent parfois avec des noms différents dans la littérature. Par exemple, Casati et al. [Alonso et al. 2003] utilisent le terme « protocole de collaboration » pour se référer au concept de chorégraphie. De façon similaire, en ebXML [UN/CEFACT and OASIS 1999], une famille de standards pour le développement de services pour le commerce électronique, l'interface fournie est appelée « profil de protocole de collaboration » (*collaboration protocol profile* en anglais) alors que l'interface requise est appelée « accord de protocole de collaboration » (*collaboration protocol agreement*).

Le lecteur intéressé par les problèmes liés à la détection d'incompatibilités entre interfaces fournies et interfaces requises peut se référer à [Martens 2005]. La résolution de telles incompatibilités fait l'objet de recherches en cours (voir par exemple [Benatallah et al. 2005a, Fauvet and Ait-Bachir 2006]).

4.3 Description de services Web

Dans cette section sont discutées les questions liées à la description des interfaces des services. Cette description peut porter sur les aspects structurels (section 4.3.1) et/ou comportementaux (section 4.3.2) des services, aussi bien que sur leurs aspects non-fonctionnels (section 4.3.3).

Les langages présentés dans cette section sont fondés sur XML (*eXtensible Markup Language*) [W3C-XML]. XML est un standard de représentation de données semi-structurées. Dans un document XML, la structure des données est fournie par le biais de l'utilisation de balises (comme en SGML *Standard Generalized Markup Language* [Goldfarb 1990], mais en s'affranchissant des aspects liés à la présentation des données). Cette structure n'est pas aussi rigide que celle d'une base de données relationnelle par exemple. Dans un document XML, les données peuvent être définies selon un schéma, mais cela n'est pas obligatoire et le schéma peut laisser quelques parties partiellement spécifiées.

Une problématique généralement associée à la description de services est celle de leur publication et de leur découverte. Un standard proposé pour s'attaquer à cette problématique est UDDI [OASIS UDDI 2005]. UDDI définit une interface de programmation pour publier des descriptions de services dans des répertoires dédiés, pour soumettre des requêtes à base de mots-clés sur ces répertoires et pour naviguer au travers des descriptions obtenues par le biais de ces requêtes. Étant donnée l'existence de moteurs de recherche sophistiqués aussi bien pour des Intranets qu'au niveau de l'Internet tout entier, et d'autres technologies pour la gestion de répertoires tels que LDAP [Koutsonikola and Vakali 2004], la valeur ajoutée apportée par UDDI est difficile à identifier. Peu d'architectures à base

de services s'appuient aujourd'hui sur UDDI et le futur de ce standard est incertain. Par conséquent, nous ne détaillons pas UDDI dans ce chapitre, malgré les associations souvent faites entre UDDI et la problématique de la description de services.

4.3.1 Description fonctionnelle et structurelle : WSDL

WSDL (*Web Services Description Language*) [W3C-WSD-Group] est un langage de la famille XML permettant de décrire les types de données supportés et les fonctions offertes par un service Web. L'objectif est de fournir la description, en XML, des services indépendamment de la plate-forme et du langage utilisés et sous une forme que des personnes ou des programmes peuvent interpréter. Les descriptions WSDL sont en fait l'équivalent des interfaces IDL (*Interface Definition Language*) de CORBA par exemple.

Dans le langage WSDL, un service est vu comme une collection de messages pour les échanges et d'une collection de points d'entrée. Un point d'entrée consiste en la description abstraite d'une interface et de son implantation. La description abstraite contient : (i) la définition des messages qui sont consommés et générés par le service (les entrées et les sorties), et (ii) la signature des opérations offertes par le service. La mise en correspondance (*implementation binding*) entre l'interface et son implantation est fournie. Elle contient essentiellement l'indication du protocole utilisé pour échanger des messages avec le service (par exemple SOAP au-dessus de HTTP) et les associations entre la description de l'interface abstraite du service et les types de messages supportés par le protocole de communication sous-jacent (par exemple SOAP). La description WSDL de l'interface donnée figure 4.2 du service de réception d'une commande offert par le fournisseur est ébauchée ci-dessous.

L'élément **definitions** constitue la racine du document et fournit les espaces de noms.

```
<?xml version="1.0"?>
<definitions name="RecevoirBdC"
  targetNamespace="http://www.yothuyindi.fr:8080/exemple/fournisseur.wsdl"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema/"
  xmlns:wns="http://www.yothuyindi.fr:8080/exemple/fournisseur.wsdl"
  xmlns:xsdl="http://www.yothuyindi.fr:8080/exemple/fournisseur.xsd"
  xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
  xmlns="http://schemas.xmlsoap.org/wsdl/">
```

Les types de données, paramètres d'entrée et/ou de sortie sont éventuellement décrits ensuite en XMLSchema [W3C-XMLSchema]. La description d'une commande est décrite ci-dessous comme une date et une liste non vide de produits à commander, chacun dans une quantité non nulle donnée.

```
<types>
  <schema targetNamespace="http://www.yothuyindi.fr:8080/exemple/fournisseur.xsd">
    <xsd:complexType name="Commande"><xsd:sequence>
      <xsd:element name="dateCommande" type="xsd:date">
      <xsd:element name="LigneDeCommande" minOccurs="1" maxOccurs="unbounded">
        <xsd:complexType><xsd:sequence>
          <xsd:element name="RéférenceProduit" type="xsd:string"/>
          <xsd:element name="Quantité" type="xsd:positiveInteger"/>
        </xsd:sequence></xsd:complexType></xsd:element>
      </xsd:sequence></xsd:complexType>
    </schema>
  </types>
```

```
...</schema>
</types>
```

Vient ensuite la description de la liste des définitions des messages échangés indépendamment de l'implantation du service et du protocole utilisé. Ici, le document décrit deux messages pour l'interaction : le premier message est la requête reçue par le service, l'autre est l'accusé de réception renvoyé. Les valeurs fournies pour les attributs `element` sont des noms de types XML Schema ou définis dans la partie `types` ci-dessus.

```
<message name="SoumissionBdC">
    <part name="body" element="xsd1:Commande"/>
</message>
<message name="RécépisséBdC">
    <part name="body" element="xsd:string"/>
</message>
```

Les opérations offertes par le service sont exposées par le biais de points d'entrée. Un point d'entrée (élément `portType`) fournit la signature de chaque opération et doit par la suite être associé à une implantation particulière (voir plus loin la description de la partie `binding`). WSDL permet l'existence de plusieurs points d'entrée dans un même document. Ainsi, la même opération peut être rendue disponible au travers d'implantations différentes.

```
<portType name=pt_RecevoirBdC>
    <operation name="op_Commande">
        <input message="wsn:SoumissionBdC">
        <output message="wsn:RécépisséBdC">
    </operation>
    <operation name=...
    ...
</portType>
```

Dans la description ci-dessus les valeurs des attributs `message` font référence à un message défini plus haut (élément `message` du document WSDL).

La définition de la signature d'une opération s'appuie sur trois sous-éléments possibles : `input` pour le message en entrée, `output` pour celui en sortie et `fault` pour un message d'erreur émis par le service. Les combinaisons possibles de ces sous-éléments permettent de décrire divers modèles d'opérations :

- Réception de message (sous-élément `input`) : le service reçoit un message envoyé par un client ;
- Réception - émission (sous-élément `input` suivi de `output` et éventuellement de `fault`) : le service reçoit une requête sous forme d'un message puis émet une réponse sous forme d'un retour de message. C'est le modèle classique RPC (*Remote Procedure Call*, voir chapitre 1) ;
- Émission - retour (sous-élément `output` suivi de `input` et éventuellement de `fault`) : le service envoie une requête à un client (suite à une inscription préalable) sous forme d'un message ; le client répond sous forme d'un message de retour.
- Émission (sous-élément `output`) : le service envoie un message (d'alerte par exemple) à un client.

La dernière partie du document fixe la mise en correspondance entre chaque point d'entrée (un ensemble d'opérations) et son implantation, et permet de définir quels services sont associés à quelle mise en correspondance. Dans le fragment donné ci-dessous, l'implantation des points d'entrée s'appuie sur SOAP.

```

<binding name="bd_opCommande" type=pt_RecevoirBdC>
  <soap:binding style="document" transport="http://schemas.xmlsoap.org/soap/http"/>
  <operation name="op_Commande">
    <soap:operation soapAction="http://www.yothuyindi.fr/exemple/op_Commande"/>
    <input>
      <soap:body
        use="literal"
        namespace="http://www.yothuyindi.fr/exemple/fournisseur.xsd"
        encodingStyle="http://schemas.xmlsoap.org/soap/encoding"/>
    </input>
    <output>
      <soap:body
        use="literal"
        namespace="http://www.yothuyindi.fr/exemple/fournisseur.xsd"
        encodingStyle="http://schemas.xmlsoap.org/soap/encoding"/>
    </output>
  </operation>
</binding>

<service name=ServiceFournisseur>
  <documentation> Service de réception des commandes </documentation>
  <port name="PortSoumissionCommande" binding="bd_opCommande">
    <soap:address location="//www.yothuyindi.fr/exemple/fournisseur"/>
  </port>
</service>
<!-- fermeture de la balise racine -->
</wsdl:definitions>
```

Le document final est obtenu par la concaténation des extraits fournis ci-dessus.

Comme le suggère l'exemple traité ici, la production de documents WSDL est un travail fastidieux dont le programmeur peut être déchargé grâce à l'utilisation d'outils pour la génération automatique de la description WSDL. Ces outils prennent en entrée la description du service sous forme d'une classe Java (ou une archive jar) ou d'un objet COM.

4.3.2 Description comportementale

La portée de WSDL est limitée la description des types des données incluses dans les messages qu'un service est capable de recevoir ou d'émettre. Dans de nombreuses applications, ces descriptions uniquement structurelles n'offrent pas assez d'information sur les contraintes régissant les interactions dans lesquelles un service peut ou est censé s'engager. Dans certains cas, ces contraintes sont assez simples, comme par exemple : « le fournisseur n'envoie le bordereau de livraison qu'après avoir reçu la confirmation du paiement ». D'autres fois, ces contraintes peuvent être relativement complexes. Par exemple, ci-dessous est donnée une description résumée d'une des interfaces définie par le

consortium RosettaNet⁵ pour le traitement de bons de commandes :

Lorsqu'un fournisseur reçoit un « Bon de Commande (BdC) » de la part d'un client, il doit répondre avec un « Récépissé de BdC ». Par la suite, le fournisseur enverra zéro, une ou plusieurs « Réponses à BdC » au client jusqu'à avoir donné une réponse (une acceptation ou un rejet) pour chacune des lignes du BdC. Au cours de ce processus, le fournisseur peut recevoir une « Annulation de BdC » de la part du client, dans ce dernier cas le fournisseur répond avec un « Récépissé d'Annulation de BdC ». Dès lors, le fournisseur ne doit plus envoyer de « Réponses à BdC » au client.

En WSDL, il serait possible de définir chacun des messages échangés dans le protocole ci-dessus. Cependant, il ne serait pas possible d'exprimer que le fournisseur peut envoyer « plusieurs réponses à un BdC jusqu'à recevoir une annulation ». Dans des systèmes tels que CORBA, ces dépendances comportementales doivent être documentées en langue naturelle et c'est ensuite aux développeurs d'assurer que : (i) les services remplissent les contraintes exprimées dans leur documentation ; et (ii) les applications utilisent les services conformément avec leur documentation.

Plusieurs initiatives de recherche, de développement et de standardisation sont en cours afin de définir des langages de description de services prenant en compte des aspects comportementaux, et d'utiliser ces descriptions « comportementales » lors du développement et de l'exécution des services. De telles initiatives s'appuient sur des résultats de recherche dans le domaine des systèmes à base de composants [Yellin and Strom 1997] et des systèmes orientés-processus [Dumas et al. 2005]. Deux efforts de standardisation dans ce domaine sont sur le point d'aboutir : le langage de description de processus métiers exécutables pour services Web (*Web Services Business Process Execution Language*, BPEL) [Andrews et al. 2003] et le langage de description de choréographies de services Web (*Web Services Choreography Definition Language*, WS-CDL) [Kavantzas et al. 2005].

BPEL est un langage fondé sur XML et permettant de décrire aussi bien des interfaces comportementales (au sens défini dans la section 4.2) que des orchestrations complètement exécutables. En quelques mots, BPEL permet de décrire des actions communicationnelles (envois et réceptions de message dont les types sont décrits en WSDL et XML Schema), et de lier ces actions au travers d'opérateurs de flot de contrôle (par exemple la séquence, le choix, l'itération, et les clauses *throw/catch* pour le traitement des exceptions). Nous montrons ci-dessous, un extrait du code BPEL correspondant à l'interface RosettaNet énoncée plus haut :

```
<sequence>
  <receive operation="BdC"/>
  ...
  <while ...>
    <invoke operation="reponse-a-BdC"/>
  </while>
  ...
  <onMessage operation="annulation-de-BdC" ...>
    <throw faultName="annulationEnCours"/>
  </onMessage>
```

⁵Voir <http://www.rosettanet.com>.

```

...
<catch faultName="annulationEnCours">
  <invoke operation="RecepisseAnnulation-de-BdC" .../>
</catch>
...
</sequence>
```

Dans cet extrait, on distingue deux types d'actions communicationnelles : `receive` pour la réception, et `invoke` pour l'envoi. Ces actions sont composées en utilisant les opérateurs de séquencement et d'itération (`sequence` and `while`) que l'on retrouve dans les langages de programmation impératifs. En outre, à tout moment (clause `onMessage`) un message de type `annulation-de-BDC` peut être reçu et alors une exception est levée et ensuite attrapée au travers des constructions `throw` et `catch` que l'on retrouve aussi dans des langages de programmation. On peut remarquer que le code BPEL ci-dessus se limite à décrire des actions communicationnelles et leurs dépendances. Si l'on voulait utiliser BPEL pour planter le service correspondant, beaucoup plus de détails seraient nécessaires.

En tant que langage d'implantation de services, BPEL est incorporé dans plusieurs outils de construction d'applications à base de services (voir Section 4.4). Cependant, en tant que langage de description d'interfaces, l'outillage autour de BPEL est presqu'inexistant. Quelques travaux de recherche montrent comment des interfaces comportementales décrites en BPEL peuvent être utilisées pour la vérification statique [Fu et al. 2004] ainsi que pour le suivi et analyse d'exécutions de services, que ce soit en temps réel [Baresi et al. 2004] ou a posteriori [Aalst et al. 2005]. Cependant, ces techniques n'ont pas encore été adoptées dans des outils commerciaux.

Ce dernier commentaire s'applique également à WS-CDL, qui essaye d'aller plus loin que BPEL en s'attaquant non seulement à la description d'interfaces comportementales, mais aussi à la description de chorégraphies à différents niveaux de détails, allant de la modélisation conceptuelle jusqu'à l'implantation. L'idée de WS-CDL est que les chorégraphies décrites dans un langage semi-exécutables, peuvent être vérifiées statiquement, testées par simulation, et dans le cas des chorégraphies définies au niveau le plus détaillé elle peuvent être utilisées pour générer automatiquement les interfaces correspondant à chacun des rôles mis en jeu. A ce jour, les outils de développement de services basés sur WS-CDL sont très loin d'avoir atteint leur maturité⁶. Pour un aperçu et une analyse critique de WS-CDL, voir [Barros et al. 2005b].

4.3.3 Description d'aspects non-fonctionnels : « WS-Policy »

Les services Web étant généralement développés par des équipes indépendantes, ont besoin d'être décrits de façon précise selon des conventions standards, et de telle manière que leurs descriptions puissent être utilisées au maximum pour automatiser le processus de développement et de déploiement des futurs services devant interagir avec un service existant. WSDL et BPEL permettent de décrire les opérations fournies par un service Web, les types de données des messages devant être échangés pour invoquer ces opérations, et les dépendances comportementales entre ces opérations. Cependant, ils ne permettent pas de décrire des aspects non-fonctionnels des services tels que leur capacité à garantir

⁶Voir par exemple Pi4Tech : <http://www.pi4tech.com>.

une certaine qualité de service par rapport à des préoccupations telles que la sécurité, la fiabilité, la journalisation des accès ou la gestion de transactions (voir le chapitre 1).

Ce manque est en partie compensé par le concept de *politiques d'usage*. Une politique d'usage est une énonciation explicite des possibilités et des restrictions d'usage d'un service Web. « WS-Policy »est un langage extensible permettant d'exprimer des politiques (ou règles) d'usage sous forme de conjonctions et de disjonctions (au sens logique) d'*assertions* [Kaler and Nadalin]. Dans ce contexte, une assertion est une donnée par laquelle un service exprime qu'il permet aux clients (ou qu'il requiert des clients) de procéder d'une certaine manière lorsqu'ils accèdent aux opérations du service. « WS-Policy »ne définit pas des types d'assertions d'usages particuliers. Cependant, ces types d'assertions sont définis par d'autres standards proposés tels que « WS-Security-Policy », « WS-Reliable-Messaging »et « WS-Addressing », qui définissent respectivement des types d'assertion liés à la sécurité, la fiabilité, et l'adressage. Ainsi, en utilisant les types d'assertion définis dans « WS-Security-Policy », il est possible d'exprimer qu'un service requiert que tous les messages soient signés ou qu'ils soient signés et encryptés avec une clé publique donnée. De même, en utilisant les types d'assertion définis dans « WS-Reliable-Messaging », on peut exprimer qu'un service donné opère selon un protocole de renvoi de messages permettant d'assurer que les messages arrivent au moins une fois, ou exactement une fois. Enfin, les types d'assertion définis dans « WS-Addressing »permettent par exemple d'exprimer qu'un service peut envoyer le résultat d'un appel d'opération à une entité différente de celle ayant initié l'appel.

« WS-Policy »constitue un pas en avant vers des modèles de développement d'applications réparties basées sur des descriptions détaillées couvrant des aspects aussi bien fonctionnels que non-fonctionnels. Cependant, pour réaliser la vision de services Web comme entités indépendantes, davantage d'efforts de recherche sont nécessaires afin de déboucher sur des modèles et des langages permettant de décrire l'environnement organisationnel dans lequel les services Web évoluent. Par exemple, pour faciliter l'adoption des services Web dans des applications de type *business-to-business*, il serait souhaitable d'avoir des descriptions de services couvrant des garanties de disponibilité et de temps de réponse, ainsi que des politiques de prix et de pénalités, de modalités de paiement, de réputation, etc. (voir [O'Sullivan et al. 2002]). Il ne faut pas s'attendre à ce que toutes ces propriétés fassent l'objet de standardisation ou soient prises en compte dans des outils de développement de services car la plupart d'entre elles touchent à des aspects socio-techniques et varient selon les domaines d'application. Cependant, de telles descriptions non-fonctionnelles pourraient constituer un outil d'évaluation et de sélection lors des phases initiales (c'est-à-dire au cours de l'analyse du domaine) dans la construction d'applications à base de services.

4.4 Implantation de services Web

Dans cette section, nous discutons des principaux outils permettant la mise en œuvre de services Web. Nous décrivons tout d'abord les principes de base du protocole SOAP (voir section 4.4.1), puis la spécification d'en-têtes permettant de transmettre des informations particulières (voir section 4.4.2). Nous donnons ensuite, dans la section 4.4.3, des éléments de comparaisons de SOAP avec REST, une solution concurrente pour l'implantation de services Web. Dans la section 4.4.4 nous introduisons rapidement deux implantations de

SOAP. Finalement la section 4.4.5 discute de BPEL un langage orienté processus.

4.4.1 SOAP (*Simple Object Access Protocol*)

Les interactions entre services Web s'effectuent par le biais d'envois de messages structurés au format XML. Le protocole SOAP (*Simple Object Access Protocol*) [W3C-XMLP-Group] fournit le cadre permettant ces échanges. SOAP est originellement issu de tentatives précédentes visant à standardiser l'appel de procédures à distance, et en particulier de XML-RPC [Winer 1999]. Mais à la différence des technologies RPC, SOAP n'est pas fondamentalement lié à la notion d'appel de procédure. En effet, SOAP vise à faciliter l'échange de messages XML, sans se limiter à des messages dont le contenu encode des paramètres d'appel de procédure et sans favoriser des échanges bidirectionnels de type requête-réponse comme c'est le cas des protocoles RPC. Dans le jargon des services Web, SOAP permet d'encoder des interactions orientées-RPC mais aussi des interactions orientées-document.

Une autre caractéristique de SOAP est de faire abstraction de la couche de transport sous-jacente. Bien que la pratique la plus répandue soit d'utiliser SOAP au-dessus de HTTP, il existe aussi des implantations de SOAP au-dessus d'autres protocoles tels que le protocole d'échange de messages électroniques SMTP, et les protocoles de transport orientés-message de Microsoft et IBM, à savoir MSMQ et MQSeries respectivement.

La manière d'implanter SOAP au-dessus d'un protocole de transport donné est appelée une *liaison SOAP* (« SOAP binding » en anglais). Une liaison SOAP définit, en particulier, l'encodage des messages (nécessaire en particulier lorsque le protocole sous-jacent utilise un format binaire), la méthode pour l'échange de messages, l'encodage des noms d'opérations (appelés « SOAP Actions »), et la façon dont différents messages (y compris les messages d'erreur) appartenant à la même interaction sont corrélés. Par exemple, la liaison SOAP au-dessus de HTTP définit que les messages sont encodés dans un « type de média » appelé « application/soap+xml » (c'est-à-dire en XML avec quelques extensions), que le nom de l'opération correspondant à une requête est donné dans un en-tête HTTP appelé « SOAPAction », et que les messages dans une interaction sont échangés au travers des méthodes POST et GET fournies par HTTP. D'autres règles (dont nous ne donnons pas les détails) définissent la manière dont les messages appartenant à un même échange (y compris les messages d'erreur) sont corrélés en exploitant les caractéristiques des méthodes POST et GET.

Outre la définition échanges de messages en faisant abstraction de la couche de transport, SOAP définit une structure standard de messages dans laquelle le contenu des messages est séparé des métadonnées liées à l'échange des messages. Ainsi, un message SOAP est constitué de deux parties : un en-tête et un corps. L'en-tête indique l'objet du message (l'appel d'une opération ou le retour de résultats), la description de l'expéditeur, et l'information requise pour acheminer le message au destinataire. Le corps du message peut contenir : (i) un document quelconque ; (ii) l'appel d'une opération offerte par le service destinataire, avec les valeurs pour les paramètres d'entrée ; (iii) les valeurs produites en résultat d'un appel ; ou bien (iv) un message d'erreur. Ainsi, SOAP offre diverses possibilités d'interactions entre les services : soit des échanges de documents, soit des interactions de type RPC.

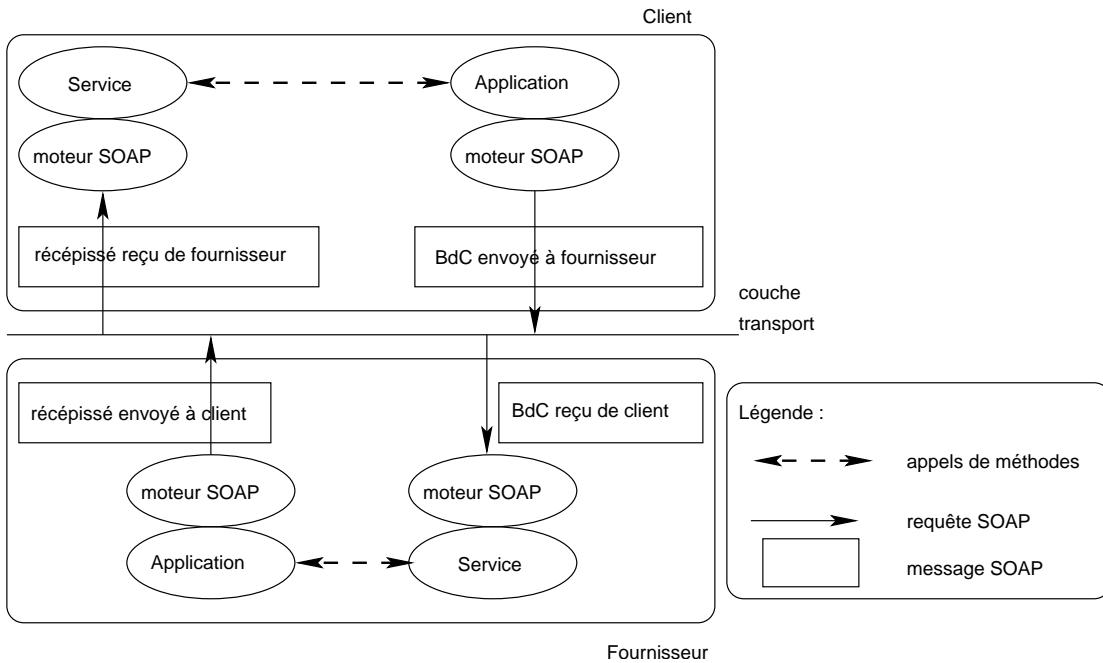


Figure 4.5 – Interactions entre les partenaires Client et Fournisseur

La figure 4.5 schématisé un scénario d’interaction entre les partenaires Client et Fournisseur déjà introduits dans la section 4.2 (les interactions avec l’entrepôt ne sont pas montrées). Le client transmet une commande au fournisseur en lui envoyant un bon de commande. Ce bon de commande est transmis, par une application du client, sous forme d’une requête SOAP. Chez le fournisseur, le serveur d’application reçoit la requête (sous forme d’une requête HTTP) et la transmet au moteur SOAP. Ce dernier décode le message reçu et effectue les appels aux procédures correspondantes de l’application locale. Lorsque les vérifications de disponibilité des articles commandés et les interactions avec l’entrepôt sont terminées, l’application du fournisseur transmet, de la même manière, la facture au serveur d’application du client (les serveurs d’application ne sont pas figurés).

Il existe plusieurs mécanismes pour construire, analyser, et échanger des messages SOAP dans des langages variés tels que Java, C++, Perl, C#, etc. Ces implantations permettent de générer les en-têtes de messages SOAP et de mettre en correspondance le contenu du corps du message avec les structures de données définies dans le langage hôte (voir Section 4.4.4).

4.4.2 Extensions de SOAP : spécifications WS-*

Comme indiqué ci-dessus, l’en-tête d’un message SOAP est destiné à contenir des métadonnées sur l’échange des messages. Ces métadonnées prennent la forme d’un nombre de (sous-)en-têtes, chacun encodé sous forme d’un élément XML. Chaque sous-en-tête permet de transmettre des informations liées à l’adressage, à la fiabilité, à la sécurité, ou à d’autres propriétés traditionnellement associées aux interactions distribuées (voir le chapitre 1).

Il est en principe possible d’inclure n’importe quel ensemble d’en-têtes dans un message

SOAP. Un fournisseur peut en théorie exprimer que son service est capable de recevoir des en-têtes X, Y, Z, avec une certaine sémantique pour chacun, et les programmeurs des applications qui accèdent à ce service sont alors libres de les utiliser ou pas. De façon symétrique, une application peut envoyer un message SOAP avec n'importe quel ensemble d'en-têtes, et c'est au service destinataire de déterminer ceux qu'il peut interpréter et ceux qu'il peut ignorer. L'attribut XML `mustUnderstand` peut être associé à chaque en-tête pour exprimer si cet en-tête doit obligatoirement être interprété par le destinataire ou si il peut être ignoré. Le destinataire est tenu de renvoyer un message d'erreur s'il détecte un en-tête qui devrait être interprété obligatoirement, et qu'il ne lui est pas possible d'interpréter.

Différentes spécifications (certaines en passe de devenir des standards), connues généralement sous le nom collectif WS-*, définissent des ensembles d'en-têtes SOAP perçus comme étant particulièrement importants pour un grand nombre d'applications. Trois des spécifications les plus avancées en terme de leur standardisation sont WS-Addressing (pour l'adressage), WS-Security (pour l'authentification et la non-répudiation) et WS-Reliable-Messaging (pour le renvoi répété de messages afin de garantir leur livraison fiable et ordonnée). Par exemple, WS-Addressing définit les (sous-)en-têtes suivants :

- `MessageID` : où l'on peut associer un identificateur au message.
- `RelatesTo` : où l'on peut faire référence à l'identificateur d'un message antérieur.
- `From`, `To` : qui fixent l'expéditeur et le destinataire du message.
- `Reply-to` : qui donne l'adresse de retour, c'est-à-dire l'URL à laquelle des réponses éventuelles au message doivent être envoyées. Cet en-tête est utile par exemple lorsque la réponse à une requête est envoyée au travers d'une deuxième connexion HTTP établie ultérieurement (selon le mode d'interaction dit *asynchrone*), au lieu d'être incluse dans le message de retour de la connexion HTTP courante (selon le mode d'interaction dit *synchrone*).

A titre d'exemple, le message HTTP suivant correspond à l'envoi d'un bon de commande par un client, d'après l'interaction décrite dans la section 4.2. Ce message inclut un identificateur de message et une adresse à laquelle l'éventuelle réponse doit être expédiée ultérieurement.

```
POST /orabpel/default/Supplier HTTP/1.0
Content-Type: text/xml; charset=utf-8
Accept: application/soap+xml, application/dime, multipart/related, text/*
User-Agent: Axis/#axisVersion#
SOAPAction: "BdC"
...
<soapenv:Envelope
    xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
    <soapenv:Header>
        <MessageID
            xmlns="http://schemas.xmlsoap.org/ws/2003/03/addressing"
            xmlns:orabpel="http://schemas.oracle.com/bpel" >
            bpel://www.yothuyindi.fr/default/Customer~1.1/301-BpInv0-BpSeq0.3-3
        </MessageID>
        <ReplyTo xmlns="http://schemas.xmlsoap.org/ws/2003/03/addressing">
            <Address>
                http://GA2550:9710/orabpel/default/Customer/1.1/Supplier/SupplierRequester
            </Address>
        </ReplyTo>
    </soapenv:Header>
</soapenv:Envelope>
```

```

        </Address>
    </ReplyTo>
</soapenv:Header>
<soapenv:Body>
    <BonDeCommande> ... </BonDeCommande>
</soapenv:Body>
</soapenv:Envelope>
```

La réponse du fournisseur peut alors être envoyée au travers d'une deuxième connexion HTTP. Cette réponse doit être envoyée à l'adresse indiquée dans l'en-tête « ReplyTo » du message ci-dessus, et doit se référer à l'identificateur du message du message ci-dessus au travers de l'en-tête « RelatesTo ».

Un exemple de message de réponse est donné ci-dessous :

```

POST /orabpel/default/Customer/1.1/Supplier/SupplierRequester HTTP/1.0
Content-Type: text/xml; charset=utf-8
Accept: application/soap+xml, application/dime, multipart/related, text/*
User-Agent: Axis/#axisVersion#
Host: GA2550:9710
SOAPAction: "ResponseBdC"
...

<soapenv:Envelope
    xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
    <soapenv:Header>
        <RelatesTo xmlns="http://schemas.xmlsoap.org/ws/2003/03/addressing">
            bpel://www.yothuyindi.fr/default/Customer~1.1/301-BpInv0-BpSeq0.3-3
        </RelatesTo>
    </soapenv:Header>
    <soapenv:Body>
        <ReponseBonDeCommande>
            ...
        </ReponseBonDeCommande>
    </soapenv:Body>
</soapenv:Envelope>
```

Pour une description relativement complète de WS-Addressing et autres spécifications WS-*, le lecteur peut se reporter à [Weerawarana et al. 2005].

4.4.3 Le débat SOAP vs. REST

Dans la section 4.4.1, nous avons indiqué que la façon la plus répandue d'utiliser SOAP consiste à s'appuyer sur le protocole HTTP en utilisant la liaison SOAP-HTTP. Dans cette liaison, il est possible d'associer plusieurs opérations à la même URL. Par exemple, un service de ventes peut être placé à l'URL www.unserviceevente.fr/serviceweb, et toutes les opérations de ce service (demande de devis, placement de bon de commande, suivi de bon de commande, etc.) peuvent être fournies sur cette même URL. Une application donnée peut alors appeler chacune de ces opérations en utilisant la méthode HTTP POST, et en incluant le nom de l'opération concernée par la requête dans l'en-tête HTTP « SOAPAction ».

Cette approche, ainsi que d'autres caractéristiques de SOAP, ont fait l'objet de nombreuses critiques. Les arguments principaux mis en avant contre SOAP reposent sur le fait que : (i) il rajoute peu de fonctionnalités au-dessus de ce qu'il est déjà possible de faire avec HTTP et XML (sans les extensions apportées par SOAP) ; et (ii) en associant plusieurs opérations à une même URL, il rend difficile, voire impossible, l'utilisation de l'infrastructure de « caching » associée au protocole HTTP, qui constitue sans doute l'un des points forts de HTTP.

Une approche alternative pour l'implantation de services Web appelée REST (REpresentational State Transfer) a été définie [Fielding 2000]. Dans cette approche, chaque opération d'un service est associée à une URL distincte et l'accès à chaque URL peut être réalisé en utilisant l'une des quatre méthodes fournies par HTTP : POST, GET, PUT et DELETE. Le contenu des messages est alors encodé en XML, et la distinction entre en-tête et contenu de message est laissée à la charge des applications qui requièrent cette distinction. Le résultat est un ensemble de conventions plus simples que celles de SOAP, et la possibilité d'utiliser des bibliothèques existantes pour l'échange de messages sur HTTP, éliminant ainsi le besoin de plates-formes implantant SOAP, qui dans certains cas peuvent être considérées comme étant plus « lourdes » et plus difficiles à utiliser.

Plusieurs services Web populaires disponibles à l'heure actuelle utilisent l'approche REST (appelée aussi « XML sur HTTP »). Ceci est le cas notamment de la maison de ventes au enchères en ligne eBay, qui rend une partie de ses fonctionnalités, accessibles sous forme de services Web « style REST »(voir <http://developer.ebay.com/rest>). Il en est de même du site de vente par Internet Amazon (voir <http://www.amazon.com/webservices>).

Le débat SOAP vs. REST est encore ouvert et les avis divergent considérablement dans la communauté. Les avantages et désavantages relatifs de SOAP et REST peuvent être résumés comme suit :

- REST ne requiert pas d'infrastructure spécifique pour le développement et l'exécution des services Web. Il repose sur l'infrastructure HTTP existante, qui a fait ses preuves dans le contexte des applications HTML classiques. De nombreux outils de programmation d'applications au-dessus de HTTP existent, et peuvent être combinés avec des outils de manipulation de documents XML pour construire des services Web « style REST ». Dans le cas de SOAP, il est nécessaire d'utiliser des outils spécifiques (voir la section 4.4.4) qui sont parfois difficiles à déployer et n'ont pas complètement fait leurs preuves.
- SOAP peut opérer au-dessus d'autres protocoles que HTTP, en particulier sur des protocoles permettant des communications asynchrones telles que SMTP, JMS, MSMQ et MQSeries.
- Lorsqu'il est utilisé en conjonction avec des spécifications WS-*, SOAP permet de prendre en compte des aspects liés à la sécurité, la fiabilité et l'adressage et routage de messages. Bien sûr, il serait possible de faire de même en utilisant l'approche REST, mais les standards et l'infrastructure correspondant ne sont pas en place.

En conclusion, l'approche REST est viable (voire préférable) dans le cas de services Web avec les caractéristiques suivantes : (i) ils n'ont besoin d'être exposés que sur HTTP ou HTTPS ; (ii) les communications asynchrones (et donc l'adressage explicite) ne sont pas requises ; (iii) ils ne requièrent pas de garanties de sécurité au delà de celles offertes

par HTTPS ; et (iv) la fiabilité peut facilement être traitée au niveau des applications.

4.4.4 Exemples d'implantations de SOAP

Nous donnons ci-après un aperçu de deux implantations *Open Source* du protocole SOAP, qui se distinguent par leur principe : l'une, Axis, est à base de bibliothèques, l'autre, Beehive, est à base d'annotations (voir [W3C-XMLP-Group] pour obtenir une liste des implantations de SOAP).

Axis [Axis] est un projet du groupe Web Services d'Apache [Apache]. Le moteur Axis, qui joue le rôle de client et de serveur SOAP, se présente sous forme d'un ensemble de bibliothèques : une version Java est actuellement disponible, une autre en C++ est en cours de développement.

La version Java d'Axis est constituée d'un ensemble de bibliothèques qui implantent l'API Java JAX-RPC d'une part, et fournissent d'autre part, des outils pour faire de la journalisation, de la génération et de l'analyse de documents WSDL. Ces bibliothèques peuvent être empaquetées comme une application Web, puis rendues disponibles par le biais d'une *servlet*.

Axis propose un premier mode de déploiement grâce auquel un programme Java est immédiatement exposé comme un service Web. Il suffit de placer ce programme (non compilé) dans le répertoire dédié à l'application Web Axis. Cette méthode, si elle a le mérite d'être simple, a pour inconvénients d'obliger le fournisseur à montrer le code source et de ne pas être adaptée au déploiement d'applications complexes. Palliant ces inconvénients, le second mode de déploiement s'appuie sur la description des éléments d'un service (classes et méthodes) qu'il faut exposer. Le moteur Axis propose une opération de déploiement (et une autre inverse de suppression) qui prend en entrée le programme Java compilé et la description du déploiement (*Web Service Deployment Descriptor*). Cette seconde méthode, spécifique à Axis, permet de spécifier les classes et les méthodes à exposer. À l'inverse de la première méthode, elle permet surtout de pouvoir spécifier la durée de vie des objets générés par l'exécution du service et de permettre l'utilisation de composants Java (*Java Beans*). La durée de vie des objets générés par l'exécution du service peut être associée soit à la requête, soit à l'ensemble des exécutions du service, soit enfin à la session. Axis prend à sa charge la sérialisation/désérialisation des classes Java à condition qu'elles aient été implantées dans des composants Java.

Beehive [Beehive] est un autre projet Apache dont l'objectif global est de définir un modèle d'annotations pour Java dans la perspective de réduire la quantité de code Java à produire pour développer des applications J2EE (voir chapitre 5).

Une partie du projet Beehive est consacrée à l'implantation de JSR181 (*Java Specification Request - Web Services Metadata for the Java Platform*) [JSR 181] qui fixe un modèle d'annotations pour la construction de services Web développés en Java. L'idée est d'offrir un outil pour le développement de services Web qui seront ensuite rendus disponibles par le biais de n'importe quel moteur SOAP, comme par exemple Axis.

Beehive s'appuie sur trois types d'annotations : `@WebService` pour spécifier quelle classe est exposée comme service Web ; `@WebMethod` pour indiquer les opérations du service, et pour chacune la méthode qui l'implante. Pour chaque opération, l'annotation `@WebParam` permet d'en déclarer les paramètres.

4.4.5 Implantation à base d'un langage spécifique : BPEL

BPEL est aujourd'hui un standard de facto pour implanter des services Web selon un point de vue orienté processus. Des plates-formes matures et reconnues, telles que BEA WebLogic, IBM WebSphere, Microsoft BizTalk, SAP XI et l'outil de gestion de processus BPEL d'Oracle supportent BPEL à des degrés divers démontrant ainsi l'intérêt réel de ce langage. ActiveBPEL est une autre proposition assez complète dans le domaine de l'*Open Source*⁷.

Comme brièvement indiqué dans la section 4.3, en BPEL, la logique des interactions entre un service et son environnement est décrite par le biais d'une composition d'actions élémentaires de communication (émission, réception ou émission/réception). Ces actions sont reliées les unes aux autres par un flot de contrôle spécifié par des constructions telles que la composition parallèle (avec un nombre fixé de branches), séquentielle, et conditionnelle, des règles de type événement-action et des clauses de capture/transmission d'erreur. La manipulation des données s'effectue par le biais de variables, comme dans un langage de programmation impérative.

Plus précisément, un processus exprimé en BPEL est construit à partir d'activités primitives qui peuvent être composées pour définir d'autres activités plus complexes. Les activités primitives sont : la réception (`receive`) qui bloque l'exécution jusqu'à l'arrivée d'un message correspondant à une opération ; l'appel (`invoke`) et la réponse (`reply`) qui chacune effectue un envoi de message correspondant à une opération ; l'affectation (`assign`) qui associe une expression (écrite en XPath ou XSLT) à une variable ; l'attente (`wait`) qui bloque l'exécution pour une période de temps fixée ; la transmission (`throw`) d'erreur et la fin (`exit`) pour terminer l'exécution.

La plupart des opérateurs de composition proposés par BPEL correspondent à ceux déjà présents dans les langages de programmation impérative : la séquence (`sequence`) pour imposer un ordre dans l'exécution, le choix (`switch`) pour exprimer un branchement conditionnel, et l'itération (`while`). Le concept de bloc issu des langages de programmation structurée est présent dans BPEL par le biais de la notion de portée (`scope`).

En plus de ces opérateurs de composition, BPEL offre plusieurs constructions pour la programmation concurrente : la composition parallèle (`flow`) pour exécuter plusieurs activités en parallèle ; la sélection (`pick`) pour choisir l'un parmi les messages entrants ou les *timers* ; des dépendances de précédences (`link`) peuvent être définies entre des activités qui autrement s'exécuteraient en parallèle. Finalement, des règles de type événement-action (`event handlers`) peuvent être associées à des blocs (`scope`). Ces règles sont déclenchées par la réception d'un message ou à l'expiration d'un délai, à n'importe quel moment au cours de l'exécution du bloc correspondant.

Dans le langage BPEL, tout est service : un processus exécutable est l'implantation d'un service qui s'appuie sur d'autres services. Lorsqu'une ressource, telle qu'une base de données, un fichier, ou une application patrimoniale, est utilisée par un service, il est nécessaire d'exposer le SGBD, le système de gestion de fichiers, ou l'application comme des services. Pour pouvoir utiliser BPEL dans des environnements où toutes les ressources ne sont pas forcément exposées comme des services, il est quelques fois judicieux de casser le paradigme « tout service » de BPEL. C'est dans cet esprit qu'a été définie

⁷Pour obtenir la liste des outils mentionnés ici, voir <http://en.wikipedia.org/wiki/BPEL>

BPELJ [Blow et al. 2004], une extension de BPEL avec laquelle il est possible d'intégrer du code Java dans un programme BPEL. Cette approche est similaire à celle des JSPs (*Java Server Pages*) où du code Java est inclus dans des programmes HTML. La considération de BPEL dans les deux plates-formes .Net et Java laisse à penser que d'autres dialectes proches de BPEL vont émerger.

Ci-dessous, nous fournissons un squelette du processus exécutable BPEL correspondant au service « fournisseur » modélisé dans la figure 4.2. Ce squelette montre comment les opérateurs de flot de contrôle **sequence**, **switch** et **flow** sont combinés pour exprimer l'ordre dans lequel les messages sont échangés entre le fournisseur d'une part, et le client et l'entrepôt d'autre part. En particulier, les actions **envoyerBordereauExpédition** et **recevoirOrdreDeTransfert** peuvent s'exécuter en parallèle ou dans n'importe quel ordre puisqu'elles sont emboîtées dans une activité de type **flow**. Le squelette montre aussi comment BPEL est lié avec WSDL : les types et les opérations définis en WSDL dans la section 4.3.1 sont utilisés ici pour définir les types des variables et les types des messages envoyés et reçus par les actions **receive**, **reply** et **invoke**. De plus, les expéditeurs et destinataires des messages envoyés sont spécifiés par le biais des liens de communication (« partner links ») qui devront être associés aux services décrits dans la description WSDL.

```
<process name="ProcessusFournisseur">
  <partnerLinks>
    <partnerLink name="client" ... />
    <partnerLink name="entrepôt" ... />
  </partnerLinks>

  <variables>
    <variable name="BdC" type="xsld:Commande"/>
    <variable name="réponseBDC" messageType= "string" />
    <variable name="disponibilité" type="string"/>
  </variables>

  <sequence>
    <receive name="recevoirBdC"
      partnerLink="client" portType="pt_Commande"
      operation="op_Commande" variable="BdC"
      createInstance="yes"/>
    <invoke name="demanderDisponibilité"
      partnerLink="entrepôt"
      ...
      inputVariable="BdC"
      outputVariable="disponibilité"/>

    <switch>
      <case ....> <!-- cas disponible -->
        <!-- Initialiser la variable réponseBdC avec réponse positive -->
        <reply name="RépondreBdC"
          partnerLink="client"
          portType="pt_Commande"
          operation="op_Commande"
          inputVariable="réponseBDC"/>
    </switch>
    <flow>
      <invoke name="envoyerBordereauExpédition" .../>
      <receive name="recevoirOrdreDeTransfert" .../>
    </flow>
  </sequence>
</process>
```

```

</flow>
</case>
<case ...> <!-- cas indisponible -->
    <!-- Initialiser la variable réponseBdC avec réponse négative -->
    ...
</case>
</switch>
</sequence>
</process>

```

Bien que BPEL offre des constructions spécifiques pour le développement de services Web, sa complexité pose problème. Comme le montre [Wohed et al. 2003] l'opérateur `link` est dans une certaine mesure redondant avec les opérateurs `switch` et `flow` dans le sens que tout processus BPEL écrit en utilisant `switch` et `flow` peut être réécrit en utilisant un seul `flow` et un certain nombre de `links`. De plus, BPEL manque d'abstractions de haut niveau lorsqu'il s'agit de développer des services qui mettent en jeu des `multicasts` avec des conditions de synchronisation partielles, comme par exemple dans le cas d'un service « client » qui requiert des devis de plusieurs services « fournisseurs » [Barros et al. 2005a]. On peut s'attendre à ce que dans le futur, des efforts de recherche portent sur la résolution de ces problèmes et proposent soit des extensions de BPEL, soit des langages alternatifs.

4.5 Perspectives

4.5.1 Fouille d'interface, test de conformité et adaptation

Dans les architectures à base de services, nous l'avons déjà vu, la description structurale et comportementale des services s'appuie sur la notion d'interface, il est donc d'attendu que les services respectent leur interface. Ces dernières peuvent ainsi être perçues comme un contrat entre les services. Cependant, les services sont indépendants les uns des autres et ne peuvent exercer aucun contrôle les uns sur les autres. Pour ces raisons, un service ne peut jamais être sûr que les autres se comportent comme convenu dans leur interface. Tout ce dont un service peut être sûr au sujet des autres services avec lesquels il interagit s'appuie sur l'ensemble des messages qu'il envoie et de ceux qu'il reçoit. L'existence de traces des événements qui se sont passés ainsi que la donnée des interfaces permettent de poser la question de la confrontation de « ce qu'il s'est passé » à « ce qui était prévu qu'il se passe ». Cette question peut être étudiée selon deux points de vue. Le premier consiste à prendre les interfaces des services comme étant la référence, puisqu'elles spécifient la manière dont les services partenaires doivent se comporter. La question est alors de savoir si les événements enregistrés dans la trace sont cohérents avec les interfaces. Par exemple, la trace peut contenir une séquence d'événements impossible selon les interfaces, ce qui traduit une violation du contrat. Le second point de vue est inverse : la trace des événements est supposée correcte car elle reflète ce qui se passe réellement. Dans ce cas, se pose le problème de savoir si les interfaces ne sont plus valides et en conséquence si elles doivent être modifiées. Il est alors extrêmement utile de pouvoir dériver les nouvelles interfaces des traces. Lorsque le problème est étudié selon la dimension structurelle des interfaces, cela revient à comparer un ensemble de messages en XML à un schéma exprimé dans un langage tel que XMLSchema [W3C-XMLSchemal]

(voir aussi [Bertino et al. 2004], un état de l'art sur ce sujet). Moins d'attention a été consacrée à la résolution des problèmes de test de conformité et de fouille d'interfaces en prenant le point de vue de la dimension comportementale des interfaces. Il est possible d'envisager la résolution de ces problèmes en appliquant des méthodes de fouille de processus [Aalst et al. 2003] ou de tests de conformité de processus [Rozinat and Aalst 2005]. Quelques investigations préliminaires sur le problème de la fouille d'interfaces comportementales sont rapportées dans [Dustdar et al. 2004, Gaaloul et al. 2004].

L'adaptation des interfaces est un autre problème saillant, complémentaire à celui de la conformité. Lorsqu'il s'avère, que ce soit *a priori* par une comparaison, ou *a posteriori* par un test de conformité, que l'interface fournie d'un service ne correspond pas à l'interface que ses partenaires requièrent, il y a deux solutions : (1) adopter l'interface requise comme interface fournie et modifier le service en conséquence ; (2) introduire un adaptateur qui réconcilie l'interface fournie avec celle requise par les partenaires. La première solution est en général mauvaise car le même service peut interagir avec plusieurs autres services partenaires qui considèrent son interface originale. Ce qui conduit à la situation où le même service peut participer à diverses collaborations qui nécessitent différentes interfaces fournies. La seconde solution pallie ce défaut par la fourniture d'autant d'adaptateurs que d'interfaces requises. Comme nous l'avons vu dans la section 4.3, une interface peut être décrite selon la dimension structurelle, comportementale ou encore non-fonctionnelle. Ainsi l'adaptation doit être étudiée selon chacune de ces dimensions. Le problème de l'adaptation structurelle se ramène essentiellement à celui de la réconciliation entre des types de messages. Il existe sur ce sujet de très nombreuses études et plusieurs systèmes sont commercialisés (par exemple Microsoft's BizTalk Mapper). A l'inverse, le problème de l'adaptation selon la dimension comportementale est en cours d'étude [Benatallah et al. 2005a, Altenhofen et al. 2005, Fauvet and Ait-Bachir 2006]. L'adaptation des règles d'usage (par exemple, la réconciliation de différentes politiques de sécurité) est par contre l'objet de peu de recherche. La technologie des services Web gagnant en maturité et tendant à être utilisée dans le cadre de l'intégration de projets à grande échelle, l'adaptation des règles d'usage devrait très vite devenir un sujet important et crucial.

4.5.2 Transactions

Certains services web, en particulier dans le domaine du commerce électronique, ont des propriétés transactionnelles inhérentes [Baïna et al. 2004]. Ceci est le cas notamment des services associés à la gestion de ressources (au sens large), comme par exemple la réservation de chambres d'hôtel, de places de spectacle, de services professionnels, etc. En principe, les propriétés transactionnelles de ces services peuvent être exploitées lors de leur composition pour répondre à des contraintes et des préférences établies par le concepteur et l'utilisateur final. Aujourd'hui cependant, les langages et outils disponibles permettant de programmer des transactions sur des services Web ne fournissent pas de concepts de haut niveau pour : (i) exprimer les propriétés transactionnelles désirées au niveau du service composé ; (ii) assurer ces propriétés de façon automatisée en exploitant les propriétés transactionnelles des services composants.

L'exécution de services composés avec propriétés transactionnelles s'appuie sur l'exécution de transactions distribuées, complexes, souvent de longue durée, qui éventuellement peuvent mettre en œuvre des mécanismes de compensation (une opération

de compensation est une opération dont l'objectif est d'annuler les effets d'une autre opération qui n'a pas pu être terminée avec succès). De nombreux modèles de transactions ont été proposés dans le domaine des bases de données, des systèmes distribués et des environnements coopératifs (voir par exemple [Elmagarmid 1990, Gray and Reuter 1993, Alonso et al. 2003, Papazoglou 2003]).

Il est bien connu que les approches traditionnelles pour assurer les propriétés ACID d'une transaction (*Atomicity, Consistency, Isolation, Durability*) ne sont typiquement pas adéquates pour les transactions de longue durée telles que celles rencontrées dans le domaine des services web, puisqu'il n'est pas acceptable de verrouiller des ressources dans une transaction qui s'exécute sur une durée prolongée. De plus, le protocole de validation à deux phases, couramment utilisé dans les systèmes distribués, n'est pas applicable aux services composites car, dans ce protocole, il est fait l'hypothèse que tous les partenaires de la transaction supportent les opérations de préparation et de validation indispensables à sa mise en œuvre ; ce qui n'est pas toujours le cas dans le cadre des services web. Dans ce contexte, il peut être approprié de relaxer les contraintes d'atomicité *tout-ou-rien*. A cela s'ajoutent des problèmes d'intégration, puisque chaque service composant s'appuie sur un système de gestion de transactions choisi ou conçu pour le composant, considéré individuellement. Lorsqu'un service est intégré comme composant, il est fortement probable que son système de gestion de transactions ne réponde pas aux besoins de la composition vue comme un tout. Ce dernier aspect est la motivation principale pour l'émergence de protocoles tels que *WS-Coordination* [Cabrera et al. 2002a], *WS-AtomicTransaction*⁸ [Cabrera et al. 2002b] et pour la conduite de travaux de recherche (voir par exemple [Arregui et al. 2000, Hagen and Alonso 2000, Vidyasankar and Vossen 2003, Limthanmaphon and Zhang 2004, Fauvet et al. 2005]).

4.5.3 Sélection dynamique

Il est courant que plusieurs services offrent les mêmes fonctionnalités (la même capacité de service). Le concept de « Communauté de services » a été proposé dans la perspective de composer un nombre potentiellement grand de services Web et ce, de manière dynamique [Benatallah et al. 2005b]. Une communauté décrit les capacités de services sans faire référence au fournisseur du service Web. Ainsi, une composition s'appuie sur des communautés et non plus sur des services directement. Au moment de l'exécution, la sélection d'un service parmi ceux disponibles est opérée par la communauté correspondante. Pour que les services deviennent disponibles au travers d'une communauté, ils doivent s'enregistrer auprès d'elle. Les services peuvent rejoindre et quitter une communauté à n'importe quel moment. Une communauté possède une interface au travers de laquelle sont accessibles les opérations qu'elle offre. Un service Web peut s'enregistrer auprès d'une ou de plusieurs communautés. Et une communauté peut s'enregistrer auprès d'une autre communauté.

Le choix d'un service entrant dans une composition peut ainsi être effectué dynamiquement, au moment de l'exécution du service composé, de sorte que l'ensemble des partenaires intervenant dans la composition n'est pas connu a priori. Cette possibilité n'est pas sans impact sur les problèmes soulevés dans les deux sections ci-dessus (voir sections 4.5.1 et 4.5.2) : par exemple, un service peut choisir d'interagir avec un autre par le biais de la

⁸<http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnWebsrv/html/wsacoord.asp>

sélection dynamique, au moment de l'exécution, en fonction de critères de coût, de qualité de services, de règles d'usages, ou encore en fonction des propriétés transactionnelles exposées (pour augmenter les chances de terminer la transaction avec succès par exemple).

Chapitre 5

La plate-forme J2EE

Ce chapitre se propose de fournir une vue synthétique des grands principes qui gouvernent la mise en œuvre de l'environnement J2EE [J2EE 2005]. Cet environnement, proposé dans le contexte de Java par Sun Microsystem, offre un support au développement, au déploiement, ainsi qu'à l'exécution d'applications s'exécutant en mode serveur, comme par exemple des applications Web ou des applications réparties offrant des prises de service.

Le chapitre commence par une introduction de l'environnement J2EE, suivie par une présentation des principes de construction et d'assemblage d'application J2EE à base de composants. Les trois sections qui suivent introduisent les principales fonctions techniques offerte par l'environnement, permettant de garantir un certain nombre de propriétés nécessaires aux applications d'entreprise visées, ayant souvent un caractère critique. Deux sections sont ensuite consacrées aux deux principaux modèles de programmation de composants applicatifs, à savoir d'une part les composants de présentation Web permettant de gérer la logique d'interaction avec les utilisateurs munis d'un navigateur, et d'autre part les composants portant la logique dite métier. Ces derniers incluent notamment les composants représentant le modèle d'informations métier, généralement projeté dans une base de données relationnelle. La dernière section conclut sur la synthèse présentée et dessine quelques perspectives sur les évolutions en cours de cet environnement.

5.1 Introduction

La synthèse proposée par ce chapitre s'attache à étudier les principes structurants qui gouvernent la mise en œuvre de J2EE. La vue qui est donnée de l'environnement J2EE ne se veut pas exhaustive. Certains aspects couverts par le standard J2EE sont de simples déclinaisons Java d'autres standards plus généraux. C'est le cas par exemple de la couverture des standards de Corba de l'OMG autour de la gestion d'objets répartis, des standards du W3C autour de la gestion de documents XML ou encore du support des services Web. Ces derniers sont présentés dans le chapitre 4.

5.1.1 Historique

L'environnement Java pour l'entreprise a commencé à émerger assez rapidement après les débuts de Java au milieu des années 90. A son origine, Java était destiné aux environnements contraints (par exemple des petits équipements électroniques). Il a en fait percé dans l'environnement du Web, notamment dans les navigateurs pour le support d'interfaces graphiques riches (notion d'appliquette Java, en anglais *applet*). Les premières déclinaisons de Java dans l'environnement des serveurs sont apparues en 1997 avec les *servlets*, dont l'objectif est la construction programmatique de pages Web, puis avec les *Enterprise Java Beans* dont l'objectif est le support de code métier nécessitant un contexte d'exécution transactionnel (cf. [Gray and Reuter 1993]).

Après ces premiers pas et un relatif succès de ces technologies, Sun a structuré l'offre technique autour des serveurs d'application Java à travers le standard J2EE. L'objectif de ce dernier est de fédérer dans un cadre cohérent toutes les technologies nécessaires à la mise en oeuvre des applications de l'entreprise (applications orientées « serveur »). La première version des spécifications de J2EE est publiée en 1999. Suivront alors la version 1.3 de J2EE en 2001, puis la version 1.4 en 2003 incluant un support complet des standards XML et le support des services Web.

Java est désormais bien installé dans l'écosystème des applications d'entreprise et est devenu le principal concurrent de l'environnement .NET de Microsoft dont le chapitre 6 donne un aperçu. Le langage lui-même a acquis de la perennité puisqu'on compte désormais plus de 4 millions de développeurs Java dans le monde.

5.1.2 J2EE, pour quoi faire ?

J2EE a été conçu comme un environnement pour développer, déployer et exécuter des applications réparties pour le monde de l'entreprise. Ce contexte de l'entreprise se caractérise généralement par la nécessité d'assurer des niveaux de qualité de service tels que la sûreté de fonctionnement, la résistance à des charges d'exécution importantes ou encore la sécurité. Nous voyons dans la suite comment ces différents aspects sont pris en charge.

J2EE est intimement lié au langage Java et à l'environnement d'exécution standard J2SE [J2SE 2005] qui l'accompagne. En fait, J2EE est construit comme une agrégation cohérente de fonctions spécifiées dans d'autres normes répondant à différents besoins des applications d'entreprise. Toutes ces normes se matérialisent sous la forme d'API permettant d'utiliser lesdites fonctions, mais aussi de systèmes de description des applications construites pour cet environnement ainsi que de format de paquets utiles au déploiement.

Parmi les principales forces de l'environnement J2EE, deux nous paraissent primordiales :

- Le spectre fonctionnel couvert. Le développeur dispose probablement d'une très large palette de fonctions pour construire des applications d'entreprise, l'un des objectifs étant de garantir un niveau élevé de productivité. C'est notamment le cas de la connectivité, pour laquelle J2EE offre des mécanismes tels que l'appel de procédure à distance (en anglais *Remote Procedure Call*), les systèmes de communication asynchrone (en anglais *Message-Oriented Middleware*), l'accès aux bases de données, ou encore des protocoles divers. Cela positionne l'environnement comme une base logi-

cielle pertinente pour l'intégration d'applications (en anglais *Enterprise Application Integration*).

- Une offre produit importante. De grands éditeurs (BEA, IBM, Oracle, Sun, Macromedia, etc) ont bâti leur offre sur ce socle intergiciel. Par ailleurs, le monde de l'*open source* participe lui aussi de l'abondance de l'offre à travers des produits comme JBoss [JBoss Group 2003], JOnAS [The Objectweb Consortium 2000] et Geronimo [The Apache Software Foundation 2006]. C'est d'autant plus rassurant pour les utilisateurs qui investissent dans la technologie que la portabilité des applications est réellement avérée, si l'utilisateur reste dans le cadre des fonctions standard évidemment. Sun met à disposition pour cela des outils de vérification de la conformité d'applications au standard J2EE [Sun 2005].

La richesse et la puissance de l'environnement J2EE en font aussi un défaut car la phase d'apprentissage reste lourde si l'objectif est la connaissance de l'ensemble. Il est aussi nécessaire de bien maîtriser les concepts sous-jacents à J2EE (notamment les principes des transactions) pour l'utiliser efficacement.

5.1.3 Principes d'architecture

La principale cible fonctionnelle de l'environnement J2EE est l'application Web mettant en œuvre des pages dynamiques, c'est à dire des pages calculées dont le contenu dépend du contexte (par exemple l'utilisateur). Outre l'ensemble des fonctions nécessaires à la mise en œuvre de telles applications, J2EE définit un cadre architectural, dit *multiétage* (en anglais *multitier*), permettant d'organiser leur code.

Ce cadre architectural se base sur la notion de composant et d'assemblage de composants. Les composants sont des bibliothèques de code Java qui décrivent les services qu'ils fournissent, les contrats qu'ils respectent (comportement transactionnel, persistance, etc), ainsi que leurs dépendances vis-à-vis d'autres composants ou des éléments de l'environnement d'exécution (par exemple le système transactionnel). Les assemblages sont eux-mêmes décrits avec les composants à travers la définition des liaisons entre composants. Ils sont matérialisés par des paquetages qui contiennent à la fois le code des composants ainsi que toutes les informations descriptives sur ces composants et leur assemblage. Les contrats associés aux composants sont pris en charge par la notion de conteneur lors du déploiement et de l'exécution des composants. Un conteneur correspond à la notion de canevas telle que présentée dans la section 2.2.1.

La vue architecturale proposée correspond à une décomposition de la chaîne d'exécution d'application du terminal jusqu'aux différents serveurs intervenant dans l'exécution d'une application. Les éléments proposés par cette décomposition sont décrits ci-après.

L'étage client

L'étage client est représenté par le terminal et prend en charge l'interaction avec l'utilisateur. Du point de cette interaction, deux architectures sont possibles :

- Architecture client léger : dans ce cas, le terminal embarque un mécanisme qui permet d'interpréter les informations de présentation et d'interaction avec l'utilisateur produite par le serveur. Cela peut être un navigateur qui interprète des pages XML (XHTML, WML, ou VoiceXML suivant la modalité utilisée) produite par le serveur

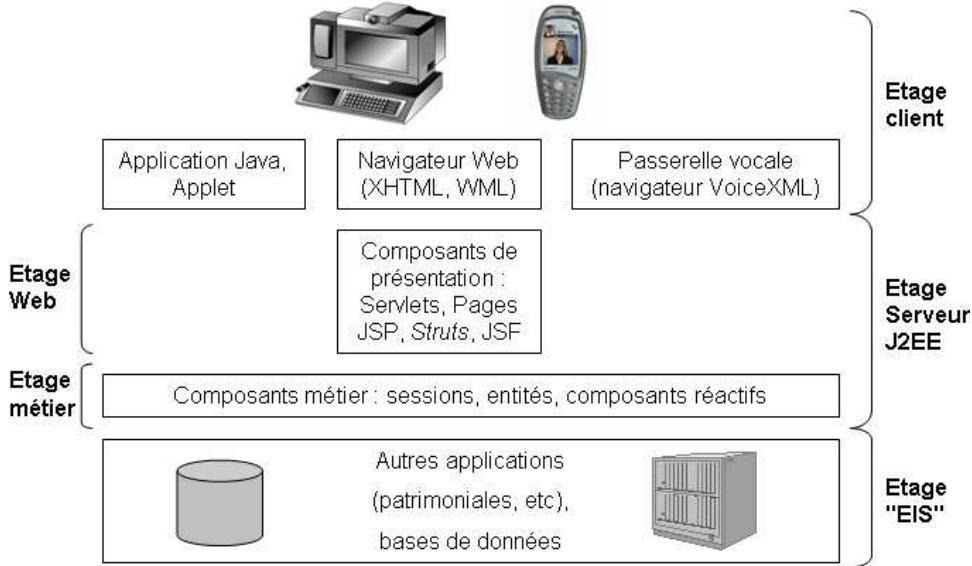


Figure 5.1 – Architecture multiétages dans l'environnement J2EE

J2EE. La communication avec le serveur passe généralement par le protocole HTTP dans le cas du Web ou des interfaces vocales, sachant que dans ce dernier cas, le navigateur VoiceXML est lui-même exécuté par un serveur vocal connecté au terminal par un canal voix classique (RTC, GSM, etc). Dans le cas du WAP (pages WML), le protocole utilisé est WTP qui ne fait pas partie des protocoles supportés en standard par l'environnement J2EE.

- Architecture client riche (ou client lourd) : dans ce cas, le terminal exécute une application Java utilisant généralement des couches graphiques évoluées de J2SE ou d'autres (AWT, Swing, SWT, etc). La communication avec la partie serveur peut passer ici par les différents moyens disponibles dans le cadre J2EE : RMI, Web Services, voire JMS. Une telle application peut d'ailleurs elle-même être exécutée sous la forme d'une *applet* dans le contexte d'un navigateur supportant une JVM. Cela peut en simplifier le déploiement même si cela pose des problèmes de compatibilité de version de JVM ou encore de gestion de politique de sécurité. D'autres approches technologiques émergent actuellement comme par exemple AJAX qui s'appuie sur la machine Javascript dont le support est généralisé dans les navigateurs Web et sur des interactions à base de services Web entre le client riche et le code serveur.

Le choix entre ces deux architectures est un problème de compromis entre la facilité de déploiement, la complexité de gestion de versions de logiciels avancés au niveau du terminal d'un côté, et l'expérience ergonomique perçue par l'utilisateur d'autre part, notamment pour les interfaces graphiques. Dans la seconde architecture, l'environnement J2EE fournit un support d'exécution pour aider la partie cliente à se lier facilement avec la partie serveur.

L'étage serveur

L'étage serveur exécute le code applicatif J2EE pour le compte de plusieurs utilisateurs simultanément. Ce même code est à son tour décomposé en deux étages distincts :

- L'étage présentation : cet étage exécute le code des interfaces utilisateur de type client léger. Il produit généralement des pages XML qui sont retournées à un navigateur représentant l'étage client qui se charge alors de les interpréter pour construire le rendu attendu.
- L'étage métier : cet étage correspond au code définissant le *métier* ou la fonction même de l'application. L'environnement J2EE propose plusieurs types de composants pour mettre en œuvre cet étage. Les composants de session gère le code métier représentant une session d'interaction entre un utilisateur et l'application, ces sessions pouvant ou non contenir des données propres. Les composants de données encapsulent l'accès aux bases de données relationnelles. Enfin les composants réactifs permettent au code métier de réagir sur l'arrivée d'événements provenant d'applications externes ou d'événements internes à l'application (programmation asynchrone dans le cadre d'une application J2EE).

L'étage information

L'étage information (étage EIS dans la figure 5.1) se compose des systèmes qui fournissent les informations de l'entreprise nécessaires aux applications. Ces informations peuvent être fournies par d'autres applications patrimoniales, ou directement par des bases de données.

Ces principes d'architecture ont un impact sur l'organisation des différentes phases du cycle de vie d'une application J2EE : sur la phase de développement avec les principes d'organisation du code mais aussi sur les phases de déploiement et d'exécution.

5.1.4 Modèles de programmation et conteneurs

Un des grands intérêts de l'environnement J2EE est qu'il fournit des modèles de programmation pour le développeur, simplifiant au maximum l'utilisation des services proposés en permettant d'exprimer des contrats déclaratifs associés aux composants. En effet, certains de ces services peuvent être compliqués à mettre en œuvre. C'est le cas pour la gestion de la sécurité, pour la gestion de données de sessions Web, pour la gestion des transactions, ou encore la persistance des objets métier dans les bases de données. L'objectif est donc de rendre l'utilisation de ces services la plus transparente possible au programmeur.

C'est le rôle dévolu aux conteneurs qui s'interposent entre les composants applicatifs et les services qu'ils utilisent. Cette interposition utilise généralement le patron d'architecture d'interception tel que décrit en 2.3.5. Dans ce cas, le programmeur manipule explicitement les intercepteurs qui implantent la même interface que l'objet intercepté plutôt que cet objet. Cela signifie notamment que dans le code de la classe de l'objet intercepté, le programmeur doit toujours passer par l'intercepteur s'il veut que les contrats demandés soit pris en compte.

L'autre rôle pris en charge par les conteneurs concerne le déploiement des composants qu'ils gèrent. De ce point de vue, le conteneur fournit aux composants le moyen de se lier

aux autres composants ou aux ressources dont ils ont besoin pour leur exécution.

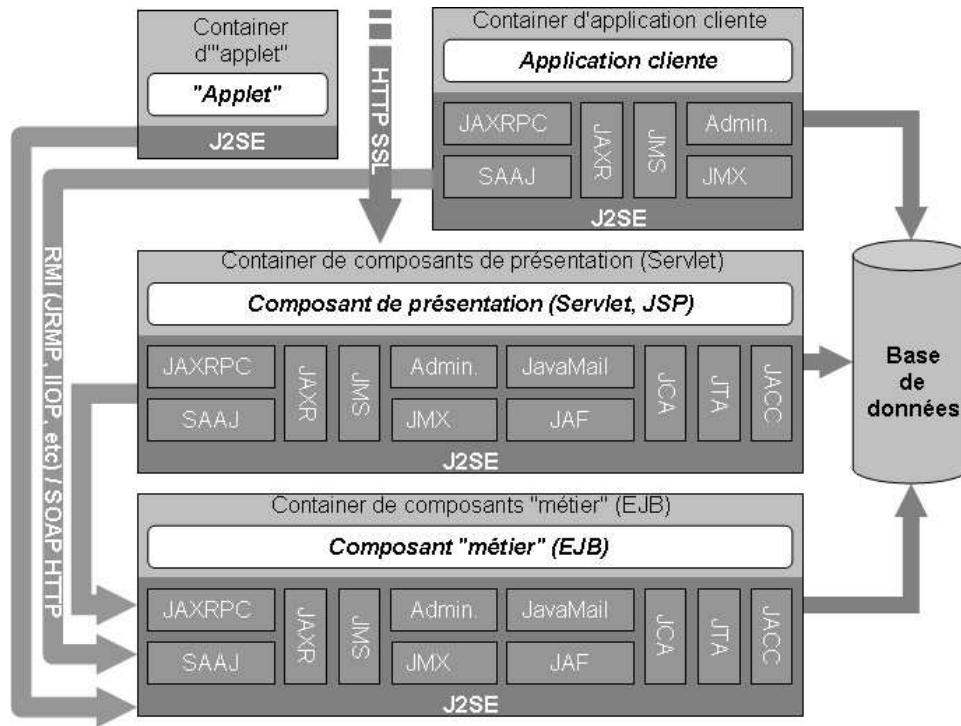


Figure 5.2 – Conteneurs J2EE

Quatre types de conteneurs sont proposés par l'environnement J2EE. Deux sont liés à l'étage client et deux autres à l'étage serveur J2EE. Pour l'étage client, il s'agit du conteneur d'application cliente J2EE et du conteneur d'*applet*, ce dernier ne proposant l'accès à aucune des fonctions de l'environnement J2EE. Côté serveur, il y a le conteneur de composants de présentation / servlets (conteneur Web) et le conteneur de composants métier (conteneur EJB).

Comme le montre la figure 5.2, les conteneurs (sauf le conteneur d'*applet*) donnent accès aux services de l'environnement J2EE. Parmi ces services, certains peuvent être appelés depuis n'importe quel étage client ou serveur. Ces services comprennent :

- La pile *Web Service* : elle est composée de trois parties. La partie basse (SAAJ) est le canevas SOAP pour Java avec le support des attachements pour le transport de données binaires. Il permet d'adapter SOAP à différents protocoles de transport des appels. La partie haute (JAX-RPC) implante la fonction de RPC au-dessus de SOAP. Enfin, la troisième partie (JAXR) donne accès aux annuaires de *Web Services* (par exemple un annuaire UDDI).
- Le support de communication asynchrone : les communications asynchrones sont un élément important pour l'intégration d'applications. La spécification JMS définit la manière de gérer des files de messages, que ce soit à travers des queues garantissant l'acheminement d'un message d'un producteur à un consommateur de messages, ou à travers des sujets (*topic*) permettant la publication et la réception anonyme de messages (tous les consommateurs reçoivent tous les messages). Elle définit aussi

différents niveaux de qualité de service concernant la délivrance de messages (par exemple aucune perte).

- Le support d'administration : il permet à travers la spécification JMX de définir des composants administrables. Cette fonction d'instrumentation est ainsi disponible aussi bien au niveau des composants de la plate-forme J2EE (les services disponibles) que des composants applicatifs. Un agent d'administration permet généralement d'agrégner l'accès à ces composants administrables pour les outils d'administration (par exemple une console d'administration graphique).

L'autre groupe contient les services qui ne sont accessibles qu'au code des composants s'exécutant dans le serveur J2EE. Ces services comprennent :

- Le support des transactions : il permet de contrôler la démarcation des transactions, c'est à dire leur démarrage (`begin`) et leur terminaison (`rollback` ou `commit`). La spécification JTA offre ainsi le moyen de garantir l'atomicité d'un ensemble d'actions sur des systèmes gérant des données critiques (comme par exemple des bases de données ou des files de messages).
- Le support des connecteurs J2EE : il permet de se connecter à des systèmes externes gérant des ressources qui peuvent être critiques. Il supporte deux modes d'interactions : les interactions à l'initiative du serveur J2EE ou celles à l'initiative du système externe. Il définit des contrats pour les connecteurs qui permettent à la plate-forme J2EE de gérer les ressources de connexion, de contrôler l'accès à ces ressources ou encore de gérer ces ressources en mode transactionnel.
- Le support d'envoi de messages électroniques : il permet d'envoyer des *e-mail* à partir d'une application J2EE, offrant ainsi un moyen de notifier des évènements à des utilisateurs externes. La spécification JavaMail offre donc les APIs pour effectuer ce type d'action et fournit un moyen de spécialiser la manière de traiter ces envois (interface fournisseur / SPI).
- Le support d'autorisation pour les conteneurs : il permet d'ajouter au niveau des conteneurs des modules définissant des politiques de sécurité, concernant pour l'essentiel l'autorisation d'accès. La spécification JACC définit la manière d'installer, de configurer et d'utiliser (vérification de droits d'accès) un module fournissant cette politique d'accès.

La plupart de ces fonctions sont décrites plus en détail et illustrées dans la suite du chapitre.

5.1.5 Les acteurs dans l'environnement J2EE : organisation des rôles

L'environnement J2EE prend en charge une grande partie du cycle de vie d'une application. Il propose des solutions pour le développement de l'application, pour son assemblage puisqu'il se base sur la notion de composants, pour son déploiement, et enfin pour son exécution et son administration. A un premier niveau, nous distinguons deux types d'acteurs : les fournisseurs de technologies et leurs utilisateurs. Voici une liste des fournisseurs ainsi que leur rôle :

- Le fournisseur de la plate-forme serveur : il fournit la plate-forme logicielle implantant les différents services proposés par l'environnement J2EE. Cette plate-forme implante aussi les mécanismes permettant de déployer et d'exécuter les composants

(cf. conteneurs). Elle implante aussi les interfaces nécessaires à l'administration de la plate-forme elle-même, sachant qu'elles sont standardisées.

- Le fournisseur d'outils de développement : il fournit généralement des outils permettant de faciliter le développement et l'empaquetage des applications. Concernant la partie développement, on retrouve des outils de mise au point mais aussi des outils de profilage pour les performances. Des outils d'aide à l'empaquetage (fabrication des paquetages introduits précédemment) sont aussi fournis car le paquetage est un passage obligé pour déployer le code sur des plates-formes J2EE, voire pour les phases de mise au point.
- Le fournisseur d'outils d'administration : lors de la mise en exploitation d'une application J2EE, il faut disposer d'outils permettant de l'administrer en même temps que la plate-forme J2EE elle-même. Même si certaines parties sont intimement liées à la plate-forme, on trouve des outils de ce type. C'est le cas d'agents d'agrégation de fonctions ou d'informations d'administration, ou encore des consoles d'administration. Ces outils sont souvent fortement adhérents aux plates-formes qu'ils ciblent car une bonne partie des fonctions à gérer sont spécifiques. C'est le cas par exemple des capacités de *clustering* qu'on retrouve dans la plupart des plates-formes mais qui sont indépendantes du modèle de programmation J2EE.

Certains grands éditeurs comme BEA, IBM, ou Sun cumulent les trois rôles mais il est néanmoins possible de trouver des fournisseurs positionnés sur un seul, notamment dans le monde *open source* où les acteurs sont plus spécialisés.

Du côté des utilisateurs, même si nous pourrions détailler beaucoup plus les rôles, les acteurs qui nous semblent les plus pertinents vis-à-vis de J2EE sont les suivants :

- Le développeur de composants : il crée et valide des composants applicatifs. Il s'appuie pour cela sur les APIs des différents services proposés par l'environnement J2EE (cf. section suivante). Il utilise les outils d'aide au développement et à la mise au point tels que présentés précédemment.
- L'assembleur d'application ou de composants : il décrit l'ensemble des composants qui composent une application ou une partie d'application (il peut s'agir d'un assemblage partiel), ainsi que les liens qui les unissent. Ces assemblages sont ensuite empaquetés afin d'être pris en charge par les conteneurs adéquats suivant le type des composants qu'ils contiennent (par exemple paquetages de composants Web ou paquetages de composants métiers ou les deux).
- L'administrateur d'application : c'est lui qui déploie l'application. Il définit pour cela les liaisons effectives des composants vers les ressources qu'ils utilisent (par exemple une base de données). Il utilise pour cela les outils d'administration tels que décrits précédemment pour effectuer ces différentes actions : déploiement, configuration, paramétrisation des services de la plate-forme (par exemple, dimensionnement des *pools* de ressources).

Il est clair que le développeur doit dans la plupart des cas maîtriser aussi les deux autres rôles. En effet, lorsqu'il met au point les composants qu'il développe, il doit les empaqueter pour pouvoir les déployer sur le plate-forme qui va lui servir à exécuter ses essais. De la même manière, s'il doit effectuer des essais de performance ou de montée en charge, il va devoir agir sur le paramétrage de la plate-forme, endossant ainsi le rôle dévolu à l'administrateur.

Cette définition des différents rôles fait ressortir l'ampleur des compétences qui doivent être mises en œuvre par le développeur J2EE. Il doit maîtriser la plate-forme, au minimum les outils de développement associés, les services de l'environnement J2EE ainsi que les bons principes d'architecture à mettre en œuvre. C'est un défi majeur en termes de formation nécessaire pour avoir un développeur efficace, même si dans de grosses structures ces développeurs peuvent être spécialisés, par exemple par étage.

5.1.6 Références utiles

Même s'il s'agit d'un outil puissant, J2EE comme tout environnement informatique qui se respecte n'a d'intérêt que s'il est correctement mis en œuvre. La manière d'architecturer des applications J2EE est un point primordial, bien traité dans des ouvrages sur les bonnes pratiques architecturales [Alur et al. 2003] ou sur les mauvaises [Dudney et al. 2003].

5.2 Approche à composants

L'environnement J2EE propose de construire des applications par parties qui sont ensuite assemblées pour former un tout cohérent et complet. Ce tout est ensuite empaqueté dans un format de paquet standard, qui peut être déployé sur n'importe quel serveur J2EE. Cette section décrit les principes de cette démarche de décomposition et d'assemblage.

5.2.1 Modèle de composants

Le modèle de composants proposé par J2EE est un modèle plat à deux niveaux : le niveau de base définit les composants J2EE et le niveau supérieur une application J2EE qui est un assemblage des composants du premier niveau.

Nous supposons comme acquis le principe de gestion de référence pour la mise en liaison d'entités logicielles à partir d'un service de nommage. Dans le cadre de J2EE, cet aspect est porté par la spécification JNDI. L'utilisation de contextes de nommage et de noms est courante tout au long du présent chapitre, et particulièrement dans cette section traitant de la définition et de l'assemblage de composants.

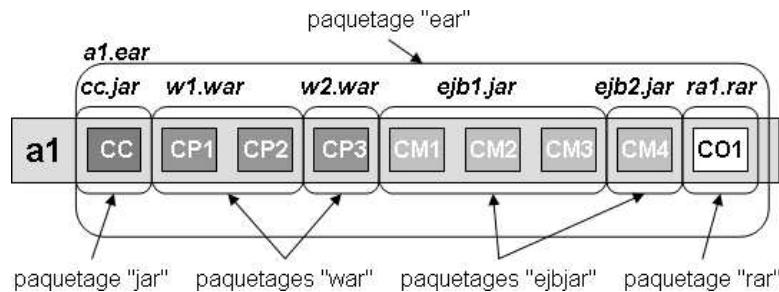


Figure 5.3 – Empaquetage de composants J2EE

Un composant J2EE est un ensemble logiciel défini par des classes, des interfaces et des informations descriptives. Comme dans les autres modèles de composants

[Szyperski and Pfister 1997], il se définit comme un élément logiciel proposant des interfaces de service (en général une seule dans le cas de J2EE) et explicitant ses dépendances vis-à-vis d'autres composants en exhibant ses interfaces requises. Il peut aussi définir un certain nombre de paramètres permettant d'adapter sa configuration à différents contextes de déploiement. Toutes ces informations sur le composant sont décrites dans un descripteur qui est utilisé lors de son assemblage au sein d'une application et lors de son déploiement.

Une application J2EE est donc un assemblage de composants qui sont empaquetés dans un paquetage qui lui est associé. Les deux processus d'assemblage et d'empaquetage sont néanmoins relativement indépendants. La figure 5.4 montre bien que les liaisons créées entre les composants à l'assemblage font fi des frontières définies entre les paquetages tels que définis dans la figure 5.3 ; un composant d'un paquetage peut très bien être relié à un composant d'un autre paquetage. Il reste que la politique d'empaquetage suit généralement une logique de pré-assemblage. Au minimum, une application J2EE distingue les paquetages par étage client, Web, métier, et pour chaque connecteur. Cela signifie qu'il y a toujours au minimum deux niveaux d'empaquetage : le niveau applicatif et le niveau d'empaquetage de composants par étage.

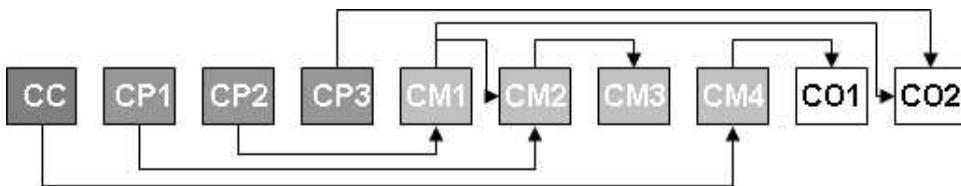


Figure 5.4 – Assemblage de composants J2EE

Les deux sections qui suivent décrivent comment s'organisent les deux processus d'empaquetage et d'assemblage d'application.

5.2.2 Empaquetage de composants et d'applications

Le paquetage est l'unité de déploiement dans l'environnement J2EE. Cet environnement présente deux niveaux d'empaquetage : le paquetage de l'application J2EE et les modules des différents étages empaquetés dans des sous-paquetages. Hormis lorsqu'une application contient un étage client, elle est déployée dans une plate-forme J2EE unique. L'environnement J2EE ne propose donc que du déploiement centralisé de composants assemblés statiquement au moment de ce déploiement.

Tous les paquetages J2EE sont bâtis comme des fichiers d'archive Java dont le suffixe est généralement `.jar`. Néanmoins, pour des raisons de lisibilité, J2EE distingue les suffixes des paquetages par étage comme le montre la figure 5.3. La principale différence entre ces paquetages est le fichier de description de leur contenu, aussi nommé descripteur de déploiement. La liste ci-dessous énumère les différents éléments déployables dans un environnement J2EE (applications et composants) accompagnés des noms types des descripteurs associés ainsi que de l'extension du fichier d'archive utilisée :

Application : `application.xml .ear`

Application cliente : `client.xml .jar`

Module Web : `web.xml .war`

Module métier : `ejb-jar.xml .jar`

Connecteur : `ra.xml .rar`

Dans tous les cas, il s'agit d'archives Java classiques contenant un répertoire `META-INF` avec un fichier `MANIFEST.MF` donnant des informations sur le paquetage et le descripteur de déploiement correspondant, par exemple le fichier `web.xml` pour un module Web. Ces archives contiennent aussi les répertoires où se trouvent les binaires des classes et des interfaces des composants définis dans le paquetage, ainsi que toute autre ressource nécessaire à l'exécution de ces composants (par exemple, des fichiers HTML ou JSP).

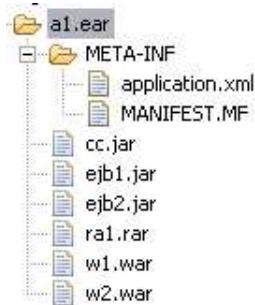


Figure 5.5 – Une archive d'application J2EE

Le descripteur XML qui suit est celui de l'application J2EE A1 de la figure 5.3. Il correspond au contenu du fichier `application.xml` du répertoire `META-INF` du paquetage `a1.ear`. La définition n'est pas complète. Elle montre la définition de deux des modules composant l'application A1. Le premier est le module Web défini dans `w1.war` et le second est le module métier défini dans `ejb1.jar`. Comme le montre la figure 5.5, les paquetages de ces modules doivent donc être stockés à la racine du fichier `a1.ear` pour pouvoir être pris en charge au déploiement.

```
<application>
    <display-name>A1</display-name>
    <description>Application description</description>
    ...
    <module>
        <web>
            <web-uri>w1.war</web-uri>
            <context-root>monsite_a1</context-root>
        </web>
    </module>
    ...
    <module>
        <ejb>ejb1.jar</ejb>
    </module>
    ...
</application>
```

Nous ne donnons pas une vue exhaustive de ces descripteurs de déploiement, la suite du chapitre montrant des éléments d'autres types de descripteur pour illustrer diverses constructions J2EE.

5.2.3 Principes d'assemblage

Le principe de base d'une approche à composants est que leur code soit aussi indépendant que possible du contexte dans lequel ils vont être déployés et exécutés, de façon à les rendre notamment le plus réutilisables possible. J2EE ne déroge pas à cette règle.

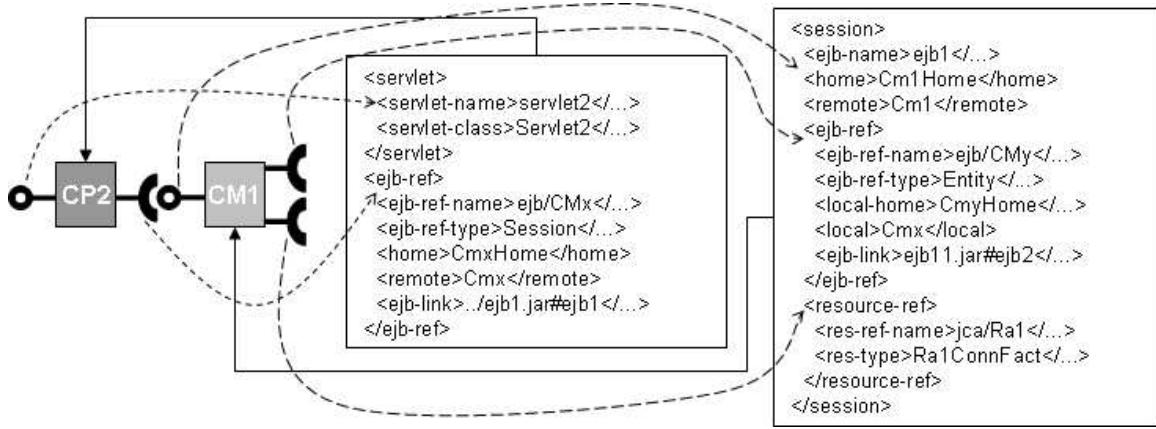


Figure 5.6 – Description d'un assemblage entre deux composants

Comme le montre la figure 5.6, l'assemblage entre les composants s'effectue d'abord au niveau de la description des composants. Cette description contient la définition des interfaces fournies et des interfaces requises (les dépendances vers d'autres composants).

En fait, la définition de ces composants est spécifique au type de composant. Par exemple, un *servlet* ne définit pas d'interface fournie puisqu'on la connaît exactement : c'est `javax.servlet.Servlet`. Un composant métier définit des interfaces métier fournies, comme le composant **CM1** avec l'interface `Cm1Home`, sachant que l'interface `Cm1` n'est là que pour signaler l'interface métier supportée par les instances de ce composant. La liaison avec un composant se fait la plupart du temps à travers une interface de type *usine* (cf. section 2.3.2). Dans un monde objet tel que celui de Java, il est naturel de constater qu'un composant logiciel gère des instances d'un même type. Par ailleurs, ce patron d'architecture est utilisé par le serveur d'application J2EE pour avoir le contrôle sur le cycle de vie des instances par interception des fonctions de l'usine. Cela lui permet d'opérer la gestion de ressources adéquate à l'exécution et d'optimiser par exemple cette gestion pour des raisons de montée en charge. Pour les composants auxquels on va pouvoir se lier, deux informations importantes sont définies dans le descripteur :

- Information de nommage : un nom est associé à chaque composant. Ce nom identifie de façon unique un composant dans le contexte du paquetage dans lequel il est défini. Ce nom est ensuite utilisé par les autres composants qui veulent y faire référence. Un exemple d'une telle définition est l'élément `ejb-name` utilisé comme nom du composant **CM1** dans la figure 5.6.
- Information de typage : une ou deux interfaces sont généralement associées à un composant. Une seule est vraiment importante : c'est celle qui permet de se lier au composant. C'est généralement une interface *usine* comme l'est l'interface `Ejb1Home` associée au composant **CM1** dans la figure 5.6. Cette interface est utilisée par la suite

pour vérifier la conformité d'un assemblage avant le déploiement d'une application, cet assemblage étant comme nous l'avons déjà signalé statique.

L'expression des dépendances est elle aussi spécifique au type de composant décrit. Dans la figure 5.6, nous pouvons observer que la dépendance vers le composant métier CM1 est défini en dehors de la définition du composant de présentation CP2. En effet, dans le cas des *servlets*, les dépendances exprimées sont communes à tous les composants de présentation définis dans un même paquetage. Dans le cas d'un composant métier, ces dépendances sont associées au composant lui-même. L'expression des dépendances est spécifiques au type de composant référencé. Malgré le manque d'homogénéité dans l'expression de principes communs, pour un type de référence donné, ces dépendances sont définies de la même manière quel que soit le type de composant dans lequel elles sont utilisées (par exemple, le référençage d'un composant métier s'exprime de la même manière dans un descripteur `web.xml` que dans un descripteur `ejb-jar.xml`). Les éléments importants exprimés pour ces dépendances sont les suivants :

- Le nom de la référence : c'est le nom qui va être utilisé dans le code du composant pour récupérer la liaison effective vers le composant référencé. Il est bien souvent préfixé par le type de composant ou de ressource auquel on se lie. La figure 5.6 montre deux types de références : deux références vers des composants métier dont le nom est préfixé par `ejb/` et une référence vers un connecteur préfixée par `jca/`.
- Le type du composant auquel on cherche à se lier : il permet de vérifier la validité des liaisons définies à l'assemblage (cas d'un lien vers un composant métier). Il suffit pour cela de vérifier que le type du composant référencé, par exemple `Cm1Home` de `CM1`, est bien conforme au type du composant attendu par le référentiel, ici `CmxHome` dans la référence de `CP2` (c'est à dire `Cm1Home` est `CmxHome` ou bien l'étend (relation d'héritage)).
- Le lien vers le composant utilisé dans le cas d'une référence vers un composant métier : l'élément `ejb-link` utilisé dans ce cas exprime une mise en liaison effective de deux composants. Il utilise le nom identifiant le composant dans le cadre de l'application en question. L'exemple de la figure 5.6 montre le lien du composant `CP2` vers le composant métier `CM1`. Ce lien est défini par `../ejb1.jar/ejb1` de façon à identifier complètement le composant dans le paquetage de l'application à partir du paquetage dans lequel la référence est définie (ici le paquetage `w1.war`).

Dans le cas des références vers d'autres ressources comme une queue JMS, une source de données JDBC, ou encore une usine à connection JCA, la définition de la liaison effective n'est pas couverte par le standard J2EE. Elle est donc spécifique à chaque produit. Elle est bien souvent définie dans un descripteur supplémentaire propre au produit.

Une fois que les liaisons ont été définies dans les descripteurs de déploiement, elles doivent être activées dans le code à l'exécution. Ceci est généralement fait à la création d'une instance d'un composant. Au déploiement d'un composant, l'environnement J2EE lui associe un contexte de nommage propre dans lequel sont définies toutes les liaisons vers les autres composants ou vers les ressources requises par celui-ci. Il est accessible par tous les composants sous un nom unique "`java:comp/env`". Les liaisons dans le code sont généralement mises en œuvre au moment de la création d'une instance. Prenons le cas du composant de présentation `CP2` pour lequel la méthode `init` est appelée par le conteneur

de `servlet` lors de la création de `servlet2` :

```
// Définition de la variable contenant le lien vers l'usine à instances
// d'un composant CMx qui dans le cas présent sera le composant CM1
private CmxHome usineCmx;

// Appelée à la création de l'instance du composant \emph{servlet}
public void init(ServletConfig config) throws ServletException {
    Context envContext = new InitialContext().lookup("java:comp/env");
    usineCmx = (CmxHome) PortableRemoteObject.narrow(
        envContext.lookup("ejb/CMx"),
        CmxHome);
    ...
}
```

Il est clair ici que le code du composant référençant est indépendant de celui du composant référencé. En effet, aucune information relative à CM1 n'est présente dans le code de CP2. Pour lier CP2 à un autre CMx, il suffit de changer le lien dans son descripteur de déploiement en spécifiant un autre composant : par exemple, on se lie à un ejb1b avec `<ejb-link>../ejb1b.jar/ejb1b</ejb-link>`.

Le mécanisme de mise en liaison dans le code est donc le même quel que soit le type de composant ou de ressource : il s'effectue par un `lookup` approprié sur le contexte correspondant à l'environnement propre au composant. L'exemple ci-dessus est compliqué par l'utilisation d'une fonction de conversion de type, nécessaire dans le cas où on se lie à un composant offrant une interface accessible à distance (interface `Remote`).

Il faut noter que ce mécanisme est en train d'évoluer quelque peu dans le cadre des nouvelles versions des spécifications, notamment dans le cadre d'EJB 3.0. En effet, l'affection de la variable de liaison à partir du `lookup` pourra être prise en charge par le conteneur. Il suffira pour cela d'annoter la variable ou un *accesseur* à celle-ci pour définir cette liaison (injection de la mise en liaison par le conteneur). L'utilisation des annotations est généralisée dans cette version des spécifications, ce qui fait que les informations de description d'un composant qui étaient auparavant présentes dans le descripteur de déploiement sont de retour dans son code, même si elles sont isolées du code proprement dit à travers ce mécanisme d'annotation. Cette nouvelle approche a ses avantages et ses inconvénients. Elle rend le code moins lisible à cause de la surcharge due aux annotations. Par contre, la description d'un composant est attachée à ce dernier plutôt que d'être agglomérée dans un descripteur général. Cela devrait faciliter l'émergence de bibliothèques de composants et de vrais outils d'assemblage s'appuyant sur celles-ci, donnant ainsi accès à un véritable environnement de réutilisation de code. Dans une telle démarche, on priviliege l'utilisation des annotations pour toutes les informations relatives au comportement du composant (propriétés déclaratives telles que les propriétés transactionnels ou les propriétés liés à la persistance), et l'utilisation des descripteurs pour toutes les informations d'assemblage.

5.2.4 Déploiement d'application

Après avoir vu comment une application est empaquetée et assemblée dans les sections précédentes, cette section décrit l'environnement proposé pour déployer une application

J2EE au niveau du serveur et au niveau des clients.

Un aspect important du cadre architectural défini par J2EE est que les étages proposés sont strictement ordonnés par rapport à la chaîne d'invocation applicative (du client vers les serveurs). Cet ordre définit une chaîne de dépendances fonctionnelles unidirectionnelles : par exemple, l'étage Web dépend de l'étage métier et jamais l'inverse.

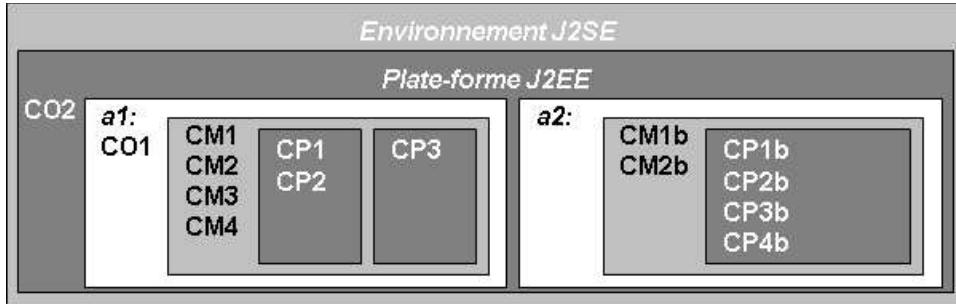


Figure 5.7 – Hiérarchie des chargeurs de classes dans un serveur J2EE

Ces principes d'organisation sont garantis par l'environnement J2EE à travers l'organisation d'une hiérarchie de chargeurs de classes Java. En effet, on a affaire à un jeu de poupées russes comme le montre la figure 5.7. On peut distinguer deux types de chargeurs.

Les premiers contiennent le code commun à toutes les applications déployées dans un serveur J2EE. Au niveau de ces chargeurs, on a généralement une hiérarchie distinguant le code de J2SE à la base, puis celui du serveur J2EE avec ses services de base, puis un dernier contenant le code des ressources mises à disposition des applications (dans l'exemple, le chargeur en question contient le code du composant C02).

Les autres chargeurs sont spécifiques à chaque application et sont organisés en deux niveaux :

- Le chargeur du code métier : tous les composants du code métier d'une application, quels que soient les paquetages dans lesquels ils ont été embarqués, sont chargés dans le même espace. Pour l'application A1, on voit dans la figure 5.7 que tous les composants métier (CM1, CM2, CM3, et CM4) ont leur code de l'espace d'un chargeur unique. A partir de ce chargeur, ceux-ci ont accès au code de tous les chargeurs partagés, donc à toutes les ressources J2EE, à la plate-forme J2EE, ainsi qu'au code de J2SE. Il n'ont en aucun cas accès au code de l'étage de présentation de l'application.
- Les chargeurs de code Web : concernant le code de présentation, un chargeur différent est créé pour chaque paquetage Web déployé. Ainsi, le code de CP1 et de CP2 sont embarqués dans le même chargeur, alors que celui de CP3 est dans un chargeur différent. Ces chargeurs ont accès au code de tous les autres étages, étage métier inclus. Le seul point notable est qu'il y a une isolation entre les différents composants de présentation qui ont été empaquetés indépendamment les uns des autres. Cette fonction ne paraît néanmoins pas apporter grand chose dans l'environnement, les différents éléments de la présentation étant généralement empaquetés ensemble (il peut être nécessaire de partager des informations concernant les transitions entre les différentes pages d'une présentation, ou encore des informations de session).

L'important avec ce modèle de chargement du code est que le code d'une application, principalement composé du code métier et du code de présentation, peut être chargé/déchargé indépendamment des autres éléments de l'environnement J2EE. Ceci est rendu possible par l'isolation du code d'une application dans un chargeur dédié comme le montre la figure 5.7 (il suffit alors de supprimer ce chargeur pour décharger l'application). C'est important notamment en phase de développement où il peut être lourd de relancer tout le serveur J2EE à chaque fois qu'on veut relancer l'application après une modification. De la même manière, on veut pouvoir descendre dans certains cas à un grain de recharge-ment encore plus fin, par exemple lors de la mise au point de l'étage de présentation, en ne rechargeant que le code de ce étage. Nous verrons par la suite que ce type de fonction est proposé en standard dans certains cas, comme par exemple les JSP (cf. section 5.5.4).

Les sections qui suivent s'attachent à présenter les différents éléments programmatiques qui sont mis à la disposition du programmeur d'applications J2EE. Elles se focalisent pour l'essentiel sur les éléments concernant la programmation de l'étage de présentation (interfaces *servlet*) et de l'étage métier (interfaces EJB), avec un point sur trois aspects prépondérants requis par les applications d'entreprise : les mécanismes permettant de faire communiquer et d'intégrer les différentes applications, les transactions comme premier support à la sûreté de fonctionnement, et la sécurité pour gérer l'authentification des utilisateurs et leurs autorisations d'accès aux ressources de l'entreprise.

5.3 Applications réparties et intégration

Parmi les fonctions importantes offertes par l'environnement J2EE, un part importante concerne les moyens d'interconnecter les applications entre elles, mais aussi ceux permettant à ces applications de s'intégrer à des systèmes existants, tels que des systèmes patrimoniaux exhibant des modes de couplage très spécifique.

Nous étudions dans un premier temps les mécanismes de communication standard pour faire communiquer des applications à travers des abstractions de haut niveau. Il s'agit soit de mécanismes synchrones de type appel de procédure à distance (cf. chapitre 1, section 1.3), soit de mécanismes asynchrones comme de simples envois de message non bloquant ou de techniques de « publication/abonnement » (en anglais *publish and subscribe*).

5.3.1 Appel de procédure à distance

Deux mécanismes de communication de type RPC sont supportés. Le premier est RMI et est dès l'origine fortement couplé à Java [Sun Microsystems 2003]. Le second, introduit dans la version 1.4 des spécifications, concerne le support des services Web. Dans les deux cas, J2EE permet de projeter une interface Java dans ces deux mondes, offrant ainsi un niveau de transparence important.

Java RMI

Dans le cas de RMI, les règles de projection sont relativement simples. La sémantique de l'invocation de méthode n'est pas exactement la même qu'une invocation locale (pure Java) de l'interface. En effet, concernant le passage d'objets en paramètre d'invocation de méthode, dans un monde pur Java, il s'agit dans tous les cas d'un passage par référence.

Dans le cas de RMI, les objets répartis (objets RMI implantant l'interface `Remote`) sont passés par référence alors que les autres sont passés par valeur.

Concernant la partie encodage des invocations à distance, RMI propose deux protocoles. L'un, JRMP, est le protocole natif de Java RMI qui fait d'ailleurs partie de Java de base (c'est à dire J2SE). L'autre est IIOP, le protocole de Corba. C'est ce protocole qui a été choisi pour l'interopérabilité entre les serveurs J2EE interagissant en mode RMI.

Web Services

La projection vers les services Web est encore plus restrictive que dans le cas de RMI. En effet, il n'existe pour l'instant (spécification en cours) aucun moyen de passer des références vers d'autres services Web lors de l'invocation d'un service Web. Il n'y a donc que du passage par valeur, avec des contraintes sur l'encodage XML des objets passés en paramètre. Par exemple, si un objet est référencé plusieurs fois dans le graphe à encoder, l'encodage amène à ce qu'il y a autant d'instances différentes de l'objet en question dans le graphe décodé que de références dans le graphe d'origine.

Concernant l'encodage XML des invocations à distance, c'est le mode *document/wrapped* spécifié dans le cadre du WS-I qui a été retenu comme protocole d'interopérabilité.

5.3.2 Communication asynchrone : JMS

Le support de communication asynchrone s'appuie sur les spécifications JMS. Celles-ci offrent deux paradigmes de communication.

- Le mode point à point permet, à travers le paradigme de file de messages (queue en anglais), à un utilisateur JMS de produire des messages dans une file et à un autre de les consommer dans cette même file.
- Le paradigme de sujet d'intérêt (en anglais *topic*), les communications sont multi-points. Plusieurs utilisateurs peuvent publier des messages sur le sujet alors que plusieurs autres peuvent s'abonner pour les recevoir.

Dans les deux cas, les spécifications donnent les moyens de définir des propriétés concernant la délivrance des messages. Par exemple, garantir l'acheminement (pas de perte), garantir l'ordre de réception par rapport à l'ordre d'émission, garantir qu'il n'y a pas de réception multiple, etc. Enfin, il est prévu une intégration en mode transactionnel du support JMS dans J2EE, permettant de garantir de l'atomicité entre la consommation d'un message et des actions faites dans une base de données par exemple suite à cette consommation.

Concernant le protocole d'encodage des messages dans les systèmes JMS, rien n'est spécifié. Cela signifie qu'il n'y a pas d'interopérabilité possible entre différents supports JMS : on ne peut pas produire dans une queue avec une première implantation et consommer avec une autre implantation.

5.3.3 Connecteurs au standard JCA

L'environnement J2EE propose un cadre pour l'intégration de systèmes externes à travers des connecteurs. Il permet de supporter des interactions dans deux modes : soit c'est l'environnement J2EE qui est à l'initiative de l'interaction, soit c'est le système externe.

Dans le second cas, les messages reçus par le connecteur sont ensuite dirigés vers des composants EJB réactifs (cf. section 5.6.3) pour être traités.

Un connecteur JCA doit planter des API conforme à cette spécification pour pouvoir s'intégrer à un serveur J2EE. Suivant les interfaces qu'il implante, il peut s'intégrer principalement à trois supports techniques fournis pour l'environnement J2EE :

- Gestion des réserves de connexions : Un des principes partagés par tous les types de connecteurs est qu'un connecteur gère des connexions permettant de communiquer avec le système externe auquel il donne accès. Ces connexions sont généralement lourdes et coûteuses à mettre en place. En conséquence, le serveur J2EE prend en charge le recyclage des connexions pour éviter leur mise en place et leur destruction lors de chaque échange avec le système externe (gestion d'une réserve de connexions par le serveur).
- Gestion des transactions : L'utilisation de connecteurs se fait bien souvent pour intégrer l'environnement J2EE avec d'autres systèmes critiques, offrant eux-mêmes des capacités transactionnelles. Suivant le mode d'interaction, cela signifie que soit le serveur J2EE doit inclure les interactions avec le système externe dans une transaction répartie initialisée par J2EE, soit à l'inverse que le système externe interagissant avec le serveur J2EE doit inclure les actions effectuées dans l'environnement J2EE dans une transaction répartie initialisée par le système externe. JCA propose pour cela différents niveaux d'intégration transactionnelle : pas de support transactionnel, support de transaction locale (transactions à validation une phase), et support de transaction globale répartie (transaction à validation à deux phases).
- Gestion de la sécurité :

Ce standard est maintenant utiliser de façon usuel comme mécanisme d'intégration standard de sous-systèmes J2EE, même si ces derniers ne sont pas transactionnels. C'est en tout cas souvent ce qui se passe pour les connecteurs standard présenté ci-dessous (cf. section 5.3.4), ou encore les connecteurs JMS présentés dans la section 5.3.2. Notons que l'objectif des contrats requis par l'environnement J2EE pour ce type de composant est de permettre au serveur de garder la responsabilité de la gestion des ressources. Cela permet ainsi d'avoir une vision et donc des politiques de gestion de ressources plus globales, de façon à exploiter au mieux cette gestion suivant le contexte d'utilisation.

En terme d'intégration, une limite de la spécification JCA est qu'elle ne propose rien concernant la gestion du cycle de vie de ce type de composant. Cela peut poser des problèmes quant à l'initialisation de tels composants ou encore vis-à-vis de l'ordre dans lequel ils sont activés.

5.3.4 Connecteurs de base

Nous donnons dans cette section un aperçu rapide de deux connecteurs de base utilisés dans l'environnement J2EE : le connecteur d'accès aux bases de données relationnelles, et le connecteur d'accès au système de messagerie électronique.

Connecteur JDBC d'accès aux bases de données relationnelles

JDBC correspond à l'interface standard permettant d'accéder à des base de données relationnelles à partir de l'environnement Java de base (J2SE). Il permet d'effectuer tous les

échanges nécessaires avec de telles bases, en s'appuyant bien sûr sur le standard SQL (voir <http://www.jcc.com/sql.htm> comme point d'entrée pour de nombreuses références). Les échanges se font donc sur la base d'émissions de requêtes SQL vers la base et récupération en retour d'appel (interaction synchrone).

Pour ce faire, un pilote JDBC dédié au produit base de données à accéder est chargé comme un connecteur dans l'environnement J2EE. Il permet d'adapter les appels à l'API JDBC au format d'échange avec le produit en question (par exemple MySQL). En effet, ces échanges se font généralement grâce à un protocole réseau spécifique au produit.

Connecteur d'accès au *mail*

L'autre connecteur de base proposé dans J2EE concerne l'accès à la messagerie électronique. Il est ainsi possible à partir d'une application J2EE d'envoyer des messages électroniques de façon standard quelque soit le système de messagerie sous-jacent. En effet, l'API JavaMail [Sun JSR-000904 2000] offre une API *fournisseur* permettant d'intégrer l'adaptateur adéquat pour interagir avec le système de messagerie choisi au déploiement de l'application. Il est ainsi possible à une application d'envoyer des messages vers une messagerie X400 ou vers une messagerie SMTP sans avoir à modifier le code applicatif.

5.4 Gestions des transactions

La sûreté de fonctionnement est un souci constant pour les applications d'entreprise. Les transactions offrent un modèle programmatique permettant de voir une application comme une suite de transitions amenant les ressources gérées par celle-ci d'un état cohérent à un autre, avec la possibilité de revenir à tout moment à un état cohérent en cas d'échec lors d'une telle transition.

5.4.1 Notion de transaction

Les applications d'entreprise sont généralement dites critiques dans le sens où elles manipulent des informations dont on cherche à garantir la cohérence dans le temps. Pour garantir cette cohérence, ce type d'application ainsi que les systèmes qui les supportent mettent en œuvre la notion de transaction [Gray and Reuter 1993]. Cela permet de manipuler ces ressources dites critiques (comme par exemple des bases de données ou des queues de messages) dans le cadre de séquences opératoires (les transactions) garantissant les propriétés dites ACID (Atomicité / Cohérence / Isolation / Durabilité). Ces séquences sont encadrées par des ordres de démarcation des transactions : démarrage d'une transaction puis terminaison de celle-ci soit par une validation en cas de succès de la séquence d'exécution encadrée, soit par une annulation en cas d'échec.

Cette fonction de l'environnement J2EE est définie par le standard JTA. Il spécifie comment une application peut accéder à des capacités transactionnelles de façon indépendante de l'implantation des ressources transactionnelles qu'elle utilise. Cette spécification couvre l'ensemble des APIs permettant de faire interagir le gestionnaire de transactions avec les parties impliquées dans le système exécutant ces transactions potentiellement réparties à savoir : l'application transactionnelle, le serveur J2EE hébergeant cette application et le gestionnaire qui contrôle l'accès aux ressources partagées (par exemple l'accès à

différentes bases de données). Dans le cas des transactions réparties, JTA définit les interfaces nécessaires à l'utilisation de ressources supportant le protocole XA (protocole de validation à 2 phases).

5.4.2 Manipulation des transactions dans un serveur J2EE

Un serveur J2EE fournit le moyen de manipuler des séquences transactionnelles quel que soit l'étage dans lequel il est mis en œuvre. Le moyen standard utilisé pour cela est l'interface `UserTransaction`, qui permet d'effectuer les principales opérations sur les transactions : les opérations de démarcation, c'est à dire `begin` pour démarrer une transaction, `commit` pour la valider, et `rollback` pour l'annuler.

Le démarrage d'une transaction a pour effet d'associer une transaction au contexte d'exécution courant (le *thread* courant). La démarcation de transaction dans ce modèle n'a donc de sens que dans le cadre de ce contexte d'exécution. C'est donc ce même contexte qui doit exécuter l'opération de terminaison, validation ou annulation.

Pour manipuler de façon cohérente la démarcation de transaction, il est conseillé de la mettre en œuvre au niveau programmatique de manière très localisée. Dans ce cas, les opérations de démarcation seront par exemple appelées à partir de la même opération. Cela clarifie la démarcation mais les problèmes nécessitant l'annulation de la transaction peuvent néanmoins intervenir dans n'importe laquelle des opérations exécutées entre ces bornes, à n'importe quelle profondeur d'appel dans le code ainsi déroulé. Pour pallier le problème, l'interface `UserTransaction` fournit l'opération `setRollbackOnly` permettant de forcer une issue fatale pour la transaction courante.

Enfin, hormis l'accès au statut de la transaction courante (opération `getStatus`), la dernière fonction fournie pour manipuler une transaction est la possibilité de lui associer un temps d'exécution maximal (opération `setTransactionTimeout`). Le système transactionnel peut avoir une valeur par défaut (dépendant de l'implantation) pour ce temps d'exécution. Dans tous les cas, si ce temps est expiré avant la fin de la transaction, le système transactionnel annule la transaction et affecte le statut annulé au contexte transactionnel courant.

Ces opérations de démarcation sont souvent difficiles à manipuler dans le contexte d'une application. L'environnement J2EE propose un moyen plus déclaratif et aussi plus sûr pour les mettre en œuvre, notamment dans le cadre du support des composants métier (support EJB) : en effet, il est possible de spécifier qu'une opération est transactionnelle. Le conteneur prend alors en charge le démarrage de la transaction en début d'opération et la terminaison de celle-ci en sortie d'opération : validation ou annulation suivant l'état d'exécution de l'opération.

Des interfaces de plus bas niveau sont fournies par le système transactionnel de l'environnement J2EE. Elles permettent par exemple de suspendre et réactiver des contextes dans le cadre du contexte d'exécution courant. L'utilisation de ces mécanismes sort du contexte d'une utilisation standard et sont hors sujet pour notre présentation de cette fonction de l'environnement J2EE.

5.4.3 Mise en relation des composants J2EE avec le système transactionnel

Pour les composants applicatifs s'exécutant dans le cadre d'un serveur J2EE, il est possible de récupérer une référence vers le gestionnaire de transaction. Cela se fait à travers l'interface JNDI, sachant que ce gestionnaire y est enregistré sous le nom `java:comp/UserTransaction`. Voici un exemple de récupération du lien vers le gestionnaire de transaction :

```
UserTransaction systx;
// Un composant \emph{servlet} se liant au gestionnaire de transaction
// lors de son initialisation.
public void init(ServletConfig config) throws ServletException {
    systx = new InitialContext().lookup("java:comp/UserTransaction");
    systx.begin();          // démarrage d'une transaction
    ...
    ...
    systx.commit();         // validation de la transaction courante
    ...
}
```

Cette liaison est faite ici à l'initialisation d'un composant de présentation. L'accès à ce contexte JNDI peut néanmoins s'opérer dans n'importe quelle séquence de code s'exécutant dans le cadre d'un serveur J2EE.

5.5 Programmation de l'étage de présentation Web

L'étage de présentation de l'environnement J2EE fournit le moyen de générer des présentations de type Web, c'est à dire produisant des pages HTML en réponse à des requêtes HTTP. L'intérêt est ici que la production de ces pages est effectuée de façon programmatique, offrant ainsi une dynamicité et une interactivité beaucoup plus importante sur le contenu produit que des fichiers statiques. Ce modèle programmatique est défini par les spécifications des *servlets*. Outre leur capacité à traiter les pages dynamiques, les moteurs de *servlets* sont aussi capables de traiter des contenus statiques comme nous le verrons dans la suite.

Le support fourni par J2EE pour l'étage de présentation se décompose en une pile de fonctions présentant différents niveaux d'abstraction. Les spécifications actuelles proposent trois niveaux :

- Au niveau le plus bas, on retrouve le support des *servlets* qui offrent les mécanismes programmatiques de base pour traiter des requêtes réseau venant de connexions IP (cf. figure 5.8). C'est donc pour l'essentiel un support orienté serveur. Les APIs définissant ce support appartiennent au package Java `javax.servlet`.
- Au-dessus de ce niveau, on retrouve la déclinaison pour le protocole HTTP de ces interfaces *servlet*. Les APIs couvrant cette déclinaison appartiennent au package Java `javax.servlet.http`. D'autres déclinaisons protocolaires existent, notamment celle pour SIP [Handley et al. 2000, Sun JSR-000116 2003].
- Le plus haut niveau (partie JSP) propose des mécanismes de création de pages HTML ou tout autres pages XML (XHTML, VoiceXML, WML, etc) à base de documents

type, mixant le code HTML et le code Java pour la définition des parties dynamiques de ces pages. Même s'il est principalement utilisé pour construire des pages HTML téléchargées à travers HTTP, le support JSP est indépendant du protocole d'échange de pages avec le client. Il est donc par principe indépendant de HTTP, même s'il contient aussi une déclinaison pour HTTP. Les APIs relevant du support JSP appartiennent aux *packages* Java `javax.servlet.jsp`, `javax.servlet.jsp.el`, et `javax.servlet.jsp.tagext`. Les deux derniers *packages* contiennent des fonctions avancées améliorant et/ou simplifiant la définition de pages JSP.

Les deux premiers niveaux d'abstraction sont couverts par la spécification des *servlets* dans [Sun JSR-000154 2003]. Le niveau JSP est couvert par une spécification complémentaire dans [Sun JSR-000152 2003].

Il y a évidemment des dépendances fonctionnelles des niveaux les plus hauts vers ceux du dessous. La partie JSP dans sa définition dépend du niveau *servlet* de base et est déclinée pour les *servlets* HTTP. Elle pourrait néanmoins être déclinée pour d'autres protocoles, par exemple WTP pour le monde mobile WAP. Nous détaillons dans la suite les trois niveaux énumérés précédemment, sachant que la plupart des utilisateurs s'appuient sur le niveau le plus haut, que ce soit à base du standard JSP ou d'autres moteurs de templates comme XMLC proposé par le consortium ObjectWeb [Objectweb Enhydra 2005]. Des canevas logiciels d'encore plus haut niveau et basés sur le patron architectural *modèle/vue/contrôleur* (MVC) pour les interfaces graphiques existent. Ils sont largement utilisés mais non standard. Nous pouvons citer par exemple STRUTS [The Apache Software Foundation 2005] qui est de loin le plus répandu, ou encore JSF qui est standardisé dans le cadre de J2EE [Sun JSR-000127 2004].

5.5.1 Principes d'architecture des composants de présentation Web

Les principes de base des composants J2EE est qu'ils sont pris en charge par un environnement de déploiement et d'exécution qui leur est dédié et qu'on appelle conteneur (cf. section 5.1.4).

Concernant le rôle lors du déploiement, le conteneur prend en charge un paquetage dont le format lui est spécifique. Dans le cas du conteneur de composants Web, il s'agit généralement de paquetage `.war`. A partir de ce paquetage, il charge le code des composants inclus dans le paquetage et met en place les chaînes de liaison qui vont permettre d'activer ces composants à partir d'un discriminant qui lui est associé (par exemple une URL dans le cas d'une *servlet* HTTP).

A l'exécution, le conteneur prend en charge les liens réseau sur lesquels les requêtes et les réponses vont être transmis (par exemple, ouverture d'un canal TCP/IP sur le port `8080` dans la figure 5.8). Il encapsule ensuite ces requêtes et réponses réseau dans des objets respectant les interfaces spécifiées par les *servlet*, et active le composant *servlet* adéquat pour traiter ces requêtes.

5.5.2 *Servlets* : organisation de la pile protocolaire

En première approximation, le modèle programmatique proposé par les *servlets* consiste à fournir un mécanisme de répartition ou de démultiplexage de requêtes réseau vers l'objet *servlet* de traitement de cette requête (cf. figure 5.8 montrant le démultiplexage

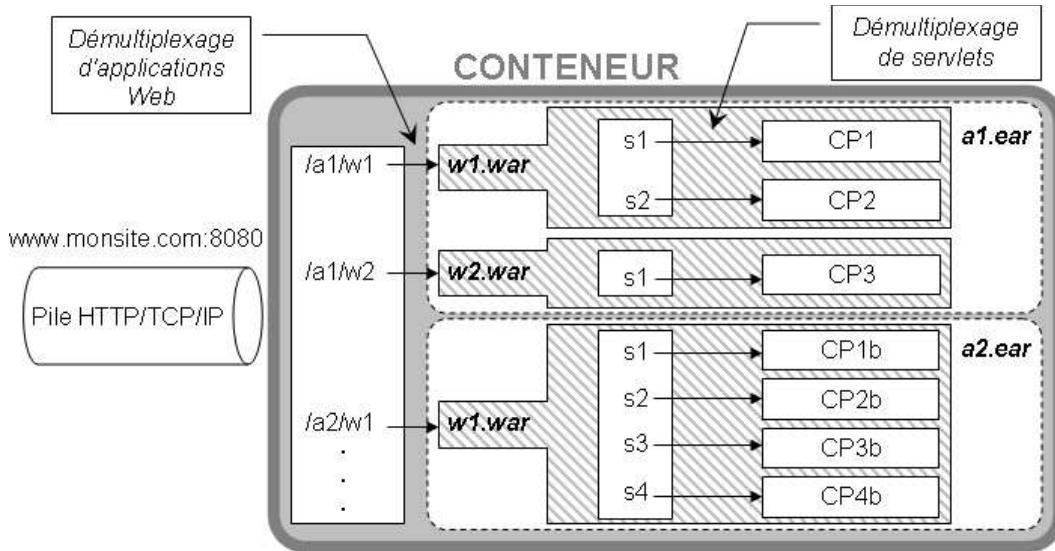


Figure 5.8 – Le démultiplexage des composants de présentation (*servlets*)

à deux niveaux effectué par un conteneur de *servlet* HTTP). Une fois ce démultiplexage effectué par le moteur de *servlets*, la *servlet* sélectionnée est activée pour opérer le traitement grâce à l'opération `service`. Nous verrons par la suite qu'il existe aussi des mécanismes pour intervenir sur ce processus de demultiplexage.

Concernant l'opération de traitement de requête, le message à traiter ainsi que le message de réponse sont abstraits par le modèle au travers de deux interfaces permettant leur manipulation, à savoir `ServletRequest` et `ServletResponse`. Ces deux interfaces donnent accès à la fois aux informations d'en-tête des messages requises par les *servlets* et aux contenus de ces messages. Elles déterminent donc les principes de bases de construction d'une pile protocolaire à base de *servlet*.

Gestion d'une requête de *servlet* (interface `ServletRequest`)

Parmi ces informations, beaucoup concernent l'adressage réseau. De telles adresses spécifient généralement trois informations de base : l'adresse IP, le port, et le nom de la machine en question (i.e., nom DNS).

- L'en-tête de la requête

Il contient un certain nombre d'adresses réseau telles que celles de la machine *client* d'où provient la requête (machine à l'origine de la requête ou le dernier *proxy* l'ayant relayé), celle de la machine *serveur* (la machine à qui était destiné le message à l'origine), ou encore celle de la machine de traitement (la machine sur laquelle l'opération de `service` s'exécute actuellement). Ces adresses permettent de traiter différemment les messages suivant le chemin qu'ils ont emprunté. Parmi ces informations d'en-tête, nous retrouvons aussi le protocole utilisé par la requête, identifié par le nom et la version identifiant le protocole applicatif mis en œuvre, soit au-dessus de TCP/IP, soit au-dessus de UDP/IP (par exemple HTTP/1.1). Ce protocole est donc pris en charge avec une architecture de pile conforme à la spécification *servlet* que nous

sommes en train de décrire.

- Le contenu de la requête

Il est représenté par plusieurs informations. La première est la taille du contenu du message. Une autre information donne le type de ce contenu qui est un type MIME, ou qui est nul dans le cas où il n'est pas connu. Enfin, on donne le moyen d'accéder à ce contenu, soit à travers la lecture d'un flux binaire, soit à travers la lecture d'un flux de caractères. Dans le cas d'un contenu de type texte, le type d'encodage de l'information est aussi défini.

- Des attributs associés à la requête

Au cours des phases de traitement de la requête, il est possible de gérer des attributs permettant de caractériser la requête vis-à-vis des phases suivantes. Cette gestion autorise la création, la modification, la destruction d'attributs, ou encore leur énumération.

Gestion d'une réponse de *servlet* (interface `ServletResponse`)

Les informations gérées dans le cadre d'une réponse sont pour l'essentiel de deux ordres : celles qui concernent le contenu de cette réponse, et celles permettant de contrôler le flux d'informations lié au contenu retourné au client de la *servlet*.

- Le contenu de la réponse

Il est représenté par plusieurs informations de façon symétrique à une requête. La première est la taille du contenu de la réponse. Une autre information spécifie le type de cette réponse qui est un type MIME. Enfin, on donne le moyen de produire cette réponse, soit à travers l'écriture d'un flux binaire, soit à travers l'écriture d'un flux de caractères. Dans le cas d'un contenu de type texte, le type d'encodage de l'information est aussi défini.

- Le contrôle du flux d'informations renvoyés au client

Un point important est la possibilité de gérer le flux de données émis vers le client auquel la *servlet* répond. Dans le cas de réponse de grande taille, comme par exemple le renvoi d'une image, cela permet de ne pas saturer la mémoire du serveur pour le seul bénéfice de la requête en question (un serveur peut gérer un nombre important de requêtes simultanées). Pour cela, il est possible de définir la taille du tampon utilisé pour gérer les données de réponse. Les données du tampon sont alors émises vers le client soit lors d'une demande explicite (opération `flush`), soit lorsque ce tampon est plein. Il est important de noter que toutes les métainformations associées à la réponse (cf. point précédent) doivent avoir été définie avant la première émission ; en effet, l'en-tête de la réponse (contenant notamment la taille de la réponse) est évidemment émis à ce moment là.

Organisation du conteneur et cycle de vie des *servlets*

Pour finir de définir les principes de base des *servlets*, la figure 5.9 nous montre le cycle de vie d'une *servlet*, ainsi que la manière dont celle-ci est pilotée par son conteneur. En fait, cette figure présente les trois structures de base de l'architecture de *servlet* :

- Le conteneur de *servlets*

Le conteneur remplit deux fonctions majeures. Il permet de déployer une *servlet*, c'est à dire charger la classe associée et créer une instance qui est alors liée à un chemin d'activation (cf. mécanisme de démultiplexage). C'est à ce moment-là que

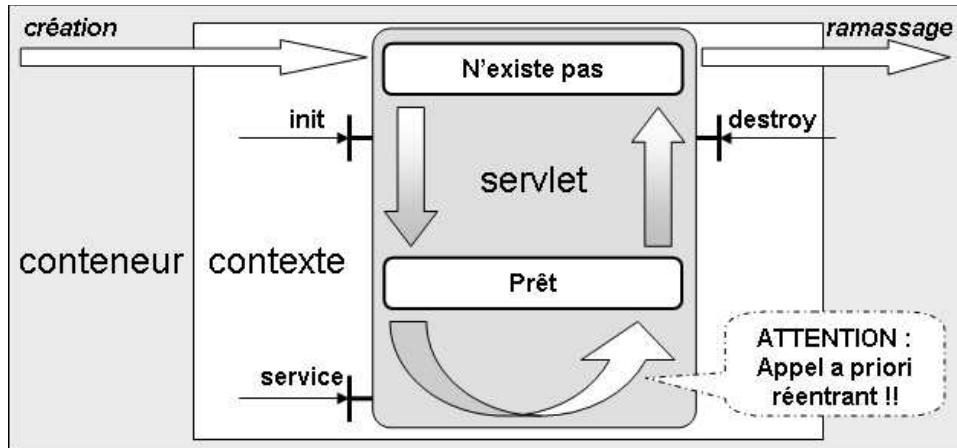


Figure 5.9 – Le cycle de vie des *servlets*

la méthode `init` est appelée par le conteneur pour initialiser la *servlet* en question. Elle est alors prête à être activée. De façon symétrique, le conteneur peut éliminer une *servlet* ; à ce moment-là, il appelle `destroy` et l'instance de la *servlet* est alors ramassé. L'autre fonction concerne l'activation de la *servlet* lors de l'arrivée d'un message. Le rôle du conteneur est alors d'encapsuler ce message dans les interfaces telles que présentées précédemment, et à partir d'informations récupérées dans ce message, de déterminer le contexte dans lequel se trouve la *servlet* à laquelle le traitement du message est délégué.

- Le contexte de *servlets*

Un contexte contient un certain nombre de *servlets* qui y ont été déployées auxquelles un nom est associé et qui permet d'identifier celle-ci dans le contexte. Au contexte lui-même est associé un nom qui sert de préfixe lors du processus de démultiplexage. Il offre aussi à ces *servlets* de gérer un ensemble d'attributs qui leur permettent de partager des informations (communication entre *servlets*). Il permet aussi de manipuler le système de démultiplexage (par exemple, retrouver une *servlet* d'un nom donné ou le contexte auquel elle appartient).

- La *servlet*

Une fois qu'une *servlet* est prête (initialisation terminée), le conteneur peut l'activer au rythme qu'il juge bon (invocation de `service` en parallèle). Comme au niveau contexte, la *servlet* gère aussi un ensemble d'attributs accessible aux requêtes traitées par `service`.

Filtres de *servlets*

Il est possible d'associer des filtres pour effectuer des traitements préalablement à l'exécution d'une *servlet*. Ces filtres peuvent agir sur les données d'en-tête lié à une requête que sur son contenu, ainsi bien sûr que sur la réponse qui doit être produite par la *servlet*. Ces filtres doivent au préalable avoir été enregistrés auprès du conteneur et associés à une ou plusieurs *servlets*. Le conteneur crée alors une chaîne de filtres pour chaque *servlet* qui a été déployée et exécute cette chaîne lors de chaque activation d'un traitement de requête.

Contrôle du démultiplexage

Lors de l'exécution d'une *servlet*, il est possible d'utiliser d'autres ressources d'exécution permettant de déléguer tout ou partie son traitement : les *RequestDispatcher*. Ces ressources peuvent être récupérées par la *servlet* auprès du conteneur à partir du nom auquel est associé la ressource en question.

A partir de là, cette ressource peut être utilisée soit pour agir à la place de la *servlet* en cours d'exécution (voir la méthode `forward`), soit pour inclure une partie de la page (voir la méthode `include`) générée par cette *servlet* (par exemple un bandeau commun à toutes les pages d'un site). Dans le cas d'un `forward`, aucune information ne doit avoir été envoyée au client par la *servlet* au moment de son activation (génération d'une exception).

5.5.3 Servlets HTTP

L'extension des interfaces *servlet* pour le support du protocole HTTP suit évidemment les principes de construction d'une pile de *servlets*. Elle expose notamment toutes les informations d'en-tête et de contenu relatives aux échanges HTTP et nécessaires au traitement de ce type de requête. Elle introduit en plus une notion supplémentaire permettant de gérer des informations communes à une suite d'échanges HTTP, la notion de session, bien utile dans la mesure où HTTP est un protocole sans état.

Démultiplexage de requêtes HTTP

Comme nous l'avons expliqué précédemment, le processus de démultiplexage des requêtes se fait en deux étapes. Dans le cas de requêtes HTTP, la figure 5.8 montre que le support de *servlets* doit d'abord déterminer l'application Web, puis la *servlet* concernée dans cette application. Le démultiplexage s'appuie sur le décodage d'URL invoquée par la requête HTTP. Une telle URL est de la forme :

```
http://machine[:port]chemin[;infosupp]?requête]
```

Par exemple, l'URL `http ://www.monsite.com :8080/a1/w2/s1?p1=val1&p2=val2` permet d'activer une des *servlets* de la figure 5.8. Nous observons dans cet exemple qu'un conteneur de *servlets* HTTP est présent sur la machine `www.monsite.com` et qu'il attend les requêtes HTTP sur le port 8080. Le chemin identifiant la ressource Web à invoquer est `/a1/w2/s1`. Le reste de l'URL, commençant au caractère ?, permet de dire que cette ressource Web est une requête à laquelle est passée deux paramètres `p1` et `p2` auxquels sont respectivement associés les valeurs `val1` et `val2` (ces paramètres sont transformés en attributs de la requête `HttpServletRequest` construite par le conteneur).

C'est la partie chemin qui sert au processus de démultiplexage mis en œuvre par le conteneur de *servlets*. En effet, lors du déploiement d'une application Web (contenant en ensemble de *servlets* à déployer), elle définit un contexte auquel est associé un nom correspondant à la racine du chemin de démultiplexage des *servlets* qu'elle contient. Par exemple, le déploiement de l'application Web spécifiée par `w2.war` définit un contexte associé à la racine `/a1/w2` (racine définie dans le descripteur de déploiement contenu dans le paquetage). Les *servlets* qu'elle contient sont ensuite créées dans ce contexte où leur est associé un nom défini lui aussi dans le descripteur de déploiement ; il y a dans le cas présent une seule *servlet* appelée `s1`. Le processus de démultiplexage consiste donc à rechercher un

contexte dont le nom correspond à un préfixe du chemin d'une URL puis à rechercher dans ce contexte une *servlet* dont le nom correspond au reste du chemin auquel on a soustrait ce préfixe.

Nous n'allons pas détailler les informations relatives à HTTP mais simplement rappeler brièvement en quoi elles consistent. Elles sont pour l'essentiel introduites dans le cadre des interfaces de manipulation des requêtes et des réponses, correspondant à des extensions des interfaces préalablement définies par les *servlets* de base. Par ailleurs, la classe abstraite `HttpServlet` propose un raffinement de l'opération `service` en appels vers des opérations correspondant aux actions définies par le protocole HTTP, à savoir `doGet` pour une requête GET, `doPost` pour une requête POST, `doHeader` pour une requête HEADER, etc. Ces opérations ont la même signature que l'opération `service`, ayant en paramètre les objets représentant la requête et la réponse.

Gestion d'une requête HTTP (interface `HttpServletRequest`)

Les informations mises à disposition par cette interface correspondent aux données véhiculées par une requête HTTP, et notamment :

- Les informations sur l'URL. Il s'agit de toutes les informations relatives à l'URL d'invocation et plus particulièrement les éléments qui la compose tels qu'ils ont été définis précédemment (URI ou chemin, les informations supplémentaires s'il y en a (ce qui est entre ; et ?), la requête s'il y en a une (ce qui suit ?), ou encore le nom du contexte de la *servlet*).
- Les *cookies*. Il s'agit de tous les *cookies* qui ont été envoyés par le client dans la requête.
- L'utilisateur. Il s'agit du nom de l'utilisateur à l'origine de la requête s'il a été authentifié.
- Les paramètres d'en-tête HTTP. Il s'agit de paramètres passés en en-tête de la requête HTTP spécifiant par exemple des dates ou des informations sur les langues supportées par l'utilisateur. On peut récupérer ces valeurs sous différentes formes telles que entier, chaîne de caractères, date (long).
- La session. Il s'agit de la session dans laquelle cette requête s'exécute. En général, s'il n'y en a pas encore, celle-ci est créée par le conteneur. Cette information n'est pas définie par HTTP (spécifique aux *servlets*). Elle est transportée dans la requête soit sous la forme d'un *cookie*, soit dans les informations supplémentaires de l'URL d'invocation (cf. partie de l'URL entre ; et ?).

Gestion d'une réponse HTTP (interface `HttpServletResponse`)

Par rapport à une réponse de *servlet* de base, cette interface donne accès à des opérations permettant soit de formatter l'en-tête du message de réponse HTTP, soit de formatter les URL diffusées dans le contenu.

Dans le premier cas, il s'agit de définir dans la réponse les *cookies* retournés, le code de statut de la réponse, des paramètres d'en-tête. Cela peut aussi consister à retourner directement une réponse d'erreur ou une réponse de redirection de la requête vers une autre URL.

Dans le second cas, il s'agit d'opérations qui permettent d'encoder les informations de session dans les URL de dialogue diffusées dans les pages de réponse, permettant ainsi de transporter systématiquement cette information entre le client et le serveur pour toutes les interactions les concernant. Cette approche permet d'éviter le passage par les *cookies*, permettant ainsi de gérer des sessions même si l'utilisateur a désactivé les *cookies* dans son navigateur par exemple.

Gestion de session HTTP

La notion de session est l'ajout majeur de la couche *servlet* HTTP. Elle permet de maintenir un contexte dédié aux échanges entre un utilisateur et l'application Web à laquelle il accède. L'identificateur de cette session doit donc être communiqué lors de chaque interaction de l'utilisateur avec l'application. Les *servlets* HTTP offrent pour cela deux méthodes que nous avons déjà introduites auparavant :

- Par échange de *cookie* : lors de la première interaction avec l'application Web, le conteneur crée une session et renvoie son identificateur dans la réponse au client. Le protocole des *cookies*, mis en œuvre par les navigateurs Web, assure ensuite que le client diffuse à l'application ce *cookie* lors de chaque nouvelle interaction.
- Par réécriture des URL : cette méthode consiste à encoder dans toutes les URL diffusées dans les pages demandées par le client, l'identificateur de la session qui lui est associée. Cela nous ramène donc à la solution précédente puisque chaque nouvelle interaction avec l'application passe alors par une URL contenant l'identificateur de session.

Le point important vis-à-vis des interfaces de manipulation de session est qu'elles sont indépendantes de l'une ou l'autre des solutions présentées ci-dessus. Notons qu'une session est liée au contexte d'une application Web (par exemple /a1/w2 dans notre exemple d'URL). Cela signifie que cette session n'est pas connue si on invoque une *servlet* d'un autre contexte.

Une session définit un contexte composé dans un ensemble modifiable d'attributs eux-mêmes modifiables. Par exemple, il est possible de référencer un composant métier session à partir d'un tel attribut, donnant ainsi accès à une vision conceptuelle plus élaborée des informations de session. Enfin, comme pour les *servlets*, il est aussi possible d'observer les événements du cycle de vie d'une session (création, destruction, modifications).

Cela clôt la description de l'environnement d'exécution de *servlets* du point de vue de l'encapsulation de couche protocolaire à travers des APIs spécialisées par protocole, où les éléments architecturaux de base qui sont spécialisés sont la *servlet* représentant du protocole applicatif, et les requêtes et les réponses permettant de gérer les flux d'informations liés à ces protocoles.

5.5.4 JSP : construction de pages HTML dynamiques

Dans la plupart des utilisations des *servlets* actuelles, le contenu de la réponse se matérialise sous la forme d'une page HTML ou XML. Or la production de telle page de façon programmatique est un exercice laborieux. En effet, la production des parties statiques consiste à aligner des lignes de code du type `out.println("<tag>mon contenu</tag>")` ; dont la lisibilité est plus que douteuse.

L'environnement *servlet* propose de remédier à ce problème à l'aide du mécanisme de JSP. Les JSP sont des patrons de pages contenant des contenus statiques dans leur forme final (le <tag>mon contenu</tag> de l'exemple précédent), et définissant des contenus dynamiques à partir de directives donnant accès au code Java permettant de manipuler les objets métiers de l'environnement du serveur d'application. L'échange d'information entre l'environnement JSP et l'environnement Java du serveur passe par l'utilisation du patron JavaBeans [B. Stearns 2000a, B. Stearns 2000b].

L'exemple suivant nous montre une page JSP qui affiche la date du jour. Nous pouvons voir que le format de la page JSP est spécifique (ni HTML, ni XML). Néanmoins, il existe une déclinaison pure XML de ces mêmes pages.

```
<%@ page contentType="text/html; charset=UTF-8" %>
<html>
    <head><title>La date</title></head>
    <body bgcolor="white">
        <jsp:useBean id="date" class="monorg.MaDate"/>
        <jsp:setProperty name="date" property="localisation" value="French"/>
        <b>Nous sommes le : </b>${date.date}
    </body>
</html>
```

Nous observons aussi les échanges avec le monde Java par l'utilisation d'un *bean*. La directive JSP `useBean` crée un *bean* de la classe donnée en paramètre et l'affecte à la variable `date`. La directive suivante affecte la propriété `localisation` du même *bean* (appel de la méthode `setLocalisation` de la classe `MaDate` définie ci-dessous). Enfin, on récupère la chaîne de caractères correspondant à la date à afficher par l'instruction `${date.date}`.

```
package monorg;

class MaDate {
    public void setLocalisation(String loc) {
        ...
    }
    public String getDate() {
        return ...
    }
}
```

Notons que le code d'une page JSP peut accéder à toutes les informations de l'environnement de *servlet* et a donc accès à toutes les APIs de ce dernier.

Les pages JSP sont déployées comme des *servlets*. En effet, au déploiement d'une page JSP, le conteneur génère la classe *servlet* correspondante, la compile et la charge. Lors de chaque interaction avec cette *servlet*, le conteneur vérifie que la *servlet* est plus jeune que la page JSP associée. Si ce n'est pas le cas, le conteneur met à jour la classe *servlet*, supprime les anciennes instances et en recrée de nouvelles pour traiter les requêtes. Cela offre une grande souplesse notamment en phase de développement.

D'autres mécanismes sont aussi offerts par JSP, et notamment les *tags* et les librairies de *tags* ainsi que le langage d'expression. Cela sort du sujet du présent chapitre dont l'objectif est de présenter les principes des technologies fournies par l'environnement J2EE. Nous ne les présentons donc pas.

5.6 Programmation de l'étage métier

L'étage métier de l'environnement J2EE fournit le moyen de programmer la logique métier d'une application orientée serveur en lui assurant les propriétés généralement nécessaires aux applications d'entreprise, ou plus précisément aux applications dites critiques (en anglais *mission-critical applications*). Il est défini par la spécification EJB (*Entreprise Java Bean*) [Sun JSR-000153 2003], la version 2.1 de la spécification servant de référence pour la description faite dans cette section.

L'environnement EJB fournit l'ensemble des services systèmes nécessaire au support d'un premier niveau de sûreté de fonctionnement (par exemple la persistance ou les transactions), ou de propriétés de sécurité (par exemple l'authentification ou le contrôle d'accès).

Il fournit d'autres mécanismes systèmes permettant d'optimiser la gestion des ressources mises en œuvre pour l'exécution d'une application. L'objectif est ici de pouvoir régler l'allocation de ressources pour optimiser les performances d'une application.

Enfin, il offre des moyens de communication de haut niveau permettant d'intégrer des applications J2EE réparties. Ces moyens permettent d'une part des communications dites *synchrones* (émission d'une requête avec attente de réponse), ou de type appel de procédures à distance (en anglais *Remote Procedure Call (RPC)*). Il peut s'agir ici soit de RMI (*Remote Method Invocation* ou support d'objets répartis pour Java), soit de *Web Services* (appel de procédures distantes basé sur XML). D'autre part, ils permettent des communications dites *asynchrones* (envoi d'une requête sans attente de réponse). C'est généralement le support JMS (*Java Messaging Service*) qui est mis en œuvre à cette fin. Tous ces mécanismes de communication sont décrits plus en détail dans la section 5.3.

L'objectif du modèle de programmation est de rendre la mise en œuvre de ces différents mécanismes aussi transparente que possible pour le programmeur. La description de ce modèle fait l'objet de cette section dans laquelle nous allons étudier les trois éléments principaux de ce modèle : composants de session, composants de données (appelés aussi composants entité), et composants réactifs.

5.6.1 Principes d'architecture des composants métier

Comme pour les composants de présentation (cf. section 5.5.1), les principes d'architecture des composants J2EE se déclinent dans le cadre des composants métier (cf. figure 5.10). On retrouve un conteneur EJB qui prend en charge le déploiement et l'exécution des composants métier. Pour tous les types de composants métier, la gestion du cycle d'allocation et de libération d'instances est sous le contrôle du conteneur. C'est là un point majeur puisque c'est à partir de cet ancrage que le serveur va pouvoir optimiser la gestion des ressources.

Au déploiement, le conteneur prend en charge des paquetages spécifiques qui sont des *.jar* avec une structure et des informations précises (cf. section 5.2.2). Il charge le code de ces composants métier et les rend aptes à être liés à d'autres composants afin qu'ils puissent être activés (appels synchrones ou asynchrones). Cette phase de mise en aptitude à la liaison procède de deux actions et met en œuvre le patron d'architecture *export/bind* comme utilisé dans [Dumont et al. 1998] :

- La première action consiste à exporter les références à ces composants déployés sous un nom discriminant dans un espace de noms associé à l'application en cours de

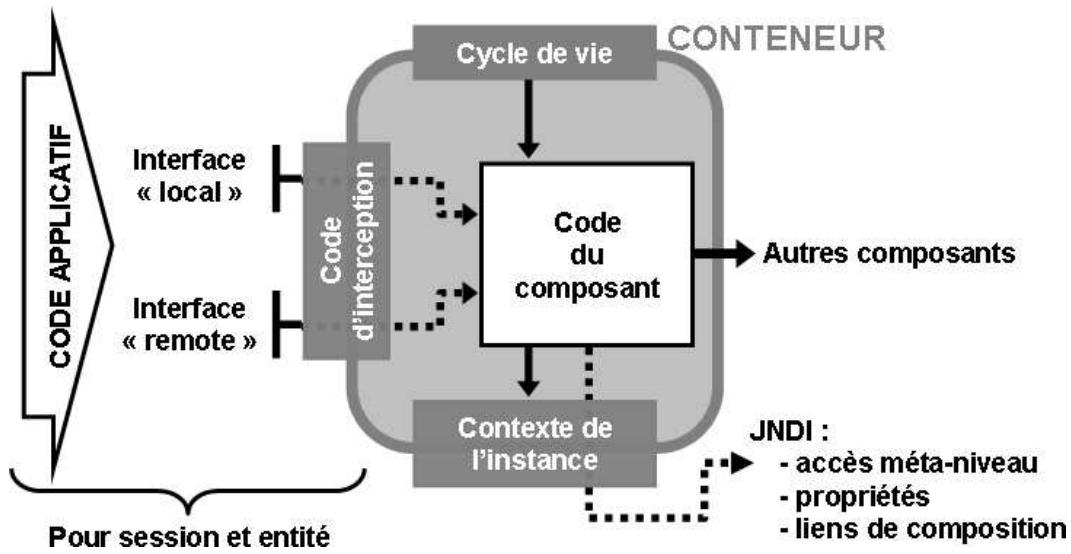


Figure 5.10 – Les principes d'architecture d'un composant métier EJB

déploiement. Cela se fait à l'aide d'un annuaire dédié accessible par JNDI.

- La deuxième action consiste à exporter les dépendances de ces composants dans un espace de noms associé au composant lui-même et à y associer la référence vers le composant auquel se lier grâce à son nom dans l'espace de nommage de l'application.

Rappelons que le découplage en deux actions permet de rendre le code indépendant du processus d'assemblage (pas de lien dans le code avec un composant particulier). A partir de là, les composants vont pouvoir accéder aux composants auxquels ils sont liés en effectuant la phase de liaison effective (phase de *bind*) dans le code applicatif (voir code utilisant *lookup* dans la section 5.2.3 équivalent au *bind*). Notons que cette phase est prise en charge par le conteneur dans la version 3 des EJB (injection des références par le conteneur dans une variable du code du composant) et devient donc transparente pour le code applicatif.

A l'exécution, le conteneur prend en charge le fait que le composant est activé de façon locale (appel à l'intérieur de la même JVM), de façon distante (appel à distance), ou sur arrivée d'un événement (messagerie asynchrone). Il prend aussi en charge les aspects techniques associés à ces composants comme l'activation de transactions, la vérification de contraintes de sécurité, ou les échanges de données avec un espace de sécurisation des données (en général une base de données relationnelles). Nous détaillons cela dans les sections qui suivent pour chaque catégories de composants métier.

La figure 5.10 nous montre que le conteneur met en œuvre du code (pour gérer les aspects techniques qu'il supporte de façon transparente au code applicatif) par interception des appels aux interfaces métier des composants (seulement pour les composants de session et les composants entité). Malheureusement, cette interception n'est pas transparente dans le modèle de programmation. La spécification fait l'hypothèse architecturale qu'il existe un objet d'interception indépendant de l'instance du composant métier, cet objet implantant l'interface métier. Dans ce cas, le comportement fonctionnel est différent suivant qu'on appelle une opération métier sur l'objet d'interposition ou sur l'instance du composant.

Pour se prémunir de mauvaises manipulations entre objet d’interception et instance de composant, l’instance du composant n’implante pas l’interface métier. Cela évite de diffuser une instance d’un composant (diffusion de `this`) comme un objet supportant l’interface métier du composant. Cette contrainte est levée dans la version 3 de la spécification des EJB.

Outre le code d’interposition définie dans le paragraphe précédent, d’autres interactions exsistent entre le conteneur et les composants qu’il gère. Le conteneur a en effet la charge du cycle de vie des instances de composant et signale à ces instances tout événement relatif à ce cycle. Enfin, un composant accède au contexte dans lequel une de ces instances s’exécute (cf. *Contexte de l’instance* dans la figure 5.10). Cela lui permet d’accéder par exemple au contexte de sécurité dans lequel elle s’exécute ou encore d’accéder à certains services (service transactionnel ou service d’échéancier).

5.6.2 Composants de session

Comme leur nom l’indique, d’un point de vue conceptuel, les composants de session ont pour objectif de représenter une session d’utilisation de l’application au niveau du serveur. Etant donné que l’utilisation de l’environnement J2EE ambitionne pour l’essentiel le support des applications Web, de telles sessions impliquent un utilisateur unique. En effet, le modèle d’application Web correspond à l’interaction entre un utilisateur unique et l’application qu’il utilise, généralement à travers un navigateur Web.

Dans le cadre des applications Web, les composants session sont souvent utilisés pour mettre en œuvre le patron d’architecture FAÇADE introduit dans [Gamma et al. 1994]. En effet, parmi les règles de bonne utilisation de J2EE, on retrouve souvent ce patron qui permet de présenter un point d’entrée unique aux applications utilisant ce code métier. Cela simplifie notamment l’utilisation de ce code dans un contexte réparti, en limitant le nombre d’objets qui peuvent être engagés dans des interactions à distance.

L’unicité de l’utilisateur dans le cadre d’une session applicative est un des défauts majeurs du modèle car d’autres formes de sessions existent où peuvent interagir plusieurs utilisateurs. C’est le cas par exemple des sessions de discussion interactives de type *chat* ou de téléphonie impliquant deux utilisateurs, voir des sessions de conférence, de travail coopératif ou de jeux en réseau impliquant potentiellement plus de deux utilisateurs. Cette contrainte spécifiée par le modèle de composant session des EJB sert de garde-fou pour la mise en œuvre de session Web (contexte d’exécution mono-programmé dont le comportement correspond à un enchainement séquentiel d’une page HTML dynamique à une autre). Elle n’est pas contraignante vis-à-vis des autres aspects techniques pris en charge (il supporte généralement la multi-programmation) par ces composants et pourrait donc être facilement levée pour traiter d’autres types de session.

Sessions sans état

Les spécifications EJB distinguent deux types de composants de session : les composants de session sans état et ceux avec état. Concernant la première catégorie, elle sert à supporter des processus métier sans mémoire. Ces processus peuvent être mis en œuvre localement ou à distance à l’aide de RMI ou de Web Services.

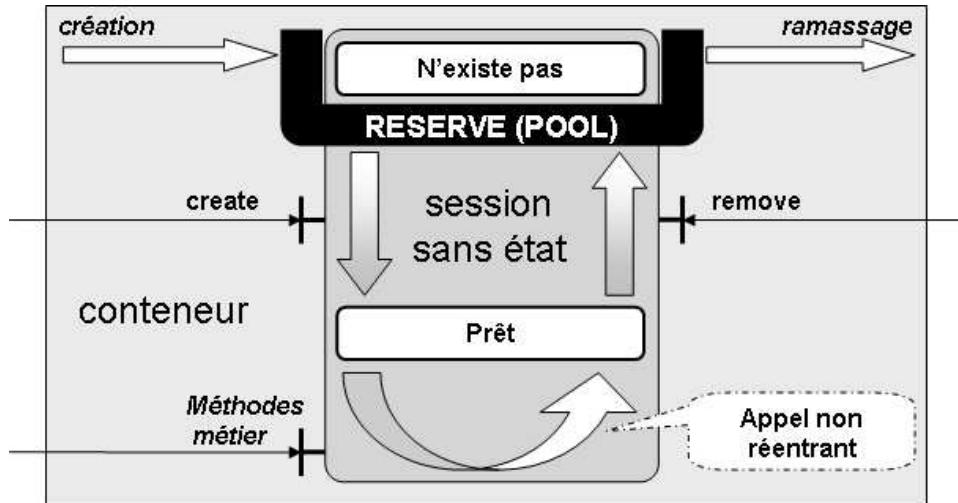


Figure 5.11 – Le cycle de vie d'un composant métier EJB de session sans état

Comme le montre la figure 5.11, un composant de session sans état est relativement primitif dans son fonctionnement. Des instances sont allouées pour effectuer des opérations dans le cadre d'une session et libérées ensuite. Leur cycle de vie contient deux états : l'état non alloué ou non existant et l'état prêt dans lequel les opérations de son interface métier peuvent être activées.

Deux aspects techniques peuvent être pris en charge pour ces composants : le support des transactions et le support de l'accès distant. Pour les transactions, le niveau de transparence offert permet de caler la démarcation transactionnelle sur l'activation d'une opération de l'interface d'un composant : le démarrage de la transaction s'opère à l'appel de l'opération et la terminaison à la sortie de l'opération. Différents comportements transactionnels peuvent être définis :

- **NotSupported** : si le *thread* appelant l'opération avec ce comportement transactionnel est associé à un contexte transactionnel, celui-ci est suspendu le temps d'exécuter cette opération.
- **Required** : si le *thread* appelant l'opération avec ce comportement transactionnel n'est associé pas à un contexte transactionnel, un nouveau contexte est activé le temps d'exécuter cette opération. Dans le cas contraire, l'opération s'exécute dans le contexte transactionnel courant sans toucher à l'association entre le *thread* et le contexte transactionnel.
- **RequiresNew** : le *thread* appelant l'opération avec ce comportement transactionnel suspend le contexte transactionnel courant s'il existe et associe un nouveau contexte transactionnel le temps d'exécuter cette opération.
- **Mandatory** : si le *thread* appelant l'opération avec ce comportement transactionnel n'est pas associé à un contexte transactionnel, une exception est levée par le conteneur.
- **Supports** : si le *thread* appelant l'opération avec ce comportement transactionnel est associé à un contexte transactionnel, cette opération s'exécute dedans sans toucher à l'association entre le *thread* et le contexte transactionnel.

- **Never** : si le *thread* appelant l’opération avec ce comportement transactionnel est associé à un contexte transactionnel, une exception est levée par le conteneur.

Concernant le support des accès distants, il est explicite dans le modèle de programmation : l’interface métier doit étendre l’interface `java.rmi.Remote`, les méthodes définies par l’interface métier devant supporter l’exception `java.rmi.RemoteException`. Cette distinction faite dans le modèle de programmation entre interface métier locale et distante est très contraignante. Elle est supprimée dans la version 3 de la spécification des EJB.

Le coût d’allocation d’une instance de composant de session peut être important, notamment dans le cas où le composant supporte les accès distants (mise en place de chaîne de liaison complexe faisant intervenir des ressources réseau). Par ailleurs, les instances de ce type de composant sans état ont tendance à avoir une durée de vie courte. Pour pallier le problème lié à ces deux propriétés contradictoires en terme de performance, le conteneur EJB interpose généralement une réserve d’instances de composant dans le cycle d’allocation/libération de celles-ci 2.3.3. Un autre intérêt de cette technique est qu’elle permet d’opérer du contrôle d’admission sur l’application, notamment lorsque ce type de composant est utilisé avec le patron d’architecture FAÇADE. Cette approche est possible parce que ces composants sont sans état et qu’il n’ont donc pas d’état initial implicite, comme ça peut être le cas pour les composants de session avec état décrit dans la section suivante.

Sessions avec état

Les composants de session avec état supportent exactement les mêmes fonctions et aspects techniques que ceux sans état. La différence est qu’ils possèdent une mémoire liée à l’activité d’une session. De ce fait, ces instances ont tendance à avoir une durée de vie plus longue que celles sans état. De plus, les instances d’un tel composant ont une identité propre

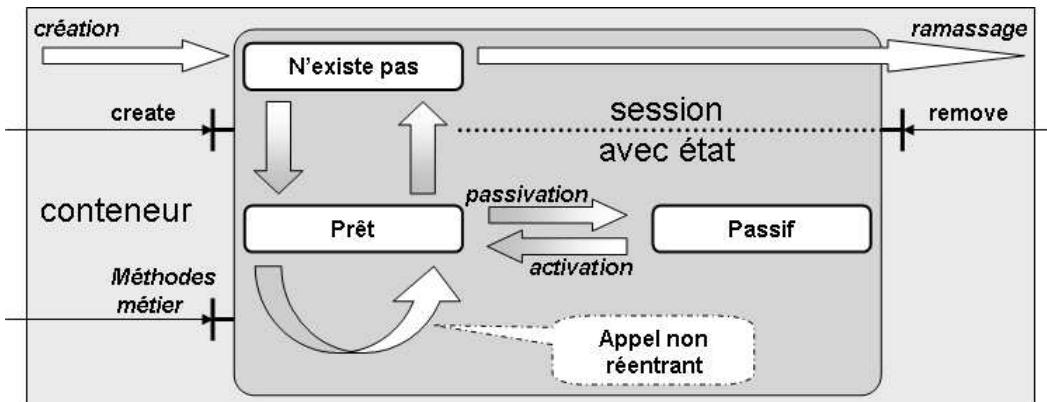


Figure 5.12 – Le cycle de vie d’un composant métier EJB de session avec état

Etant données les propriétés que doivent supporter ces composants, il est proposé de pouvoir les *passiver* de façon à éviter d’emcombrer la mémoire du serveur dans le cas d’un nombre important de sessions à gérer en parallèle. Cette étape, correspondant à un nouvel état dans le cycle de vie du composant de session (cf. figure 5.12), implique que lors d’une

saturation de la mémoire du serveur, certaines instances de composants de session avec état peuvent être vidées dans un espace mémoire secondaire, le choix de ces instances dépendant de politiques de gestion de ressources mises en œuvre par le serveur, par exemple choix des instances les moins récemment utilisées (en anglais *Least Recently Used* ou LRU).

5.6.3 Composants réactifs

Les composants EJB réactifs (en anglais *Message Driven Bean* ou MDB) sont très proches des composants de session sans état. En effet, ils sont eux-mêmes sans état (neutralité de l'instance traitant un message) et leur cycle de vie présente les deux mêmes états. Dans l'état prêt, le composant réactif ne fait que réagir à des événements gérés par le conteneur qui les fait traiter par une instance du composant à l'aide de l'opération `onMessage`.

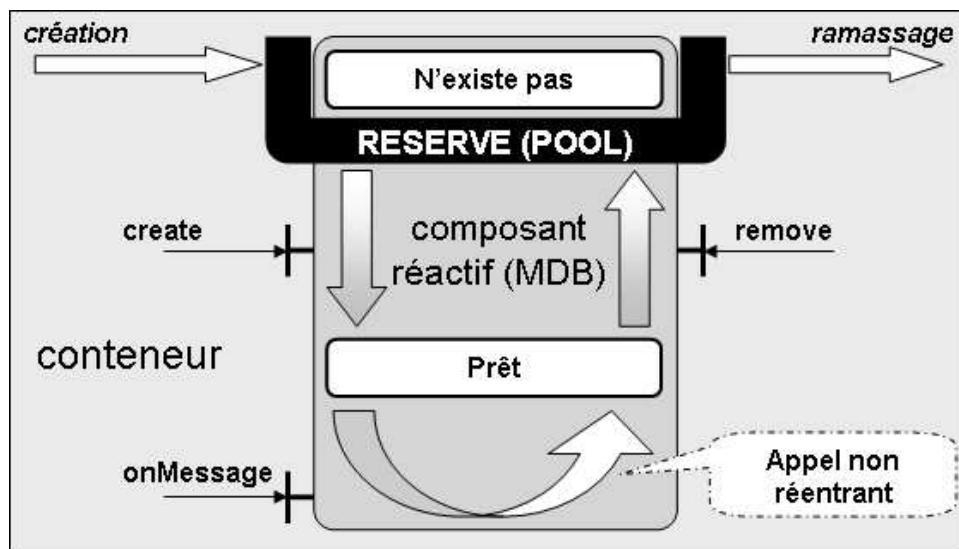


Figure 5.13 – Le cycle de vie d'un composant EJB réactif

Comme pour les composants de présentation (les *servlets*), les composants réactifs ne peuvent pas être invoqués de l'extérieur du conteneur, ce qui n'est pas le cas des composants de session pour lesquels l'allocation et la libération d'instance ou encore l'activation du code métier sont pilotées de l'extérieur du conteneur. Nous verrons que c'est aussi le cas pour les composants entité.

Les composants réactifs supportent des comportements transactionnels plus limités que les composants de session. En fait, seuls deux comportements transactionnels peuvent être spécifiés pour l'opération `onMessage` :

- **Required** : un contexte transactionnel est alloué et est associé au *thread* utilisé pour exécuter `onMessage` le temps de son exécution. Le message est normalement consommé de manière transactionnelle (dépendant des capacités techniques de l'émetteur de message). Cela signifie que si la transaction exécutée par `onMessage` échoue, une nouvelle tentative de consommation du message sera effectuée ultérieurement (nouvelle activation de `onMessage`).

- `NotSupported` : l'opération `onMessage` est exécutée hors de tout contexte transactionnel et consomme le message qui a été délivré.

5.6.4 Composants entité

Les composants entité sont des composants qui représentent des informations persistantes d'une application d'entreprise, c'est-à-dire des informations dont la durée de vie est supérieure à la session qui les a créées ou même à l'application elle-même (l'information persiste à l'arrêt de l'application). Ces informations sont généralement stockées dans une base de données relationnelle et manipulées à travers l'API JDBC (pour *Java DataBase Connectivity*). Cette interface fournit le niveau d'abstraction correspondant aux fonctions d'une base de données relationnelle. Or, l'environnement EJB ambitionne de fournir une vue objet métier de ces données relationnelles au niveau du modèle de programmation. C'est l'objectif des composants entité que de fournir cette couche d'adaptation des objets applicatifs métier vers leur stockage relationnel.

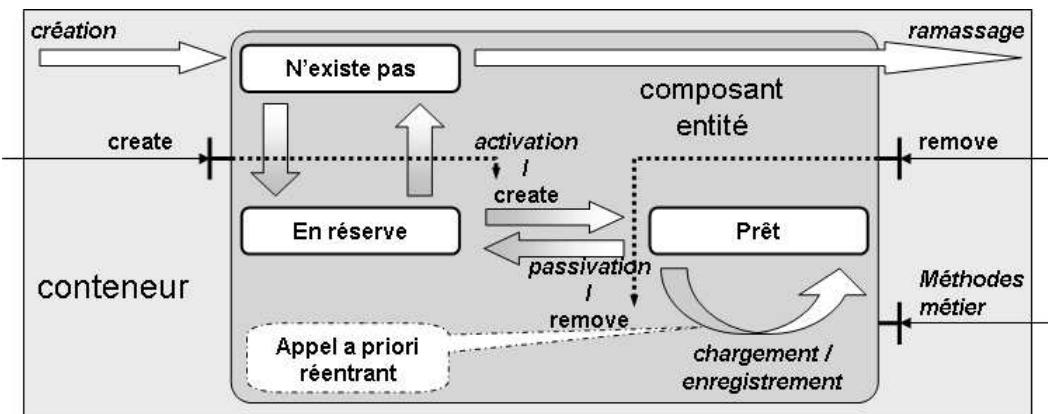


Figure 5.14 – Le cycle de vie d'un composant EJB entité

Le cycle de vie d'un composant entité est assez semblable aux précédents. La principale différence consiste à exhiber le passage par une réserve d'instances pour leur allocation. Étant donné que ces instances peuvent être manipulées en grand nombre et de manière récurrente d'une transaction à l'autre, il est important d'optimiser leur processus d'allocation. En fait, les instances qui sont dans la réserve ont a priori exécuté tout leur cycle de configuration (mise en liaison avec les ressources dont elles dépendent). Lors de l'allocation d'une instance, le conteneur en cherche une disponible dans la réserve avant d'allouer un nouvel objet Java à configurer.

Pour passer dans l'état *Prêt*, il faut qu'une instance soit liée à son représentant dans la base (en général un n-uplet dans une table). Cette liaison est définie par le nom dans la base de ce représentant, c'est à dire sa clé primaire. Cette liaison peut être créée de plusieurs manières :

- Création d'une nouvelle instance : la création d'une nouvelle instance de composant entité crée implicitement un représentant dans la base et donc provoque la création d'un nouveau nom définissant la liaison.

- Navigation à travers une référence vers un objet : lorsque le conteneur traverse une référence vers une autre objet persistant, il doit alors activer une instance de composant entité si elle n'existe pas déjà. Il va alors allouer cette instance et lui associer le nom de l'objet à atteindre en récupérant le nom de son représentant dans l'état de l'objet référençant, définissant ainsi la liaison.
- Parcours du résultat d'un *finder* : un *finder* permet de récupérer le résultat d'une requête EJB-QL qui est généralement une collection d'instances d'un composant entité. La mécanique sous-jacente émet vers la base de données une requête SQL qui remonte la collection de noms des représentants sélectionnés. A partir de là le conteneur peut définir les liaisons entre les instances et leur représentant lorsque l'utilisateur parcourt sa collection d'instances.

Une fois une liaison établie d'une des manières précédentes, le conteneur prend en charge la synchronisation entre l'instance et son représentant en base de données de façon transparente pour le programmeur (cf. actions *chargement* et *enregistrement* dans la figure 5.14 équivalent en anglais des actions *load* et *store*). L'action de *chargement* est généralement exécutée après la mise en liaison et avant tout autre appel vers des opérations métier. L'opération inverse d'*enregistrement* intervient quant à elle soit lors de la validation de transaction, soit lors d'opérations de passivation (voir les paragraphes qui suivent).

Les instances ne retournent dans la réserve ou ne sont détruites qu'après une des deux opérations suivantes :

- Exécution de *remove* déclenchée depuis le code applicatif : cette opération provoque la destruction du représentant de la base de données et donc du nom associé. La liaison avec l'instance de composant est rompue et cette instance retourne alors dans la réserve.
- Opération de passivation déclenchée par le conteneur : lorsqu'il considère que la mémoire est trop encombrée, le conteneur peut décider d'en éliminer certaines instances de composants entité, suivant des politiques qui lui sont propres (par exemple *LRU* comme précédemment pour les composants de session avec état). Dans ce cas, l'instance est généralement synchronisée avec son représentant si nécessaire, déliée et renvoyée dans la réserve, voire ramassée par la JVM.

Outre la transparence de projection entre composants entité et base de donnée, les transactions sont supportées de la même manière que pour les composants de session au niveau des opérations métier. Cette fonction est en principe peu utilisée dans ce cadre car les transactions sont des séquences d'opérations opérant à une autre granularité (processus métier). Néanmoins, les composants entité peuvent aussi être utilisés pour supporter des sessions persistantes (cas de sessions particulièrement longues).

5.6.5 Gestion des événements relatifs aux composants métier

Les composants métier ont la possibilité de récupérer un certain nombre d'événements qui peuvent être utiles à leur logique propre. Trois catégories d'événements sont décrites dans les sous-sections qui suivent. Dans tous les cas, les événements sont gérés/émis par le conteneur vers des instances de composants métier. Le même patron architectural est utilisé : les composants intéressés par des événements doivent planter une interface spécifique qui permet au conteneur d'activer l'instance de composant intéressée pour qu'elle traite l'événement en question. Dans le cas de la version 3 de la spécification, il n'y a

pas besoin d'implanter des interfaces. Il suffit de définir une correspondance entre des opérations du composant et les événements qui peuvent être traités.

Evénements relatifs au cycle de vie des instances de composant

Les premiers événements qui peuvent être traités par les instances de composants sont ceux relatifs leur cycle de vie qui est géré par le conteneur. Ils dépendent du type de composants. Ils sont spécifiés par des interfaces associées à chacun de ces types : `javax.ejb.SessionBean` pour les composants de session, `javax.ejb.MessageDrivenBean` pour les composants réactifs et `javax.ejb.EntityBean` pour les composants entité. Comme ces interfaces typent le composant, l'une d'entre elles doit obligatoirement être implantée par un composant métier, même si aucun traitement n'est effectué pour ces événements. Cette contrainte est levée dans la version 3 de la spécification EJB.

Pour les composants de session, quatre événements sont émis par le conteneur. Seulement deux d'entre eux sont émis pour les sessions sans état, à savoir `setSessionContext` et `ejbRemove` :

- `setSessionContext` : cet événement est émis à la création d'une instance de composant de session. Il permet à l'instance de récupérer le contexte qui lui est associé.
- `ejbRemove` : cet événement est émis à la destruction d'une instance de composant de session. L'étape suivante est généralement la prise en charge de cette instance par le ramasse-miette.
- `ejbPassivate` : cet événement est émis lorsqu'une instance de composant de session avec état va être rendu passif (stockage dans un espace mémoire secondaire).
- `ejbActivate` : cet événement est émis lorsqu'une instance de composant de session avec état est réactivée, c'est ramenée en mémoire principale avec restauration de son suivant les règles définies par la spécification EJB (un composant de session avec état doit être `Serializable` et les variables définies comme `transient` peuvent aussi être sauvegardées et restaurées sous certaines conditions).

Pour les composants réactifs, deux événements seulement sont émis par le conteneur :

- `setMessageDrivenContext` : similaire à l'événement `setSessionContext` des composants de session.
- `ejbRemove` : similaire à l'événement `ejbRemove` des composants de session.

Pour les composants entité, sept événements sont émis par le conteneur :

- `setEntityContext` : similaire à l'événement `setSessionContext` des composants de session.
- `unsetEntityContext` : similaire à l'événement `ejbRemove` des composants de session.
- `ejbRemove` : l'événement est émis lorsque le représentant de stockage associé à cette instance est éliminé de la base de données.
- `ejbLoad` : l'événement est émis lorsque l'état de l'instance de composant est chargé de la base de données vers la mémoire principale.
- `ejbStore` : l'événement est émis lorsque l'état de l'instance de composant est enregistré dans la base de données à partir de la mémoire principale.
- `ejbPassivate` : similaire à l'événement `ejbPassivate` des composants de session, mise à part que l'espace secondaire vers lequel est envoyé l'état de l'instance est la base de données sous-jacente, et notamment le représentant de stockage de l'instance en question.

- **ejbActivate** : similaire à l'événement **ejbActivate** des composants de session, mise à part que l'espace secondaire depuis lequel l'état de l'instance est réactivé est la base de données sous-jacente, et notamment le représentant de stockage de l'instance en question.

La définition de ces événements de même que celle des cycles de vie aurait mérité un traitement plus systématique. En effet, certains événements communs sont définis avec le même patron de nommage (pourquoi pas le même nom...) comme **set...Context**. D'autres événements communs changent de nom suivant le type de composant (par exemple **ejbRemove** des composants de session et des composants réactifs devient **unsetEntityContext** pour les composants entité, pour lesquels **ejbRemove** prend une autre sémantique). Concernant les cycles de vie, ceux des composants réactifs et des composants de session sans état sont exactement les mêmes, sachant que le cycle de vie des composants de session avec état étend le précédent, de même que celui des composants entité étant lui-même celui des sessions avec état.

Evénements relatifs aux transactions

Il est possible d'être averti des événements relatifs aux démarcations transactionnelles, notamment lorsque celles-ci sont prises en charge de façon transparente par le conteneur. Le composant doit pour cela implanter l'interface **javax.ejb.SessionSynchronization** qui définit les événements suivants :

- **afterBegin** : l'événement est émis avant l'exécution de n'importe quelle opération métier lorsqu'une instance de composant s'exécute dans un nouveau contexte transactionnel.
- **beforeCompletion** : l'événement est émis lorsqu'une instance de composant s'exécute dans un contexte transactionnel qui est sur le point d'être validé ou annulé.
- **afterCompletion** : l'événement est émis lorsqu'une instance de composant s'exécute dans un contexte transactionnel qui vient d'être validé ou annulé.

Ces événements peuvent être émis seulement dans le cadre des composants de session avec état. Là encore, le manque de vision systématique est dommageable car de tels événements pourraient aussi bien intéresser d'autres types de composants qui supportent la démarcation gérée par le conteneur (en fait, tous les composants EJB !!). Il est toujours possible d'obtenir cette fonction en interagissant directement avec le service transactionnel pour qu'une instance de composant s'enregistre comme une *Synchronization*.

Evénements temporels (horloges / échéanciers)

Il est possible d'associer des événements temporels aux composants EJB. Pour que les composants supportent de tels événements, il faut qu'ils implantent l'interface **javax.ejb.TimedObject**. Lors de l'échéance d'un événement temporelle, l'opération **ejbTimeout** de l'interface en question est invoquée par le conteneur, avec en paramètre l'événement temporel arrivé à échéance.

Deux types d'échéancier peuvent être définis. Le premier type permet de définir une échéance unique, c'est à dire qu'un seul événement est émis à l'échéance. Le second type permet de définir des échéanciers périodiques, c'est à dire qu'un événement est émis à période fixe.

Le service d'échéanciers (en anglais *Timer Service*) est accessible aux composants (sauf pour les composants de session avec état qui ne sont pas pris en compte dans la version 2.1 de la spécification EJB) ou plus précisément à leurs instances à travers leur contexte. Il permet donc de créer de nouveaux échéanciers qui sont associés aux instances de composants de la manière suivante :

- Pour les composants de session sans état et les composants réactifs : les échéanciers créés ne sont en fait pas associés à une instance particulière puisque dans le cas de ces composants, il n'y a pas de distinction entre les instances. Lorsqu'un événement temporel est émis par le conteneur, celui-ci active une instance quelconque du composant pour le traiter.
- Pour les composants entité : les échéanciers sont effectivement associés à l'instance qui les crée. En fait, l'association est faite avec l'identifiant (basé sur la clé primaire) de l'instance. Cela permet au conteneur d'activer la bonne instance de composant pour traiter les échéances d'événements.

Dans tous les cas, les échéanciers ainsi définis sont persistants. En effet, ils doivent survivre à l'arrêt ou à la panne du serveur qui les héberge. Si des échéances sont intervenues pendant de tels arrêts, tous les événements correspondants doivent être émis lors du redémarrage.

On comprend bien que cette approche ne peut pas être utilisée pour les composants de session avec état car les instances de ceux-ci ont une identité propre mais ne survivent pas à l'arrêt du serveur qui les héberge. Des échéanciers persistants n'ont donc pas de sens pour de tels composants. Leur support dans de futures versions de la spécification ne doit néanmoins pas poser de problème.

5.7 Conclusion et perspectives

Ce chapitre présente une synthèse des principaux principes et des principales fonctionnalités de l'environnement J2EE. Cette synthèse se concentre sur les fonctions qui permettent de développer et d'intégrer des applications d'entreprise avec des garanties de sûreté de fonctionnement (support des transactions et notamment des transactions réparties) et de sécurité (support de l'authentification et du contrôle d'accès).

La synthèse insiste aussi sur les principes de d'architecture et de construction du code proposés par l'environnement J2EE. Le support de composant est omniprésent dans la solution proposée même s'il n'est pas appliqué de façon homogène et systématique (approche ad hoc avec différentes notions de composant suivant les spécifications considérées telles que *servlet* ou *EJB*).

On peut néanmoins affirmer que la notion de composant J2EE est un relatif échec. En effet, il n'existe pratiquement pas d'environnement de développement permettant d'effectuer de l'assemblage d'applications à partir de bibliothèques de composants J2EE existant.

Par ailleurs, la mise en œuvre des modèles de composants J2EE est considérée comme lourde par de nombreux développeurs, même si les environnements de développement tels que Eclipse (cf. <http://www.eclipse.org>) ou NetBeans (cf. <http://www.netbeans.org>) exhibent des fonctions avancées pour faciliter ces développements. La facilité de programmation est d'ailleurs un des principaux axes d'amélioration à l'œuvre dans la version 1.5 de J2EE qui doit être finalisée bientôt. C'est

notamment le cas pour les EJB où la version 3.0 des spécifications simplifie largement le développement.

Chapitre 6

La plate-forme .NET

.NET¹ est en passe de devenir une technologie fréquemment utilisée pour la réalisation de projets informatiques. Dans ce chapitre nous abordons les principaux principes de cette plate-forme, diverse et parfois complexe intégrant les standards actuels. La section d'introduction met en évidence la complétude et la richesse de l'ensemble des bibliothèques de la plate-forme .NET. Elle nous permet de fixer le vocabulaire pour différencier les éléments de la plate-forme normalisés par Microsoft de ceux qui sont restés propriétaires afin de pouvoir très rapidement comparer les différentes plates-formes de développement. La section 6.2 insiste sur l'architecture générale de la plate-forme et sur ses principales caractéristiques. La section 6.2.4 présente le modèle de composants et la machine virtuelle permettant leur exécution. Les trois sections suivantes font respectivement le lien entre les composants .NET et les services techniques (6.4), l'étage d'accès aux données (6.4.1) et l'étage de présentation (6.3.2) présents dans toute architecture à N étages (N-tier). La section 6.3 est constituée d'un petit exemple permettant d'illustrer les principaux principes de la plate-forme. Le chapitre se termine par une évaluation qualitative et quantitative et par une présentation de l'évolution future de cette plate-forme.

6.1 Historique, définitions et principes

6.1.1 Généralité

Microsoft .NET est une architecture logicielle destinée à faciliter la création, le déploiement des applications en général, mais plus spécifiquement des applications Web ou des services Web. Cette architecture logicielle concerne aussi bien les clients que les serveurs qui vont dialoguer entre eux à l'aide d'XML. Elle se compose de quatre principaux éléments :

- un modèle de programmation qui prend en compte les problèmes liés aux déploiements des applications (gestion de version, sécurité) ;

¹une partie de ce texte a été publiée dans les *Techniques de l'Ingénieur* - Traité “Technologies logicielles et architecture des systèmes” - vol. H3540 - février 2006

- des outils de développement dont le cœur est constitué de Visual Studio .NET ;
- un ensemble de systèmes serveurs représentés par Windows Server 2003, SQL Server ou Microsoft Biztalk Server qui intègrent, exécutent et gèrent les services Web et les applications ;
- un ensemble de systèmes clients représenté par Windows XP, Windows CE ou Microsoft Office 2003.

En fait, Microsoft .NET est essentiellement un environnement de développement et d'exécution reprenant, entre autres, les concepts de la machine virtuelle de Java, via le CLR (*Common Language Runtime*). Le principe, bien connu, est le suivant : la compilation du code source génère un objet intermédiaire dans le langage MSIL (*Microsoft Intermediate Language*) indépendant de toute architecture de processeur et de tout système d'exploitation hôte. Cet objet intermédiaire, est ensuite compilé à la volée, au sein du CLR, au moyen d'un compilateur JIT (*Just In Time*) qui le transforme en code machine lié au processeur sur lequel il réside. Il est alors exécuté.

Au contraire de Java, pour lequel le code intermédiaire (*byte-code*) est lié à un seul langage source, le code intermédiaire de la plate-forme .NET est commun à un ensemble de langages (C#, VB², C++, etc.). Le CLR en charge de son exécution contient un ensemble de classes de base liées à la gestion de la sécurité, gestion de la mémoire, des processus et des *threads*.

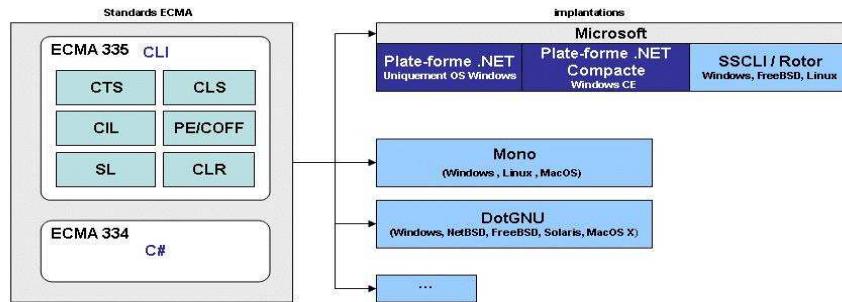


Figure 6.1 – Les différentes plates-formes .NET

La figure 6.1 résume les différents aspects de ces normes qui concernent le langage de programmation C# “inventé” pour l'occasion ³ et la plupart des interfaces internes ⁴ de la plate-forme. Cette même figure présente les principales implantations de ces spécifications.

²VB.NET est la nouvelle version de Visual Basic, il intègre la notion d'objets qui existe dans .NET et les programmeurs de l'actuelle version 6.0 sont invités à faire migrer leurs codes

³ECMA 334 puis ISO/IEC 23270 : C# Langage Specification

⁴ECMA 335 puis ISO/IEC 23271 : Common Language Interface (CLI)

6.1.2 Implantations

Rotor - SSCLI

Le code source partagé CLI (*Common Language Infrastructure*)⁵ [Stutz et al. 2003] correspond à une implémentation fonctionnelle des spécifications du langage C# et de la CLI normalisées par l'ISO. Cette implémentation va au delà de ces spécifications en proposant un environnement d'exécution, des bibliothèques de classes, le compilateur C# et JScript, un débogueur, des outils, la couche PAL (Platform Adaptation Layer), une version portable du système et portée sur Windows XP, FreeBSD et MacOS X, ainsi qu'une suite complète de test. Elle est proposée à des fins non commerciales via la licence SSCLI dont les principales caractéristiques sont :

- le logiciel doit être utilisé à des fins non commerciales (enseignement, recherche universitaire ou expérimentations personnelles). Il peut toutefois être distribué au sein d'ouvrages ou autres supports de cours liés à l'enseignement, ou publié sur des sites Web dont le but est de former à son utilisation.
- toutes utilisations du logiciel à des fins commerciales sont prohibées.
- le logiciel peut être modifié et le résultat de ces modifications peut être distribué dans un but non commercial à la condition de ne pas accorder de droit supplémentaire.
- le logiciel est livré sans garanties.

Autres implantations

Suite à l'adoption officielle de ces deux standards par l'ECMA en décembre 2001, et plus tard par l'ISO2 en avril 2003, la réaction ne se fit pas attendre. Différentes implantations du CLI basées sur la spécification ECMA ont vu le jour permettant ainsi l'exécution et le développement d'applications .NET sur des systèmes d'exploitation non Microsoft comme Linux, FreeBSD et MacOS. Nous pouvons par exemple citer deux initiatives logiciels libres (*open-source*), indépendantes de Microsoft, et ayant une réelle importance : Mono⁶ [Edd Dumbill and M. Bornstein 2004] et DotGNU⁷ [Weatherley and Gopal 2003].

6.2 Architecture générale et concepts

6.2.1 CLR

Le composant central de l'architecture .NET est le CLR (*Common Language Runtime*) [Stutz et al. 2003] qui est un environnement d'exécution pour des applications réparties écrites dans des langages différents. En effet, avec .NET on peut véritablement parler d'interopérabilité entre langages. Cette interopérabilité s'appuie sur le CLS (*Common Language Specification*) qui définit un ensemble de règles que tout compilateur de la plate-forme doit respecter. Le CLS utilise entre autres un système de types unifié permettant d'avoir le même système de type entre les différents langages et le même système de type entre les types prédéfinis et les types définis par l'utilisateur.

⁵Rotor : <http://msdn.microsoft.com/net/sscli/>

⁶Mono : <http://www.go-mono.org>

⁷DotGNU : http://www.southern-storm.com.au/portable_net.html

Un autre concept majeur, lié au CLR est celui de code géré, c'est à dire de code exécuté sous le contrôle de la machine virtuelle. Dans cet environnement, un ensemble de règles garantissent que les applications se comporteront d'une manière uniforme, et ce indépendamment du langage ayant servi à les écrire.

Pour répondre à ce souci d'interopérabilité entre les langages, la plate-forme .NET contient une bibliothèque de classes très complète, utilisable depuis tous les langages et permettant aux développeurs d'utiliser une même interface de programmation pour toutes les fonctions offertes.

Dans .NET, le langage lui-même est essentiellement une interface syntaxique des bibliothèques définies dans le CLR. Grâce au jeu commun de ces bibliothèques, tous les langages disposent théoriquement des mêmes capacités car ils doivent, exception faite de la déclaration des variables, passer par ces bibliothèques pour créer des *threads*, les synchroniser, sérialiser des données, accéder à internet, etc.

6.2.2 MSIL et JIT

Pour permettre la compilation de différents langages sur la plate-forme .NET, Microsoft a défini une sorte de langage d'assemblage, indépendant de tout processeur baptisé MSIL (*Microsoft Intermediate Language*). Pour compiler une application destinée à .NET, le compilateur concerné lit le code source dans un langage respectant les spécifications CLS, puis génère du code MSIL. Ce code est traduit en langage machine lorsque l'application est exécutée pour la première fois. Le code MSIL produit contient un en-tête au standard PE (Win-32 Portable Executable) qui permet de différencier les exécutables .NET des autres exécutables de l'OS Windows en chargeant au démarrage de l'application un ensemble de DLL spécifiques précisées dans cet en-tête.

Pour compiler à l'exécution le code MSIL, la plate-forme .NET utilise un compilateur à la volée (JIT Compiler - *just-in-time compiler*) qui place le code produit dans un cache. Sur la plate-forme .NET, trois JIT sont disponibles :

- Génération de code à l'installation : le code MSIL est traduit en code machine lors de la phase de compilation comme le fait tout compilateur.
- JIT : les méthodes sont compilées une par une à la volée lors de leur première exécution, ou lorsqu'elles ne sont plus en cache.
- EconoJIT : jit permettant de minimiser la place mémoire utilisée pour exécuter un programme et qui sera utilisé pour les équipements portables.

Les compilateurs de la plate-forme .NET incorporent dans l'exécutible produit différentes métadonnées. Ces métadonnées décrivent l'exécutible produit et lui sont systématiquement incorporées pour pouvoir être atteintes lors de l'exécution. Il est ainsi possible dynamiquement de connaître les différents types déclarés par un exécutible, les méthodes qu'il implémente, etc. La lecture des métadonnées s'appelle réflexion (voir 2.4) et la plate-forme .NET fournit de nombreuses classes permettant de manipuler de manière réflexive les exécutables. Voici un exemple de code qui crée dans un premier temps un exécutable et regarde par la suite les métadonnées.

Fichier calculatrice.cs

```
using System;
```

```
class MaCalculatrice {
    public int Add(int a, int b) {
        return a+b;
    }
    public int Sub(int a, int b) {
        return a-b;
    }
}
```

Fichier application.cs

```
public class MonApplication {
    static void Main (string[] args) {
        MaCalculatrice calculette = new MaCalculatrice ();
        int i = 10, j = 20;
        Console.WriteLine ("{i} + {j} =", i, j, calculette.Add (i, j));
    }
}
```

La compilation de l'application, s'effectue par étape :

- génération de la bibliothèque calculatrice.dll
- génération de l'exécutable application.exe

```
% csc.exe /target:library /out:calculatrice.dll calculatrice.cs
% csc.exe /target:exe /reference:calculatrice.dll application.cs
```

```
using System;
using System.IO;
using System.Reflection;

public class Meta {
    public static int Main () {
        // lire l'assembly
        Assembly a = Assembly.LoadFrom ("application.exe");

        // lire tous les modules de l'assembly
        Module[] modules = a.GetModules();

        // inspecter tous le premier module
        Module module = modules[0];

        // lire tous les types du premier module
        Type[] types = module.GetTypes();

        // inspecter tous les types
        foreach (Type type in types) {
            Console.WriteLine("Type {0} a cette méthode : ", type.Name);

            // inspecter toutes les méthodes du type
            MethodInfo[] mInfo = type.GetMethods();
```

```

        foreach (MethodInfo mi in mInfo) {
            Console.WriteLine (" {0}", mi);
        }
    }
    return 0;
}
}

```

La compilation de l'application permettant de lire l'application “application.exe” puis son exécution, s'effectue de la manière suivante :

```

% csc.exe meta.cs
% meta.exe

Type MaCalculatrice a cette méthode :
Int32 Add(Int32, Int32)
System.Type GetType()
System.String ToString()
Boolean Equals(System.Object)
Int32 GetHashCode()
Type MonApplication a cette méthode :
System.Type GetType()
System.String ToString()
Boolean Equals(System.Object)
Int32 GetHashCode()

```

L'outil ILDASM (*Intermediate Language Disassembler*) qui permet d'afficher de manière hiérarchique les informations sur une application utilise essentiellement la réflexion. La figure 6.1 montre à quoi ressemble le programme précédent sous ILDASM.

6.2.3 Sécurité et déploiement des composants

La sécurité est cruciale pour toutes les applications. Elle s'applique dès le chargement d'une classe par le CLR, via la vérification de différents éléments liés aux règles de sécurité définies dans la plate-forme : règles d'accès, contraintes de cohérence, localisation du fichier, etc. Des contrôles de sécurité garantissent que telle ou telle portion de code est habilitée à accéder à telle ou telle ressource. Pour cela, le code de sécurité vérifie rôles et données d'identification. Les contrôles de sécurité peuvent être effectués entre processus ou entre machines afin d'assurer la sécurité même dans un contexte distribué.

Windows nous a habitués à un déploiement hasardeux pour lequel il est nécessaire de gérer de multiples fichiers binaires, la base de registres, des composants COM, des bibliothèques comme ODBC. Fort heureusement, .NET change cette approche et le déploiement des applications repose sur la copie dans un répertoire spécifique, le GAC (*Global Assemblies Cache*), ou dans l'arborescence utilisateur, des fichiers exécutables produits. Ces fichiers exécutables (mais pas toujours) “assembly⁸” sont un paquet regroupant des éléments devant être déployé simultanément. Ce sont généralement des fichiers exécutables répartis sur un ou plusieurs fichiers physiques. Les problèmes antérieurs à .NET ont été éliminés et il n'est plus nécessaire d'enregistrer des composants dans la base de

⁸Un assembly est une unité de déploiement.

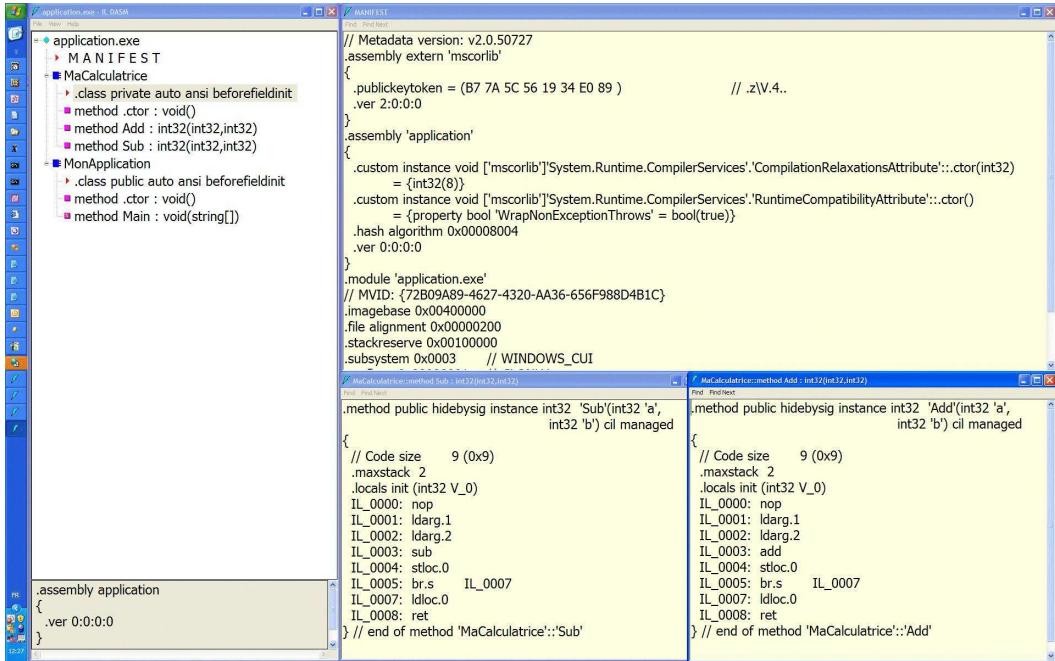


Figure 6.2 – Le programme de 6.2.2 analysé avec ILDASM

registre. .NET permet aussi de gérer dans le GAC plusieurs versions du même fichier et de rediriger à la demande les appels vers un assemblage vers une version plus récente si c'est souhaité par le concepteur de l'application. En effet, grâce aux métadonnées et à la réflexion, tous les composants sont désormais auto-descriptifs, y compris leur version. Cela à pour conséquence, que toute application installée est automatiquement associée aux fichiers faisant partie de son assemblage.

Une partie de la sécurité du code repose sur la possibilité de signer celui-ci à l'aide d'une clé privée afin de pouvoir identifier ultérieurement la personne ou l'organisation qui a écrit ce code à l'aide de la clé publique associée. Si l'exécution dans le répertoire courant ne nécessite pas la signature du code, toute publication dans le GAC ainsi que la gestion des versions dans celui-ci le nécessite.

6.2.4 Composants .NET

Composants

La brique de base d'une application .NET est le composant (littéralement assemblage ou *assembly*). Il peut s'agir d'un exécutable (.exe) ou d'une bibliothèque (.dll) contenant des ressources, un manifeste, des métadonnées, des types et du code exécutable sous la forme d'une liste d'instructions en langage intermédiaire. Ces métadonnées sont utilisées par l'environnement d'exécution (run-time). La définition des classes peut être obtenue directement à partir du composant, en examinant les métadonnées. Au contraire de l'approche Java RMI et de Corba, aucun autre fichier n'est nécessaire pour utiliser un composant et par conséquent, aucun fichier IDL, aucune bibliothèque de type ou couple talon client/ser-

veur (*stub/skeleton*) n'est nécessaire pour accéder à un composant depuis un autre langage ou un autre processus. Il n'est pas nécessaire de déployer des informations de configuration séparées pour identifier les attributs de services requis par le développeur. Plus important encore, les métadonnées étant générées à partir du code source durant le processus de compilation et stockées avec le code exécutable, elles ne sont jamais désynchronisées par rapport à celui-ci.

Le rôle du composant est de plusieurs natures.

- C'est une unité de déploiement unique. C'est une unité d'exécution unique. Chaque composant est chargé dans un domaine d'application différent lui permettant ainsi d'être isolé des autres composants de l'application.
- Il intègre un mécanisme de gestion de version spécifique.
- Il permet également de définir la visibilité des types qu'il contient afin d'exposer publiquement ou non certaines fonctionnalités vers d'autres domaines d'application.
- Il permet de sécuriser l'ensemble des ressources qu'il contient de manière homogène car c'est à la fois une unité de déploiement, d'exécution (au sein d'un domaine d'application) et une unité pour la gestion des versions.

Déploiement des composants

Le modèle de déploiement d'applications de la plate-forme .NET règle les problèmes liés à la complexité de l'installation d'applications, à la gestion des versions de DLL dans le monde Windows ou du positionnement de la variable **CLASSPATH** dans le monde Java. Ce modèle utilise la notion de composant qui permet de déployer de façon unitaire un groupe de ressources et de types auquel est associé un ensemble de métadonnées appelé manifeste (*assembly manifest*).

Le manifeste comprend entre autres la liste des types et des ressources visibles en dehors du composant et des informations sur les dépendances entre les différentes ressources qu'il utilise. Les développeurs peuvent aussi spécifier des stratégies de gestion des versions pour indiquer si l'environnement d'exécution doit charger la dernière version d'un composant, une version spécifique ou la version utilisée lors de la conception. Ils peuvent aussi préciser la stratégie, spécifique à chaque application, pour la gestion de la recherche et le chargement des composants.

La plate-forme .NET permet le déploiement côté à côté (*side by side*) des composants : plusieurs copies d'un composant, mais pas nécessairement de la même version, peuvent résider sur un même système. Les assemblages peuvent être privés ou partagés par plusieurs applications et il est possible de déployer plusieurs versions d'un composant simultanément sur une même station. Les informations de configuration d'applications définissent l'emplacement de recherche des composants et permettent ainsi à l'environnement d'exécution de charger des versions différentes du même composant pour deux applications différentes exécutées simultanément. Ceci élimine les problèmes relatifs à la compatibilité des versions de composants et améliore la stabilité globale du système.

Si nécessaire, les administrateurs peuvent ajouter aux composants lors du déploiement des informations de configuration telles que des stratégies de gestion des versions différentes. Cependant, les informations d'origine fournies au moment de la conception ne sont jamais perdues. En raison du caractère auto-descriptif des composants, aucun enregistrement explicite auprès du système d'exploitation n'est nécessaire. Le déploiement

d'une application peut se limiter à la copie de fichiers dans une arborescence de répertoires. Les informations de configuration sont stockées dans des fichiers XML qui peuvent être modifiés avec tout éditeur de texte. La tâche se complique légèrement si des composants non-gérés (*unmanaged components*) sont nécessaires au fonctionnement de l'application.

Le terme "*unmanaged*" rencontré ci-dessus fait référence au processus de compilation des composants .NET.

- Soit la compilation du composant respecte toutes les spécifications de la plate-forme .NET et alors l'exécution de ce composant est complètement prise en charge par la machine virtuelle .NET et l'on parle alors de code "*managed*". Le composant associé est lui aussi déclaré "*managed*".
- Soit la compilation du composant ne respecte pas toutes les spécifications de la plate-forme .NET, et c'est particulièrement le cas si l'on souhaite reprendre sans modification du code existant (C ou C++ par exemple) manipulant directement des pointeurs. Alors la machine virtuelle .NET isole ce composant dans un espace spécifique, n'assure pas pour ce composant une gestion mémoire spécifique (pas de ramassage des miettes) et l'on parle alors de code "*unmanaged*" et de composants "*unmanaged*".

Gestion de version

L'installation d'une version de la plate-forme .NET sur une machine conduit à la création d'un cache global pour les composants .NET (GAC : Global Assembly Cache). Ce cache est un répertoire (e.g. `c:\Windows\assembly`) dans lequel se trouvent tous les composants .NET destinés à être partagés entre les différentes applications s'exécutant sur cette machine. Il contient les composants de base de la plate-forme .NET de même que les composants déposés par les différents programmeurs. Le fait d'être placé dans le GAC plutôt que dans le répertoire d'une application donnée permet à un composant d'être partagé et donc d'être localisé de manière unique sans ambiguïté sur la machine. Ce répertoire spécifique peut aussi contenir plusieurs versions d'un même composant ce qui permet de résoudre les problèmes classiques liés à l'utilisation des DLL Windows standard où chaque nouvelle version d'une bibliothèque écrase l'ancienne conduisant à une compatibilité aléatoire des applications.

L'installation d'un composant dans le GAC se fait en plusieurs étapes :

- affectation d'un numéro de version selon un format bien défini (128 bits) `Major.Minor.Build.Revision` (par exemple (2.0.142.20) ;
- signature du composant afin de lui donner un nom unique (*strong name*) ;
- installation du composant dans le GAC ;
- mise au point de la sécurité.

6.2.5 Assemblage de composants .NET

Après une présentation très rapide des systèmes de type de .NET, cette section, reprise du document d'habilitation d'Antoine Beugnard [Beugnard 2005] [Beugnard 2006] est une critique détaillée des aspects "multi-langages" du framework. On compare dans un premier temps le comportement des langages à objets vis-à-vis de la liaison dynamique et dans

un second temps la capacité de la plate-forme à avoir un comportement neutre pour la programmation par composition.

Le système de type CTS

L'un des objectifs de la plate-forme logicielle .NET est d'unifier les processus de développement et en particulier de résoudre le problème de l'hétérogénéité des langages de programmation. Pour atteindre cet objectif, la plate-forme .NET s'appuie sur un système de types commun. Ce système de types est capable d'exprimer la sémantique de la plupart des langages de programmation actuels et peut être vu comme un sur-ensemble des constructions que l'on retrouve dans la plupart des langages comme Java et C++. Cette spécification impose que chaque langage ciblant la plate-forme .NET doit respecter ce système de types. Ceci permet par exemple à un développeur C# d'utiliser une classe écrite par un développeur VB.NET, celle-ci héritant d'une classe Java (ou plutôt J# l'équivalent pour .NET). Chaque langage utilise ce système commun de type qui est appliqué à son propre modèle. Par exemple, chaque langage va utiliser le type `System.Int` afin que tous les int (ceux de C, de C++, de VB, de J# ou de C#) aient la même taille. Le code source est compilé vers un langage intermédiaire unique (MSIL : Microsoft Intermediate Language) pour le moniteur d'exécution .NET (run-time).

A l'exécution, le langage d'origine importe peu, et par exemple, des exceptions peuvent être signalées à partir de code écrit dans un langage et interceptées dans du code écrit dans un autre langage. Les opérations telles que la mise au point ou le profilage fonctionnent indépendamment des langages utilisés pour écrire le code. Les fournisseurs de bibliothèques n'ont ainsi plus besoin de créer des versions différentes pour chaque langage de programmation ou compilateur et les développeurs d'applications ne sont plus limités aux bibliothèques développées pour le langage de programmation qu'ils utilisent.

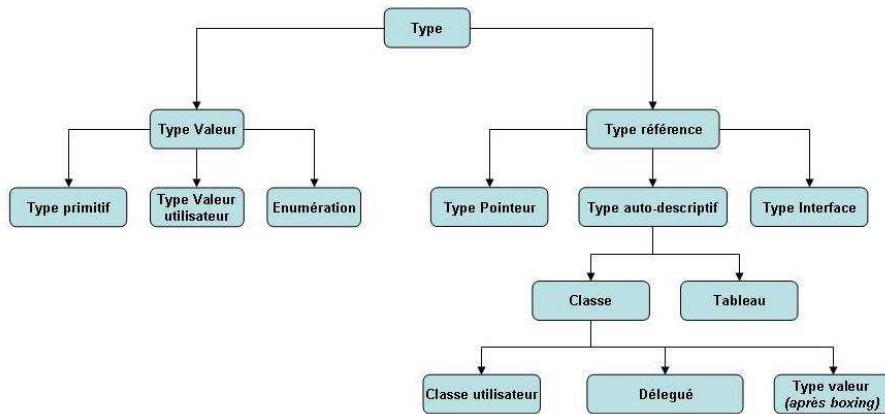


Figure 6.3 – Le système de types de la plate-forme .NET

Comme pour la plupart des langages de programmation à objets, le modèle de type de .NET distingue (voir figure 6.3) le type valeur (ce sont les types de base : `bool`, `char`, `int` et `float` mais aussi les types énumérés (`enum`) et les types structurés (`struct`)) et le type référence qui permet de désigner les objets (classes, interfaces, types délégués

et tableaux). Les valeurs sont directement allouées dans la pile. La valeur d'un tel type contient directement les bits représentant la donnée valeur. Par exemple une valeur de type `int` correspondra à une case mémoire de 8 octets. Afin de garantir l'intégrité des types à l'exécution, le CLR garde la trace des types contenus dans la pile. Les références sont allouées dans le tas.

Un type valeur contient directement sa valeur tandis qu'une référence désigne l'objet qui contient la valeur. On retrouve ici la différence entre variable et pointeur dans certains langages de programmation. La conséquence de cette distinction est que deux variables sont nécessairement distinctes et que la modification de la valeur de l'une n'aura pas d'incidence sur l'autre tandis que deux références peuvent désigner le même objet et que la modification de la valeur de celui-ci sera aussi perçue par l'autre.

Par exemple au type `System.Boolean` du moniteur d'exécution .NET correspond le type C# `bool` et le type VB `Boolean`. Idem pour les types `System.Byte`, `System.Char`, `System.Object` qui sont respectivement couplés avec les types `byte`, `char` et `object` de C# et les types `Byte`, `Char` et `Object` de VB.

Dans .NET les deux types valeur et référence sont unifiés pour les différents types de base du langage. Il est ainsi possible de transformer un type valeur en un type de référence par le mécanisme appelé empaquetage ou objectification (*boxing*). Celui-ci transforme le type primitif en son homologue objet. L'opération inverse est appelée dépaquetage (*unboxing*) (voir figure 6.4)

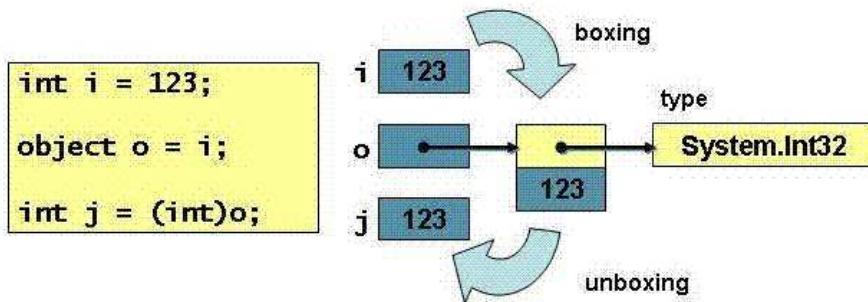


Figure 6.4 – Boxing et Unboxing

6.2.6 Le langage C#

Comme nous l'avons vu, la plate-forme .NET n'impose a priori aucun langage de programmation. En effet, toute compilation conduit à un ensemble de types suivant la spécification du système de types commun .NET (CTS) et le code se retrouve sous forme de langage intermédiaire (MSIL). Malgré cela un nouveau langage nommé C# [Gunnerson and Wienholt] [Ben Albahari et al. 2002] est apparu en même temps que la plate-forme. Ce langage a fait également l'objet d'une normalisation auprès de l'ECMA et de l'ISO. C'est le langage de référence de la plateforme .NET puisqu'il permet d'en manipuler tous les concepts : des instructions du MSIL au système de types .NET (classes, structures, énumérations, interfaces, délégués, tableaux, opérateurs, constructeurs, des-

tructeurs, accesseurs (*setter, getter*), etc.). C'est pourquoi, en dépit de sa nouveauté, de nombreux développeurs apprennent ce langage pour programmer des applications .NET.

Pour un programmeur C++ ou Java, la lecture d'un code C# n'est pas déstabilisante au premier abord puisque le langage ressemble fortement à un mélange entre ces langages avec le meilleur de chacun et la simplification des fonctionnalités complexes comme l'héritage multiple en C++. Des exemples de code seront donnés dans les sections suivantes.

6.3 Applications distribuées

La plateforme .NET offre différents mécanismes permettant de distribuer les applications. Même si le framework est principalement orienté vers les services web, d'autres mécanismes sont disponibles pour le développeur. Par exemple .NET Remoting est une proposition beaucoup plus efficace que les services web pour la distribution d'application. Il est également possible d'utiliser la plateforme de service COM+ ou encore l'intergiciel orienté messages MSMQ. La sortie de la version 2.0 du framework a donné naissance à la nouvelle plateforme de distribution de Microsoft «Windows Communication Framework» (nom de code Indigo). Elle vise à unifier le développement et le déploiement des applications distribuées sur la plateforme .NET. Lorsque l'on développe un service distant, aucune technologie de distribution n'est à privilégier de manière systématique. Le choix entre les différentes technologies dépend des contraintes de chaque application (rapidité, standardisation, disponibilité, montée en charge) mais aussi des contraintes de fonctionnement (en intranet, volonté d'avoir des clients légers ou lourd). Dans ce chapitre nous présentons les principales technologies offertes dans .NET pour la communication.

6.3.1 Le canevas .NET Remoting

Equivalent de la technologie RMI pour la plateforme JAVA, le Remoting .NET permet le dialogue entre applications situées dans des domaines d'applications distincts, en particulier sur des stations différentes. L'architecture du Remoting .NET est conçue de manière extrêmement modulaire. Il est en effet possible d'insérer ses propres mécanismes à de nombreux endroits de la chaîne de traitement exécutée lors de l'envoi d'un message depuis le composant client vers le composant cible. Dans ce canevas, ont été séparés les aspects liés au transport de messages, à l'empaquetage des données et à l'appel à distance. Il est par exemple possible en implantant certaines classes du framework de faire communiquer un client .NET avec un composant RMI ou encore un composant CORBA. De même, le même code d'un composant pourra dialoguer sur un canal http en sérialisant ses données en XML ou alors dialoguer sur un canal TCP en sérialisant les données envoyées dans un format binaire plus compact.

En .NET Remoting, lors d'un dialogue client-serveur, des objets peuvent être passés en paramètres d'un appel de méthode. D'autres valeurs peuvent être retournées au client en réponse à sa requête. Il existe alors deux modes de passage de paramètres :

- **Passage par valeur** : dans ce mode c'est une copie intégrale de l'objet qui est envoyée au client. Si le client modifie cet objet une fois récupéré, les modifications ne seront pas propagées côté serveur. Ce type d'objet implémente l'interface ISerializable. On parle alors d'objet «sérialisable».

- **Passage par référence** : dans ce mode, c'est une référence vers un objet qui est envoyée au client. Une modification de l'objet soit du côté serveur, soit du côté client entraînera la modification de l'objet désigné dans le domaine d'application où réside l'objet. Ce type d'objet implémente la classe `MarshalByRefObject`. On parle alors de service .NET remoting ou encore de composant .NET Remoting.

Un composant .NET Remoting peut être activé de différentes manières. La création d'une instance du composant sur le serveur sera soit à la charge du serveur lui-même (on parle alors de service «activé par le serveur») soit par le client (on parle de service «activé par le client»). Le cycle de vie des services *Remoting* est également variable. On distingue ainsi trois types de service .NET Remoting :

- Les services « **activés par le serveur à requête unique** » (mode `SingleCall`) Pour chaque invocation d'une méthode du service par un client, une nouvelle instance spécifique du composant est créée pour répondre à la requête. C'est le serveur qui est en charge de l'instanciation du composant.
- Les services « **activés par le serveur et partagés** » (mode `Singleton`) La même instance du composant serveur est utilisée pour répondre aux différentes requêtes des différents clients. Ce mode permet un partage des données entre les clients. C'est le serveur qui est en charge de l'instanciation du composant.
- Les services « **activés par le client** » Dans ce mode c'est au client de demander explicitement la création d'une instance du composant dans l'application serveur par un appel explicite à l'opérateur `new`. Le composant serveur répondra alors aux différentes requêtes d'un même client

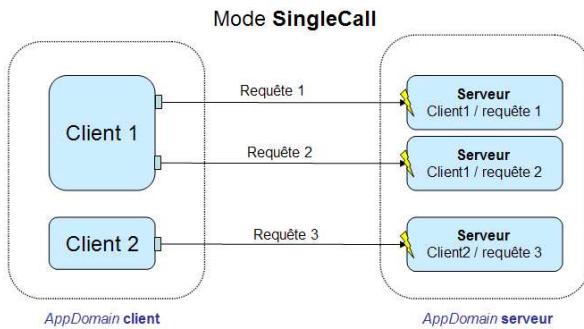


Figure 6.5 – Activation côté serveur - SingleCall

Les applications .NET Remoting dialoguent à travers un canal de communication sur lequel circulent des messages. Le framework offre la possibilité de définir par programmation le type de canal utilisé de même que la manière dont les messages sont encodés sur ce canal. Deux types de canaux sont fournis au développeur par le framework : le canal `HttpChannel` permettant d'établir un dialogue via le protocole HTTP et le canal `TCPChannel` permettant un dialogue sur une socket TCP. Au niveau de l'encodage, deux formateurs de messages sont également fournis. Le premier (`BinaryFormatter`) permet un encodage binaire des données sur le canal. L'autre (`SOAPFormatter`) permet un encodage des messages sous forme de document SOAP. L'utilisateur peut en outre implémenter ses propres canaux et ses propres formateurs en implémentant respectivement les interfaces

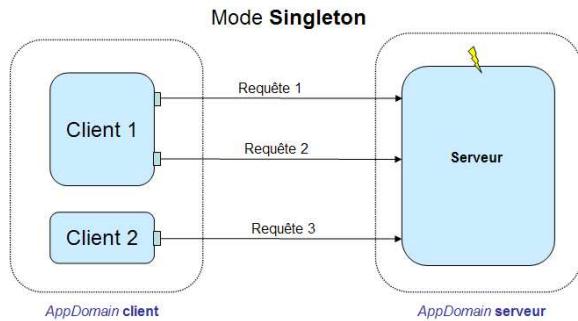


Figure 6.6 – Activation côté serveur - Singleton

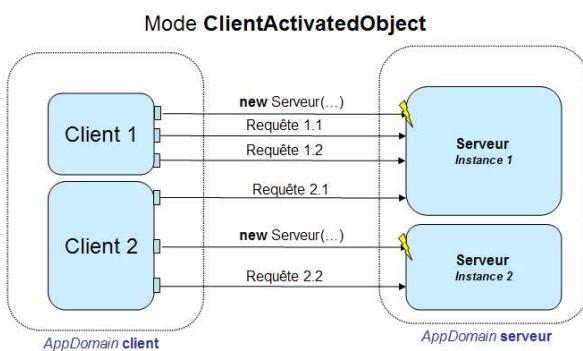


Figure 6.7 – Activation par le client

IChannel et **IRemotingFormatter**. Le type de dialogue entre un client et un serveur est défini par le couple (Canal, Formateur) qui est donné par le programmeur de l'application. Il est ainsi possible de faire circuler des messages SOAP sur un canal TCP ou encore encoder les messages sous un format binaire compact sur un canal HTTP.

Création d'un service .NET Remoting

Pour comprendre le processus de développement d'un service en .NET Remoting, prenons le cas d'un service très simple afin de s'abstraire des mécanismes spécifiques à l'implémentation du service. Considérons un service permettant d'effectuer deux opérations : la division réelle et la division entière. Dans un premier temps il s'agit de définir l'interface du service ainsi que les classes des données échangées. Ces classes de données exposées par le service (paramètres et valeur de retour) doivent pouvoir permettre à leurs instances d'être passées par valeur afin de pouvoir traverser les frontières du service. On parle alors d'objets « sérialisables ». En particulier les types de données simples fournis par le canevas (**int**, **double**, **string**) sont tous sérialisables. Dans le cas de la division entière nous désirons retourner au client un objet contenant le quotient et le reste de la division euclidienne de A par B. Pour cela nous marquons la classe **QuotientEtReste** par l'attribut **[Serializable]**.

Fichier EchoContract.cs

```
namespace Div.ServiceContract {
    [Serializable]
    public class QuotientEtReste {
        public int Quotient;
        public int Reste;
    }

    public interface IDivService {
        double Division(double A, double B);

        QuotientEtReste DivisionEntiere(int A, int B);
    }
}
```

Ces deux classes définissent la base du «contrat» entre le client et le service. Lorsque le service est activé côté serveur (mode **SingleCall** ou **Singleton**), seules ces classes seront partagées par les deux parties.

Pour créer le service lui même il faut ensuite coder une classe implémentant cette interface et la faire hériter de la classe du classe **System.MarshalByRefObject** indiquant au runtime .NET que les instances de cette classe ne seront pas sérialisées par copie mais par référence.

```
namespace Div.ServiceImpl {
    using Div.ServiceContract;

    public class DivService: MarshalByRefObject, IDivService {
        string mName;
```

```

static int mCounter=0;

public DivService()      : this("Anonyme_"+mCounter++){}

public DivService(string name) {
    this.mName=name;
    Console.WriteLine("new DivService(\"+mName+\")");
}

public double Division(double A, double B) {
    Console.WriteLine(mName+.Division("+A+","+B+"));
    return A/B;
}

public QuotientEtReste DivisionEntiere(int A, int B) {
    Console.WriteLine(mName+.DivisionEntiere("+A+","+B+"));
    QuotientEtReste RESULT=new QuotientEtReste();
    RESULT.Quotient=A/B;
    RESULT.Reste=A%B;
    return RESULT;
}
}
}
}

```

Notre service possède deux constructeurs. Le constructeur par défaut (sans paramètre) est obligatoire si l'on publie le service en mode « activé côté serveur ». En effet ce sera au runtime du *Remoting* d'instancier de manière transparente les instances de ce service. Nous définissons également un constructeur prenant une chaîne de caractère en paramètre et ce afin que le service soit identifié lorsque nous le testerons. Ce constructeur nous permettra de voir les différences à l'exécution entre les différents types de cycle de vie du service (lors de l'instanciation).

Il reste maintenant à créer l'application qui hébergera notre service. Nous allons créer deux applications console. L'une hébergera le service sous les deux modes « activé côté serveur » SingleCall et Singleton. La deuxième application hébergera les instances du service en mode «activé par le client». Il existe deux manières de spécifier au framework ces paramètres : soit par programmation soit à l'aide de fichier de configuration. L'utilisation de fichier de configuration est la méthode à privilégier puisqu'elle permet de modifier les paramètres de déploiement sans recompiler l'application. Nous présentons ici l'approche par programmation. La deuxième approche par fichier de configuration sera présentée dans le chapitre suivant .

Application hôte N° 1 en mode « activé par le serveur »

```

namespace Div.Host.SAO {
    using Div.ServiceContract;
    using Div.ServiceImpl;

    class DivHost {
        [STAThread]

```

```

        static void Main(string[] args) {
            // création et enregistrement du canal HTTP 8081
            ChannelServices.RegisterChannel(new HttpChannel(8081));
            RemotingConfiguration.RegisterWellKnownServiceType(typeof(DivService),
                "division/singleton", WellKnownObjectMode.Singleton);
            RemotingConfiguration.RegisterWellKnownServiceType(typeof(DivService),
                "division/singlecall", WellKnownObjectMode.SingleCall);
            RemotingConfiguration.RegisterActivatedServiceType(
                typeof(Div.ServiceImpl.DivService));
            Console.WriteLine("Appuyez sur [Entrée] pour quitter.");
            Console.ReadLine();
        }
    }
}

```

Tout d'abord nous indiquons au *Remoting* le canal de transport à utiliser. Nous créons une instance de canal HTTP sur le port 8081 que nous enregistrons auprès du gestionnaire de canal **ChannelServices**. Dans un deuxième temps nous devons enregistrer notre service auprès du *Remoting*. C'est le rôle de la méthode **RegisterWellKnownServiceType** pour les services « activés par le serveur ». Cette méthode prend en paramètre la classe des futures instances du service, une adresse URI relative au canal de transport utilisé, ainsi que le mode de publication choisi pour le service. Nous appelons cette méthode une première fois pour publier le service en mode Singleton et une seconde fois pour publier le service en mode SingleCall.

Voyons à présent comment la publication du service s'effectue en mode « activé par le client »

Application hôte N ° 2 en mode « activé par le client »

```

namespace Div.CAO.Host {
    using Div.ServiceContract;
    using Div.ServiceImpl;

    class DivHost {
        [STAThread]
        static void Main(string[] args) {
            // création et enregistrement du canal HTTP 8082
            ChannelServices.RegisterChannel(new HttpChannel(8082));
            RemotingConfiguration.RegisterActivatedServiceType
                (typeof(Div.ServiceImpl.DivService));
            Console.WriteLine("Appuyez sur [Entrée] pour quitter.");
            Console.ReadLine();
        }
    }
}

```

La différence réside dans la manière de publier le service. Cette fois c'est la méthode **RegisterActivatedServiceType** qui est utilisée. Elle indique au *Remoting* qu'il s'agit d'une classe de service à enregistrer dans le mode « activé par le client » et ne prend en paramètre que la classe du service.

Création d'un client .NET Remoting

Voyons à présent comment dialoguer avec nos services dans les différents modes. Le premier exemple montre comment dialoguer avec les deux services publiés en mode Singleton et SingleCall. Afin d'obtenir une référence distante sur le service, le client utilise la méthode `GetObject` de la classe `Activator`, en lui passant en paramètres l'interface et l'adresse URI du service distant. Grâce à ces paramètres le *Remoting* est en mesure de localiser l'application hébergeant le service et de créer une référence transparente pour le client. Une fois la référence obtenue le client peut invoquer les méthodes du service comme ci celui-ci était présent localement. A chaque invocation de méthode l'appel sera transporté jusqu'au serveur qui déterminera l'instance qui devra répondre à l'appel client. S'il s'agit du composant `SingleCall` le *Remoting* créera une nouvelle instance du service sur le serveur à chaque appel client. Dans le cas du service `Singleton` le *Remoting* créera une instance du service lors du premier appel de méthode du client

Application cliente N° 1 (dialoguant en mode SingleCall et Singleton)

```
class Client {
    [STAThread]
    static void Main(string[] args) {
        IDivService service;
        service=(IDivService)
            Activator.GetObject(typeof(Div.ServiceContract.IDivService),
                "http://localhost:8081/division/singleton");
        Test("1)Mode Singleton",service);

        service=(IDivService)
            Activator.GetObject(typeof(Div.ServiceContract.IDivService),
                "http://localhost:8081/division/singlecall");
        Test("2)MODE SingleCall",service);
    }

    static void Test(string testName, IDivService service) {
        Console.WriteLine(testName);

        double RESULT1=service.Division(100.0,70.0);
        Console.WriteLine("100.0 / 70.0="+ RESULT1);

        QuotientEtReste QR=service.DivisionEntiere(100,70);
        Console.WriteLine("100 / 70 -> Q="+ QR.Quotient+" R="+QR.Rest);

        Console.WriteLine("Appuyez sur [Entrée] pour continuer.");
        Console.ReadLine();
    }
}
```

En mode « activé par le client », l'obtention de la référence distante sur le service est assez différente. Cette fois c'est la méthode `CreateInstance` de la classe `Activator` qui est utilisée. Celle-ci prend en paramètre l'url (dans un tableau d'objet) de l'application qui héberge les instances du service. C'est à ce moment que le serveur crée l'instance

qui servira le client. Le client peut alors appeler différentes méthodes sur cette même instance du service. Le *Remoting* utilise un système paramétrable de « bail » permettant de contrôler le cycle de vie de cette instance. Lorsque le bail expire après un certain temps, le serveur détruira l’instance. Chaque invocation du client sur le service renouvelera ce bail pour une durée donnée. Ce mécanisme de bail existe également dans les mode SingleCall et Singleton.

Application cliente N° 21 (dialoguant en mode SingleCall et Singleton)

```
[STAThread]
static void Main(string[] args) {
    object[] url = { new UrlAttribute("http://localhost:8082") };
    object[] param = { "Icar" };
    IDivService service=(DivService)
    Activator.CreateInstance(typeof(DivService),param,url);
    Test("1) MODE CAO",service);
}
```

Evolution future

Un aspect à prendre en compte lorsque l’on utilise le .NET Remoting est le couplage qu’il créé entre le client et le serveur. En effet ces deux entités doivent partager un même *assembly* .NET : celui contenant l’interface du service. Les changements de version du service impactant l’interface implique également la modification du client. Dans le nouveau framework Indigo, le découplage entre consommateur et fournisseur de service devient réel puisque les deux entités partageront non plus une interface, mais plutôt un contrat de service dans un format standard tel que WSDL.

6.3.2 Les services Web en .NET

Les services web ayant déjà été présenté dans un chapitre précédent (cf. chapitre 4, cette section présente uniquement leur mise en œuvre dans le cadre de la plate-forme .NET à l’aide de ASP.NET (nouvelle version d’ASP - Active Server Pages). ASP.NET permet d’intégrer dans une page html du code .NET. Ce code est exécuté par le serveur Web lorsque la page est demandée. Il va de soi, que le serveur Web doit être, pour cela, capable d’exécuter du code .NET. Deux suffixes de fichiers sont utilisés :

- “*aspx*” pour les pages html qui contiennent du code .NET devant être interprétée par le serveur web. Les pages sont envoyées au client dans le format HTML.
- “*asmx*” pour des pages qui contiennent le code du service web. Ce service est exécuté par le serveur web. Les pages sont envoyées au client dans le format SOAP.

Cette section présente essentiellement la mise en œuvre des services web dans l’architecture .NET.

Un modèle de développement web unifié, ASP.NET

JSP (Java Server Pages) a déjà été une réponse à la proposition ASP de Microsoft. Aujourd’hui une nouvelle étape est franchie. Les applications web conçues sur la plate-forme .NET partagent avec toutes les autres applications .NET un même modèle d’application.

Dans ce modèle, une application web est un ensemble d'URL dont la racine est une URL de base. Il englobe donc les applications web générant des pages pour un navigateur et les services web. Ce modèle de programmation d'applications web est une évolution d'ASP (Active Server Pages). ASP.NET utilise les avantages du Common Language Runtime et de la plate-forme .NET pour fournir un environnement fiable, robuste et évolutif aux applications web. ASP.NET bénéficie aussi des avantages du modèle d'assemblage du Common Language Runtime pour simplifier le développement d'applications. Il fournit en outre, des services pour faciliter le développement d'applications (tels que les services de gestion d'état) et des modèles de programmation haut niveau (tels que ASP+ Web Forms et ASP.NET Services Web).

L'environnement d'exécution HTTP dans la plate-forme .NET est au cœur d'ASP.NET pour le traitement des requêtes HTTP. Cette machine virtuelle haute performance est basée sur une architecture de bas niveau similaire à l'architecture ISAPI traditionnellement fournie par Microsoft Internet Information Services (IIS). Cet environnement d'exécution HTTP est constitué de " managed code " exécuté dans un processus spécifique. Il est responsable du traitement de toutes les requêtes HTTP, de la résolution de l'URL de chaque requête vers une application et de la distribution de la requête à l'application pour traitement complémentaire. ASP.NET utilise le modèle de déploiement de la plate-forme .NET basé sur les assemblages et permet la mise à jour des applications à chaud. Les administrateurs n'ont pas besoin d'arrêter le serveur Web ou l'application pour mettre à jour les fichiers d'application. Ceux-ci ne sont jamais verrouillés de façon à pouvoir être remplacés même lors de l'exécution de l'application. Lorsque les fichiers sont mis à jour, le système bascule vers la nouvelle version. Le système détecte les modifications de fichiers, lance une nouvelle instance de l'application avec le nouveau code et commence à router les requêtes entrantes vers cette application. Lorsque toutes les requêtes en suspens traitées par l'instance de l'application existante ont été traitées, cette instance est arrêtée.

Le traitement des requêtes HTTP

Au sein d'une application, les requêtes HTTP sont routées via un pipeline de modules HTTP et en dernier lieu, par un gestionnaire de requêtes. Les modules et gestionnaires de requêtes HTTP sont simplement des classes gérées qui implantent des interfaces spécifiques définies par ASP.NET. L'architecture en pipeline facilite grandement l'ajout de services aux applications : il suffit de fournir un module HTTP. Par exemple, la sécurité, la gestion d'état et des traces sont implantées en tant que modules HTTP. Les modèles de programmation de haut niveau, tels que les services web et Web Forms, sont généralement implantés comme gestionnaires de requêtes. Une application peut être associée à plusieurs gestionnaires de requêtes (un par URL), mais toutes les requêtes HTTP sont routées au travers du même pipeline. Le web constitue un modèle fondamentalement sans état et sans corrélation entre les requêtes HTTP. Ceci peut rendre l'écriture d'applications web difficile, car les applications nécessitent habituellement de conserver leur état entre plusieurs requêtes. ASP.NET améliore les services de gestion d'état introduits par ASP pour fournir trois types d'état aux applications web : application, session et utilisateur. L'état " application ", tout comme pour ASP, est spécifique à une instance de l'application et n'est pas persistant. L'état " session " est spécifique à une session utilisateur de l'application. Contrairement à l'état session de ASP, l'état session de ASP.NET est stocké dans

un processus distinct et peut même être configuré pour être stocké sur un ordinateur distinct. Ceci rend l'état session utilisable lorsqu'une application est déployée sur un ensemble de serveurs web. L'état utilisateur est similaire à l'état session mais en général, il n'expire jamais et est persistant. L'état utilisateur est donc utile pour stocker les préférences de l'utilisateur et d'autres informations de personnalisation. Tous les services de gestion d'état sont implantés en tant que modules HTTP et peuvent en conséquence être facilement ajoutés ou supprimés du pipeline d'une application. Si des services de gestion d'état autres que ceux fournis par ASP.NET sont nécessaires, ils peuvent être fournis par un module tiers.

ASP.NET fournit aussi des services de cache pour améliorer les performances. Un cache de sortie permet d'enregistrer les pages générées en entier et un cache partiel permet de stocker des fragments de pages. Des classes sont fournies pour permettre aux applications, aux modules HTTP et aux gestionnaires de requêtes de stocker à la demande des objets arbitraires dans le cache.

Création d'un Service Web XML en ASP.NET

Le framework .NET offre de nombreux mécanismes simplifiant au maximum la création de service web. En particulier tous les aspects liés à SOAP et WSDL peuvent devenir complètement transparents pour le développeur. De plus s'utilisation de l'environnement de développement de Microsoft Visual Studio pousse encore plus loin la simplicité de création et de déploiement de service web en quelques click. Nous montrons cependant ici comment se passent les étapes de développement sans l'aide de cet IDE afin de ne pas trop masquer les mécanismes impliqués.

Les étapes du processus de développement et de déploiement d'un service web en .NET sont les suivantes :

- Implémenter la classe du service en le faisant hériter de la classe System.Web.Services.WebService
- Décorer l'implémentation en faisant précéder chaque déclaration de méthode devant être exposée par l'attribut [WebMethod]
- Créer un répertoire virtuel sur le serveur (exemple IIS).
- Compiler le service (dans le répertoire virtuel)
- Déployer le service sur le serveur web

Tout d'abord il s'agit d'écrire le service lui même. Reprenons l'exemple de la section précédente sur le service de division. Voici le code C# permettant d'en faire un service web

DivWebService.cs Le service web

```
using System; using System.Web.Services;

namespace Div {
    public class DivWebService: WebService {
        [WebMethod]
        public double Division(double A, double B) {
            return A/B;
        }
    }
}
```

```

    public void UneMethodeNonPubliee() {
    }
}

}

```

Pour qu'une classe puisse être exposée via un service web, celle-ci doit hériter de la classe du framework `System.Web.WebServices.WebService`. De plus chaque méthode qui devra être publiée devra être précédée de l'attribut `[WebMethod]`. Nous pouvons remarquer dans notre exemple que la méthode `UneMethodeNonPubliee` ne sera pas publiée dans l'interface du service web car sa signature n'est pas précédée de l'attribut `[WebMethod]`. Ainsi cette méthode ne sera pas accessible aux clients. Nous compilerons le code dans une bibliothèque `DivWS.dll` grâce à la ligne de commande suivante :

```
csc.exe /out:bin\DIVWS.dll /t:library DivService.cs
```

Pour pouvoir tester notre service une page `div.asmx` utilisant la notion de "*code behind*" doit être mise en œuvre. Nous spécifions dans son entête qu'il s'agit d'un web service ainsi que le nom du fichier source C# correspondant à cette classe à des fins de débogage.

Le fichier `div.asmx`

```
<%@ WebService Class="Div.DivWebService"%>
```

A présent, il nous faut déployer notre service au sein d'une application web hébergée par le serveur IIS de Microsoft. Pour cela nous pouvons utiliser la console d'administration de Windows afin de créer une nouvelle application. Nous la nommerons par exemple `icar`. Dès lors l'application aura comme racine distante l'url `http://localhost/icar`. Créons un répertoire `bin` dans lequel nous plaçons la bibliothèque compilée précédemment. La page `aspx` du service sera placé à la racine de l'application. Le service est alors opérationnel. Il peut immédiatement être testé en utilisant un navigateur web quelconque et en allant à l'adresse `http://localhost/icar/div.asmx?WSDL`. Cette page permet de visualiser le contrat WSDL de notre service. Pour tester le service lui-même il faut se rendre à l'adresse `http://localhost/icar/div.asmx`. .NET génère automatiquement un formulaire web qui permet d'appeler les différentes `WebMethod` du service web.

Utilisation de Services Web

Nous allons voir à présent comment consommer un service web. Nous prendrons comme exemple un service Web assez pratique : celui offert par Google®. Avant toute chose il faut disposer de la description WSDL de ce service. Celle-ci se trouve à l'adresse `http://api.google.com/GoogleSearch.wsdl`

Le framework .NET fournit un utilitaire `wsdl.exe` permettant à partir de cette description de générer les classes proxy permettant de consommer le service web. Voici la ligne de commande permettant d'obtenir le code C# du proxy :

```
wsdl.exe /o:google.cs http://api.google.com/GoogleSearch.wsdl
```

Le fichier généré `google.cs` contient différentes classes : celles du proxy vers le service Web et les classes de données manipulées par le service.

Voici à présent un programme client simple permettant d'effectuer une correction d'un mot mal orthographié.

```
class TestGoogle {
    static void Main(string[] args) {
        string GOOGLE_KEY="-xxxxx-";
        string phrase="Intergiciel";
        GoogleSearchService service=new GoogleSearchService();
        string rep=service.doSpellingSuggestion(GOOGLE_KEY,phrase);
        Console.WriteLine("Correction: "+rep);
        Console.ReadLine();
    }
}
```

Le programme une fois exécuté affichera le texte corrigé « Intergiciel ». Dans notre exemple nous utilisons l'opération `doSpellingSuggestion` du Web Service en lui donnant comme entrée d'une part une clé d'authentification (fournie par Google© permettant d'accéder au service web après enregistrement en ligne) et en deuxième paramètre la phrase à corriger.

6.3.3 L'Asynchronisme

Il existe deux possibilités pour effectuer des appels asynchrones de méthode dans .NET. La première utilise les appels asynchrones de méthode disponible dans la plate-forme .NET et la seconde repose sur les files de messages disponible dans la plate-forme Windows. Nous détaillons ci-après ces deux possibilités.

Appel asynchrone

Pour pouvoir effectuer en .NET des appels asynchrones de méthode, il est nécessaire d'utiliser le principe des délégues qui fournit pour chacun d'entre eux deux méthodes : `BeginInvoke()` et `EndInvoke()`. `BeginInvoke()` permet d'effectuer l'appel asynchrone et `EndInvoke()` permet de récupérer les résultats de cet appel. La synchronisation entre l'appel et la lecture du résultat est à la charge du programmeur qui peut :

- effectuer une attente bloquante du résultat en appelant la méthode `EndInvoke()`,
- être prévenu de la disponibilité du résultat en transmettant lors de l'appel, le code à exécuter pour récupérer le résultat.

Voici un exemple illustrant ces deux possibilités :

```
class Exemple {
    public delegate int Prototype(int a, int b);

    static int PGCD(int a, int b) {
        // calcul du pgcd
        return c;
    }
}
```

```

static void FinCalcul(IAsyncResult A) {
    // récupération du résultat
    Prototype p = (Prototype) (IAsyncResult A).AsyncDelegate;
    int c = p.EndInvoke(A);
}

static void Main () {
    // création de l'instance du délégué
    Prototype p = New Prototype (PGCD);
    int c;

    // appel synchrone du délégué
    c = p (10, 25);

    // appel asynchrone du délégué
    IAsyncResult A = p.BeginInvoke (10, 25, null, null);

    // lecture bloquante du résultat
    c = p.EndInvoke(A);

    // appel asynchrone du délégué en transmettant la procédure à
    // exécuter à la fin de l'appel (méthode de finalisation)
    IAsyncResult A = p.BeginInvoke (10, 25, new AsyncCallback(FinCalcul), null);
}
}

```

Le dernier paramètre, non utilisé dans l'exemple ci-dessus, de la procédure `BeginInvoke` permet de transmettre la référence d'un objet utilisable simultanément dans le *thread* qui déclenche l'appel asynchrone et dans celui qui exécute la méthode de finalisation.

Les Files de message

Contrairement à l'appel asynchrone de .NET, le client et le serveur ne sont plus obligés d'être connectés et les requêtes des clients sont stockées dans une file sous forme de messages. Il est par contre nécessaire d'installer sur les stations hôtes le mécanisme de gestion de file de messages (Microsoft Message Queue - MSMQ).

6.4 Composants et services techniques

Les services techniques, de plus en plus nombreux, sont destinés à simplifier la tâche des programmeurs en leur évitant d'avoir à développer tout ce qui peut être mis en commun. Dans la plate-forme .Net, ces services reposent sur Enterprise Services qui est la nouvelle appellation de COM+ et ils sont présent dans la plate-forme support (Windows) que .NET utilise. Les principaux services techniques permettent l'accès aux données, la gestion de la sécurité applicative, la mise en œuvre de transactions distribuées. Ce sont ces services qui nous allons détailler dans cette section.

6.4.1 Accès aux données

Pratiquement toutes les applications utilisent un système de persistance au sens large du terme. Dans la plate-forme .NET, cette persistance peut être réalisée de plusieurs manières :

- par sérialisation de l'état de l'objet sous forme de flots d'octets ou en XML ;
- par sauvegarde de la valeur de l'état de l'objet dans une base de données relationnelle.

L'appellation ADO (*ActiveX Data Object*) englobe à la fois l'ensemble des classes du framework .NET utilisées pour manipuler les données contenues dans les SGBD relationnels et la philosophie d'utilisation de ces classes.

ADO permet la connexion avec une base de données en mode :

- connecté : si l'application charge des données de la base au fur et à mesure de ces besoins et envoie les modifications à ces données dès qu'elles surviennent.
- déconnecté : si l'application charge des données de la base, les exploite hors ligne et les met à jour si nécessaire ultérieurement. Elle peut aussi récupérer ses données de manière externe à la base, par exemple depuis un fichier XML, les manipuler et les sauvegarder dans la base ; ou réciproquement.

La faiblesse du mode connecté réside dans les multiples accès qui sont effectués sur la base de données, générant pas là même de très nombreux accès réseaux. Le mode déconnecté n'a pas cet inconvénient mais est beaucoup plus consommateur de mémoire utilisée pour stocker de manière temporaire les données nécessaires.

Parmi toutes les classes qu'ADO regroupe, DataSet joue un rôle prépondérant. Les instances de cette classe représentent une partie de la base. Pour chacun de ces tables, l'instance contient des lignes extraites et éventuellement les contraintes d'intégrités associées à la table. Cette classe est à utiliser pour tout travail en mode déconnecté. Cette classe permet aussi le passage d'un DataSet à un document XML ou réciproquement.

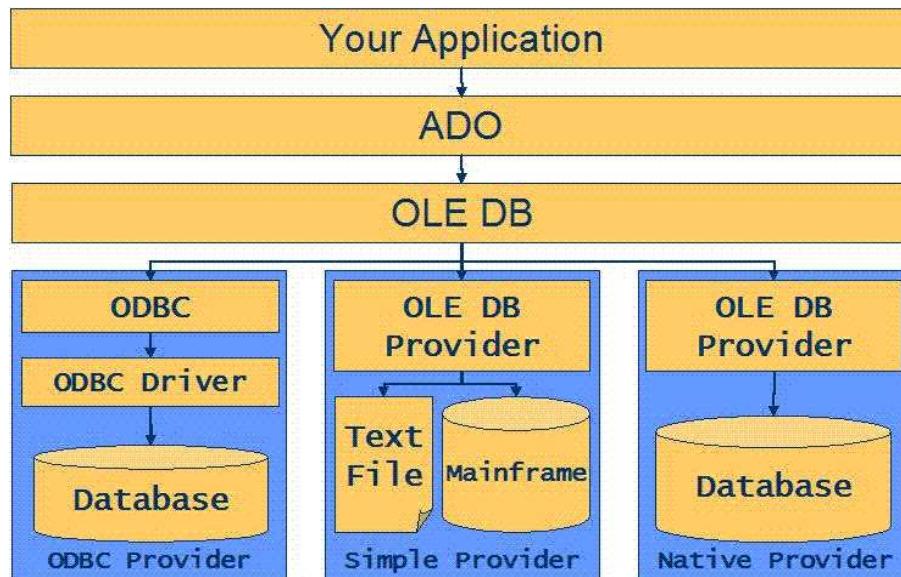


Figure 6.8 – Accès aux données

Exemple de manipulation d'un DataSet :

```
// créer le DataSet et définir les tables
DataSet dataset = new DataSet();
dataset.DataSetName = "BookAuthors";

DataTable authors = new DataTable("Author");
DataTable books = new DataTable("Book");

// définir les colonnes et les clés
DataColumn id = authors.Columns.Add("ID", typeof(Int32));
id.AutoIncrement = true;
authors.PrimaryKey = new DataColumn[] {id};

DataColumn name = new authors.Columns.Add("Name",typeof(String));

DataColumn isbn = books.Columns.Add("ISBN", typeof(String));
books.PrimaryKey = new DataColumn[] {isbn};

DataColumn title = books.Columns.Add("Title", typeof(String));
DataColumn authid = books.Columns.Add("AuthID",typeof(Int32));
DataColumn[] foreignkey = new DataColumn[] {authid};

// ajouter les tables au DataSet
dataset.Tables.Add (authors);
dataset.Tables.Add (books);

// Ajouter des données et sauvegarder le DataSet
DataRow shkspr = authors.NewRow();
shkspr["Name"] = "William Shakespeare";
authors.Rows.Add(shkspr);

DataRelation bookauth = new DataRelation("BookAuthors",
                                         authors.PrimaryKey, foreignkey);
dataset.Relations.Add (bookauth);

DataRow row = books.NewRow();
row["AuthID"] = shkspr["ID"];
row["ISBN"] = "1000-XYZ";
row["Title"] = "MacBeth";
books.Rows.Add(row);

dataset.AcceptChanges();
```

6.4.2 Sécurité

Les services de sécurité intégrés à la plate-forme permettent de s'assurer que seuls les utilisateurs autorisés accèdent aux ressources d'un ordinateur et seul le code ayant acquis les droits suffisants peut exécuter les actions nécessitant ces droits. Ceci améliore la sécurité globale du système ainsi que sa fiabilité. L'environnement d'exécution étant utilisé pour charger le code, créer les objets et effectuer les appels de méthodes, il peut contrôler la

sécurité et assurer l'application des stratégies de sécurité lorsque du " managed code " est chargé et exécuté. La plate-forme .NET propose une sécurité d'accès au code, mais également une sécurité basée sur les rôles.

Grâce à la sécurité d'accès au code, les développeurs peuvent spécifier les autorisations requises par leur code pour effectuer une tâche. Le code peut par exemple nécessiter une autorisation d'écriture sur un fichier ou d'accès aux variables d'environnement. Ces informations, ainsi que celles concernant l'identité du code, sont stockées au niveau du composant. Lors du chargement et des appels de méthodes, l'environnement d'exécution vérifie que le code peut se voir accorder les autorisations qu'il a demandées. Si ce n'est pas le cas, une violation de sécurité est déclarée sous la forme du signalement d'une exception. Les stratégies d'attribution d'autorisations (connues sous le nom de trust policies) sont établies par les administrateurs des systèmes et sont basées sur des informations relatives au code telles que son origine ou les demandes d'autorisation de l'assemblage. Les développeurs peuvent également spécifier explicitement les autorisations qu'ils ne souhaitent pas, pour éviter toute utilisation malveillante du code. Il est possible de programmer des contrôles de sécurité dynamiques si les autorisations requises dépendent d'informations inconnues avant l'exécution.

Outre la sécurité d'accès au code, l'environnement d'exécution prend en charge la sécurité basée sur les rôles. La sécurité basée sur les rôles est conçue sur le même modèle d'autorisations que la sécurité d'accès au code, à ceci près que les autorisations sont basées sur l'identité de l'utilisateur et non sur l'identité du code. Les rôles représentent des catégories d'utilisateurs et sont définis lors du développement ou du déploiement. Des stratégies d'attribution d'autorisations sont assignées à chaque rôle. Lors de l'exécution, l'identité de l'utilisateur pour lequel le code est exécuté est déterminée. L'environnement d'exécution détermine les rôles auxquels l'utilisateur appartient et accorde ensuite les autorisations en fonction de ces rôles.

Sécurité d'accès utilisateur La sécurité d'accès utilisateur sert à déterminer les droits associés à l'identité en cours. Il est possible de vérifier ce qu'un appelant a le droit de faire de nombreuses façons. Dans les applications avec une interface utilisateur, il est possible d'emprunter l'identité de l'appelant, mais il est aussi possible que le code utilise un compte de " service " spécifique par modification de l'identité en cours d'exécution (modification de l'objet *Principal*).

Les droits de l'utilisateur sur l'environnement et la plate-forme sont généralement contrôlés à l'aide de listes de contrôle d'accès qui sont comparées à l'identité Windows du thread en cours.

La technologie .NET fournit une infrastructure complète et extensible pour la gestion de la sécurité d'accès utilisateur y compris des identités, des permissions et la notion d'objet *Principal* et de rôles. Pour garantir que les utilisateurs appartenant à un certain rôle appellent une méthode donnée, il suffit de définir un attribut sur la méthode de classe, comme indiqué dans le code suivant :

```
[PrincipalPermission(SecurityAction.Demand, role="Managers")]
public void managersReservedMethod () {
    // Ce code ne s'exécutera pas si le Principal associé au
    // thread n'a pas 'Managers' lorsque IsInRole est appelé
```

}

Si le composant est basé sur un déploiement dans Enterprise Services et authentifie les utilisateurs via Windows, il est aussi possible d'utiliser la gestion des rôles Enterprise Services comme indiqué dans le code ci-dessous.

```
[SecurityRole("Managers")]
public void managersReservedMethod () {
    // Ce code ne s'exécutera pas si le Principal associé à
    // l'utilisateur n'est pas dans le groupe 'Managers'
}
```

Sécurité d'accès au code La sécurité d'accès au code permet de définir les droits de l'assembly, mais il est également possible de décider du lancement ou non du code, en fonction du code essayant d'y accéder. Par exemple, cela permet d'empêcher que des objets soient appelés à partir de scripts lancés de manière inopportunne par une personne disposant de priviléges suffisants. Vous pouvez contrôler l'accès au code à l'aide des éléments suivants :

- le répertoire d'installation de l'application ;
- le hachage cryptographique de l'assembly ;
- la signature numérique de l'éditeur de l'assembly ;
- le site dont provient l'assembly ;
- le nom fort cryptographique de l'assembly ;
- l'URL dont provient l'assembly ;
- la zone dont provient l'assembly.

Les politiques de sécurité peuvent être appliquées à l'entreprise, l'ordinateur, l'utilisateur et l'application. Les zones définies par la technologie .NET sont les suivantes : Internet, intranet, Poste de travail, NoZone, de confiance et non fiable.

Implémentation de contrôles d'autorisation complexes Dans certains cas, une application doit effectuer des contrôles d'autorisation complexes reposant sur une politique d'autorisation qui requiert des combinaisons de permissions combinant simultanément la sécurité utilisateur, la sécurité d'accès au code mais aussi des permissions liées aux paramètres de l'appel. Ces types de contrôles d'autorisation sont à effectuer dans le code de l'application sous forme de programmes et requièrent généralement un développement conséquent.

Communication sécurisée Outre l'authentification des utilisateurs et des requêtes d'autorisation, il est aussi nécessaire de s'assurer que la communication entre les différents éléments d'une application soit sécurisée afin d'empêcher toute attaque de "reniflage" ou de falsification lors de la transmission ou du placement des données dans une file d'attente.

Une communication sécurisée implique la sécurisation des transferts de données entre les composants distants. Pour ce faire, .NET, comme la plupart des plates-formes, offre les options suivantes pour :

- Sécuriser l'ensemble du canal :

- Protocole SSL (Secure Sockets Layer) recommandé pour les communications avec le protocole HTTP. Il s'agit d'un standard très répandu et il est généralement permis d'ouvrir des ports SSL sur les pare-feu.
 - IPSec. Ce mécanisme constitue un choix judicieux lorsque les deux points d'entrée de la communication sont bien connus et sous le contrôle de la même entité administrative.
 - Réseau privé virtuel (VPN) qui permet d'établir un transport IP point à point sur Internet (ou d'autres réseaux).
- **Sécurisation des données :**
- Signature d'un message. Cela permet de détecter les messages falsifiés. Les signatures peuvent être utilisées pour l'authentification au sein d'un même processus.
 - Cryptage de l'ensemble d'un message. De cette façon, l'ensemble du message devient illisible en cas de compromission des paquets du réseau. Le cryptage d'un message à l'aide d'algorithmes appropriés permet également de détecter les messages falsifiés.
 - Cryptage des parties sensibles d'un message si seulement des parties du message doivent être protégées.

6.4.3 Transaction

Pour gérer des transactions distribuées avec plusieurs bases de données, il n'est pas possible dans la technologie .NET d'utiliser ADO.NET 6.4.1 et il faut, comme pour la sécurité utiliser le gestionnaire de transactions disponible dans Enterprise Services (DTC : Distributed Transaction Coordinator). Pour illustrer la mise en œuvre des transactions nous allons prendre l'exemple très classique du transfert bancaire entre deux comptes. L'architecture de l'application est décrite dans la figure 6.9.

Ci-dessous, la partie essentielle du source de l'application :

```
public class Programme {
    static void Main () {
        TdF transfert = new TdF ();
        transfert.Transfert ("Michel", "David", 1960, 1979, 1000) ;
        // transfert 1000 euros du compte de Michel au compte de David
    }
}

[Transaction(TransactionOption.RequiresNew)]
public class TdF : ServicedComponent {
    // hérite d'un objet Services Enterprises
    public void Transfert (String ctx_client1, ctx_client2,
                           int ID_client1, ID_client2, montant) {
        Compte client1 = new Compte (); Compte client2 = new Compte ();
        // création de la connection à la base de données
        client1.Connect (ctx_client1); client2.Connect (ctx_client2);
        // connection à la base de données pour lecture du solde du compte
        int solde_client1 = client1.ConsulteCompte (ID_client1);
        int solde_client2 = client2.ConsulteCompte (ID_client2);
        // modification locale des montants des comptes
    }
}
```

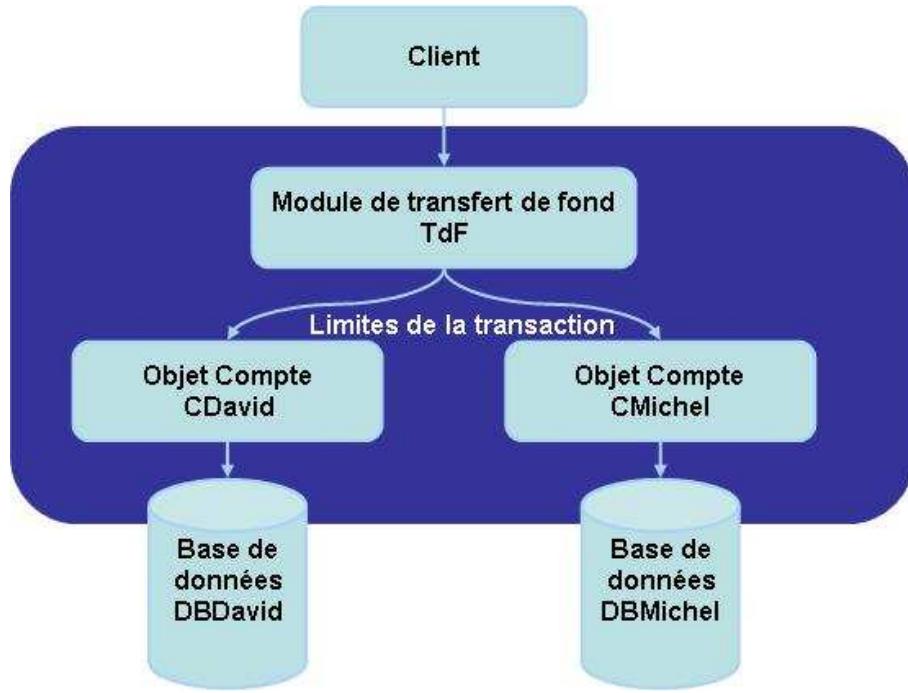


Figure 6.9 – Application de transfert entre comptes

```

        solde_client1 -= montant; solde_client2 += montant;
        // connection à la base de données pour mise à jour du compte
        client1.MetAJour (ID_client1, solde_client1);
        client2.MetAJour (ID_client2, solde_client2);
    }
}

[Transaction(TransactionOption.Required)
public classe Compte : ServiceComponent {
    public void Connect (String ctx_client) {
        // ouverture de la connexion à la base de donnée
    }
    public int ConsulteCompte (int ID_client) {
        // connexion à la base de données pour lire le solde du compte
    }
    [autocomplete(true)]
    public void MetAJour (int ID_Client, int montant) {
        // connexion à la base de données pour modification du solde du compte
        // fermeture de la connexion à la base de données
    }
}

```

Il n'est pas nécessaire d'insérer dans le code du programme une manipulation explicite des transactions, celle-ci se faisant par l'utilisation de l'attribut Transaction sur une classe (dans l'exemple sur les classes TdF et Compte). Un composant peut adopter cinq comportements différents face à la création ou à l'utilisation d'une transaction. Ils sont décrits

par les cinq options de l'attribut Transaction suivant :

- Disabled : les instances de ce composant n'utilisent pas de transaction et ne doivent pas être appelées pendant une transaction.
- NotSupported : les instances de ce composant n'utilisent pas de transaction mais peuvent être utilisées pendant une transaction. Attention, la transaction n'est pas propagée aux instances de composants appelées par un composant en mode transactionnel NotSupported
- Supported : les instances de ce composant n'utilisent pas de transaction et peuvent être utilisées pendant une transaction. La transaction est propagée aux instances de composants appelées par un composant en mode transactionnel Supported.
- Required : les instances de ce composant doivent être exécutées en mode transactionnel. Soit l'appel de méthode véhiculait un contexte transactionnel et cette instance est enrôlée dans la transaction, soit l'appel n'était pas dans une transaction et une transaction est alors créée pour exécuter cette méthode. Toutes les connexions avec des bases de données utilisées pendant cet appel seront enrôlées dans la transaction courante.
- RequiredNew : un appel de méthode sur une instance de ce composant provoque la création d'une nouvelle transaction. Toutes les connexions avec des bases de données utilisées pendant cette transaction seront enrôlées dans celle-ci.

Il est aussi possible de modifier l'option transactionnelle de chaque instance par programme et de créer explicitement par programme des transactions.

Si la transaction a été créée par le moniteur transactionnel de COM+ suite aux attributs posés sur les composants, la fin de la transaction correspond à la fin de la méthode ayant provoqué sa création. Il est possible de signaler le mode de terminaison d'une méthode exécutée dans une transaction. L'attribut AutoComplete positionné sur la méthode MetAJour () permet de provoquer un arrêt de la transaction si cette méthode ne se termine pas normalement. En cas de terminaison normale la transaction se poursuit. Il est aussi possible de provoquer par programme un arrêt de la transaction :

```
// pas d'attribut pour cette méthode
public void MetAJour (int ID_Client, int montant) {
    ContextUtil.DeactivateOnReturn = true;
    // connexion à la base de données pour modification du solde du compte
    // fermeture de la connexion à la base de données
    // si (l'opération s'est bien déroulée) alors la transaction peut continuer
    ContextUtil.SetComplete ();
    // sinon la transaction doit être arrêtée
    ContextUtil.SetAbort ();
}
```

6.5 Exemple : Mise en œuvre d'une application complète

6.5.1 Architecture générale

Afin de décrire l'application d'exemple que nous allons mettre en œuvre, nous nous appuierons sur le cahier des charges suivant :

- Nous disposons (virtuellement) d'une bibliothèque de synthèse vocale (1) pour la plateforme Windows XP que nous appellerons TTS. Nous pouvons écrire une application (2) pour Windows XP qui inclut directement cette bibliothèque.
- Certaines autres applications (3) de l'Intranet sécurisé nécessitent également l'utilisation d'une bibliothèque de synthèse vocale, mais nous ne possédons pas de version de la bibliothèque pour ces plateformes (exemple d'un PDA). Nous créerons donc un service .NET Remoting (4) encapsulant les fonctionnalités de la bibliothèque TTS. Ce service sera distribué via un canal TCP Binaire permettant un accès rapide pour les clients de l'Intranet.
- Nous désirons offrir ce service à des applications tierces en dehors de la zone sécurisée de l'Intranet (5). Nous voulons mettre à disposition ce service de Synthèse vocale pour tout type de plate-forme (un client Java sous linux par exemple). Nous mettons alors en place un Service Web (6) constituant ainsi une passerelle standardisée vers le service .NET Remoting.

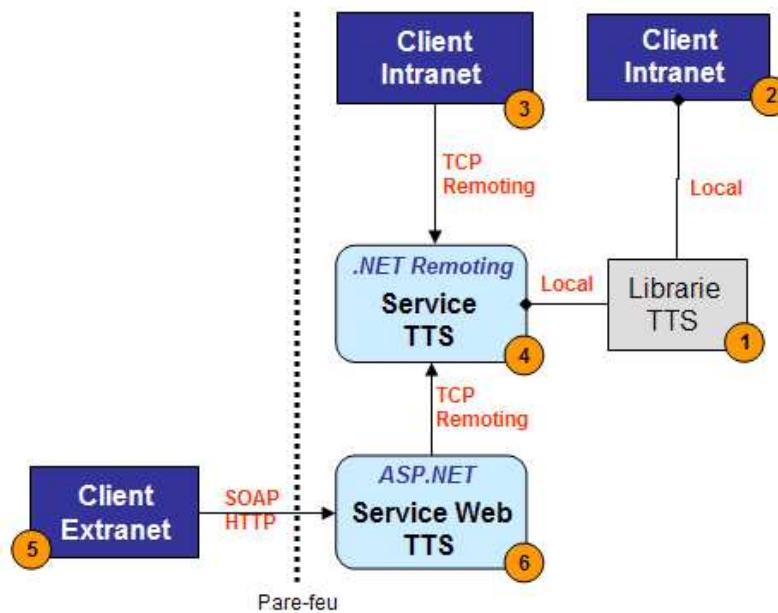


Figure 6.10 – Architecture générale de l'application

L'avantage de déporter le traitement d'un service réside souvent dans le fait que les clients de ce service ne disposent pas eux-mêmes des fonctionnalités leur permettant d'effectuer ce traitement sur le terminal sur lequel ils tournent.

6.5.2 Implémentation et déploiement des services

Etapes de la mise en œuvre

Avant de décrire en détails comment implémenter cette architecture, nous présentons ici le processus de développement utilisé. Nous développerons cette application en essayant de modulariser le plus possible les différentes parties de l'architecture. Les choix de conception

sont donnés ici à titre indicatif mais peuvent être mis en œuvre de manière assez générale.

Nous commencerons tout d'abord par définir et implémenter les types de données échangées par les différentes entités. Dans un deuxième temps, nous définirons l'interface du service. Ces deux premières bibliothèques constitueront le sous ensemble minimal à partager entre client et serveur dans notre application. À partir de ces deux briques, nous développerons et déployerons le service .NET Remoting. Cela consistera en une bibliothèque pour l'implémentation du service ainsi qu'une application pour héberger le service. Ensuite, nous écrirons un client pour ce service afin de le tester. Afin de dépasser les frontières de l'intranet, nous implémenterons ensuite le service web qui redirigera les appels clients vers le service .NET remoting. Enfin nous écrirons un client simple de ce service web.

Les types de données échangées

Le client enverra sa requête sous la forme d'un objet Sentence contenant un texte à convertir vocalement ainsi que la langue dans laquelle la synthèse vocale s'effectue. En réponse le service renverra un objet TTSResult dans lequel le flux audio sera représenté par un tableau d'octets.

```
using System;
using System.Xml;
using System.Xml.Serialization;

namespace TTS.Schema {
    [Serializable]
    [XmlType("sentence")]
    public class Sentence {
        public Sentence(){}
        public Sentence(string lang, string text) {
            this.Language=lang;
            this.Content=text;
        }

        [XmlAttribute("lang")]
        public string Language;

        [XmlElement("content")]
        public string Content;
    }

    [Serializable]
    [XmlType("result")]
    public class TTSResult {
        public TTSResult(){}
        public TTSResult(byte[] stream) {
            this.AudioStream=stream;
        }

        [XmlElement("audio")]
        public byte[] AudioStream;
```

```

    }
}

```

Chaque instance de ces données devra traverser les frontières du service et donc pouvoir être sérialisé. Les deux classes sont donc marquées par l'attribut C# `Serializable`. Selon la configuration dans laquelle sera publié notre service, la sérialisation pourra prendre plusieurs formes. Si les données sont sérialisées via un formateur binaire, le résultat sera une série d'octets suivant une spécification propriétaire du Remoting .NET. Dans le cas où le formateur de données sera un formateur SOAP, les données seront encodées sous forme de document XML. Le framework s'occupe par défaut de tout ce qui a trait à cette sérialisation. Par contre, si l'on veut pouvoir contrôler le format de ce document XML, nous devons marquer la classe ainsi que ses champs et méthodes par des attributs de sérialisation. Par exemple une instance de la classe `sentence` contenant "EN" dans le champ `Language` et "Bonjour" dans le champ `Content` sera transformée en XML comme suit :

```

<sentence lang="EN">
  <content>BONJOUR</content>
</sentence>

```

Si aucun attribut de contrôle de la sérialisation XML n'avait été employé, le framework aurait par défaut utilisé les noms de classe, champs et méthode, ce qui formaterait les instances comme ci-dessous :

```

<Sentence Language="EN">
  <Content>BONJOUR</Content>
</Sentence>

```

L'interface du service (partagée entre client et serveur)

```

using System;

namespace TTS.ServiceInterface {
  using TTS.Schema;

  public interface ISpeechService {
    TTSResult Speak(Sentence sentence);
  }
}

```

L'implémentation du service .NET Remoting (4)

Il s'agit ici d'implémenter l'interface du service sous la forme d'une classe héritant de la classe `MarshalByRefObject` afin qu'elle puisse être passée par référence.

```

using System;
using TTSLibrary;
using TTS.Schema;
using TTS.ServiceInterface;

```

```

namespace TTS.Service.Remoting {
    public class SpeechService: MarshalByRefObject, ISpeechService {
        public SpeechService() {
            Console.WriteLine("Instance de 'SpeechService' créée.");
        }

        public TTSResult Speak(Sentence sentence) {
            TTSService speaker=new TTSService(sentence.Language);
            return new TTSResult(speaker.DoSpeak(sentence.Content));
        }
    }
}

```

Déploiement du service .NET Remoting

Pour rendre le service accessible, nous devons choisir de le déployer dans un conteneur. Ce conteneur peut être au choix une application, un service Windows ou encore le serveur web IIS. Nous allons déployer notre service dans une application Console. Cette application une fois exécutée hébergera les instances du service.

Nous avons vu dans la section précédente comment paramétrier ce déploiement par programmation. Nous montrons ici comment effectuer ce paramétrage à l'aide d'un fichier de configuration. Cette méthode est beaucoup plus souple puisque le service n'a pas besoin d'être recompilé afin d'être déployé différemment. L'application hôte consistera uniquement à charger le fichier de configuration et à se bloquer (ici par l'appel à `Console.ReadLine()`) afin que l'application soit toujours disponible.

```

using System;
using System.Runtime.Remoting;

namespace TTS.Service.Remoting {
    public class MainClass {
        [STAThread]
        static void Main(string[] args) {
            RemotingConfiguration.Configure("server.config");
            Console.WriteLine("Appuyez sur [Entrée] pour quitter.");
            Console.ReadLine();
        }
    }
}

```

En .NET, chaque application dispose de son propre fichier de configuration. Ce fichier respecte une grammaire XML composée de différentes sections. En particulier nous nous intéressons ici à la section `<system.runtime.remoting>` permettant de paramétrier le mode d'exécution de notre service. La partie `<channels>` permet de spécifier l'ensemble des canaux de communication gérés par le framework .NET Remoting au sein de cette application. Nous utiliserons ici deux canaux différents permettant aux clients d'atteindre le service via deux ports différents : un canal ouvert sur le port 9000 et dialoguant via le protocole HTTP, un autre dialoguant sur le port 9001 via un canal

TCP. Vient ensuite la section <services> permettant de décrire le mode de publication de notre service. Nous le déployons dans notre exemple dans un mode Singleton en spécifiant l'assembly contenant l'implémentation du service ainsi que l'adresse relative à l'URL implicitement définie par les canaux de communications. Dans notre exemple le service sera accessible respectivement aux adresses `http ://localhost :9000/speech` et `tcp ://localhost :9001/speech`.

Fichier de configuration du serveur (server.config)

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <system.runtime.remoting>
    <application>
      <channels>
        <channel port="9000" ref="http" />
        <channel port="9001" ref="tcp" />
      </channels>
      <service>
        <wellknown
          mode="Singleton"
          type="TTS.Service.Remoting.SpeechService, TTS.Service.Remoting"
          objectUri="speech" />
      </service>
    </application>
  </system.runtime.remoting>
</configuration>
```

Via ce fichier de configuration, le composant en mode singleton sera accessible de deux manières différentes : soit via un canal HTTP sur port 9000, soit via un canal TCP sur le port 9001. Les deux URI correspondantes seront respectivement `http ://localhost :9000/speech` et `tcp ://localhost :9001/speech`.

6.5.3 Client du service .NET Remoting (3)

Le Code du client

```
ISpeechService service=(ISpeechService)Activator.GetObject(typeof(ISpeechService),url);
TTSResult result=service.Speak(new Sentence("fr",textBox1.Text));
```

Fichier de configuration du client (client.config)

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <system.runtime.remoting>
    <application>
      <client>
        <wellknown
          type="TTS.ServiceInterface.ISpeechService, TTS.ServiceInterface"
          url="http://localhost:9000/speech"/>
      </client>
    </application>
  </system.runtime.remoting>
</configuration>
```

```
</application>
</system.runtime.remoting>
</configuration>
```

6.5.4 Implementation du Service WEB (6)

Nous allons à présent créer le service web permettant d'accéder aux fonctionnalités de notre service en dehors de l'intranet de l'entreprise. Selon notre schéma, ce service web a pour rôle de rediriger ses requêtes vers le service .NET Remoting implémenté précédemment. Ainsi son code fonctionnel est très similaire au code d'un client .NET Remoting.

Le Code du service web

La classe du web service

```
using System;
using System.Runtime.Remoting;
using System.Runtime.Remoting.Channels;
using System.Runtime.Remoting.Channels.Tcp;
using System.Web.Services;

using TTS.Schema;
using TTS.ServiceInterface;

public class TTSService : WebService {
    [WebMethod]
    public TTSResult Speak(Sentence sentence) {
        //RemotingConfiguration.Configure("client.config");
        ISpeechService remoting_service=
            (ISpeechService)
            Activator.GetObject(typeof(ISpeechService),
                "tcp://localhost:9001/speech");
        return remoting_service.Speak(sentence);

    }
}
```

La page d'accès au webservice

```
<%@ WebService CodeBehind="TTSWebService.cs" Class="TTSService"%>
```

Compilation du service web

Notre service web dépend des différentes bibliothèques compilées précédemment. Il faut placer ces bibliothèques dans le répertoire **bin** de l'application web hébergement le service web.

Structure du répertoire de l'application

```
\www_icar
  \bin
    TTS.Schema.dll      --> Schémas de donnée
    TTSSpeechService.dll --> Interface du service .NET Remoting
    TTSWebService.dll   --> Le web service compilé
    tts.asmx            --> La page d'accès
    TTSWebService.aspx  --> Le code c# du web service
```

La page d'accès au webservice

```
set PATH=C:\WINDOWS\Microsoft.NET\Framework\v1.1.4322;%PATH%
SET LIB_SCHEMA=bin\TTS.Schema.dll SET
LIB_INTERFACE=bin\TTS.SpeechService.dll
SET OUTPUT=bin/TTSWebService.dll SET SOURCES=TTSWebService.cs
csc.exe /reference:%LIB_SCHEMA% /reference:%LIB_INTERFACE% \
/reference:System.Runtime.Remoting.dll /out:%OUTPUT% \
/t:library %SOURCES%
pause
```

Déploiement du service web

Le service doit être hébergé dans une application ASP.NET sous le contrôle du serveur web IIS de Microsoft. La console d'administration des services Internet permet de créer ce répertoire que nous nommerons **icar**. L'arborescence de cette application doit contenir un sous-répertoire **bin** contenant tous les assemblages .NET nécessaires au fonctionnement du service web. Dans notre cas, ce répertoire devra contenir la bibliothèque contenant le web service **TTS.WebService.dll**, la bibliothèque **TTS.ServiceContract** ainsi que la bibliothèque **TTS.SpeechService** contenant l'interface du service. Dans un deuxième temps nous écrivons une page web avec l'extension **asmx** permettant de décrire le service web. Nous nommerons cette page **ttsservice.asmx**.

Le service est alors disponible à l'url **http://hostname:80/icar/ttsservice.asmx**. Sa description WSDL peut être récupérée à l'adresse **http://hostname:80/icar/ttsservice.asmx?WSDL**. Notre service web est désormais prêt à l'emploi.

6.5.5 Ecriture du client du Service web

Génération du proxy du service web via l'outil en ligne de commande **wsdl.exe**.

```
using System;
namespace TTS.ClientWebService {
  /// <summary>
  /// Description résumée de Class1.
  /// </summary>
```

```

class Class1 {
    /// <summary>
    /// Point d'entrée principal de l'application.
    /// </summary>
    [STAThread]
    static void Main(string[] args) {
        TTS.WebService.sentence request = new TTS.WebService.sentence();
        request.content = "Salut";
        request.lang = "FR";
        TTS.WebService.result result =
            new TTS.WebService.TTSService().Speak(request);
        Console.ReadLine();
    }
}
}

```

6.6 Evaluation

6.6.1 .Net et les langages à objets

Comme on l'a vu dans les parties précédentes, .NET offre, grâce à une machine virtuelle, les mécanismes nécessaires à la représentation de données et aux échanges de données entre modules écrits dans des langages différents. Le but de cette première partie d'une évaluation de .NET est de s'assurer que les comportements des principaux langages (VB, C++, C# et J#) de la plate-forme .NET sont conformes à ceux attendus, en particulier pour l'un des points essentiels de la programmation objet : la redéfinition et la surcharge.

On parle de redéfinition lorsqu'une méthode possède plusieurs définitions qui peuvent participer à la liaison dynamique. Le compilateur ne peut pas choisir au moment de la compilation et met en place une procédure de *lookup* qui sera déclenchée lors de l'exécution du programme pour sélectionner la méthode appropriée ; en général, celle qui se trouve le plus bas dans le graphe d'héritage. Pour la surcharge il s'agit en fait de l'utilisation d'un même nom de méthode mais pour des méthodes que le compilateur considère comme différentes et qui ne sont donc pas éligibles en même temps lors de la liaison dynamique. Les langages à objets font des choix différents ; les redéfinitions peuvent être invariantes, covariantes ou contravariantes et la surcharge est parfois interdite. Pour mettre ces différences en évidence, Antoine Beugnard⁹ propose une expérimentation simple.

Nous considérons deux classes : Up, et Down qui est une sous-classe de Up. Chacune des deux classes offre deux méthodes cv et ctv qui requièrent un paramètre unique. Les paramètres sont des instances de trois classes : Top, Middle qui est une sous-classe de Top et Bottom qui est une sous-classe de Middle. Pour tester tous les cas (covariants et contravariants) nous définissons les classes Up et Down selon la figure 6.11. La méthode cv paraît covariante car les classes où elles sont définies et les classes des paramètres varient dans le même sens, tandis que la méthode ctv paraît contravariante puisque les classes des paramètres varient dans le sens inverse des classes de définition de ctv.

Notre but n'est pas ici de discuter l'avantage de tel ou tel choix fait par les langages à objet, mais simplement de montrer qu'ils offrent de nombreuses sémantiques et que le

⁹L'ensemble de l'étude est disponible à l'adresse : <http://perso-info.enst-bretagne.fr/~beugnard/papier/lb-sem.shtml>

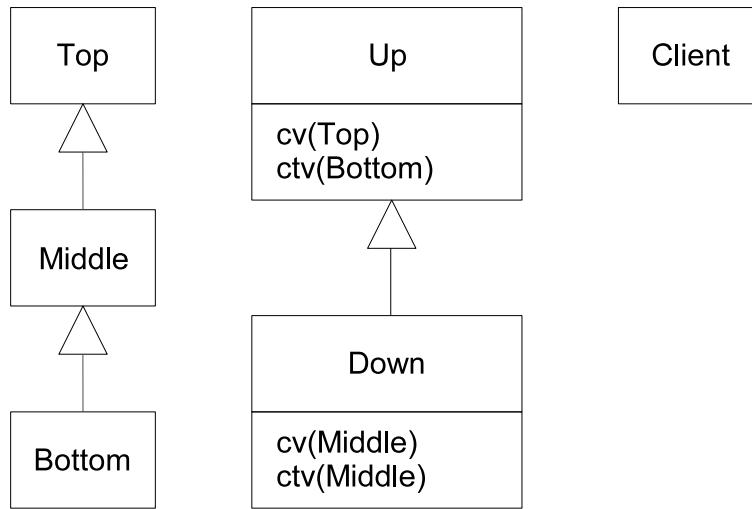


Figure 6.11 – Les classes utilisées pour l’expérimentation

framework .Net les respecte. Pour illustrer cela on crée, pour chacun des langages étudié, un programme client qui instancie les objets puis essaye tous les appels possibles : les deux méthodes (`cv` et `ctv`) étant appelées successivement avec les trois classes de paramètres possibles (`Top`, `Middle` et `Bottom`) pour les trois objets récepteurs possibles : `u` de type `Up` instancié avec un objet de classe `Up`, `d` de type `Down` instancié avec un objet de classe `Down` et `ud` de type `Up` mais instancié avec un objet de classe `Down`. Le dernier cas est autorisé car `Down` est une sous-classe de `Up`. Ce code peut parfois ne pas compiler ou ne pas s’exécuter lorsque le système de type l’interdit.

Les résultats sont présentés sous la forme d’un tableau de 6 lignes (une par appel possible) et 3 colonnes (une par objet récepteur possible). Chaque case contient la classe dans laquelle a été trouvée la méthode effectivement exécutée. Les sources et les résultats des exécutions peuvent être lus sur <http://perso-info.enst-bretagne.fr/~beugnard/papier/lb-sem.shtml>. Ces tableaux sont appelés *signature du langage* (vis-à-vis de la redéfinition et de la surcharge).

Nous reproduisons dans les tables 6.1 à 6.4 les signatures pour C++, C#, J# et VB.Net. Ce sont les mêmes qu’avec d’autres compilateurs comme `gcc` ou `javac` (jusqu’à la version 1.3) par exemple.

C++	u	d	ud
<code>cv(Top)</code>	<code>Up</code>	Compil. error	<code>Up</code>
<code>cv(Middle)</code>	<code>Up</code>	<code>Down</code>	<code>Up</code>
<code>cv(Bottom)</code>	<code>Up</code>	<code>Down</code>	<code>Up</code>
<code>ctv(Top)</code>	<code>Compil. error</code>	<code>Compil. error</code>	<code>Compil. error</code>
<code>ctv(Middle)</code>	<code>Compil. error</code>	<code>Down</code>	<code>Compil. error</code>
<code>ctv(Bottom)</code>	<code>Up</code>	Down	<code>Up</code>

TAB. 6.1 – signature de C++

On notera avec intérêt que bien que proches, ces langages ont tous une deuxième co-

C#	u	d	ud
cv(Top)	Up	Up	Up
cv(Middle)	Up	Down	Up
cv(Bottom)	Up	Down	Up
ctv(Top)	Compil. error	Compil. error	Compil. error
ctv(Middle)	Compil. error	Down	Compil. error
ctv(Bottom)	Up	Down	Up

TAB. 6.2 – signature de C#

J#	u	d	ud
cv(Top)	Up	Up	Up
cv(Middle)	Up	Down	Up
cv(Bottom)	Up	Down	Up
ctv(Top)	Compil. error	Compil. error	Compil. error
ctv(Middle)	Compil. error	Down	Compil. error
ctv(Bottom)	Up	Compil. error	Up

TAB. 6.3 – signature de Java before 1.3 (et J#)

lonne différente ! La différence entre ces 4 langages est due à des interprétations différentes de la surcharge. En effet, la troisième colonne montre que `cv` et `ctv` sont en fait des surcharges ; il aurait fallu voir apparaître des `Down` dans la troisième colonne pour avoir de la liaison dynamique et donc des redéfinitions. Ces 4 langages n'acceptent que des redéfinitions invariantes. Des expériences avec d'autres langages à objets (comme ADA95, Dylan, Eiffel, OCaml ou Smalltalk) montrent une plus grande variété encore.

Ce qui nous intéresse ici est de conclure que le framework .Net permet de définir plusieurs sémantiques et que les langages existants hors du framework ont la même sémantique qu'avec le framework (C++ et J#/Java version 1.3).

6.6.2 Compositions multi-langages en .Net

Comme nous venons de le voir, les interprétations de la redéfinition et de la surcharge dans les langages à objets sont nombreuses. Quelle conséquence cela peut-il avoir lorsqu'on assemble des morceaux de logiciel écrits dans des langages différents ? Le framework .Net met en avant la notion de composants. Ces composants peuvent être écrits avec plusieurs langages et leurs interfaces doivent assurer leur interconnexion. Mais, comme on va le voir avec .NET, les modèles à composants n'ont pas un comportement neutre vis-à-vis des langages d'implémentation de chacun des composants. En effet, la différence de comportement des langages à objets en ce qui concerne la redéfinition et la surcharge n'est pas prise en compte dans les interfaces des composants. Pour mettre cela en évidence, il suffit de construire la petite expérience suivante, toujours reprise du document d'habilitation d'Antoine Beugnard.

Imaginons le découpage de la figure 6.12. Les quatre classes `Up`, `Top`, `Middle` et `Bottom` de la partie précédente définissent un framework **C1** élaboré dans le langage *L1*. Par la suite, dans le cadre d'une évolution du framework, la classe `Up` est étendue par la classe

VB.Net	u	d	ud
cv(Top)	Up	Up	Up
cv(Middle)	Up	Down	Up
cv(Bottom)	Up	Down	Up
ctv(Top)	Compil. error	Compil. error	Compil. error
ctv(Middle)	Compil. error	Down	Compil. error
ctv(Bottom)	Up	Up	Up

TAB. 6.4 – signature de Visual Basic

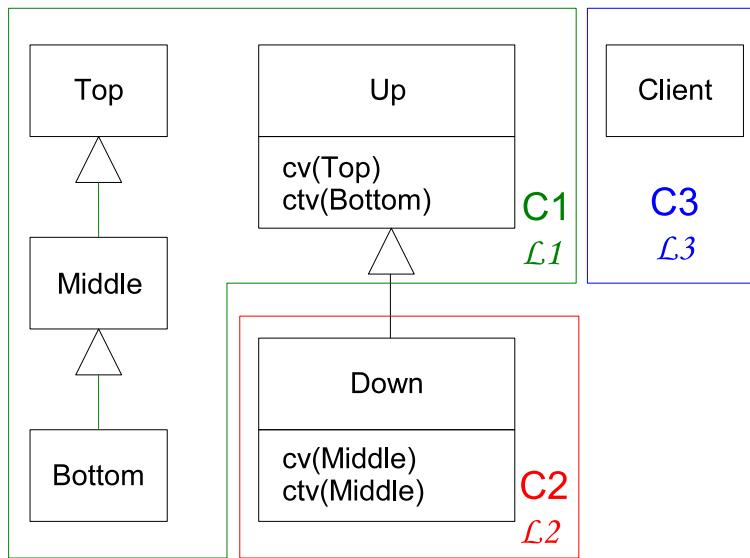


Figure 6.12 – Le découpage avec plusieurs langages

Down dans le langage L_2 pour former le composant **C2**. Cette extension, multi-langage est tout à fait permise par la plate-forme .Net. Enfin, pour compléter l’expérimentation un client **C3** est écrit dans le langage L_3 . Pour mettre en œuvre cette expérimentation nous avons utilisé les langages VB, C++ et C# qui forment le trio des langages de base de la plate-forme .Net. Nous obtenons donc un total de 27 ($3 \times 3 \times 3$) assemblages possibles. Pour présenter les résultats, nous construisons un tableau de 9 signatures pour chacun des 3 clients (**C3**). Les colonnes de chaque tableau représentent les langages d’implantation du framework de référence **C1** et les lignes, les langages d’implantation de l’extension **C2**. Chaque cellule du tableau contient la signature observée.

Dans le tableau suivant, le client est en C#.

Extension/Framework	C#	VB	C++
C#	C#	C#	C++
VB	C#	C#	C++
C++	C#	C#	C++

On constate que le comportement attendu par le client est majoritairement celui de C#, mais que dans le cas où le framework (**C1**) est écrit en C++, l’anomalie d’héritage de

la signature de C++ (erreur de compilation ligne 1, colonne 2 de la table 6.1) est toujours présente.

Dans le tableau suivant, le client est en Visual Basic.

Extension/Framework	C#	VB	C++
C#	VB	VB	C++
VB	VB	VB	C++
C++	VB	C#	C++

On constate que le comportement attendu par le client est majoritairement celui de Visual Basic, mais que dans le cas où le framework (**C1**) est écrit en C++, l'anomalie d'héritage de la signature de C++ est encore présente. Un nouveau comportement apparaît lorsque le framework est écrit en Visual Basic et l'extension en C++, le client Visual Basic observe un comportement “à la” C#.

Dans le tableau suivant, le client est en C++.

Extension/Framework	C#	VB	C++
C#	VB/C++	VB/C++	VB/C++
VB	C++	C++	C++
C++	C++	C++	C++

On constate que le comportement attendu par le client est majoritairement celui de C++, mais que dans le cas où l'extension (**C2**) est écrit en C#, un nouveau comportement apparaît : la signature observée par le client est un mélange de C++ et de VB (erreur de compilation comme C++ ligne 1, colonne 2 table 6.1 et choix de Up comme VB ligne 6 colonne 2 table 6.4). Cette signature ne correspond à aucun des 13 langages à objet étudiés par Antoine Beugnard.

Au delà de la compréhension fine des résultats, cette rapide expérience montre que pour pouvoir prévoir le comportement d'un assemblage, le client (et donc l'utilisateur) doit avoir connaissance de la manière dont la surcharge est interprétée, et donc du langage d'implémentation de la partie serveur. Il pourrait être intéressant de pousser plus avant cette expérimentation avec les services web, pour vérifier là aussi la réelle neutralité de la plate-forme vis-à-vis des langages d'implémentation.

6.6.3 Quelques éléments de performances

L'évaluation d'un langage et d'une plate-forme est quelque chose de complexe. Cette courte section ne se donne pas comme ambition d'évaluer toutes les caractéristiques mais de donner quelques éléments de référence. Une étude publiée dans www.gotdotnet.com à propos de l'application PetStore qui servait d'exemple à la plate-forme J2EE titrait : “.NET, 10 fois plus rapide que J2EE”. N'ayant pas les capacités pour reproduire sur une autre application significative cette étude nous avons dans un premier temps mesuré le cout des appels de méthodes. La figure suivante donne quelques valeurs comparatives entre un appel de méthode local en Java et en .NET, puis entre un appel distant entre Java RMI et .NET Remoting, la taille des paquets d'appels évoluant.

Attention il s'agit d'une échelle logarithmique (cf. figure 6.13). Nous n'arrivons pas à expliquer les différentes discontinuités présentes dans les mesures .NET en local.

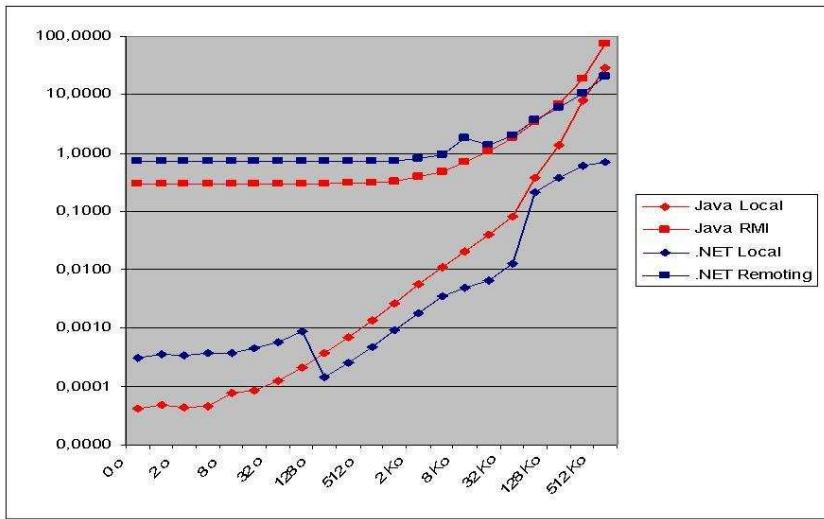


Figure 6.13 – Coût d'un appel de méthode

Voici deux autres éléments complémentaires incluant des utilisations plus complexes du langage et de la plate-forme. L'ensemble des résultats, y compris avec les principaux fichiers sources, est disponible à l'adresse : <http://www.dotnetguru.org/articles/Comparatifs/benchJ2EDotNET/J2EEvsNETBench.html>.

Etude du démarrage d'une application graphique

L'application utilisée pour ce test est simpliste par ses fonctionnalités mais permet de mesurer le temps de démarrage d'initialisation de la partie graphique d'un client lourd (cf. figure 6.14). Elle consiste à charger un formulaire permettant d'enregistrer un utilisateur, photos comprise dans une base. Sans équivoque la version C# avec Winforms a été nettement plus rapide que la version Java avec Swing.

Efficacité de la mise en œuvre d'un service web

Pour évaluer celle-ci, l'auteur du papier a choisi d'appeler une fonction permettant de trouver le minima d'un tableau d'entiers. Ce service web a été implanté en .NET et à l'aide de la plate-forme Java/axis puis en Java avec la plate-forme Glue (www.themindelectric.com). A été mesuré uniquement (cf. figure 6.15) le temps de réponse du service sans vouloir décomposer celui-ci entre les différentes activités : empaquetage/dépaquetage, transfert des données, activation du service, traitement du côté serveur.

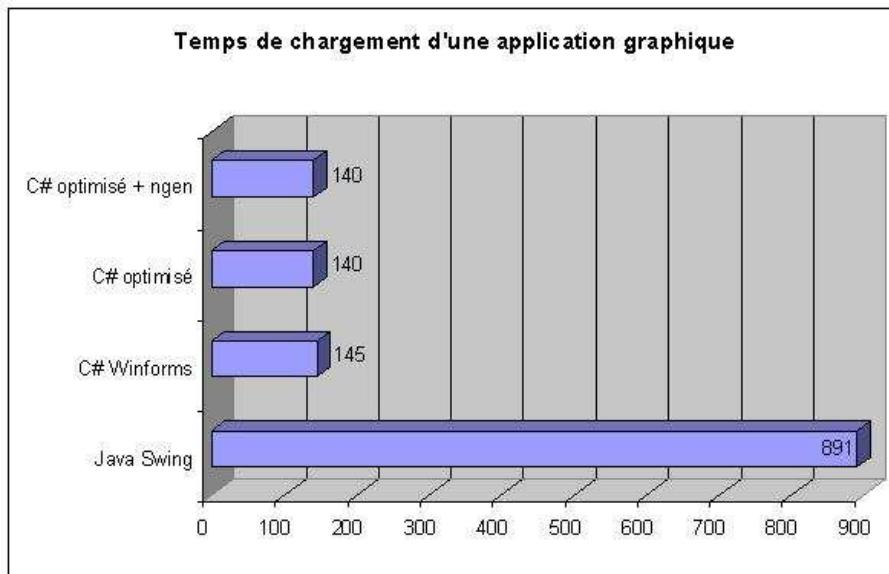


Figure 6.14 – Coût démarrage application graphique

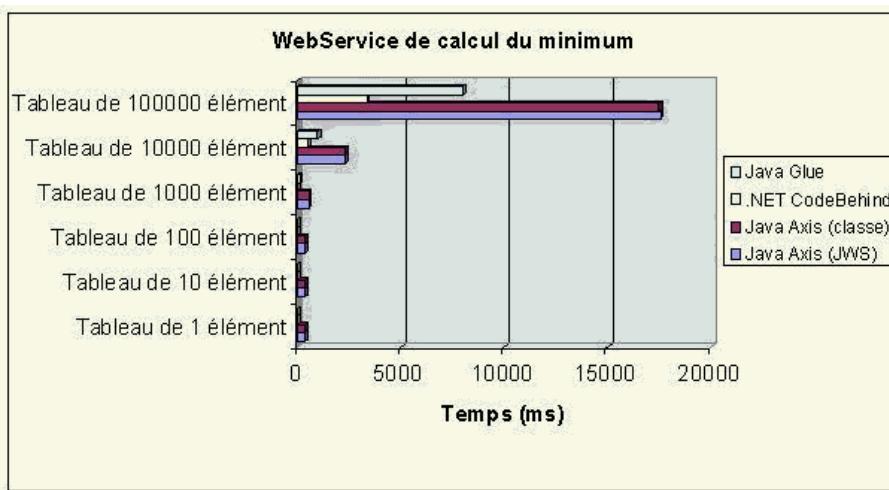


Figure 6.15 – Cout d'utilisation d'un service web

6.6.4 La plate-forme .NET 2.0 et Indigo : nouveautés à venir

Une nouvelle plate-forme de distribution

L'architecture distribuée orientée Services Web (S.O.A) repose sur quatre principes fondamentaux :

- un couplage faible entre services (fournis par SOAP et WSDL) ;
- un bus de communication synchrone (HTTP) ou asynchrone (Middleware Oriented Messages : MOM)
- une invocation des services de type échange de documents (XML, RPC)
- des processus métiers qui pilotent l'assemblage de services Web (Business Process Management : BPM).

Le futur de la plate-forme .NET (appelé Indigo) renforce encore plus le souhait de Microsoft de se positionner comme un acteur dans une approche de programmation orientée services. L'objectif d'Indigo est de choisir les meilleures abstractions pour construire des logiciels orientés processus en intégrant de manière plus étroites les services nécessaires à ces applications (bus de messages, sécurité et transaction) en s'appuyant de manière explicite sur les travaux réalisés par le consortium Oasis. Indigo fait lui-même une partie de la nouvelle version de Windows nommée Longhorn.

Pour atteindre cet objectif, Indigo unifie l'ensemble des « attributs » auparavant disponibles pour construire un service. Par exemple, pour publier un web service, il était nécessaire de faire précéder chaque méthode exposée de l'attribut `[WebMethod]`, de même, un composant qui devait utiliser le service transactionnel devait être précédé de l'attribut `[Transaction]`. Indigo propose d'introduire la notion de contrat :

```
Using System.ServiceModel;

[ServiceContract]
public class AnnuaireWebService {
    private AnnuaireService service;
    public AnnuaireWebService() {
        service=new AnnuaireService();
    }
    [OperationContract]
    public CarteDeVisite Rechercher(string nom) {
        return service.Rechercher(nom); ;
    }
}
```

Les contrats peuvent porter sur les classes ou sur les méthodes, mais aussi sur les données échangées afin de permettre le transport de données structurées. Pour chaque points de publication d'un service, sans avoir à modifier le code de celui-ci, ni du client qui le consomme il sera possible de préciser dans un fichier de configuration : le protocole de transport (par exemple : HTTP), l'adresse du service mais aussi les services techniques associés aux appels (sécurité ou transaction) basés sur les spécifications du W3C pour les Web services.

La construction d'un client Indigo sera identique (et proche de la manière de procéder actuellement pour le Web services) : à partir d'un proxy du service généré à partir du fichier de publication du service, le client pourra accéder au service de la même manière quelque soit le protocole utilisé (.NET remoting ou SOAP).

Evolution du langage de référence : C# version 2.0

C# va, avec la version deux de la plate-forme .NET, être complété par l'introduction de très nombreuses nouveautés : la généricité, les itérateurs, les méthodes anonymes et les classes partielles pour ne citer que les améliorations les plus notables.

Généricité En C++ les templates sont extrêmement utilisés et le fait qu'ils n'ont pas été implanté sous C# a été un des reproches majeurs à sa sortie. Microsoft a écouté les utilisateurs et l'a mis en place en nommant dorénavant la technique Generic. Pour contourner l'absence de généricité, les programmeurs C# utilisaient actuellement des collections d'objets non typés et sollicitaient massivement l'usage coûteux du boxing/unboxing et de cast. L'introduction de la généricité devrait permettre d'augmenter le typage statique des programmes et d'augmenter simultanément l'efficacité de ceux-ci.

La généricité sera utilisée pour simplifier l'écriture des classes de collections qui puissent être énumérable (c'est à dire interrogables dans une boucle foreach).

Méthodes génériques Les méthodes anonymes sont aussi une manière d'alléger le code en particulier dans la gestion des événements. Actuellement, il était nécessaire de déclarer un 'callback d'événement' par une méthode qui possédait un nom. Dorénavant, il suffira juste de donner le code à exécuter :

```
void Page_Load (object sender, System.EventArgs e) {
    this.Button1.Click += delegate(object dlgSender, EventArgs dlgE) {
        Label1.Text = " Vous avez cliqué sur le bouton 1 !";
    }
};
```

Classes partielles Les classes partielles permettront à une équipe de développeurs de travailler en commun : la même classe sera contenue dans plusieurs fichiers. En reprenant l'exemple de la section précédente, on trouvera par exemple dans le fichier annuaireService.cs le code suivant :

```
public partial class AnnuaireService : IAnnuaireService {
    // constructeur de la classe
    public AnnuaireService(){}
    // implementation de la méthode métier
    public CarteDeVisite Rechercher(string nom) {
        AnnuaireDAL dal=new AnnuaireDAL();
        return dal.ExtraireCarte(nom);
    }
}
```

Le fichier AnnuaireAdminService.cs, pourra lui contenir :

```
public partial class AnnuaireService : IAnnuaireAdminService {
    // implementation de la méthode métier
    public void Ajouter(CarteDeVisite carte) {
        AnnuaireDAL dal=new AnnuaireDAL();
        // on vérifie que la carte existe pas déjà
        CarteDeVisite test=dal.ExtraireCarte(carte.Nom);
        if (test==null)
```

```

        dal.AjouterCarte(carte);
    else
        throw new Exception("La carte existe déjà");
}
}

```

6.7 Conclusion

La terre est envahie de composants COM, et si l'application que vous voulez construire fait un usage intensif de cet univers là alors .NET est fait pour vous. Vous y trouverez des outils de développement sophistiqués, des langages de programmation adaptés à vos besoins, la possibilité de construire un composant en utilisant plusieurs langages de programmation grâce au langage de type commun. Mais même si vous souhaitez écrire une nouvelle application .NET peut aussi être fait pour vous, voici quelques apports :

- Point d'accès : c'est le point fort par excellence de l'approche .NET, n'importe quel client Web peut accéder à un service publié sur le Web. Il n'est pas nécessaire d'avoir une configuration spécifique, d'installer une pile de protocole, de configurer un pare-feu (firewall). L'intégration avec les standards du Web est totale, un client XML et la disponibilité prochaine du protocole SOAP permettront cet usage et rendront peut-être inutile la programmation de script CGI, ceux-ci étant remplacés par des composants .NET.
- Simplicité d'utilisation : .NET a été conçu dès l'origine comme une plate-forme à composants distribués intégrant les standards de l'Internet XML et SOAP. Pour cela il n'est pas nécessaire de d'envelopper (wrapper) les composants afin de les rendre accessibles, il n'est pas non plus nécessaire d'ajouter du code lors du déploiement pour permettre l'exécution du composant car celui-ci est inclus dans la plate-forme ou directement associé avec les composants. Un composant est un objet simple qui contient tout ce qui est nécessaire pour le manipuler : la description de ses interfaces, la description de ses services, le code pour accéder aux services et les services eux mêmes. Le déploiement est immédiat. .NET propose aussi avec ADO un modèle très évolué pour l'accès aux données acceptant le mode déconnecté, permettant l'accès à différents formats de base de données, etc. Toutes les données peuvent être accéder immédiatement depuis n'importe quel client de la plate-forme qui se charge de les transporter et de les convertir.
- Les performances : selon que l'on s'intéresse à la manière dont le service est rendu ou l'accès au service, le point de vue est diamétralement opposé. En ce qui concerne l'exécution des services, les performances sont au rendez-vous : code compilé, multi-processeur, équilibrage de charge, etc. Par contre, les appels inter-service seront lents par nature. En effet le choix d'utiliser le protocole SOAP afin de pouvoir être accédé depuis n'importe quel type de station cliente se paye au prix fort. La technologie proposée permet essentiellement de relier entre eux des composants ayant une grosse granularité.
- Multiplicité des langages de programmation : .NET ne fait pas le choix d'un langage de programmation à votre place, il vous laisse libre de choisir le langage que vous voulez utiliser en fonction de vos usages ou de votre humeur. COM permettait de faire

coopérer des composants écrits dans des langages différents, .NET permet d'intégrer les langages les uns avec les autres. Tous les langages sont a priori égaux, même si les langages de types procéduraux sont plus proches de la machine virtuelle proposée. Il par exemple est possible d'utiliser simultanément différents langages (un par classe par exemple). Le typage est complet et les différents types du Common Language Runtime, dont par exemple les threads sont disponibles dans les différents langages de programmation de la plate-forme. Des compilateurs C++, C#, VB et Jscript sont aujourd'hui disponibles et différentes compagnies ont déjà annoncé qu'elles allaient sous peu fournir des compilateurs. Il est imaginable que dans peu de temps, des compilateurs APL, ML, Cobol, Oz, Pascal, Perl, Python, Smalltalk et Java seront disponible sur la plate-forme .NET.

Chapitre 7

La plate-forme dynamique de services OSGi

Ce chapitre présente les grands principes de la plate-forme dynamique de services OSGi, ainsi que la programmation des services. Plusieurs services standardisés sont détaillés dans les domaines particuliers pour lesquels ils ont été définis. Le chapitre débute par un bref historique de la spécification OSGi et de son évolution. La section 7.2 rappelle les objectifs initiaux de cette spécification, qui se focalisaient sur un modèle d'administration d'applications déportées. La section 7.3 détaille le conditionnement et le déploiement des applications ainsi que la collaboration au travers des services. La section 7.4 présente la programmation de services OSGi. La section 7.5 détaille trois services déjà standardisés pour le développement d'applications liées à des domaines spécifiques. Le chapitre se termine par une revue des perspectives pour la spécification OSGi.

7.1 Introduction

L'*OSGi¹ Alliance* est un consortium industriel fondé en mars 1999 par une quinzaine de sociétés membres. L'intention originelle de ces sociétés était de définir une spécification ouverte pour développer et déployer des télé-services sur des passerelles résidentielles comme des décodeurs de TV numérique ou des modems-routeurs ADSL. La première version de la spécification² était strictement orientée par ce marché de niche. Néanmoins, cette première spécification s'est rapidement révélée très intéressante pour des applications diverses que personne n'imaginait à l'origine. Depuis, beaucoup d'autres sociétés se sont jointes au consortium pour l'élaboration des versions successives de la spécification, dont l'objectif s'est progressivement élargi dans la direction d'une plate-forme horizontale de déploiement et d'exécution de services. Les domaines d'application d'OSGi couvrent désormais les marchés de l'électronique grand public, de l'industrie, des moyens de transport (auto-

¹OSGi était l'acronyme de *Open Service Gateway initiative*. L'*OSGi Alliance* interdit désormais l'usage de cette dénomination.

²qui est repartie du JSR008 transféré par SUN à l'*OSGi Alliance* en novembre 1999

mobiles et transports publics), des télécommunications mobiles, de la domotique et de l'immotique. L'adoption d'OSGi par la fondation Eclipse pour le déploiement de *plugins* de son atelier de développement et sa plate-forme pour clients riches (*rich-client*) a poussé la spécification à adresser de manière plus générale le déploiement des applications Java. Malgré la généralisation des buts de l'OSGi, la plate-forme OSGi reste légère et applicable aux domaines initiaux.

7.2 Motivations

La première motivation de l'OSGi Alliance était de définir un modèle d'administration d'un parc de passerelles dans lequel l'administration est transparente à l'utilisateur bénéficiaire des services qui lui sont offerts. Ce modèle que nous allons détailler est illustré par la figure 7.1. Une passerelle est généralement un intermédiaire entre un réseau ouvert (c.a.d. l'Internet) et un réseau local privé qui est généralement dédié à un domaine d'application. Le réseau raccorde la passerelle à des équipements de l'environnement de l'usager final. Ces équipements dont la fonction est très spécialisée et figée, sont communicants mais ne sont pas généralement prévus pour être raccordés à un réseau ouvert.

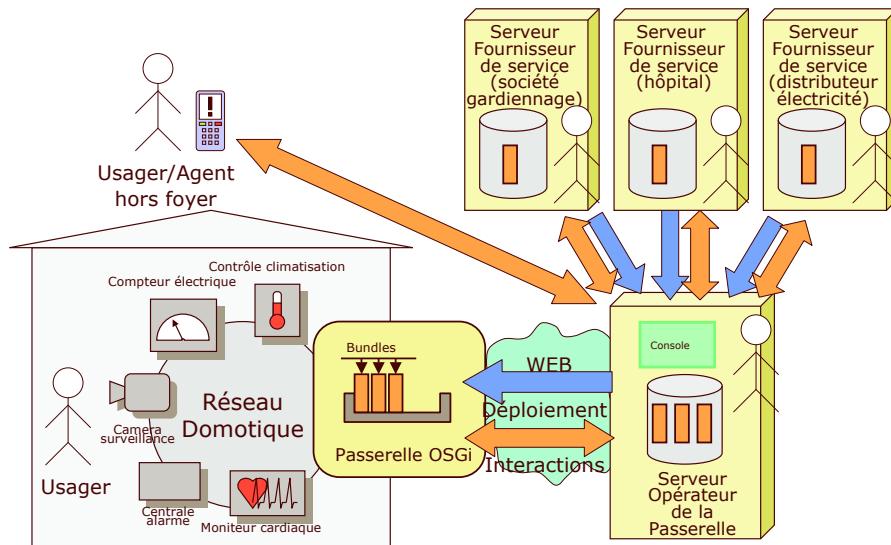


Figure 7.1 – Modèle d'administration de passerelles OSGi

La passerelle est administrée par un opérateur³. Ce dernier autorise le déploiement de services fournis par les fournisseurs de services qui se partagent ainsi la passerelle et définit les règles de partage des ressources. Dans le contexte d'un parc de passerelles constitué de plusieurs milliers d'unités (voire de plusieurs millions), l'opérateur est un système autonome qui gère le déploiement des services en fonction de profils définis pour les passerelles et du contexte environnemental de chaque passerelle.

³L'opérateur de la passerelle est vraisemblablement un opérateur de télécommunications dans le contexte de la domotique. Cependant dans le contexte de l'automobile, il est sans doute le constructeur du véhicule.

Ces services sont généralement liés aux équipements du réseau privé dont ils exploitent les données enfouies et agissent sur les fonctions. Pour des questions de sécurité, l'opérateur assure que les équipements éventuellement liés à un fournisseur (compteur électrique du distributeur d'électricité, moniteur cardiaque de l'hôpital, caméras et centrale d'alarme de la société de gardiennage) ne peuvent être utilisés que par les services du fournisseur. Les services (et leurs fournisseurs) peuvent coopérer entre eux dans l'élaboration de services plus complexes. Par exemple, en cas de malaise détecté par le moniteur cardiaque, l'hôpital peut récupérer le flux vidéo des cameras opérées par la société de gardiennage.

Les services déployés interagissent également avec les serveurs d'entreprise des fournisseurs. L'interaction peut être à l'initiative du serveur ou de la passerelle. Par exemple, le serveur du distributeur d'électricité procède à un télé-relevé en contactant le service connecté au compteur électrique. L'autre exemple est celui du service pilotant le moniteur cardiaque. Si ce dernier détecte une défibrillation du cœur du patient, il contacte le serveur de l'hôpital qui dépêche aussitôt une équipe de secours.

7.3 La plate-forme de déploiement et d'exécution de services Java

La plate-forme⁴ OSGi est une plate-forme d'exécution d'applications Java colocalisées au sein de la même machine virtuelle Java [Gong 2001]. Chaque application qui est attachée à un fournisseur de services peut éventuellement collaborer avec les applications des autres fournisseurs hébergées sur la plate-forme. La plate-forme est généralement représentée comme une surcouche au Java Runtime Environnement⁵ introduisant la dynamicité dans les relations entre les applications déployées.

Dans ce contexte de hébergement multi-fournisseur, la spécification OSGi s'est focalisée sur deux points importants qui jusqu'alors avaient été délaissés ou négligés par les plates-formes Java : le conditionnement et le déploiement des applications, et la collaboration entre les applications s'exécutant au sein de la même machine virtuelle.

Le conditionnement et le déploiement des applications Les applications sont déployées sur la plate-forme sous la forme d'unités de conditionnement et de livraison appelées *bundles*. Le déploiement avec OSGi couvre l'ensemble des opérations de déploiement [Carzaniga et al. 1998] c'est à dire le chargement et l'installation des unités mais également l'activation, la mise à jour, l'arrêt et la désinstallation. Les opérations de déploiement sont réalisables dynamiquement sans nécessairement redémarrer complètement la plate-forme. Cette propriété est très importante car la plate-forme est généralement partagée par plusieurs fournisseurs comme nous l'avons vu dans la section 7.2 et chacune des applications peut avoir une mission critique à remplir. La plate-forme contrôle les dépendances de code (et leurs versions) entre les unités de déploiement avant d'autoriser l'activation d'une application. Pour OSGi, la granularité de code est un paquetage Java.

⁴Ce terme est synonyme du terme *passerelle* qui fait référence aux objectifs initiaux de la plate-forme OSGi.

⁵au minimum J2SE ou J2ME/CDC/Foundation Profile

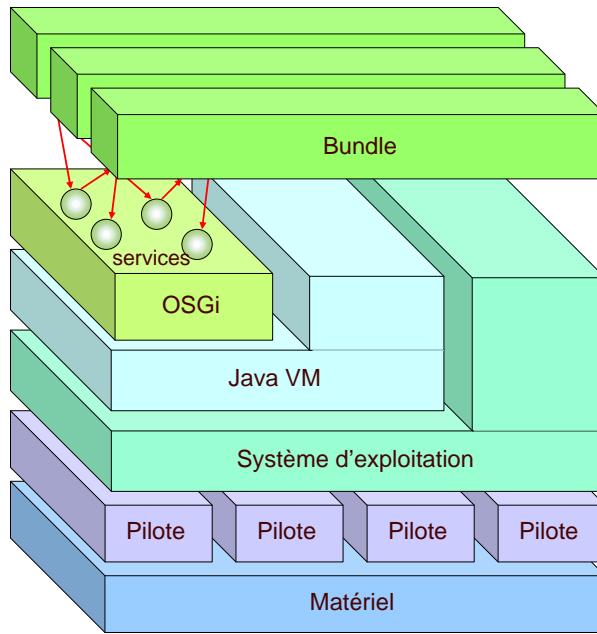


Figure 7.2 – Architecture multi-couche d'une plate-forme OSGi (avec l'aimable autorisation de Peter Kriens).

La collaboration entre les applications . Les applications sur une plate-forme OSGi collaborent par le biais de services 2.1. La plate-forme offre aux applications d'une part un registre de services qui permet de rechercher des objets offrant un contrat de service donné, et d'autre part, un mécanisme de notification des changements qui se produisent sur les services. Les changements concernent l'apparition des nouveaux services, la disparition de services en cours d'utilisation ou bien le changement de terme du contrat. Ce dernier point permet de réaliser des applications supportant l'arrêt de composants optionnels. Il constitue aussi une charge de travail supplémentaire pour le développeur d'application

La suite de cette section détaille ces deux points.

7.3.1 Conditionnement et déploiement des applications OSGi

Une application Java est classiquement conditionnée sous la forme d'un ou plusieurs fichiers JAR (Java ARchive) placés dans des répertoires et listés dans le chemin de recherche des classes, le *CLASSPATH*. Ce *CLASSPATH* est implicitement complété par les fichiers déposés dans le répertoire des extensions de l'environnement d'exécution (c.a.d. `$JRE_HOME/lib/ext`). Ce type de conditionnement conduit généralement à deux problèmes. Le premier est l'absence d'une classe dans un des fichiers JAR listés dans le *CLASSPATH* et dans le `$JRE_HOME/lib/ext` qui provoque l'abandon d'une exécution de l'application qui peut être fort avancée. Le second problème concerne l'incompatibilité de version des classes présentes dans l'environnement opérationnel avec celui qui a servi à compiler l'application. Ceci provoque également des erreurs non récupérables à l'exécution. OSGi remédie à ces deux problèmes en introduisant la notion de *bundle*.

Bundle

Le *bundle*⁶ est l'unité de déploiement⁷ versionnée dans la plate-forme OSGi. Il est conditionné sous la forme d'un fichier JAR contenant le code binaire des classes, des ressources comme des fichiers de configuration ou des images, et des bibliothèques de code natives (c.a.d. dépendant du processeur et du système d'exploitation). Le fichier de manifeste du JAR est augmenté d'une certain nombre d'entrées propres à OSGi. Ces entrées décrivent, entre autres, la classe servant de point d'entrée pour l'activation des services (*Bundle-Activator*), des informations factuelles (auteur, vendeur, nom, nom symbolique, licence, url vers le dépôt de l'archive, url vers la documentation, ...), les paquetages importés (*Import-Package*) et les paquetages exportés (*Export-Package*).

L'atome de partage de code (c.a.d. des classes) dans OSGi est le paquetage et chaque paquetage est versionné. OSGi suppose à priori une compatibilité⁸ ascendante vis à vis des versions antérieures. Les paquetages conditionnés dans le *bundle* qui n'apparaissent ni dans l'entrée *Import-Package*, ni dans l'entrée *Export-Package*, n'ont qu'une portée locale au *bundle*. Ce mécanisme assure un espace de nommage privé n'interférant pas avec ceux des autres *bundles*.

La spécification 3 d'OSGi a introduit la possibilité de démarrer un *bundle* alors que des paquetages qu'il utilisera ne sont pas exportés par un autre *bundle* au moment de l'activation. Ceci est fort utile pour les applications à *plugin* pour lesquels les *plugins* peuvent être dynamiquement installés bien après l'activation du noyau central de l'application. C'est, par exemple, le cas des *codecs* pour *Java Media Framework*, le canevas audio-vidéo de Java. L'entrée *DynamicImport-Package* liste les patrons⁹ de paquetages qui sont attendus après l'activation.

Cycle de vie des bundles

Le *bundle* suit le cycle de vie représenté par le diagramme d'état de la figure 7.3.

Installation L'installation consiste à télécharger et stocker le fichier JAR du *bundle* dans le système de fichiers de la plate-forme (appelé aussi *bundle cache*). Le *bundle* est alors dans son état initial **INSTALLED**.

Résolution des dépendances Le *bundle* passe à l'état **RESOLVED** quand les paquetages qu'il importe (listé dans *Import-Package*), sont tous exportés par des *bundles* installés et résolus sur la plate-forme. Une fois, dans cet état, le *bundle* est prêt pour exporter les paquetages listés dans l'entrée *Export-Package* du manifeste. La résolution est retardée le plus possible jusqu'à l'activation d'un *bundle* important les paquetages exportés par le précédent. Quand un paquetage identique est exporté par plusieurs *bundles*, un seul de ces

⁶baluchon en français

⁷La spécification 4 introduit le concept de *fragment bundle* qui permet de conditionner un *bundle* sous la forme de plusieurs JAR. Un *bundle* fragment est généralement utilisé pour conditionner des ressources localisées ou des bibliothèques natives dépendant du système d'exploitation et du processeur. La spécification introduit également le concept de *extension bundle* qui permet de compléter le *Boot-class-path* sans passer par le mécanisme d'import-export.

⁸La spécification 4 introduit la notion d'intervalle de versions pour limiter la compatibilité ascendante.

⁹par exemple, de la forme `com.acme.plugin.*` pour les *plugins* de la société ACME.

bundles devient l'exportateur effectif et les autres se comportent comme des importateurs. La désignation de l'exportateur effectif se fait par rapport au plus haut numéro de version de paquetage puis au plus petit identifiant d'installation de *bundles*¹⁰.

Activation L'activation d'un *bundle* devient alors possible. L'activation consiste pour la plate-forme à instancier un seul objet de la classe donnée par l'entrée `Bundle-Activator` du manifeste. La classe doit implémenter l'interface `BundleActivator` qui régit le cycle de vie du *bundle*. La méthode `start(BundleContext)` est alors invoquée avec en paramètre un objet `BundleContext` qui représente à la fois le contexte du *bundle* et le contexte de la plate-forme. La méthode `start(BundleContext)` peut rechercher des services, en enregistrer et si besoin démarrer des *threads*. En cas d'exception levée lors de l'exécution la méthode `start(BundleContext)`, le *bundle* retourne à l'état `RESOLVED`. Sinon il se trouve dans l'état `ACTIVE`. L'état `STARTING` est l'état temporaire dans lequel est le *bundle* tant que l'exécution de la méthode `start(BundleContext)` n'est pas terminée.

Arrêt L'arrêt d'un *bundle* déclenche la méthode `stop(BundleContext)` de l'interface `BundleActivator`. Cette méthode doit désenregistrer les services fournis, relâcher les services utilisés et arrêter les *threads* démarrés. Le *bundle* repasse à l'état résolu. L'objet d'activation est alors récupérable par le ramasse-miette.

Mise à jour La mise à jour d'un *bundle* provoque l'arrêt, la réinstallation, la résolution puis la réactivation en continu. Il arrive qu'une mise à jour se termine sur un échec. Le *bundle* se retrouve dans l'état `INSTALLED` dans le cas où des nouveaux paquetages sont requis et sont non disponibles dans la plate-forme, ou à l'état `RESOLVED` si la réactivation du *bundle* avec la méthode `start(BundleContext)` s'est soldée par une exception.

Désinstallation Enfin, la désinstallation du *bundle* provoque la suppression du JAR du *bundle* du système de fichier local. Cependant, certaines classes restent chargées en mémoire tant que tous les *bundles* qui en dépendent (c.a.d. qui les importent) sont actifs (c.a.d. dans l'état `ACTIVE`). Le rafraîchissement de la résolution des paquetages provoque aussi la libération de ces classes.

Bibliothèques natives

Un *bundle* OSGi peut livrer également des bibliothèques de fonctions natives. Ces bibliothèques sont décrites dans le manifeste grâce à l'entrée `Bundle-NativeCode`. Cette entrée décrit le chemin dans le fichier JAR, le processeur cible et le système d'exploitation requis. Une bibliothèque peut être liée à la JVM quand le *bundle* est activé et déliée quand le *bundle* est arrêté. Le chargeur vérifie préalablement la compatibilité du processeur et du système d'exploitation de la plate-forme avant de lier la bibliothèque. Cependant seuls les objets créés par le *bundle* peuvent invoquer les fonctions de la bibliothèque liée. L'invocation se fait via le mécanisme des JNI (*Java Native Interface*). L'invocation de

¹⁰La spécification 4 introduit de nouvelles contraintes pour guider la résolution comme par exemple, les groupements de paquetage qui ne peuvent être importés/exportés qu'ensemble.

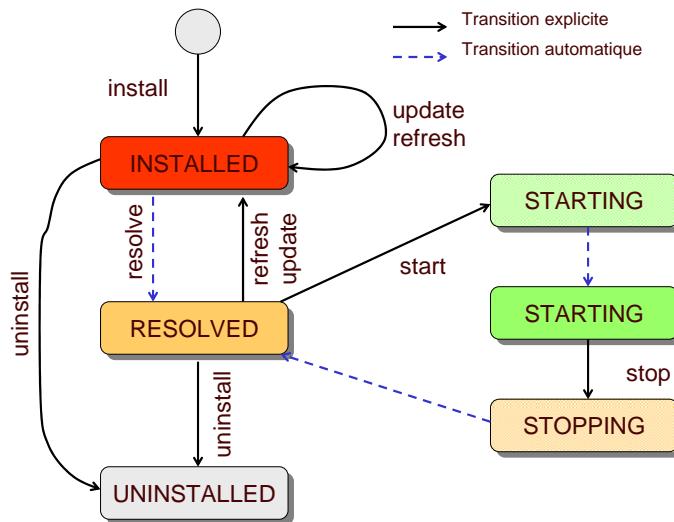


Figure 7.3 – Cycle de vie d'un *bundle* OSGi

ces fonctions natives depuis un autre *bundle* doit obligatoirement passer par un objet de service jouant le rôle de mandataire (cf. 2.3.1).

Déploiement en cascade

La plate-forme OSGi ne se charge que de l'installation d'un seul *bundle* à la fois. Si un *bundle* dépend d'autres *bundles* pour l'importation de paquetages ou l'usage de services, il est nécessaire d'installer préalablement les bundles fournissant les paquetages ou les services requis. Ces bundles peuvent à leur tour requérir l'installation d'autres bundles et ainsi de suite. L'installation de *bundles* en cascade reste à la charge d'outils tiers de déploiement. Cependant, un très récent RFE à la spécification propose un service de déploiement en cascade qui se base sur une description de bundles en terme de paquetages importés et exportés et en terme de services requis et fournis. Ce RFE s'appuie sur le `BundleRepositoryService` de l'OBR de la plateforme OSGi OSCAR. D'autres outils proposent des mécanismes de points de reprise (checkpoint) pour défaire un déploiement en chaîne ayant échoué.

7.3.2 Service

OSGi suit le paradigme de la programmation orientée service dynamique ([Bieber and Carpenter 2002], voir aussi 2.1). Popularisé par le développement des services Web 4.1, la programmation orientée service considère d'une part que tout service respectant un contrat demandé est substituable à un autre, et d'autre part que le choix d'un candidat parmi un liste de services respectant le contrat est décidé le plus tard possible à l'exécution. Le client récupère la liste des services en interrogeant un registre de services. Un fournisseur enregistre son service auprès du registre de services en y associant un contrat qualifié. La programmation orienté service dynamique considère de plus que les services utiles à un client peuvent apparaître à tout moment et que les services en cours

d'utilisation peuvent disparaître à tout moment. Il convient que le client soit sensible (en anglais *aware*) à ces changements.

Dans la plate-forme OSGi, les services sont le moyen pour les *bundles* de coopérer entre eux¹¹. Le contrat de service dans OSGi est à la fois syntaxique et quantitatif selon la classification de [Beugnard et al. 1999] (voir aussi 2.1.3). Il est constitué par une ou plusieurs interfaces Java qualifiées par un ensemble de propriétés (typées) obligatoires ou optionnelles. Les interfaces servent à la négociation syntaxique tandis que les propriétés servent à la négociation de qualité de service. Dans l'exemple fictif de la figure 7.4, le contrat est défini par l'interface `PrintService` complétée par des propriétés dont les noms sont listés dans l'interface `Constants`. L'interface `Job` n'est pas une interface de service mais néanmoins sert aux paramètres et aux retours des méthodes de l'interface de service. Le contrat de service d'une imprimante peut être à la fois l'interface `PrintService` et l'interface `FaxService`. D'autre part, seuls deux services d'impression remplissent le contrat de qualité de service défini ici par la géo-localisation au premier étage.

```
package org.device.print;

public interface PrintService {
    public Job print(InputStream in, Dictionary printParams)
                    throws PrintException;
    public Job[] list();
}

public interface Job {
    public final static int STATUS_WAITING=0;
    public final static int STATUS_ACTIVE=1;
    public final static int STATUS_STOPPED=2;
    public final static int STATUS_COMPLETED=3;
    public final static int STATUS_KILLED=4;

    public int getJobId();
    public int getRemainingPages();
    public int getPages();
    public int getStatus();

    public void kill() throws PrintException;
}

public class PrintService extends Exception {}

public interface Constants {
    public final static String TYPE="org.device.print.type";
    public final static String DPI="org.device.print.dpi";
    public final static String COLOR="org.device.print.color";
    public final static String NPM="org.device.print.npm";
}
```

¹¹ce n'est pas le seul moyen de coopération car les membres et méthodes statiques des classes exportés sont accessibles de tout importateur. Cependant ce procédé de partage doit être proscrit dans le développement d'applications car il ne respecte pas le cycle de vie des *bundles*.

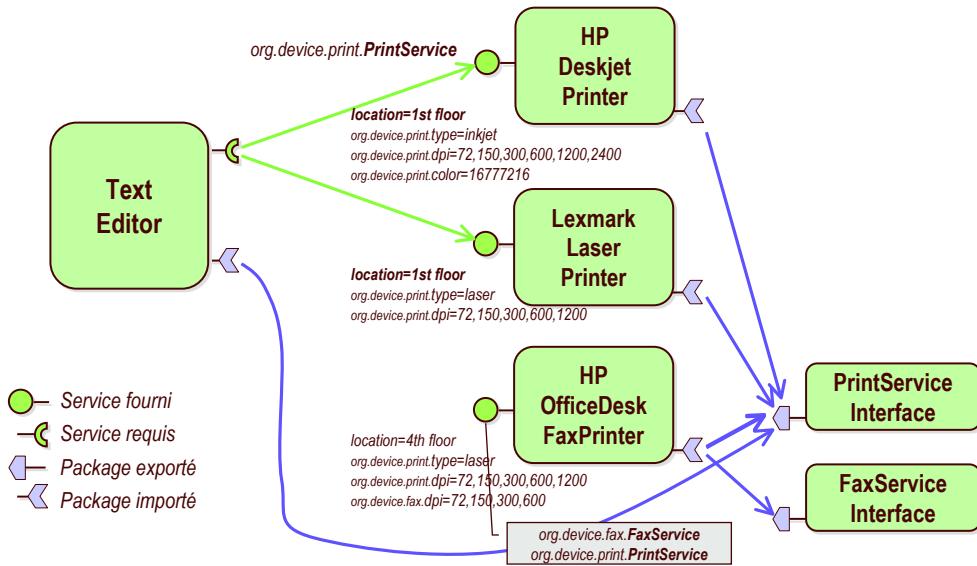


Figure 7.4 – Exemple d'une application liée par les services

Il n'existe pas de notion d'application à proprement parler dans OSGi. La plate-forme est généralement partagée par plusieurs fournisseurs qui y déplacent des *bundles*. Un *bundle* peut démarrer un *thread*¹² pour lancer une activité relative à l'application apportée par le fournisseur. Généralement, le point d'entrée d'une application est un contrat de cycle de vie¹³ qui est piloté par un gestionnaire d'application (déployé comme un *bundle*). La figure 7.5 présente un tel exemple de gestionnaire d'application *CronDeamon*. Ce *bundle* invoque périodiquement la méthode `run()` des services enregistrés dont le contrat est l'interface `java.lang.Runnable` et la propriété obligatoire `crond.pattern` (cas des *bundles* 1 et 2). Cette propriété précise la date et la période des invocations.

7.4 Programmation des services OSGi

Cette section est consacrée à la programmation des objets utilisant et fournissant des services. La programmation de ces objets est essentiellement fondée sur des API fournis par le canevas. L'essentiel de cette API concerne l'enregistrement des services, la recherche et la liaison aux services et l'observation de la dynamique des services (c.a.d. leurs apparitions et leurs disparitions). Cette dernière tache étant fastidieuse pour le développeur, plusieurs travaux se sont attachés à proposer des modèles de composants simplifiant la prise en charge de la dynamique. Les sous-sections détaillent les API. La dernière est consacrée aux modèles de composants.

¹²de préférence via un service de réserve (*pool*) de *threads*

¹³Par exemple, `java.lang.Runnable`, `java.applet.Applet`, `javax.microedition.midp.MIDlet`, `java.microedition.xlet.Xlet`...

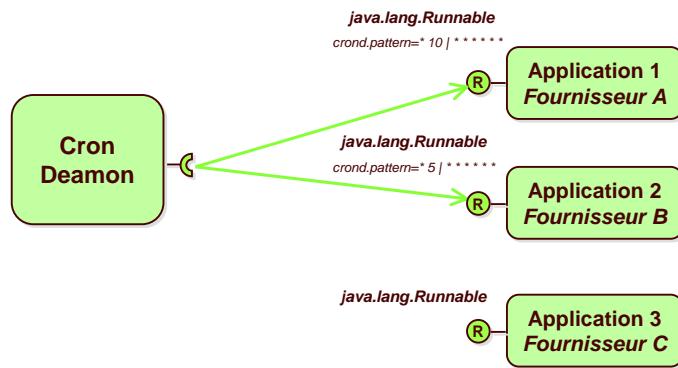


Figure 7.5 – Exemple de gestionnaire d'applications basé sur les services

7.4.1 Activation

L'entrée `Bundle-Activator` du manifeste désigne la classe qui sert de point d'entrée pour l'activation et pour l'arrêt du *bundle*. Cette classe publique doit implémenter l'interface `BundleActivator`. Pour activer le *bundle*, la plate-forme crée un chargeur de classe (*class loader*) [Liang and Bracha 1998] propre au *bundle*, puis crée une seule instance de la classe d'activation et enfin invoque la méthode `start(BundleContext)` sur cette instance. Pour arrêter le *bundle*, la plate-forme invoque la méthode `stop(BundleContext)` sur l'instance puis détruit le chargeur de classes du *bundle*¹⁴.

Le rôle de l'instance de la classe d'activation est généralement de créer les objets qui utiliseront les services fournis par d'autres bundles et qui fourniront à leur tour des services. La méthode `start(BundleContext)` se déroule donc en général de la manière suivante :

1. recherche des services nécessaires ou optionnels,
2. liaison avec les services sélectionnés,
3. création des objets implémentant les interfaces Java d'un contrat,
4. enregistrement de ces objets comme des services avec les propriétés de courtage,
5. positionnement d'écouteurs (*listener*) sur les événements de services, des *bundles* et de la plate-forme.

L'autre rôle de l'instance de la classe d'activation est celui de déactiver les services fournis, de relâcher les références vers les objets de service utilisés et de libérer les ressources utilisées (*threads*, descripteur de fichiers, *sockets*, etc.) au cours des invocations de services. La méthode `stop(BundleContext)` se déroule donc à contre-sens de la manière suivante :

1. retrait des écouteurs
2. désenregistrement des services enregistrés,
3. suppression des objets de services
4. relâche des services auxquels il est lié,
5. libération de ressources,

¹⁴La destruction du chargeur provoque le déchargement et la recyclage (*garbage collection*) de toutes les classes qu'il a chargées.

Le paramètre passé à l’invocation de ces 2 méthodes est le contexte du *bundle*. Cet objet qui implémente l’interface `BundleContext`, permet au *bundle* d’obtenir des informations sur lui-même et d’interagir avec le registre de services interne à la plate-forme. Il permet également de demander à la plate-forme l’installation, l’activation, l’arrêt, la mise à jour et la désinstallation d’un autre *bundle*. Pour des raisons de sécurité, ce privilège est néanmoins réservé aux *bundles* d’administration qui ont la permission `AdminPermission`.

7.4.2 Enregistrement de services

L’enregistrement d’un service se fait au moyen de la méthode `registerService(String[], Object, Dictionary)` du contexte. Cette méthode prend en paramètre, le tableau des interfaces du contrat, la référence vers l’objet qui remplit le service et le dictionnaire des propriétés d’enregistrement. L’objet retourné par la méthode, implémente l’interface `ServiceRegistration` et sert à modifier ultérieurement les propriétés d’enregistrement et à désenregistrer le service.

L’exemple suivant présente l’enregistrement du service d’impression par le *bundle* Lex-Mark de la figure 7.4.

```
package com.lexmark.printer.laser.impl;
import org.osgi.framework.*;

public class Activator implements BundleActivator {
    private ServiceRegistration reg=null;
    private PrintService servant=null;
    public void start(BundleContext ctxt) throws BundleException {
        servant=new PrintServiceImpl();
        Dictionary props=new Properties();
        props.put("org.device.print.type", "laser");
        props.put("org.device.print.dpi", new int[]{72,150,300,600,1200});
        props.put("location", "1st floor");
        reg=ctxt.registerService(
            "org.device.print.PrintService", servant, props);
    }
    public void stop(BundleContext ctxt) throws BundleException {
        if(reg != null) reg.unregister();
    }
}
```

7.4.3 Courtage et liaison de services

L’utilisation d’un service se fait en deux étapes : la recherche (ou courtage) des services puis la liaison aux services choisis. Lorsqu’un service n’est plus utilisé ou bien que le canevas signale son désenregistrement, le *bundle* usager doit procéder à la relâche de liaison.

Courtage Le courtage de services se fait au moyen de la méthode `getServiceReferences(String, String)` auprès de l’objet de contexte. Le premier paramètre est le nom de l’interface des services recherchés et le deuxième est un filtre sur les propriétés d’enregistrement détaillé au paragraphe suivant. Cette méthode retourne

un tableau de plusieurs objets `ServiceReference` qui décrivent les services conformes au contrat demandé.

Le courtage peut être généralement restreint à un sous-ensemble des services enregistrés au moyen d'un filtre. Ce filtre est une expression de condition de syntaxe LDAP (RFC1960) sur les propriétés enregistrées par les services. L'expression contient des expressions simples (propriétés opérateur valeur) connectées par des opérateurs logiques (`&`, `!`, `|`). Les opérateurs de comparaison sont limités à supérieur ou égal (`<=`), inférieur ou égal (`<=`), égalité (`=`), égalité approximative (`~`) et présence de la propriété (`=*`). La propriété `objectClass` représente les noms des interfaces du service.

Les sections de code ci-dessous présentent plusieurs exemples de courtage avec des filtres.

1. Tous les services d'impression.

```
refs=bundleContext.getServiceReferences(
    org.device.print.PrintService.class.getName(),
    null);
```

2. Tous les services d'impression (alternative).

```
refs=bundleContext.getServiceReferences(
    null,
    "(objectClass=org.device.print.PrintService)");
```

3. Certains services d'impression.

```
refs=bundleContext.getServiceReferences(
    org.device.print.PrintService.class.getName(),
    "(&(!(org.device.print.type=laser))"+
    "(!((org.device.print.dpi<=300))(location=*))");
```

4. Tous les services de `org.device`.

```
refs=bundleContext.getServiceReferences(
    null,
    "(objectClass=org.device.*)");
```

5. Le service d'impression et de fax au 3ième étage.

```
refs=bundleContext.getServiceReferences(
    null,
    "(&(&(objectClass=org.device.print.PrintService)"+
    "(objectClass=org.device.fax.FaxService))"+
    "(location=4th floor))");
```

Liaison La méthode `getService(ServiceReference)` de l'objet `BundleContext` permet alors d'obtenir la référence directe sur l'objet de service. OSGi n'introduit pas de mandataire 2.3.1 entre les *bundles* qui introduisent une indirection et nuisent aux performances.

Relâche Quand le service n'est plus utilisé ou quand le service se désenregistre, la méthode `ungetService()` de l'objet `ServiceReference` permet de relâcher le service. Cependant, il est nécessaire de mettre à `null` toutes les références vers l'objet de service pour celui-ci puisse être recyclé. Ce dernier point est important car cet oubli conduit à ne pas pouvoir arrêter ou mettre à jour proprement le *bundle* qui fournissait le service.

L'exemple suivant présente l'utilisation du service d'impression par le *bundle* `Editor` de l'exemple ci-dessous.

```

package org.eclipse.texteditor.impl;
import org.osgi.framework.*;

public class Activator implements BundleActivator {
    private ProgressBar progressbar;
    private PrintService servant;
    private ServiceReference ref;

    public void start(BundleContext context) throws BundleException {
        // On va voir si quelqu'un offre un PrintService ...
        ServiceReference[] refs=context.getServiceReferences
            ("org.device.print.PrintService","(location=1st floor)");
        if(refs!=null) {
            // On prend le premier offert!
            ServiceReference ref=refs[0];
            servant=(PrintService) context.getService(ref);
            if(servant!=null) {
                // On l'utilise
                Job job=servant.print(...);
                ...
                progressbar.setLevel(job.getRemainingPages()/job.getPages());
                ...
                job=null; // car il faut que l'objet puisse être recyclé
                // On le relâche quand on en a plus besoin
                servant=null; // car il faut que l'objet puisse être recyclé
                context.ungetService(ref);
                ref=null;
                ...
            } else {
                // Ce cas se produit quand le service a été
                // désenregistré dans le laps de temps
            }
        }
    }

    public void start(BundleContext context) throws BundleException {
        // on le relâche bien que la plate-forme ne l'oublie !!
        if(ref!=null){
            servant=null;
            context.ungetService(ref);
            ref=null;
        }
    }
}

```

7.4.4 Interception des demandes de liaison

L'interface `ServiceFactory` est une interface de rappel (*callback*) qui permet à l'objet qui est enregistré d'intercepter la demande de liaison (méthode `getService(ServiceReference)` du contexte). La méthode `getService(Bundle,ServiceRegistration)` de l'interface `ServiceFactory` réalise cette

interception et retourne l'objet effectif de service. Ce dernier peut être nouvellement fabriqué ou bien pris dans une réserve (*pool*) d'instances. Néanmoins, ce mécanisme n'est pas réellement une véritable usine (voir 2.3.2) car plusieurs invocations de la méthode `getService(ServiceReference)` pour le même service (c.a.d. le même objet `ServiceReference`) par un *bundle* retourne systématiquement l'objet retourné à la première invocation. Ce mécanisme est généralement utilisé pour vérifier les permissions attachées des *bundles* usagers, pour distinguer le traitement à effectuer en fonction du *bundle* usager ou bien réaliser l'activation paresseuse (*lazy activation*) de l'objet de service afin de retarder l'usage de ressources.

L'exemple ci-dessous illustre ces deux derniers cas d'utilisation. La classe `PrintFactory` instancie un nouvel objet de service `PrintService` pour chaque nouvelle demande de liaison de la part d'un *bundle* (méthode `getService()`). L'objet de service `PrintInstanceImpl` peut réaliser des traitements différenciés en fonction de l'identifiant du *bundle* qui lui a été passé lors de sa création. Par exemple, la méthode `list()` ne retourne que les travaux d'impression commandés par le *bundle* concerné. L'objet de service est enfin détruit lorsque le *bundle* usager relâche la liaison (méthode `ungetService()`). La classe `PrintLazySingleton` tarde l'instanciation de l'objet (singleton) de service jusqu'à l'interception de la première demande de liaison de la part d'un *bundle* usager (méthode `getService()`). Cette classe utilise un compteur de référence pour détruire l'objet de service et libérer les ressources qu'il détient lors que le dernier *bundle* utilisant l'objet relâche celui-ci (méthode `ungetService()`).

Le mécanisme de `ServiceFactory` ne permet pas néanmoins de fabriquer directement des objets de session. Le développeur doit alors se tourner vers l'utilisation de contexte de *thread* pour traquer les sessions et les ressources qui leur sont attachés.

```
public class PrintInstanceImpl implements PrintService {
    // identifiant du bundle usager
    private long bundleId;

    public PrintInstanceImpl(long bundleId){
        this.bundleId=bundleId;
    }
    public Job[] list() {
        // ne retourne que les travaux lancés par le bundle usager
    }
    public start() {
        // fait quelque chose
    }
    public stop() {
        // défait quelque chose
    }
}

public class PrintFactory implements ServiceFactory {

    public Object getService(Bundle bundle,
                           ServiceRegistration serviceRegistration) {
```

```

PrintInstanceImpl instance=new PrintInstanceImpl(bundle.getBundleId());
instance.start();
return instance;
}
public void ungetService(Bundle bundle,
                        ServiceRegistration serviceRegistration, Object instance) {
    ((PrintInstanceImpl)instance).stop();
}
}

public class PrintLazySingleton implements ServiceFactory {
    private int referenceCounter=0;
    private PrintInstanceImpl singleton=null;
    public Object getService(Bundle bundle,
                            ServiceRegistration serviceRegistration) {
        if(referenceCounter==0){
            singleton=new PrintInstanceImpl(null);
            singleton.start();
        }
        referenceCounter++;
        return singleton;
    }
    public void ungetService(Bundle bundle,
                           ServiceRegistration serviceRegistration, Object instance) {
        referenceCounter--;
        if(referenceCounter==0){
            singleton.stop();
            singleton=null; // pour que le singleton soit recyclé
        }
    }
}

public class Activator implements BundleActivator {
    private ServiceRegistration regFact;
    private ServiceRegistration regLazy;

    public void start(BundleContext context) throws BundleException {
        regFact = context.registerService(
            PrintService.class.getName(),
            new PrintFactory(),
            null
        );
        regLazy = context.registerService(
            PrintService.class.getName(),
            new PrintLazySingleton(),
            null
        );
    }
    public void stop(BundleContext context) throws BundleException {
        if (regFact != null) regFact.unregister();
        if (regLazy != null) regLazy.unregister();
    }
}

```

```

    }
}
}
```

7.4.5 Evénements et observateurs

Le *bundle* peut souscrire aux événements relatifs aux services, aux *bundles* et à la plate-forme au moyen d'observateurs. Un observateur peut être synchrone ou asynchrone. L'observateur synchrone s'exécute avant que l'événement soit effectif, alors que l'observateur asynchrone est exécuté alors que l'événement a lieu et se poursuit. Tous les observateurs asynchrones sont exécutés les uns à la suite des autres par un seul *thread* attaché à la plate-forme.

Les événements (interface `ServiceEvent`) relatifs aux services sont :

- l'enregistrement d'un nouveau service,
- le désenregistrement d'un service,
- la modification d'une ou plusieurs propriétés d'un service déjà enregistré.

L'interface `ServiceListener` est celle de l'observateur synchrone de ces événements.

Les événements (interface `BundleEvent`) relatifs aux *bundles* concernent les changements suivants dans le cycle de vie des *bundles* : installé, démarré, arrêté, mis à jour et désinstallé. L'interface `BundleListener` est celle de l'observateur asynchrone de ces événements. La spécification 3 a introduit l'interface `SynchronousBundleListener` d'un observateur synchrone des événements liés aux *bundles*.

Les événements (interface `FrameworkEvent`) relatifs à la plate-forme concernent les changements dans le cycle de vie de la plate-forme : redémarrage, changement de niveau de démarrage, rafraîchissement des exportateurs effectifs de package. L'interface `FrameworkListener` est celle de l'observateur asynchrone de ces événements.

L'observation est nécessaire, voire requise, pour rendre les *bundles* sensibles aux changements. Ce sont les événements de service qui occupent le plus souvent l'attention du développeur OSGi. En effet, quand un service est utilisé, il est impératif de relâcher toutes les références (directes) vers ce service quand l'événement de service indique le désenregistrement de ce service. Si la relâche n'est pas effectuée, l'objet de service ne peut pas être recyclé et le *bundle* fournisseur ne peut être correctement arrêté. Cette tâche, qui est critique pour le bon fonctionnement de la plate-forme, est fastidieuse pour le développeur. Elle est simplifiée par l'usage d'utilitaires comme le `ServiceTracker` et des modèles à composant orientés services qui sont présentés dans la section 7.4.6.

L'exemple de `ServiceListener` ci-dessous présente le code nécessaire pour gérer la liaison dynamique avec tous les services d'impression disponibles.

```

public class MyServiceListener implements ServiceListener {
    BundleContext context;
    String interfaceName;
    Set<Object> servants;
    Map<ServiceReference, Object> ref2servantMap;

    public MyServiceListener(BundleContext context, Set servants, String interfaceName) {
        this.context=context;
        this.servants=servants;
    }

    public void serviceChanged(ServiceEvent event) {
        if (event.getType() == ServiceEvent.SERVICE_ADDED) {
            // ...
        } else if (event.getType() == ServiceEvent.SERVICE_CHANGED) {
            // ...
        } else if (event.getType() == ServiceEvent.SERVICE_REMOVED) {
            // ...
        }
    }
}
```

```

        this.interfaceName=interfaceName;
        this.ref2servantMap=new HashMap();
    }

    public void open() {
        ServiceReference[] srs=context.getServiceReferences(interfaceName);
        for(int i=0;i<srs.length;i++){
            Object servant=(Object) context.getService(srs[i]);
            if(servant==null) continue;
            servants.add(servant);
            ref2servantMap.put(ref,servant);
        }
    }

    public void close() {
        Iterator iter=ref2servantMap.keys();
        while(iter.hasNext()){
            ServiceReference ref=(ServiceReference)iter.next();
            Object servant=(Object) ref2servantMap.remove(ref);
            servants.remove(servant);
            context.ungetService(ref);
        }
    }

    public void serviceChanged(ServiceEvent e) {
        ServiceReference ref = e.getServiceReference();
        String[] objectClasses=ref.getProperty("objectClass");
        for(int i=0;i<objectClasses.length;i++){
            if(!objectClasses[i].equals(interfaceName))
                continue;
            switch (e.getType()) {
                case ServiceEvent.REGISTERED: {
                    Object servant=context.getService(ref);
                    if(ref==null) continue;
                    servants.add(servant);
                    ref2servantMap.put(ref,servant);
                    break; }
                case ServiceEvent.UNREGISTERING: {
                    Object servant=ref2servantMap.remove(ref);
                    if(servant==null) continue;
                    servants.remove(servant);
                    context.ungetService(ref);
                    break; }
                case ServiceEvent.MODIFIED:
                    break;
            }
        }
    }

    public class Activator implements BundleActivator {

```

```

Set printers;
MyServiceListener listener;

public void start(BundleContext context) throws BundleException {
    servants=new HashSet();
    listener=new MyServiceListener(context,printers,PrintService.class.getName());
    listener.open();
    context.addServiceListener(listener);
    ...
    // faire quelque chose avec les servants
    ...
}

public void stop(BundleContext ctxt) throws BundleException {
    context.removeServiceListener(listener);
    listener.close();
}
}
}

```

7.4.6 Modèles de composants

L'observateur de la section de code précédente est relativement simple car un `ServiceListener` complet doit traiter les expressions de filtre pour restreindre le nombre de services et prendre en compte les changements de propriétés. Il peut éventuellement agir sur le cycle de vie de l'utilisateur des services si la disparition de ceux-ci empêche un fonctionnement normal. Cette tâche devient ingérable quand le nombre de services utilisés grandit. Le constat est que la prise en charge de la dynamique des services est une tâche très lourde pour le développeur OSGi et qu'elle conduit à un taux élevé d'erreurs de programmation qui peut provoquer un blocage partiel de la plate-forme. Une première réponse à ce constat est la classe utilitaire `ServiceTracker` (paquetage `org.osgi.util.servicetracker`) qui comme son nom l'indique suit un service. `ServiceTracker` est limité par le fait qu'il ne traque qu'un seul service et ne gère pas le cycle de vie de l'objet utilisateur des services en fonction de leurs disponibilités effectives.

Service Component Runtime

Le *Service Component Runtime* (SCR), introduit dans la spécification 4, propose un modèle à composants [Szyperski 2002] orienté services qui capture la dynamique des services et assujettit le cycle de vie de l'objet usager à la présence et à l'absence des services référencés obligatoires. Le développement est grandement simplifié par la description du composant au moyen d'un descripteur déclaratif dans une grammaire XML. Ce modèle de composant autorise l'écriture d'applications à partir de composants SCR mais également de services fournis par des *bundles* patrimoniaux.

L'exemple de la figure 7.6 présente deux composants interagissant autour d'un service d'impression.

Le composant fournisseur du service (à droite de la figure) est décrit par le descripteur de droite. Ce composant fournit un service (c'est-à-dire une facette) décrit par l'élément `service`. Ce service implémente une interface fonctionnelle (`PrintService`) et une inter-

face technique (`ConfigMBean`¹⁵). Les propriétés déclarées dans les éléments `property` sont utilisées pour qualifier le service fourni. Le fournisseur de service peut être une usine de services (non présentée dans l'exemple) qui instancie des services personnalisés à chaque composant usager au lieu du singleton commun à tous.

Le composant usager du service (à gauche de la figure) est décrit par le descripteur de gauche. Les éléments `references` indiquent que le composant référence plusieurs service obligatoires d'impression (`PrintService`) et un service optionnel de journalisation (`LogService`). L'attribut `cardinality` indique si la liaison est simple ou multiple et si elle est obligatoire ou optionnelle. Le caractère multiple de la liaison signifie que le composant peut utiliser un ensemble de services `PrintService`. Le caractère obligatoire signifie que le composant est instancié dès qu'un service `PrintService` est disponible et il est détruit dès que le dernier `PrintService` est désenregistré. Le caractère optionnel indique que le composant peut fonctionner sans nécessairement référencer le service décrit. L'attribut `target` restreint le courtage à un sous-ensemble de services `PrintService` au moyen de l'expression LDAP. Les méthodes de rappel (*callback*) pour le contrôle de la liaison sont décrites par les attributs `bind` / `unbind`. Elles sont invoquées lors de l'instanciation ou de la destruction du composant et lors de l'apparition ou de la disparition des services intéressants le composant. Ces méthodes sont cependant optionnelles comme dans le cas de la référence vers un service de journalisation. Le composant peut alors récupérer la ou les référence(s) vers le service au travers de son contexte. Les méthodes optionnelles `activate` / `deactivate` sont les méthodes de rappel pour le contrôle du cycle de vie. La méthode `activate` est invoqué au moment de l'instanciation juste après la création des liaisons et la méthode `deactivate` est invoqué au moment de la destruction juste avant la création des liaisons. L'attribut `policy` indique qu'un service qui disparaît est substituable dynamiquement par un autre service équivalent. La politique statique interdit toute substitution après l'activation du composant.

Les descripteurs de composants du *bundle* sont conditionnés et livrés dans le *bundle*. Leurs chemins sont listés dans l'entrée `Service-Component` du manifeste.

Travaux apparentés

SCR reprend à son profit les principales idées de *ServiceBinder* [Cervantes and Hall 2004][Cervantes 2004]. SCR le complète néanmoins par la possibilité de retarder l'instanciation de l'objet de service jusqu'à la demande de liaison effective par un bundle usager. Cette fonctionnalité permet de limiter l'usage des ressources de la machine hôte lorsque de nombreux services sont déployés sans être utilisés. SCR reste cependant un modèle de composants simple et non-hierarchique qui ne prend pas en charge les aspects non fonctionnels comme la persistance, la sécurité, la distribution, les sessions, etc. Le développeur OSGi doit encore gérer ceux-ci de manière ad-hoc et non transparente. Il existe néanmoins plusieurs travaux apparentés. Par exemple, FROGi [Donsez et al. 2004] propose de développer les applications selon le canevas Fractal et de conditionner celles-ci dans un ou plusieurs *bundles*. Les liaisons entre composants Fractal s'appuient sur le courtage de services OSGi lors que les composants sont déployés

¹⁵qui permet la configuration du composant via un agent JMX embarquable sur la plate-forme [Frénnot and Stefan 2004]

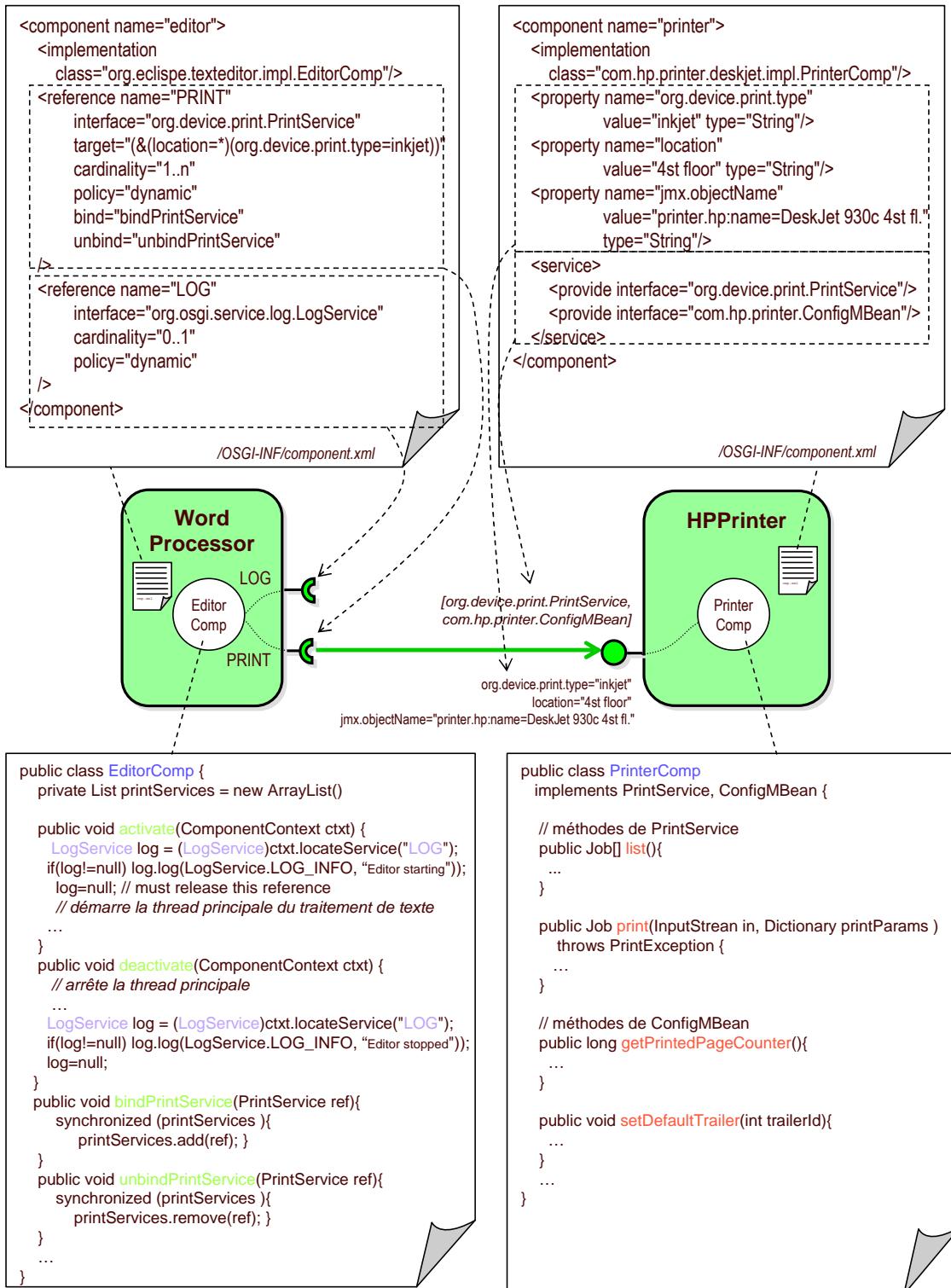


Figure 7.6 – Exemple de composants SCR

dans des bundles différents. D'autres travaux s'intéressent à l'ingénierie des conteneurs dynamiques pour les services OSGi par l'utilisation de la programmation orientée aspect ou bien encore par l'injection de code sur des POJOs (*Plain Old Java Objects*).

7.5 Services Standard

L'OSGi Alliance a progressivement enrichi les spécifications successives de services standard. Ceux-ci sont généralement optionnels par rapport au noyau OSGi (*core*) qui ne dépend d'aucun d'entre eux. Certains services sont assez généraux pour être utilisables dans n'importe quel domaine d'application tandis que les autres traitent des besoins spécifiques à un domaine comme l'automobile ou la téléphonie mobile. Le développement d'une application ne requiert en général qu'un sous-ensemble des services standard complétés par des services propriétaires.

Cette section n'a pas la vocation de présenter l'ensemble des services standardisés par l'OSGi Alliance. Le lecteur peut se reporter à la spécification pour en avoir la liste exhaustive. Cette section détaille trois d'entre eux représentatifs des domaines applicatifs : le *Http Service* pour l'administration à distance via le Web, le *Wire Admin Service* pour les applications basées sur la collecte de mesures depuis un réseau de capteurs et l'*UPnP Device Driver* dans le contexte des services à l'habitat.

7.5.1 Http Service

Le service `HttpService` (paquetage `org.osgi.service.http`) est un des trois premiers services spécifiés par l'OSGi Alliance. Ce choix s'explique par la tendance effective à administrer des équipements réseaux en s'appuyant sur HTTP via des documents HTML¹⁶. L'équipement embarque un serveur Web qui exécute les fonctions d'administration, de configuration, d'exportation des journaux d'audit, etc.

Le service `HttpService` permet à des *bundles* de publier¹⁷ une application Web (en anglais raccourci *WebApp*) constituée de servlets et de JSP respectant la spécification J2EE5.1 mais aussi des documents statiques comme des documents HTML, des icônes, des fichier JAR d'applet ou de midlet. Pour cela, le *bundle* qui implémente le serveur HTTP chargé de recevoir les requêtes, enregistre un service `HttpService`. Un *bundle* publiait une application Web se lie à ce service et enregistre chacune des *servlets* en invoquant la méthode `registerServlet()` du service `HttpService`. Les ressources statiques qui sont conditionnées dans le fichier JAR du *bundle*, sont enregistrées via un objet `HttpContext` avec la méthode `registerRessource()`. Cet objet permet un accès contrôlé et typé (type MIME) aux ressources conditionnées dans le fichier JAR d'un *bundle* via son chargeur de classes.

Une application Web peut contenir des scripts JSP (*Java Server Page*) qui seront compilés à la volée par la *servlet* embarquée `JspServlet` comme dans un serveur J2EE classique. Il est cependant recommandé de conditionner les JSP dans leur forme compilée¹⁸

¹⁶en anglais *Web-based management*

¹⁷rendre accessible via HTTP

¹⁸la tâche ANT <jspc> effectue cette compilation avant le conditionnement (*Ahead Compile Time*).

afin d'éviter le surcoût lié à la compilation elle-même et à l'embarquement d'un compilateur de JSP.

Le service `HttpService` peut également servir à publier des services Web (voir chapitre 4). Plusieurs implémentations de services Web sont basées sur le reconditionnement en *bundle* d'implémentations comme AXIS ou la très légère kSOAP.

La figure 7.7 présente un exemple d'administration sécurisée de la passerelle par HTTP. Le serveur embarqué dans le *bundle* `HttpService` sert les requêtes provenant du Web. Les requêtes GET provenant de la console sont redirigées vers le *bundle* `WebConsole` qui effectue l'action demandée et retourne un document HTML représentant l'état rafraîchi de la plate-forme. Les requêtes POST provenant du serveur de déploiement sont des messages SOAP4.4.1 dont l'élément Body contient un script de lignes de commande à exécuter. Ces requêtes SOAP sont relayées vers le *bundle* `SapService` qui décode les messages contenus et réalise l'invocation des services des *bundles* qui y sont liés. Dans le cas qui nous intéresse, le service invoqué est le service de script du *bundle* `ScriptService` qui invoque le service de *shell* du *bundle* `ShellService` pour chaque commande du script envoyé. Dans cette architecture, les commandes sont des *plugins* du *bundle* `ShellService`. Parallèlement, le service de *shell* sert à des consoles textuelles comme le *bundle* `SerialConsole` qui gère une console branchée sur le port série ou le *bundle* `SSHDConsole` qui permet un accès distant sécurisé au *shell*. En marge de cette application d'administration, le *bundle* `SoapService` peut servir d'autres services Web embarqués comme le *bundle* `SensorLoggerWebSvc` qui réalise la relève de mesures acquises sur des capteurs connectés¹⁹ à la passerelle et journalisées depuis la relève précédente. Ce type d'application dite orientée capteur, est présenté à la section 7.5.2.

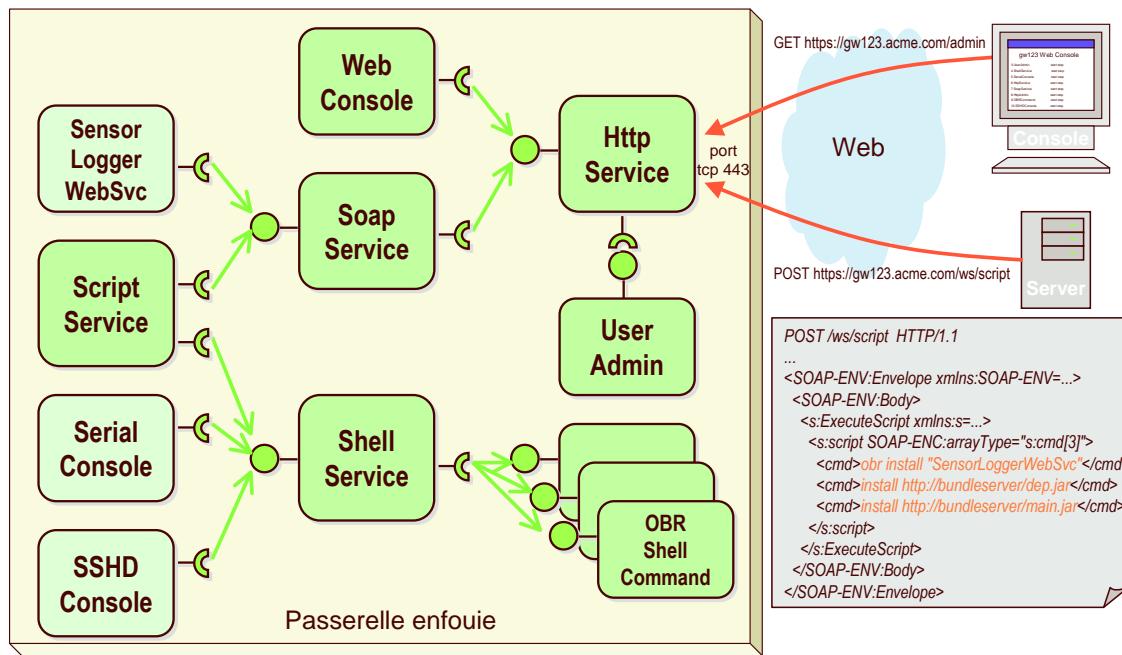


Figure 7.7 – Administration de passerelle via `HttpService`

¹⁹par exemple, le compteur électrique de la figure 7.1.

7.5.2 Wire Admin

Les applications à base de capteurs (*sensor-based services*) proposent d'acquérir, collecter, filtrer, agréger, analyser et réagir aux mesures acquises par les capteurs disséminés dans le monde réel afin de les intégrer en temps réel dans les systèmes d'information des entreprises. Ces mesures peuvent être aussi variées que l'identifiant d'un badge RFID lu par un portique, la position GPS d'un véhicule ou celle d'un prisonnier en liberté supervisée, la puissance consommée par une chaudière, la température d'une poche de transfusion sanguine ou celle du moteur d'un véhicule, l'image infrarouge d'une caméra de surveillance, le niveau de toner d'un photocopieur ou bien encore la fréquence cardiaque d'un soldat au contact de l'ennemi ou celui d'un patient en convalescence à son domicile. Ces services s'articulent principalement autour du traitement de flux de mesures acquises en début de chaîne par une myriade de capteurs.

Sous l'impulsion des fabricants d'automobile, l'OSGi Alliance a spécifié le *Wire Admin Service* (paquetage `org.osgi.service.wireadmin`) pour la construction de telles applications. Ce service maintient une topologie dynamique de liaisons (`Wire`) reliant des producteurs de mesures aux consommateurs de ces mesures.

Pour cela, un *bundle* participe à une application en enregistrant un service `Producer` pour produire des données ou un service `Consumer` pour consommer des données produites. Le service `WireAdminService` crée un objet `Wire` qui référence un service `Producer` et un service `Consumer` pour chaque liaison de la topologie de l'application. Dans une topologie, un producteur peut être lié à plusieurs consommateurs et un consommateur peut être lié à plusieurs producteurs. Un fois la liaison établie, un service `Producer` peut pousser une donnée (par exemple, une mesure récupérée sur un capteur physique) vers tous les objets `Wire` qui les propageront vers les services `Consumer`. Contrairement au modèle *push* précédent, l'initiative peut venir également de service `Consumer` qui peut demander à tous les objets `Wire` de forcer les services `Producer` à produire une donnée conformément au modèle *pull*. Le service `Wire Admin` impose à la fois la compatibilité des données échangées et leurs adaptations si celles-ci sont nécessaires. Pour cela, la notion de *flavor* décrit la liste ordonnée des classes Java des objets que peut produire un service `Producer` et qu'un service `Consumer` peut consommer. Un objet `Wire` ne peut être créé que si les services `Producer` et `Consumer` partagent une classe en commun dans les 2 listes de *flavors*. De plus, un service `Producer` doit adapter l'envoi de données vers les services `Consumer` en fonction de leur listes de *flavors*. Ainsi il pousse par le biais d'un objet `Wire` un objet du type de la liste *flavors* du service `Consumer`. Ce type est la première classe commune des deux listes *flavors* des services `Producer` et `Consumer` raccordés au `Wire`.

Afin d'éviter l'inondation des consommateurs par les producteurs, le service `WireAdmin` spécifie cinq schémas de contrôle de flot.

Ces schémas de contrôle sont :

- envoi en cas de changement de valeur
- envoi en cas d'écart absolu de la nouvelle valeur par rapport à la précédente
- envoi en cas d'écart relatif de la nouvelle valeur par rapport à la précédente
- envoi sur hystérésis encadré par un seuil bas et par un seuil haut
- envoi périodique pour un certain seuil fixé

Le contrôle de flot est réalisé soit par le service `Producer`, soit par l'objet `Wire`.

Un exemple d'application avec WireAdmin L'application à base de capteurs permet d'indiquer au conducteur d'un véhicule la distance restante et le cap vers des points d'intérêt choisis (station essence, accident...). Cette application est constituée de :

- un *bundle GPS Receiver* pilotant un récepteur de position GPS qui produit des objets **Position** qui sont des quintuplets <latitude, longitude, altitude, vitesse, cap>
- un *bundle Initial Sensor* pilotant un capteur inertiel qui produit des objets **DiffPosition** qui sont des doublets de variation instantanée de cap et de vitesse.
- un *bundle Position Correlator* corrélateur de position qui infère et produit des objets **Position** à partir d'un objet **Position** de référence consommée et une suite d'objets **DiffPosition** consommées. Ce *bundle* de médiation est utile quand le récepteur GPS perd le signal des satellites GPS (sous un tunnel par exemple) et ne produit plus d'objets **Position**.
- un *bundle POI Locator* de localisation de points d'intérêt (*Point Of Interest*) qui calcule et affiche le cap à tenir et la distance restante avec le point d'intérêt le plus proche à partir des objets **Position** produits par le *bundle Position Correlator*.
- un *bundle Wire Admin Binder* de construction de la topologie des **Wires** à partir d'un ADL dédié à ce type d'application. Ce *bundle* prend en compte la dynamique des services **Producer** et **Consumer** fournis par des *bundles* qui n'étaient pas forcément prévus initialement comme par exemple un *bundle* produisant des positions à partir des informations de positionnement des réseaux GSM.

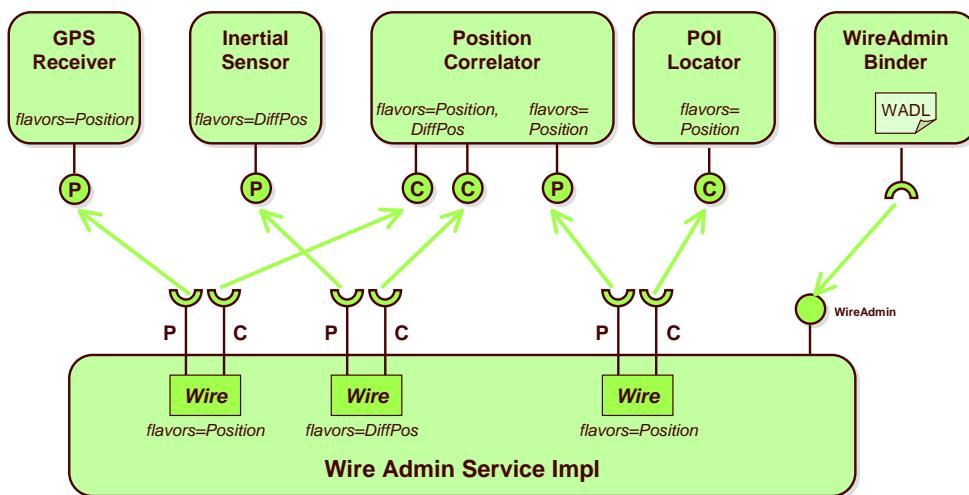


Figure 7.8 – Exemple d'application à base de capteurs

7.5.3 UPnP Device Driver

Universal Plug and Play (UPnP) Forum [UPnP Forum 2005] est un consortium industriel ouvert qui s'est formé en 1999 pour la définition de standards simplifiant la mise en réseaux d'équipements communicants dans les maisons et dans les entreprises (*SOHO* : *Small Office Home Office*). Les domaines couverts sont la micro-informatique/bureautique (imprimantes, appareil-photo...), l'électronique grand public (DVD, TV, radio, Media-

Center...), la communication (téléphones, routeur ADSL...), la domotique (alarme anti-intrusion, régulation du chauffage, volets roulants, etc.) ou bien encore l'électroménager (lave-linge, réfrigérateur...). UPnP Forum a publié une première version des protocoles réseaux requis (UPnP DA) et un certain nombre de définitions standard de périphériques (*devices*) et de leurs services associés.

Universal Plug and Play (UPnP) Device Architecture (DA) spécifie les protocoles pour des réseaux spontanés de périphériques (*devices*). Les protocoles UPnP traitent :

- la détection et le retrait dynamique des périphériques,
- leur description et celles des services qu'ils fournissent,
- l'utilisation par les points de contrôle (PDA, télévision, télécommande RF...) des services fournis
- la notification des changements de valeurs des variables d'état associées aux services.

UPnP DA a les mêmes objectifs que JINI [Waldo 1999] proposé par SUN. Cependant, UPnP DA n'est pas attaché à un langage particulier (comme JINI avec Java). Les protocoles de UPnP DA s'appuient sur XML, SOAP 1.0 et HTTP au dessus de TCP, UDP et UDP Multicast. La prochaine version d'UPnP DA devrait s'orienter vers les protocoles vers les standards actuels des services Web. La proposition DPWS (Device Profile for Web Services) candidate à UPnP v2 s'appuie sur SOAP 1.2, WSDL 1.1, XML Schema, WS-Addressing, WS-MetadataExchange, WS-Policy, WS-Security, WS-Discovery et WS-Eventing.

Le chapitre *UPnP Device Driver* de la spécification OSGi traite la manière de développer des périphériques UPnP et des points de contrôle UPnP au dessus une plate-forme OSGi. Ce chapitre décrit d'une part l'API `org.osgi.service.upnp` dont l'interface principale `UPnPDevice` représente un périphérique, et d'autre part un élément de la passerelle, l'*UPnP Base Driver*, qui assure le pont entre les points de contrôle UPnP et les périphériques UPnP présents sur le réseau IP adhoc, et les *bundles* hébergés par la plate-forme²⁰.

L'interface de service `UPnPDevice` décrit le périphérique. Cette interface contient principalement les références vers des objets `UPnPService` qui décrivent les services associés au périphérique. Cette interface `UPnPService` n'est jamais enregistrée comme un service OSGi. A son tour, l'objet `UPnPService` décrit la liste de variables d'état au moyen d'objets `UPnPStateVariable` et la liste des actions au moyen d'objets `UPnPAction`. L'invocation d'une action sur le périphérique UPnP se traduit par l'invocation de la méthode `Dictionary invoke(Dictionary args)` sur l'objet `UPnPAction` correspondant. L'API comporte également l'interface de service `UPnPEventListerner`. Un point de contrôle qui souhaite être informé des changements d'état des variables enregistre un service `UPnPEventListerner`. Les *bundles* (dont l'*UPnP Base Driver*) qui fournissent des services `UPnPDevice`, doivent alors se lier à ce service `UpnPEventListerner` afin d'effectuer les notifications. La demande de notification peut concerner soit un périphérique particulier, soit tous les périphériques d'un même type ou bien soit tous les services d'un même type.

²⁰L'Alliance avait spécifié un service pour JINI avec des objectifs similaires dans la version 3 de la spécification OSGi. Cependant ce service a été retiré de la version 4 faute d'un soutien des équipementiers.

Cas d'utilisation

Cette section présente quatre mises en œuvre possibles d'OSGi *UPnP Device Service*.

Colocalisation La première mise en œuvre colocalise un *bundle* point de contrôle et un *bundle* périphérique sur la même passerelle et aucun échange réseau n'étant opéré. Le *bundle* périphérique fournit un service *UPnPDevice*. Le *bundle* point de contrôle utilise ce service et fournit à son tour un service *UPnPEventListener* pour être informé des changements de valeur des variables d'état. Le *bundle* périphérique doit alors se lier à ce service pour pouvoir notifier les changements de valeur. Cette mise en œuvre a un intérêt limité d'un point de vue opérationnel ; cependant, elle permet de mettre au point des périphériques et des points de contrôle sans en ajouter, les problèmes pouvant provenir du réseau.

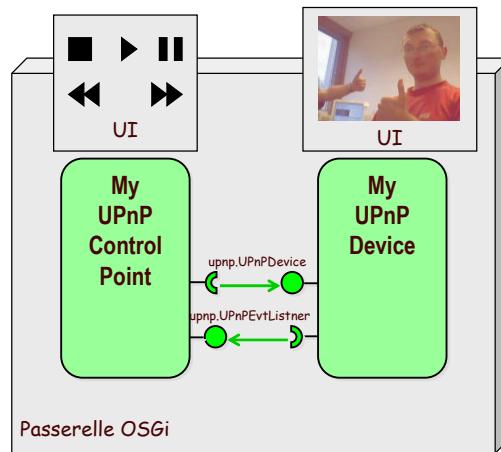


Figure 7.9 – Colocalisation d'un point de contrôle et d'un périphérique

Point de Contrôle UPnP La mise en œuvre consiste dans la réalisation d'interfaces homme-machine (IHM) plus ou moins évoluées pour le contrôle de périphériques UPnP. Ces IHM sont généralement exécutées sur des terminaux domestiques (assistant personnel, télévision, téléphone mobile, panneau sensible ou *touchpanel*) embarquant une passerelle OSGi (avec une machine virtuelle Java J2ME/CDC ou J2SE). Le *bundle UPnP Base Driver* qui fait le pont en les autres *bundles* déployés et le réseau UPnP, enregistre un service *UPnPDevice* pour chaque périphérique UPnP découvert. Le *bundle* point de contrôle sélectionne les services *UPnPDevice* pertinents pour lui et se lie à eux. Le point de contrôle enregistre à son tour un service *UPnPEventListner* quand il souhaite être notifié des changements d'état d'un périphérique concerné. Le *bundle UPnP Base Driver* se lie au service *UPnPEventListner* et annonce au périphérique sa demande de notification conformément à l'UPnP DA. La plate-forme OSGi permet alors la réalisation d'IHM dynamique réagissant à l'apparition et à disparition de périphériques dans l'environnement réseau du point de contrôle. En effet, le *bundle* contenant le code de l'IHM dédiée au périphérique n'est pas nécessairement installé au démarrage du point de contrôle et peut être chargé, installé

et démarré dynamiquement au moment de l'apparition du périphérique [Donsez 2006]. Le *bundle* sera généralement chargé depuis un site Web qui peut être celui du constructeur du périphérique contrôlé ou bien d'un site miroir ou pair-à-pair.

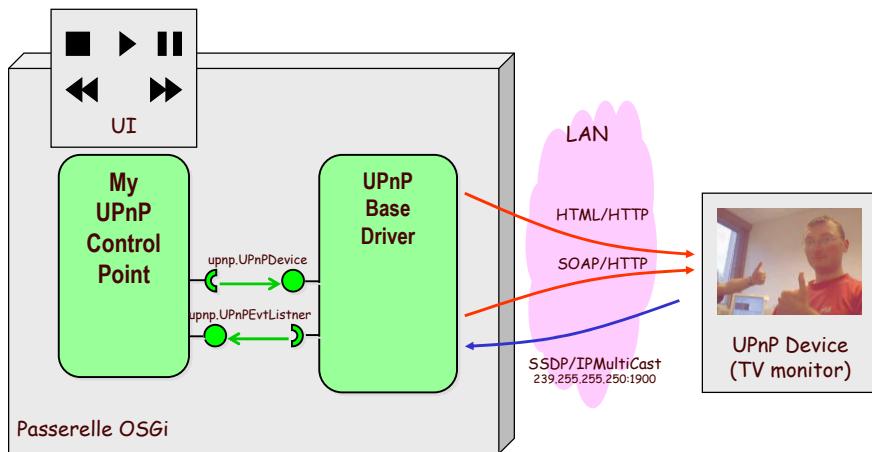


Figure 7.10 – Point de contrôle UPnP

Périphérique UPnP Cette mise en œuvre permet d'utiliser Java et OSGi pour développer des périphériques UPnP. La figure présente un décodeur de TV numérique embarquant une passerelle OSGi et une machine virtuelle Java J2ME/CDC. Le *bundle* périphérique contrôle le matériel de la STB comme la carte tuner et la carte son. Ce *bundle* enregistre tout d'abord un service UPnPDevice. A ce moment, le *bundle* UPnP Base Driver se lie à ce service et annonce l'apparition d'un nouveau périphérique dans le réseau ad hoc. Les points de contrôle externes à la passerelle peuvent alors invoquer des actions qui seront converties en appel de méthode invoke(Dictionary) sur l'objet UPnPAction par le *bundle* UPnP Base Driver. Pour chaque point de contrôle externe à la passerelle qui souhaite être notifié des changements de valeur dans le périphérique, le *bundle* UPnP Base Driver enregistre un service UPnPEventListener.

Passerelle UPnP avec les micro-mondes La dernière mise en œuvre propose d'utiliser la passerelle OSGi pour créer des ponts entre un réseau UPnP (basé sur IP) et des réseaux non-IP (comme X10, IEEE1394, OneWire™, ModBus™...). Ce dernier type de réseaux est appelé « micro-monde ». Le niveau fonctionnel des équipements raccordés est généralement très faible car la partie communication de l'équipement est fortement contrainte par son coût. La mise en œuvre est très semblable à la précédente car le *bundle* qui réalise le pontage avec le micro-monde enregistre un service UPnPDevice pour chaque équipement découvert dans le micro-monde²¹. De manière similaire à la seconde mise en œuvre, le code du service UPnPDevice peut être déployé au moment de la découverte. Le

²¹Certains réseaux comme X10 n'offrent pas de mécanismes de découverte ou de sondage. Les adresses des équipements du micro-monde doivent donc être entrées manuellement par l'installateur des équipements via l'interface de configuration de la passerelle.

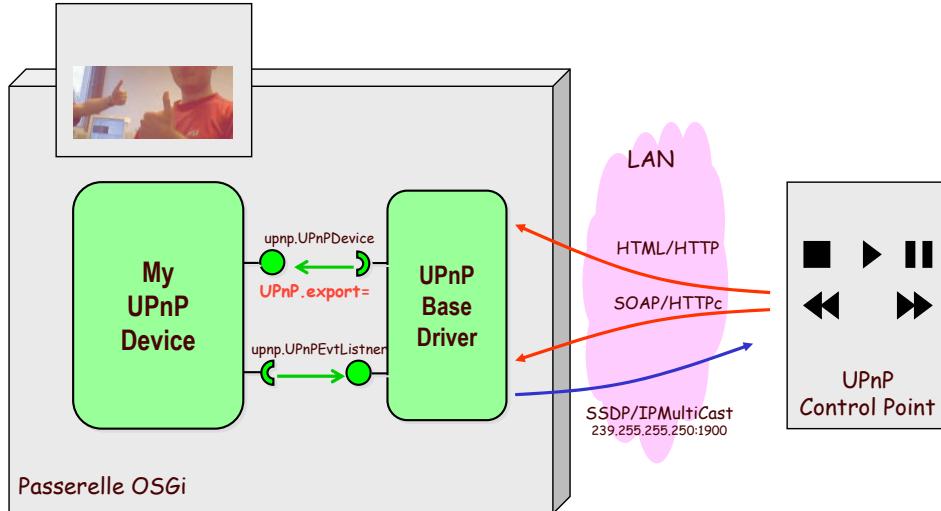


Figure 7.11 – Périphérique UPnP

bundle *UPnP Base Driver* assure quand à lui le pontage entre les services *UPnPDevice* enregistrés et le réseau UPnP.

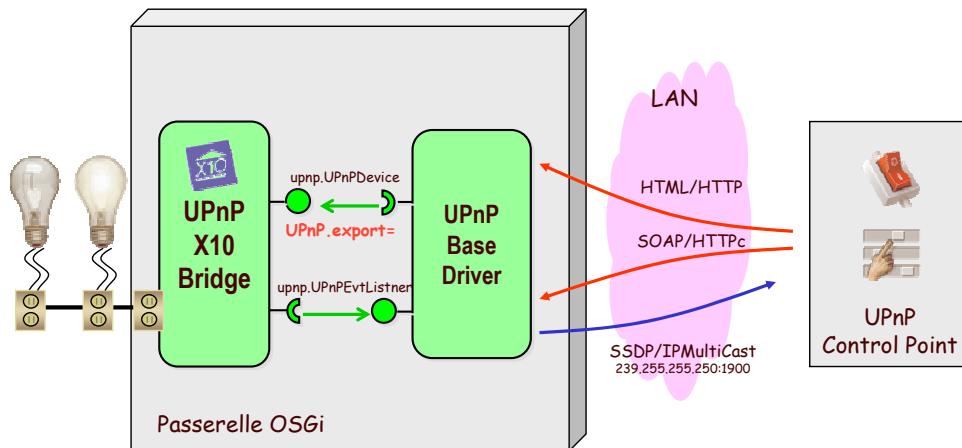


Figure 7.12 – Liaison interne entre un appareil et un point de contrôle

Pour terminer cette section, UPnP DA peut être également utilisé comme support pour découvrir et contrôler les passerelles OSGi présentes sur un réseau ad hoc. La passerelle est décrite alors comme un périphérique. Pour cela, elle embarque seulement un *bundle* d'administration fournissant un service *UPnPDevice*.

7.6 Conclusion et Perspectives

OSGi s'est imposé comme le standard de fait pour l'exécution et l'administration des plates-formes Java de services déportées. Ce chapitre a présenté les grands principes de la plate-forme dynamique de services OSGi, ainsi que la programmation des services dynamiques. Plusieurs services standardisés ont été également détaillés.

Le marché des passerelles OSGi compte déjà plusieurs fournisseurs commerciaux (IBM SMF, ProSyst mBedded, Siemens VDO TLA...) et des implémentations *open-source* matures (Oscar, Felix, Knoplatform, Equinox). La différence entre ces plates-formes tiennent surtout aux outils (développement, administration de parc, déploiement), aux *bundles* disponibles implémentant des services standard ou des services propriétaires, et aux supports par les machines virtuelles du marché pour les J2ME/CLDC, J2ME/CDC et J2SE.

De façon général, tout développeur d'application gagne à conditionner ses applications sur la forme de *bundle* OSGi afin d'éviter l'enfer du CLASSPATH²² lors du déploiement des applications. La très récente annonce du JSR 277 *Java Module System* confirme cette tendance. Il constate que les problèmes de déploiement des applications Java (J2SE et Java EE) sont liés au manque de méta-information décrivant les dépendances des unités de déploiement. Le nouveau modèle de conditionnement que vise à définir ce JSR risque fort de ressembler à la spécification 4 de OSGi sortie en Octobre 2005. La disponibilité de ce JSR dans l'édition standard de la plate-forme est prévue pour sa version 7 (appelée *Dolphin*). Ce JSR a été très récemment complété par le JSR 291 *Dynamic Component Support for Java SE* qui propose le modèle d'exécution d'OSGi R4 comme modèle d'exécution dynamique pour les applications Java. Si ces deux JSR sont adoptés et intégrés à la plate-forme Java, une bonne partie de la spécification du noyau de la plate-forme OSGi R4 sera absorbée par l'environnement d'exécution de Java.

Et côté serveur ? Jusqu'à présent, OSGi est synonyme de serveurs embarqués ou enfouis. Cependant, avec son adoption par le projet Eclipse [Gruber et al. 2005] et récemment par plusieurs projets de l'Apache Software Foundation [Rodriguez 2005], OSGi pourrait bien être demain la plate-forme de référence pour construire des serveurs d'application d'entreprise dynamiques et flexibles [Désertot et al. 2006].

Et la plate-forme Microsoft .NET alors ? Le grand concurrent de Java pourrait-il servir à construire une plate-forme dynamique de services semblable à OSGi ? Comme Java, .NET repose sur une machine virtuelle, la CLR (voir chapitre 6, section 6.2.1), qui exécute un code portable, le MSIL (voir chapitre 6, section 6.2.2), chargé dynamiquement. Cependant, le grain de déchargement des classes est le domaine d'application et les références directes entre domaines d'applications sont interdites. La réponse semble être non, pour l'instant, avec les versions actuelles de la CLR [Escoffier et al. 2006]. Mais on peut parier que Microsoft corrigera le tir si le besoin de dynamité se fait sentir dans les domaines de marché encore vierges que vise déjà OSGi.

²²par analogie avec le *DLL Hell* de Microsoft DCOM

Intergiciel et Construction d'Applications Réparties

©2006 (version du 19 janvier 2007 - 10:31)

Licence Creative Commons (<http://creativecommons.org/licenses/by-nc-nd/2.0/fr/deed.fr>)

Chapitre 8

Conclusion

Chapitre provisoirement indisponible.

Annexes

- A. Les services Web en pratique
 - B. .NET en pratique
 - C. OSGi en pratique
 - D. AOKell : une réalisation réflexive du modèle Fractal
 - E. Joram : un intergiciel de communication asynchrone
 - F. Speedo : un système de gestion de la persistance
 - G. Infrastructure M2M pour la gestion de l'énergie
- M. Dumas¹, M.-C. Fauvet²,
 - A. Aït-Bachir²
 - M. Riveill³, D. Emsellem³
 - D. Donsez²
 - L. Seinturier⁴
 - R. Balter⁵, A. Freyssinet⁵
 - P. Déchamboux⁶, Y. Bersihand⁶,
 - S. Chassande-Barrioz⁶
 - R. Balter⁵

¹Queensland University of Technology, Brisbane (Australie)

²Université Joseph Fourier, Grenoble

³Université de Nice - Sophia Antipolis

⁴Université Pierre et Marie Curie et INRIA

⁵Scalagent Distributed Technologies

⁶France Telecom R&D

Annexe A

Les Services Web en pratique

La version HTML de ce document, ainsi que les ressources qu'il utilise sont accessibles via : <http://www-clips.imag.fr/mrim/User/marie-christine.fauvet/icar06/>

Avant de commencer ce TP copier l'archive `webserv_prat_1.tar`, puis extraire les fichiers qu'elle contient. Cela aura pour effet de créer l'arborescence et les fichiers nécessaires au TP (voir section A.8.1).

Objectifs :

Nous proposons de mettre en pratique le cours dédié aux services Web par le biais d'une activité qui consiste à programmer et implanter des services web simples.

A.1 Préambule

Attention :

Les utilisateurs d'autres systèmes d'exploitation que unix/linux doivent adapter en conséquence les notations utilisées ci-après.

Outils à installer :

Pour ce TP, les outils qui doivent avoir été installés sur la machine sont listés ci-après. Les versions utilisées pour ces outils sont valides au 31 juillet 2006. Elles sont bien sûr amenées à évoluer avec le temps.

1. JDK 1.5
2. Tomcat 5.5 (à rendre accessible via le port 8080)
3. Beehive
4. Ant 1.6.5
5. Axis 1.4

Les variables d'environnement listées ci-après doivent avoir été correctement positionnées (consulter les documentations d'installation, et positionner les variables en fonction de l'installation locale) :

- `AXIS_HOME`
- `AXIS_LIB`

- AXISCLASSPATH
- ANT_HOME
- BEEHIVE_HOME
- CATALINA_HOME
- CATALINA_LIB
- CATALINACLASS
- JAVA_HOME
- CLASSPATH

NB : Les services Web seront déployés dans un serveur Apache Tomcat sur la machine locale accessible via le port 8080.

A.2 Beehive et Axis

Beehive : Le projet Beehive est lancé par Apache Group pour le développement rapide d'applications Web entre autres. La notion de programmation avec annotations est introduite par le projet JSR (Java Specification Request). L'idée est d'introduire des annotations dans du code java afin de décrire plus facilement le service Web, ses méthodes et ses paramètres. Beehive fait appel à l'outil de compilation Ant qui permet de compiler et de déployer le service Web développé en Java et JSR. Beehive s'appuie sur les bibliothèques fournies par Axis. Voir la documentation annotations JWS : <http://e-docs.bea.com/wls/docs92/webserv/annotations.html>

Axis : Le projet Axis est aussi un projet de Apache Group pour le développement et le déploiement de services Web. Axis est une bibliothèque de classes sur laquelle on s'appuie pour le développement du service et celui du client, les deux utilisant SOAP (Simple Object Application Protocol).

A.3 Déploiement d'un service

Le service `TimeNow` donne l'heure système du serveur sur lequel il s'exécute, il possède une seule opération non paramétrée : `getTimeNow`. Pour concevoir ce service Web nous allons dans un premier temps écrire le code Java de la classe `TimeNow`.

1. Dans le répertoire `webserv_prat_1/ws_time/WEB-INF/src-ws/web` créer un fichier `TimeNow.java` (voir son contenu, section A.8.2). Respecter le nom.
2. Il faut noter que ce qui est en gras après le symbole @ sont des annotations qui permettent de signaler que la classe `TimeNow` représente un service Web (`@WebService`). L'autre annotation utilisée est `@WebMethod` qui permet d'indiquer parmi les méthodes celles qui sont exposées par le service Web.
3. Dans le répertoire `webserv_prat_1/ws_time/WEB-INF/src` se trouve le fichier `build.properties`. Editer ce fichier pour modifier les lignes suivantes et les positionner comme suit (remplacer le chemin d'accès à `beehive.home` par la valeur correspondant à l'installation locale) :

```
-----build.properties-----
beehive.home=/home/fauvet/Beehive/apache-beehive-incubating-1.0m1/
service.name=TimeNow
```

-
4. Aller dans le répertoire `webserv_prat_1/ws_time` et dans le fichier `index.html` vérifier les lignes signalées ci-après :

```
-----index.html-----
<a href="web/TimeNow.jws?wsdl">WSDL</a>
<a href="web/TimeNow.jws?method=getTimeNow">appel opération</a>
-----
```

5. Compilation et déploiement avec `ant` :

Aller dans le répertoire `webserv_prat_1/ws_time/WEB-INF/src` et exécuter la commande suivante :

```
ant clean build war
```

6. Avec l'outil Tomcat Web Application Manager (*WAR file to deploy*) déployer l'application contenue dans le fichier `TimeNowWS.war` (généré par `ant` dans le répertoire `webserv_prat_1/`).
7. Observer les informations rendues disponibles au travers du lien `http://127.0.0.1:8080/TimeNowWS/ ..`

A.4 Déploiement d'un client

Il s'agit ici de développer en java un client qui va exécuter la méthode du service `TimeNow` déjà étudiée. Cette fois l'appel au service va être effectué à partir d'un programme java et non plus via un navigateur.

1. Dans le répertoire `webserv_prat_1/ws_client` créer un fichier java nommé `ClientTimeNow.java` (voir son contenu, section A.8.3). Respecter le nom.
2. Compiler le fichier : `$ javac ClientTimeNow.java`
3. Avant d'exécuter la classe compilée il faut avoir bien défini dans la variable d'environnement `CLASSPATH` le chemin d'accès au répertoire où le fichier `ClientTimeNow.class` a été placé.
4. Dans le répertoire de `ClientTimeNow.class`, exécuter l'instruction suivante :
`$ java ClientTimeNow`. A la date près, le résultat doit ressembler à ce qui suit :
`$ java ClientTimeNow`
Bonjour d'Autrans!! Ici il est : Sun Mar 05 17:38:49 CET 2006

Le message d'erreur :

- `Unable to find required classes (javax.activation.DataHandler and javax.mail.internet.MimeMultipart). Attachment support is disabled.` est sans conséquence.

A.5 Un service avec une opération paramétrée, et un client

Ce service Web affiche un message concaténé au nom passé en paramètre de l'opération `sayHelloWorldInParam(String name)` et de la date système. Cette dernière est obtenue par l'appel au service `TimeNow` réalisé précédemment. Pour concevoir ce service Web nous allons dans un premier temps écrire le code Java de la classe Hello.

1. Dans le répertoire `webserv_prat_1/hello/WEB-INF/src-ws/web/` créer un fichier `Hello.java` (voir son contenu, section A.8.4). Respecter le nom.
2. Noter que ce qui est en gras après le @ sont des annotations qui permettent de signaler que la classe `Hello` représente un service Web (`@WebService`). L'autre annotation utilisée est `@WebMethod` qui permet d'indiquer les méthodes du service Web. Dans cet exemple il n'y a qu'une méthode paramétrée (`sayHelloWorldInParam(String name)`) qui renvoie un bonjour au nom passé en paramètre (`@WebParam`).
3. Dans le répertoire `webserv_prat_1/ws_hello/WEB-INF/src/` se trouve le fichier `build.properties`. Editer ce fichier pour modifier les lignes suivantes et la positionner comme suit (remplacer le chemin d'accès à `beehive.home` par la valeur correspondant à l'installation locale) :

```
-----build.properties-----
beehive.home=/home/fauvet/Beehive/apache-beehive-incubating-1.0m1/
service.name=Hello
-----
```

4. Aller dans le répertoire `webserv_prat_1/ws_hello` et dans le fichier `index.html` vérifier les lignes signalées ci-après :

```
-----index.html-----
<a href="web/Hello.jws?wsdl">WSDL</a>
<a href="web/Hello.jws?method=sayHelloWorldInParam&name=you">
    appel opération</a>
-----
```

5. Compilation et déploiement avec `ant` :

Aller dans le répertoire `webserv_prat_1/ws_hello/WEB-INF/src/` et exécuter la commande suivante :

```
ant clean build war
```

6. Avec l'outil Tomcat Web Application Manager (*WAR file to deploy*) déployer l'application contenue dans le fichier `HelloWS.war` (généré par ant dans le répertoire `webserv_prat_1`).

7. Observer les informations rendues disponibles au travers du lien `http://127.0.0.1:8080/HelloWS/`.

Implantation d'un client :

1. Dans le répertoire `webserv_prat_1/ws_client` créer le fichier `ClientHelloInParam.java` (voir son contenu, section A.8.5).
2. Compiler et exécuter `ClientHelloInParam` avec les paramètres adéquats.

A.6 Perfect Glass Inc. et l'entrepôt

A.6.1 Version 1 : paramètres simples

Les fichiers `qDemandeeLocal.java` (voir son contenu, section A.8.6) et `EntrepotLocal.java` (voir son contenu, section A.8.7) contiennent respectivement la description des opérations `qDemandee` implantée par Perfect Glass Inc. et `Disponibilite` implantée par l'Entrepôt (étude de cas vue dans le cours).

Les codes fournis utilisent des actions de saisie définies dans `Saisie.java` à compiler et à rendre accessible (voir son contenu, section A.8.8).

Dans ces codes, on a considéré que le client (Perfect Glass Inc.) et le fournisseur (Entrepôt) étaient accessibles en local, sur la même machine.

Modifier les codes fournis afin d'implanter le fournisseur via un service web : sur le modèle de l'arborescence `ws_time`, compléter l'arborescence de racine `Entrepot` (dans le répertoire `webserv_prat_1`).

1. Créer `Entrepot.java` dans le répertoire `webserv_prat_1/Entrepot/WEB-INF/src-ws/web/`
2. Adapter les fichiers `build.properties` et `index.html`.
3. Compiler et déployer le service.
4. Tester le déploiement via le serveur web (Tomcat).

Implanter le client :

1. Créer `qDemandee.java` dans un répertoire, par exemple `webserv_prat_1/PerfectGlass`. Veiller à ce que ce dernier répertoire soit dans la variable d'environnement `CLASSPATH`.
2. Compiler et tester le client.

Voir le corrigé côté Perfect Glass Inc. et côté Entrepôt. Penser à modifier si nécessaire les paramètres concernés dans le fichier `build.properties`.

A.6.2 Version 2 : paramètres complexes

L'objectif est d'étendre le service `Hello` et son client `ClientHello`. Cette extension consiste à paramétriser l'opération `sayHello` exposée par le service `Hello` de manière à ce qu'elle reçoive un objet de la classe `Personne` et renvoie une chaîne de caractères.

Il s'agit ici d'exploiter la possibilité offerte par Axis de sérialiser/désérialiser des classes Java implantées selon le patron des *Java Beans* qui fixe le format d'écriture des sélecteurs et des constructeurs.

1. Copier et décompresser l'archive `ws_helloObjet.tar` (cliquer ici) sous la racine `webserv_prat_1/`.
2. Observer l'arborescence ainsi créée : le service s'appuie sur classe `Personne` dont le code est fourni dans le paquetage `hello` (voir `ws_helloObjet/WEB-INF/src/hello/Personne.java`).
3. Créer le client `ClientHelloObjet.java` (voir son contenu, section A.8.9) et la classe `Personne.java` (voir son contenu, section A.8.10) dans le répertoire `webserv_prat_1/ws_client`. Compiler `Personne.java` et `ClientHelloObjet.java`

4. Personnaliser les paramètres dans `build.properties` (placé dans le répertoire `webserv_prat_1/ws_helloObjet/WEB-INF/`).
5. Compiler et déployer le service. Tester.

Sur le modèle fourni, modifier le service `Entrepot` et son client `PerfectGlass`, de manière à ce que l'opération `Disponibilite` admette un paramètre de type `Produit` (à définir).

Voir le corrigé client côté Perfect Glass Inc., (classe `Produit`) et (cliquer ici) pour obtenir l'archive qui contient le service `EntrepotObjet`.

Penser à modifier en conséquence les paramètres concernés dans le fichier `build.properties`.

A.7 Composition de services

A.7.1 Composition à base d'opérations bi-directionnelles

1. Modifier le service `Hello` afin qu'il réalise l'appel au service `TimeNow` et retourne au client le message issu de la concaténation du salut et la date système. Le diagramme de séquence modélisant les interactions est décrit par la figure A.1.
2. Implanter le client (sur la base du client `ClientHelloInParam.java` déjà étudié).

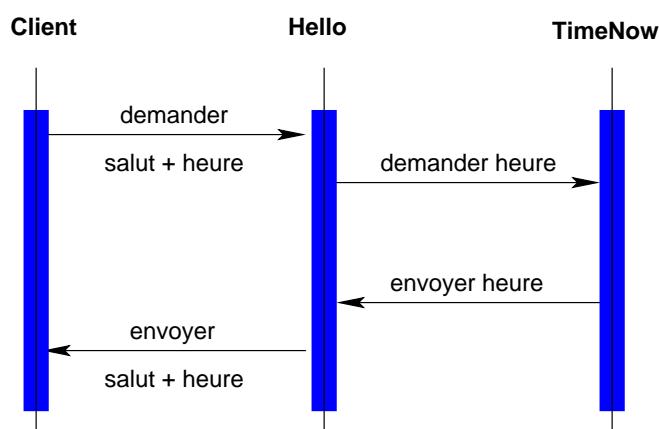


Figure A.1 – Interactions entre les services sur la base d'opérations bidirectionnelles.

Voir le corrigé pour le service et pour le client. Penser à modifier en conséquence les paramètres concernés dans le fichier `build.properties`.

A.7.2 Composition à base d'opérations uni-directionnelles

L'objectif est d'étudier l'implantation de services dont les interactions s'appuient sur des opérations unidirectionnelles. Le diagramme de séquence modélisant ces interactions, dans le cadre des échanges entre le département Ventes de Perfect Glass Inc. et l'entrepôt est décrit par la figure A.2.

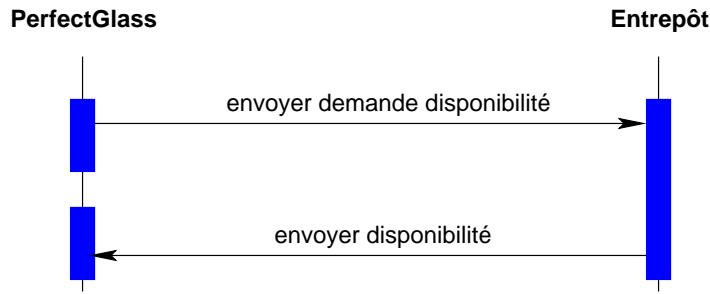


Figure A.2 – Interactions entre les services sur la base d’opérations unidirectionnelles.

1. Copier et décompresser l’archive `EntrepotUnidir.tar` (cliquer ici) sous la racine `webserv_prat_1/`
2. Copier et décompresser l’archive `PerfectUnidir.tar` (cliquer ici) sous la racine `webserv_prat_1/`. Cette archive contient le client `PerfectGlassClient.java` qui initialise les interactions, et l’arborescence du service `PerfectGlassUnidirService` qui traite la réception de la réponse envoyée par l’entrepôt.
3. Compiler le client `PerfectGlassClient.java` ainsi que la classe `Produit.java`.
4. Personnaliser les paramètres dans `build.properties`, pour chacun des deux services `EntrepotUnidir` et `PerfectGlassUnidir`.
5. Compiler et déployer les services. Tester (le message de retour est enregistré dans le fichier `PerfectGlass` placé dans `/tmp`).

Sur le modèle fourni planter les interactions décrites dans la figure A.3.

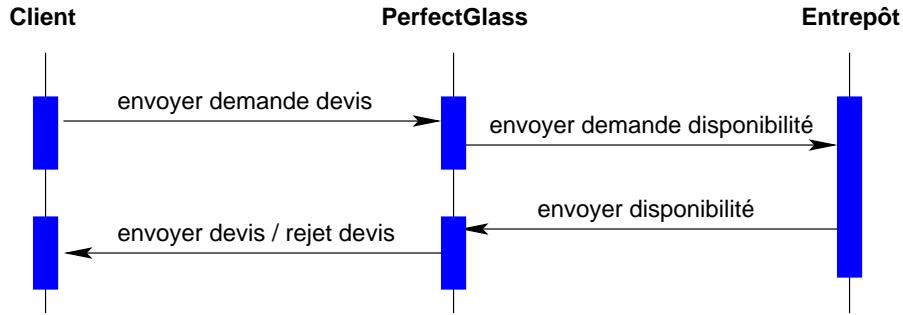


Figure A.3 – Interactions entre les services Client, PerfectGlass et Entrepôt.

A.8 Ressources

A.8.1 Arborescence de webserv_prat_1/ (telle que fournie dans l’énoncé)

```

webserv_prat_1/
webserv_prat_1/EntrepotLocal/
  
```

```

webserv_prat_1/ws_hello/
webserv_prat_1/ws_hello/WEB-INF/
webserv_prat_1/ws_hello/WEB-INF/server-config.wsdd
webserv_prat_1/ws_hello/WEB-INF/src/
webserv_prat_1/ws_hello/WEB-INF/src/build.xml
webserv_prat_1/ws_hello/WEB-INF/src/build.properties
webserv_prat_1/ws_hello/WEB-INF/web.xml
webserv_prat_1/ws_hello/WEB-INF/src-ws/
webserv_prat_1/ws_hello/WEB-INF/src-ws/web/
webserv_prat_1/ws_hello/index.html
webserv_prat_1/ws_client/
webserv_prat_1/PerfectGlassLocal/
webserv_prat_1/Entrepot/
webserv_prat_1/Entrepot/WEB-INF/
webserv_prat_1/Entrepot/WEB-INF/server-config.wsdd
webserv_prat_1/Entrepot/WEB-INF/src/
webserv_prat_1/Entrepot/WEB-INF/src/build.xml
webserv_prat_1/Entrepot/WEB-INF/src/build.properties
webserv_prat_1/Entrepot/WEB-INF/web.xml
webserv_prat_1/Entrepot/WEB-INF/src-ws/
webserv_prat_1/Entrepot/WEB-INF/src-ws/web/
webserv_prat_1/Entrepot/index.html
webserv_prat_1/ws_time/
webserv_prat_1/ws_time/happyaxis.jsp
webserv_prat_1/ws_time/WEB-INF/
webserv_prat_1/ws_time/WEB-INF/server-config.wsdd
webserv_prat_1/ws_time/WEB-INF/src/
webserv_prat_1/ws_time/WEB-INF/src/build.xml
webserv_prat_1/ws_time/WEB-INF/src/build.properties
webserv_prat_1/ws_time/WEB-INF/web.xml
webserv_prat_1/ws_time/WEB-INF/src-ws/
webserv_prat_1/ws_time/WEB-INF/src-ws/web/
webserv_prat_1/ws_time/index.html
webserv_prat_1/PerfectGlass/

```

A.8.2 Code du service TimeNow

```

package web;
import javax.jws.WebMethod;
import javax.jws.WebService;
import java.util.Date;

@WebService
public class TimeNow {
    @WebMethod
    public String getTimeNow() {
        Date d=new Date();
        String t=d.toString();
        return t;
    }
}

```

A.8.3 Code du client ClientTimeNow

```

import org.apache.axis.client.Call;
import org.apache.axis.client.Service;
import org.apache.axis.encoding.XMLType;
import org.apache.axis.utils.Options;

import javax.xml.namespace.QName;

public class ClientTimeNow {
    public static void main(String [] args) {
        try {

            // l'URI a contacter
            String endpointURL = "http://localhost:8080/TimeNowWS/web/TimeNow.jws";

            // Le service à executer
            Service service = new Service();
            Call call = (Call) service.createCall();
            call.setTargetEndpointAddress( new java.net.URL(endpointURL) );

            // l'operation du service
            call.setOperationName( new QName("TimeNow", "getTimeNow") );

            // L'appel
            String ret = (String) call.invoke( new Object[] { } );

            System.out.println("Bonjour d'Autrans !! Ici, il est " + ret);
        } catch (Exception e) {
            System.err.println("erreur "+ e.toString());
        }
    }
}

```

A.8.4 Code du service Hello

```

package web;

import javax.jws.WebMethod;
import javax.jws.WebService;
import javax.jws.WebParam;

@WebService
public class Hello {

    @WebMethod
    public String sayHelloWorldInParam(@WebParam String name ) {
        try {

            return "Hello, " + name + ". ";
        } catch (Exception e) {
            return "Erreur " + e.toString();
        }
    }
}

```

A.8.5 Code du client ClientHelloInParam

```

import org.apache.axis.client.Call;
import org.apache.axis.client.Service;
import org.apache.axis.encoding.XMLType;
import org.apache.axis.utils.Options;

import javax.xml.namespace.QName;
import javax.xml.rpc.ParameterMode;

public class ClientHelloInParam {
    public static void main(String [] args) {
        try {
            // on attend un parametre
            Options options = new Options(args);
            String textToSend;

            args = options.getRemainingArgs();
            if ((args == null) || (args.length < 1)) {
                textToSend = "l'inconnu";
            } else {
                textToSend = args[0];
            }

            // Le necessaire pour realiser l'appel :
            Service service = new Service();
            Call call = (Call) service.createCall();

            // L'URI du service a appeler
            String url = "http://127.0.0.1:8080/HelloWS/web/Hello.jws";
            call.setTargetEndpointAddress( new java.net.URL(url) );

            // L'operation a executer avec ses parametres d'entree et de sortie
            call.setOperationName( new QName("Hello", "sayHelloWorldInParam") );
            call.addParameter( "s", XMLType.XSD_STRING, ParameterMode.IN );
            call.setReturnType( org.apache.axis.encoding.XMLType.XSD_STRING );

            // L'appel
            String ret = (String) call.invoke( new Object[] { textToSend } );

            System.out.println(ret);

        } catch (Exception e) {
            System.err.println(e.toString()+" ici");
        }
    }
}

```

A.8.6 Code du programme qDemandeeLocal

```

import java.io.* ;

public class qDemandee{
    public static void main(String args[]) {
        try {
            String produit; // la référence du produit
            int quantite ; // la quantité demandée

            produit = Saisie.lire_String ("Saisie de la référence produit : ");
            quantite = Saisie.lire_int ("Saisie de la quantité demandée : ");

            System.out.println ("Vérification de disponibilité.....");

            String ret = Entrepot.Disponibilite (produit, quantite);

            System.out.println(ret);

        } catch (Exception e) {
            System.err.println(e.toString()+" ici");
        }
    }
}

```

A.8.7 Code du programme EntrepotLocal

```

class Produit {
    private String reference ;
    private int quantite ;

    public Produit() {
    }

    public void setRef(String ref) {
        reference = ref ;
    }
    public void setQuantite (int qte) {
        quantite = qte ;
    }

    public String getRef() {
        return reference;
    }

    public int getQuantite() {
        return quantite ;
    }
}

public class Entrepot {
    public static String Disponibilite (String produit, int quantite) {

```

```
try {
    String res ;
    Produit v = new Produit () ;
    Produit a = new Produit ();

    v.setRef("verres") ;
    v.setQuantite(200);

    a.setRef("assiettes") ;
    a.setQuantite(4500);

    res = "Le produit " + produit + ", en quantité " + quantite ;
    if ((v.getRef().equals (produit) && v.getQuantite() >= quantite) ||
        (a.getRef().equals (produit) && a.getQuantite() >= quantite)) {
        res = res + ", est disponible." ;
    } else {
        res = res + ", est indisponible" ;
    }
    return res ;
} catch (Exception e) {
    return "Erreur " + e.toString();
}
}
```

A.8.8 Code des procédures de Saisie

```
import java.io.*;
class Saisie {
public static String lire_String ()
{
    String ligne_lue=null ;
    try {
        InputStreamReader isr = new InputStreamReader(System.in);
        BufferedReader br = new BufferedReader(isr);
        ligne_lue = br.readLine ();
    }
    catch (IOException e) {System.err.println(e);}
    return ligne_lue;
}
public static String lire_String (String question)
{
    System.out.print (question);
    return (lire_String());
}
public static int lire_int ()
{
    return Integer.parseInt (lire_String ());
}
public static int lire_int (String question)
{
```

```

        System.out.print (question);
        return (lire_int());
    }

    public static double lire_double()
    {
        return Double.parseDouble (lire_String ());
    }
    public static double lire_double (String question)
    {
        System.out.print (question);
        return (lire_double());
    }

    public static float lire_float()
    {
        return Float.parseFloat (lire_String ());
    }
    public static float lire_float (String question)
    {
        System.out.print (question);
        return (lire_float());
    }

    public static char lire_char()
    {
        String reponse = lire_String ();
        return reponse.charAt(0);
    }
    public static char lire_char (String question)
    {
        System.out.print (question);
        return (lire_char());
    }
}

```

A.8.9 Code du client ClientHelloObjet

```

import org.apache.axis.client.Call;
import org.apache.axis.client.Service;
import org.apache.axis.encoding.XMLType;
import org.apache.axis.utils.Options;

import javax.xml.namespace.QName;
import javax.xml.rpc.ParameterMode;

public class ClientHelloObjet {
    public static void main(String [] args) {
        try {
            String nom ;
            int age ;

```

```

String adresse ;
nom = Saisie.lire_String ("Donner un nom : ");
age = Saisie.lire_int ("Donner un age : ") ;
adresse = Saisie.lire_String ("Donner une adresse : ") ;

Personne personne = new Personne () ;
personne.setAge(age) ;
personne.setAdresse(adresse) ;
personne.setNom(nom) ;

System.out.println ("Connexion .....");

Service service = new Service();
Call call = (Call) service.createCall();

String url = "http://127.0.0.1:8080/HelloObjetWS/web/HelloObjet.jws";
call.setTargetEndpointAddress( new java.net.URL(url) );

// Le necessaire pour le parametre en entree
// qn est la correspondance XML de la classe Personne
QName personneXML = new QName( "http://HelloObjetWS/web", "Personne" );

call.registerTypeMapping(Personne.class, personneXML,
    new org.apache.axis.encoding.ser.BeanSerializerFactory(
        Personne.class, personneXML),
    new org.apache.axis.encoding.ser.BeanDeserializerFactory(
        Personne.class, personneXML));

// L'operation a executer avec ses parametres d'entree et de sortie
call.setOperationName( new QName("HelloObjet", "sayHello") );
call.addParameter( "p", personneXML, ParameterMode.IN);
call.setReturnType( org.apache.axis.encoding.XMLType.XSD_STRING );

String ret = (String) call.invoke( new Object[] { personne } );

System.out.println(ret);

} catch (Exception e) {
    System.err.println(e.toString()+" ici");
}
}
}

```

A.8.10 Code de la classe Personne

```
public class Personne {  
    private String nom ;  
    private int age ;  
    private String adresse ;  
  
    public Personne () {
```

```
}

public void setNom(String n) {
    nom = n ;
}
public void setAdresse (String a) {
    adresse = a ;
}
public void setAge (int a) {
    age = a ;
}

public String getAdresse() {
    return adresse;
}

public String getNom() {
    return nom;
}

public int getAge() {
    return age;
}

}
```


Annexe B

.NET en pratique

L'objectif de cette annexe est de permettre à chacun de se familiariser un peu avec l'environnement de programmation .NET. Pour cela nous allons développer une petite application permettant à chacun de gérer ses cartes de visite (ajout, suppression et consultation) et de les rendre accessibles aux autres utilisateurs (uniquement consultation). Pour atteindre ce but, l'application se compose de deux parties : le service d'annuaire qui doit pouvoir être utilisée à distance et différentes applications clientes de type client léger ou client lourd graphique ou en ligne de commande. Pour des raisons de simplicité nous ne nous intéresserons pas aux problèmes liés à l'authentification des différentes classes d'utilisateurs.

Une version html de ce texte est disponible à l'adresse <http://rangiroa.essi.fr/riveill/enseignement/tp/carteVisite.html>. Elle contiendra les évolutions futures du sujet.

B.1 Les outils à utiliser

Comme nous allons faire des développements simples nous allons utiliser les outils mis 'gracieusement' en ligne par Microsoft et les installer les uns après les autres :

1. Le *framework .NET*¹ est généralement installé à l'adresse C:\Windows\Microsoft .Net\Framework et contient les principales classes nécessaires à l'exécution des programmes ainsi que le compilateur (**csc.exe**). Pensez à mettre à jour vos variables d'environnement avec l'adresse d'installation du *framework .NET*.
2. Le *kit de développement (SDK)*² est généralement installé à l'adresse C:\Program Files\Microsoft.Net\SDK et contient les principaux utilitaires : **xsd.exe** (pour générer la documentation associée aux classes), **ildasm.exe** (le désassembleur) ou **wsdl.exe** (le générateur de talon pour les services web).

¹Framework .Net : <http://www.microsoft.com/france/msdn/netframework/default.mspx>

²SDK : <http://www.microsoft.com/france/msdn/netframework/default.mspx>

3. L'environnement de programmation *Web Matrix*³ a comme principaux avantages d'avoir une faible empreinte mémoire, d'être gratuit, simple d'utilisation, d'inclure un serveur HTTP et de permettre la création et l'accès à des bases de données simples au format *access* sans aucune installation supplémentaire.

Ce n'est pas le seul environnement possible, en effet vous pouvez utiliser *Mono*⁴, comme implémentation du framework et *SharpDevelop*⁵ comme environnement de programmation. Par ailleurs, si vous le souhaitez, Microsoft met à disposition des programmeurs des versions gratuites de ses environnements de développement⁶. L'offre *MSDN Academic Alliance*⁷ permet aux universités d'avoir accès à Visual Studio, SQL Serveur et à l'ensemble des outils de développement commercialisés par Microsoft.

B.2 Architecture de l'application “Cartes de Visite”

Nous allons développer au cours de ce TD une application de gestion de cartes de visite qui respectera l'architecture décrite dans la figure B.1.

Cette application sera développée en C#.

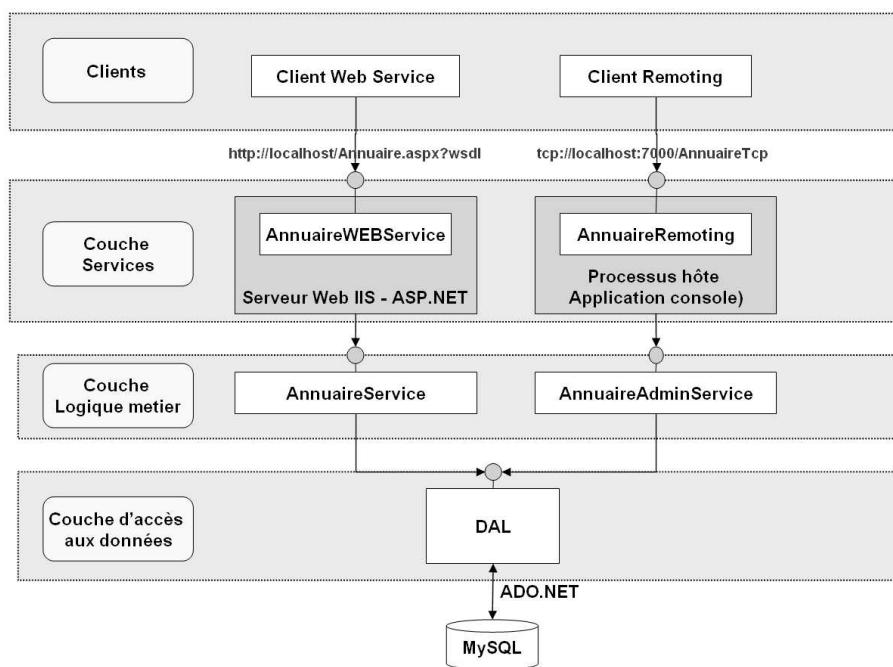


Figure B.1 – Architecture de l'application

³Web Matrix : <http://www.asp.net/webmatrix/>

⁴Mono : <http://www.go-mono.org>

⁵SharpDevelop : <http://sourceforge.net/projects/sharpdevelop/>

⁶Visual Studio Express : <http://www.microsoft.com/france/msdn/vstudio/express/default.mspx>

⁷MSDN AA : <http://www.microsoft.com/france/msdn/abonnements/academic/default.mspx>

B.3 Les objets métiers

Nous choisissons de représenter les données de chaque carte dans des instances d'une classe `CarteDeVisite` dont le code C# est donné ci-dessous :

`CarteDeVisite (CarteDeVisite.cs)`

```
using System;
using System.Xml;
using System.Xml.Serialization;

namespace MonPremierTP {
    [Serializable] [XmlType("carte-de-visite")]
    public class CarteDeVisite {
        [XmlElement("nom")] public string Nom;
        [XmlElement("prenom")] public string Prenom;
        [XmlElement("email")] public string Email;
        [NonSerialized()]
        public string Telephone;
    }
}
```

Nous verrons ultérieurement comment les différentes fiches seront initialisées par des données issues de la base de données. La description et l'utilisation des différents attributs sont présentés dans la section B.6.5.

Pour générer la bibliothèque `CarteDeVisite.dll`, la commande est la suivante :

```
csc /target:library CarteDeVisite.cs
```

B.4 La couche d'accès aux données

Cette couche permet de d'alimenter en données les différents objets métiers manipulés par l'application cartes de visite. Nous donnons uniquement ici l'interface, la description de l'implémentation sera faite ultérieurement. Voici la version 0.1 :

`AnnuaireDAL (AnnuaireDAL.cs)`

```
// volontairement le namespace n'est pas répéter...
// à vous de l'indiquer et d'utiliser les bonnes directives 'using'
[assembly: AssemblyVersion("0.1.0.0")]
public class AnnuaireDAL {
    public AnnuaireDAL() {} // constructeur
    public CarteDeVisite ExtraireCarte(string nomRecherche) {
        // opération de recherche d'une carte dans la BD
        return null;
    }
    public void AjouterCarte(CarteDeVisite carte){
        // opération d'insertion d'une nouvelle carte
    }
}
```

Pour générer la bibliothèque AnnuaireDAL.dll, la commande est la suivante :

```
csc /target:library /reference:CarteDeVisite.dll AnnuaireDAL.cs
```

B.5 La couche métier

Quand nous aurons implémenté la classe *AnnuaireDAL* décrite dans la section précédente, nous disposerons d'un moyen d'accéder aux données persistantes du service d'annuaire sans avoir à connaître la technologie de stockage employée. L'interrogation et la mise à jour des données seront faites au travers des interfaces de la *DAL (Data Access Layer)* en ne manipulant que des entités liées au domaine métier.

Pour créer l'indépendance entre conservation des données et manipulation des données, nous allons créer la couche contenant la logique du service d'annuaire. Il s'agit d'implanter deux services, l'un pour consulter les cartes de visites existantes et un autre pour en insérer de nouvelles. L'idée est ici de spécifier clairement les services sous la forme d'interface. Cette couche ne sera manipulée qu'au travers de ces seules interfaces et non par un accès direct à leur implantation. Voici les interfaces des deux services :

IAnnuaireService (IAnnuaireService.cs)

```
public interface IAnnuaireService {
    CarteDeVisite Rechercher(string nom);
}

public interface IAnnuaireAdminService {
    CarteDeVisite Rechercher(string nom);
    void Ajouter(CarteDeVisite carte);
}
```

Une fois ces interfaces définies, il ne nous reste qu'à les implanter en utilisant la couche d'accès aux données décrite précédemment.

AnnuaireService (AnnuaireService.cs)

```
public class AnnuaireService : IAnnuaireService {
    // constructeur de la classe
    public AnnuaireService() {}

    // implémentation de la méthode métier
    public CarteDeVisite Rechercher(string nom) {
        AnnuaireDAL dal=new AnnuaireDAL();
        return dal.ExtraireCarte(nom);
    }
}

public class AnnuaireAdminService : IAnnuaireAdminService {
    // constructeur de la classe
    public AnnuaireAdminService(){}
    // implémentation de la méthode métier
    public CarteDeVisite Rechercher(string nom) {
        AnnuaireDAL dal=new AnnuaireDAL();
```

```

        return dal.ExtraireCarte(nom);
    }
    public void Ajouter(CarteDeVisite carte) {
        AnnuaireDAL dal=new AnnuaireDAL();
        // on vérifie que la carte n'existe pas déjà
        CarteDeVisite test=dal.ExtraireCarte(carte.Nom);
        if (test==null)
            dal.AjouterCarte(carte);
        else
            throw new Exception("La carte existe déjà");
    }
}

```

B.6 Premières manipulations en C#

Ces expérimentations ne sont pas directement liées avec la mise en œuvre de l'application cartes de visite mais permettent de se familiariser avec le langage C# et le framework .Net.

B.6.1 Construire le diagramme UML

A cette étape du TP, nous avons écrit 4 fichiers : `CarteDeVisite.cs`, `AnnuaireDAL.cs`, `IAnnuaireService.cs` et `AnnuaireService.cs` et il peut être souhaitable de construire le diagramme UML de l'ensemble.

B.6.2 Générer les fichiers de documentation

Pour commenter le code, C# offre trois types de commentaires :

- les commentaires introduits par les balises '//' , la fin étant la fin de ligne ;
- les commentaires introduits par les balises '/*' et terminé par les balises '*/' ;
- les commentaires introduits par les balises '///' , la fin étant la fin de ligne.

Seul, ce dernier type va nous intéresser car il permette de générer des fichiers de documentation associés au code développé. Dans *Visual Studio* ou *SharpDevelop*, le fait de mettre trois slash à la suite dans un endroit valide fait apparaître un menu avec les balises XML de documentation dont nous présentons ici les principales :

- La balise *summary* sert à donner la description complète de l'élément que l'on souhaite documenter. Il peut être utilisé sur une classe, une méthode, d'une propriété ou une variable.
- La balise *param* permet de documenter les paramètres d'une méthode ou d'une propriété. Il prend en complément uniquement le nom de la variable, le type de la variable est automatiquement déterminé par le générateur de documentation.
- La balise *returns* permet de documenter la valeur de retour d'une fonction seulement.
- La balise *value* permet de décrire la valeur de retour ou d'entrée d'une propriété. Elle joue le même rôle que la balise *param* pour une fonction.
- La balise *paramref* permet d'indiquer que le mot dans le commentaire est un paramètre de la fonction afin que le générateur de documentation puisse mettre un lien vers le commentaire du paramètre.

- La balise *exception* permet d'informer sur le(s) type(s) d'exception(s) que la fonction peut lever. La propriété *cref* du tag permet de spécifier le type d'exception documenté.
- Il existe bien d'autres balises : *<c>*, *<code>*, *<example>*, *<include>*, *<list>*, *<para>*, *<permission>*, *<remarks>*, *<jsee>*, *<seealso>*. A vous de découvrir leur utilisation en lisant la documentation.

Voici un exemple de programme commenté avec ces balises :

```
// xml_AnnuaireDAL.cs
// à compiler par : csc /doc:xml_AnnuaireDAL.xml

[assembly:AssemblyVersion("0.2.0.0")]
/// Mon texte pour la classe AnnuaireDAL
/// <summary>
/// Description de la classe AnnuaireDAL .
/// </summary>
public class AnnuaireDAL {
    /// <remarks>
    /// Des commentaires plus longs peuvent être associés à un type ou un membre
    /// grâce à la balise remarks</remarks>
    /// <example> Cette classe permet d'isoler la partie métier de la persistance
    /// des données, elle comporte essentiellement deux méthodes.
    /// <code>
    /// public CarteDeVisite ExtraireCarte(string nomRecherche)
    /// public void AjouterCarte(CarteDeVisite carte)
    /// </code>
    /// </example>
    /// <summary>
    /// Constructeur de la classe AnnuaireDAL.
    /// </summary>
    public AnnuaireDAL() {} // constructeur

    /// texte pour la méthode ExtraireCarte
    /// <summary>
    /// Cette méthode très sophistiquée reste totalement à implémenter. Le paramètre
    /// d'entrée est <paramref name="nomRecherche" />
    /// </summary>
    /// <param name="nomRecherche">
    /// le paramètre d'entrée est le nom du titulaire de la carte de visite recherchée
    /// </param>
    /// <returns>
    /// le résultat est la carte de visite associée au nom
    /// <returns>
    /// <exception cref="NomInconnu">
    /// Une exception est levée si le nom est inconnu
    /// </exception>
    public CarteDeVisite ExtraireCarte(string nomRecherche) {
        // opération de recherche d'une carte dans la BD
        return null;
    }
}
```

```

/// texte pour la méthode AjouterCarte
/// <summary>
/// Description de AjouterCarte.</summary>
/// <param name="carte"> Emplacement de la description du paramètre carte</param>
/// <seealso cref="String">
/// Vous pouvez utiliser l'attribut cref sur n'importe quelle balise pour
/// faire référence à un type ou un membre,
/// et le compilateur vérifiera que cette référence existe. </seealso>
public void AjouterCarte(CarteDeVisite carte){
    // opération d'insertion d'une nouvelle carte
}

/// <summary>
/// Une propriété introduite juste pour montrer la balise "value".
/// </summary>
/// <value>donne toujours 0492965148</value>
public int UnePropriete {
    get {
        return 0492965148;
    }
}
}

```

Mettez ce code dans le fichier `commentaire1.cs` et compilez le : `csc.exe /doc :commentaire1.xml /reference :CarteDeVisite.dll commentaire1.cs` et examinez le fichier XML produit.

Maintenant, à vous de jouer en commentant les différentes classes précédemment écrites. Faites bien la différence entre les commentaires débutants par '///', pris en compte dans la génération du fichier de documentation XML et les commentaires débutant par '//' ou '/*', seulement présent dans le fichier source.

Attention : lorsque vous générerez la documentation, vous compiler aussi le code en question.

B.6.3 Compilation du code

Compilez les différents fichiers pour obtenir une dll.

```
csc.exe /out:annuaire.dll /t:library /r:AnnuaireDAL.dll /r:CarteDeVisite.dll \
AnnuaireService.cs IAnnuaireService.cs
```

Observez avec `ildasm` la *dll* produite et commentez. Qu'observez-vous : nombre de classes, fonctions membres ?

```
ildasm.exe annuaire.dll
```

B.6.4 Utilisation de la réflexivité

Nous allons utiliser la réflexivité pour imprimer à l'écran les différents éléments de la dll précédemment produite. Voici un exemple :

```

using System;
using System.IO;
using System.Reflection;

public class Meta {
    public static int Main () {
        // lire l'assembly
        Assembly a = Assembly.LoadFrom ("annuaire.dll");

        // lire les modules de l'assembly
        Module[] modules = a.GetModules();
        // inspecter tous les modules de l'assembly
        foreach (Module module in modules) {
            Console.WriteLine("Dans le module {0}, on trouve", module.Name);
            // lire tous les types du module
            Type[] types = module.GetTypes();
            // inspecter tous les types
            foreach (Type type in types) {
                Console.WriteLine("Le type {0} a cette(ces) méthode(s) : ", type.Name);
                // inspecter toutes les méthodes du type
                MethodInfo[] mInfo = type.GetMethods();
                foreach (MethodInfo mi in mInfo) {
                    Console.WriteLine (" {0}", mi);
                }
            }
        }
        return 0;
    }
}

```

Compilez cette classe (`csc <nom de fichier>.cs`) et exécutez le code, puis faites une lecture attentive du source. Avez-vous tout compris ?

B.6.5 Sérialisation binaire et XML

Voici un cours programme qui permet de sérialiser/désérialiser une carte de visite en XML et en format binaire.

```

using System;
using System.IO;
using System.Runtime.Serialization.Formatters.Binary;
using System.Xml;
using System.Xml.Serialization;

class Serialisation {
    static void Main(string[] args) {
        CarteDeVisite uneCarte = new CarteDeVisite();

        uneCarte.Nom = "RIVEILL";
        uneCarte.Prenom = "Michel";
        uneCarte.Email = "riveill@unice.fr";
    }
}

```

```

uneCarte.Telephone = "04 92 96 51 48";

// on sérialise en binaire et on sauvegarde dans un fichier
Stream stream = File.Open("Demo.bin", FileMode.Create);
BinaryFormatter formatter = new BinaryFormatter();
formatter.Serialize (stream, uneCarte);
stream.Close();

// on sérialise en XML et on sauvegarde dans un fichier
Stream stream2 = File.Open("Demo.xml", FileMode.Create);
XmlSerializer serializer = new XmlSerializer (typeof(CarteDeVisite));
serializer.Serialize(stream2, uneCarte);
stream2.Close();

// on désérialise depuis un fichier (mode binaire)
stream = File.Open ("Demo.bin", FileMode.Open);
formatter = new BinaryFormatter();
CarteDeVisite obj = (CarteDeVisite) formatter.Deserialize (stream);
stream.Close ();

// on désérialise depuis un fichier (mode XML)
stream2 = File.Open ("Demo.xml", FileMode.Open);
serializer = new XmlSerializer (typeof(CarteDeVisite));
CarteDeVisite obj2 = (CarteDeVisite) serializer.Deserialize (stream2);
stream2.Close ();

Console.WriteLine ("\t\torigine\t\tbinaire\t\tXml");

// Nom, prenom et email ont bien été sérialisés
Console.WriteLine ("uneCarte.nom (a ete serialisee)");
Console.WriteLine ("\t\t{0}\t\t{1}\t\t{2}", uneCarte.Nom, obj.Nom, obj2.Nom);

Console.WriteLine ("uneCarte.prenom (a ete serialisee)");
Console.WriteLine ("\t\t{0}\t\t{1}\t\t{2}", uneCarte.Prenom, obj.Prenom, obj2.Prenom);

Console.WriteLine ("uneCarte.email (a ete serialisee)");
Console.WriteLine ("\t\t{0}\t\t{1}\t\t{2}", uneCarte.Email, obj.Email, obj2.Email);

// téléphone n'a pas été sérialisé
Console.WriteLine ("uneCarte.telephone (n'a pas ete serialisee)");
Console.WriteLine ("\t\t{0}\t\t{1}\t\t{2}", uneCarte.Telephone,
                    obj.Telephone, obj2.Telephone);
}

}

```

Compilez et exécutez ce code. Analysez-le.

Remarque : Si vous sérialisez un tableau ou une collection, tous les objets du tableau ou de la collection sont sérialisés.

B.7 La couche de stockage physique des données

On reprend la conception de l'application. Les données stockées dans l'annuaire sont des cartes de visite contenant un nom, un prénom, un courriel et un numéro de téléphone. Pour que ceux qui ne sont pas très aguerri avec la manipulation des bases de données il est possible de créer celles-ci directement avec *Web Matrix*. Voici la manière de procéder :

1. ouvrir *web matrix*
2. aller dans la fenêtre 'Workspace'
3. sélectionner l'onglet 'Data'
4. cliquer sur l'icône 'Nouvelle Base de Données'
5. créer une base de données Access
6. créer la table en sélectionnant table, dans la BD créée
7. cliquer sur l'icône 'Nouvelle Table'
8. Pour finir, créer les entrées de la table qui doivent au moins contenir les champs suivants ainsi qu'une clé primaire non décrite ici :

```
string Nom;
string Prenom;
string Email;
string Telephone;
```

Par exemple la table obtenue peut avoir le format décrit dans la figure B.2.

Column Name	Data Type	Size	Allow Nulls
key	AutoNumber	0	False
nom	Text	50	False
prenom	Text	50	True
email	Text	50	True
telephone	Text	50	True

Figure B.2 – Format de la base de données

Il est alors possible d'initialiser la base de données en sélectionnant l'onglet 'Data'. Un exemple de contenu est donné dans la figure B.3.

B.8 Accès direct à la base de données par un client léger

Dans un premier temps et pour nous familiariser avec l'environnement, nous allons construire un premier client capable d'accéder à la base de données. La procédure est relativement simple.

1. ouvrir *web matrix*
2. créer un nouveau fichier ('File', 'New File') puis 'General', 'ASP.Net Page', et finir de remplir le formulaire (le suffixe du fichier sera aspx).

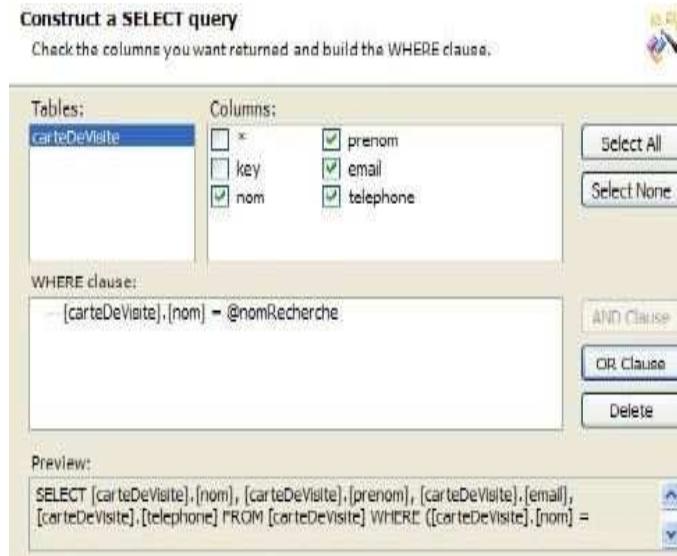
	key	nom	prenom	email	telephone
▶	1	RIVEILL	Michel	riveill@unicef	(null)
	2	BLAY	Mireille	(null)	(null)
	3	PINNA	Anne-Marie	(null)	(null)
	4	TIGLI	Jean-Yves	(null)	(null)
*	5	HUGUES	Anne-Marie	(null)	(null)

Figure B.3 – Contenu de la base de données

3. une fenêtre d'édition comportant 3 modes : 'dessin', 'html' et 'code' est à votre disposition ('all' est la fusion du mode 'html' et 'code'). En mode 'dessin' mettez sur la page une *TextBox* (saisie de chaîne de caractère), un *DataGridView* et un *Button*.
4. en mode 'html' vous pouvez renommer les identificateurs des différents éléments. Par exemple pour moi, le *TextBox* a "nom" comme identificateur, le *DataGridView* a "listeCartes" comme identificateur et le *Button* a "Button" comme id et "Chercher" comme Text.
5. écrire la méthode ExtraireCarteSQL à l'aide de l'utilitaire fournit par *Web Matrix* (procédure décrite ci-après).

Pour écrire la méthode ExtraireCarteSQL à l'aide de l'utilitaire présent dans *Web matrix* il faut être en mode 'code', puis faire glisser la zone **SELECT Data Method** de la colonne *ToolBox* à l'endroit où vous voulez insérer la méthode. Une boîte de dialogue s'ouvre. Elle permet de sélectionner la base de données, puis de construire la requête SELECT souhaitée.

Pour cela, sélectionnez les bonnes colonnes, remplissez la clause WHERE pour obtenir la requête décrite dans la figure B.4.

**Figure B.4 – Clause WHERE**

Après avoir cliqué sur le bouton suivant, testez la requête écrite. Si elle convient, donnez le nom de la méthode et demandez le résultat sous la forme d'un `DataSet`.

Pour terminer l'exemple, il reste à programmer la méthode appelée lorsque le bouton sera sélectionné. Pour cela, en mode 'dessin', cliquez 2 fois sur le `Button`. La fenêtre d'édition bascule en mode 'code' et l'on peut saisir le code associé à l'évènement 'clic sur le `Button`'. Le code de cette méthode est le suivant :

```
listeCartes.DataSource = ExtraireCarteSQL (nom.Text);
listeCartes.DataBind();
listeCartes.Visible = true;
```

Petites explications : la première ligne met à jour la liste des données du `DataSet` en appelant la méthode `ExtraireCarteSQL` construite dans l'étape 6 précédente, le paramètre est le texte de la `TextBox` contenue dans la page. Les deux lignes mettent à jour le `DataSet` et le rendent visible.

Attention : cette méthode, certes très efficace en terme de programmation ne respecte pas du tout l'architecture initiale, elle a juste permis d'utiliser la base de données créée et de nous familiariser avec *Web Matrix* pour la création de page web dynamique. La suite du TP reprend la construction pas à pas des différentes couches de l'application cartes de visite.

B.9 La couche d'accès aux données

Cette couche a pour rôle de faire abstraction de la technologie utilisée pour stocker les données. Elle doit masquer le moteur de base de données utilisé à la couche suivante, la couche métier que nous avons déjà écrit. L'avantage de cette abstraction est qu'il est possible de remplacer le moteur de SGBD utilisé par toute autre moteur de base de données ou encore par un système de stockage simple à l'aide d'un système de gestion de fichier. Si une telle modification avait lieu, seule cette couche d'accès aux données devrait être réécrite et la modification serait transparente au reste de l'application.

L'API ADO.NET est utilisée pour dialoguer avec le serveur de base de données utilisé (ici Access). Il faut maintenant implémenter la classe en charge du dialogue avec la base de données pour lire et insérer des cartes de visite. Pour ne pas avoir à écrire directement les requêtes SQL d'interrogation de la base de données nous utiliserons *Web Matrix* pour construire les fonctions nécessaire comme dans la section B.8. Pour cela :

1. reprendre la classe `AnnuaireDal`,
2. compléter la méthode `ExtraireCarte` qui s'appuie sur la fonction `ExtraireCarteSQL` de la section B.8 précédente.
3. compléter la méthode `AjouterCarte` selon le même principe en utilisant la commande SQL `INSERT` pour créer la fonction `AjouterCarteSQL`. Les fonctions `ExtraireCarteSQL` et `AjouterCarteSQL` font partie du fichier `AnnuaireDal.cs`.

```
public CarteDeVisite ExtraireCarte(string nomRecherche) {
    CarteDeVisite carte = new CarteDeVisite ();
    System.Data.DataSet requeste = ExtraireCarteSQL (nomRecherche);
```

```

        carte.Nom = requete.Tables["Table"].Rows[0]["nom"];
        carte.Prenom = requete.Tables["Table"].Rows[0]["prenom"];
        carte.Email = requete.Tables["Table"].Rows[0]["email"];
        carte.Telephone = requete.Tables["Table"].Rows[0]["telephone"];
        return carte;
    }

    public void AjouterCarte(CarteDeVisite carte) {
        AjouterCarteSQL (carte.Nom, carte.Prenom, carte.Email, carte.Telephone);
    }
}

```

B.10 La logique métier : AnnuaireService.dll

Nous avons maintenant les différents éléments pour construire la bibliothèque AnnuaireService.dll par compilation du fichier annuaireService.cs. Voici un exemple de compilation séparée :

```

REM production des différents modules
csc /t:module CarteDeVisite.cs
csc /t:module /addModule:CarteDeVisite.netmodule AnnuaireDAL.cs
csc /t:module /addModule:CarteDeVisite.netmodule IAnnuaireService.cs

REM liaison des modules produits avec la logique métier
csc /out:AnnuaireService2.dll /t:library /addModule:IAnnuaireService.netmodule \
    /addModule:CarteDeVisite.netmodule /addmodule:AnnuaireDAL.netmodule AnnuaireService.cs

```

Voici un exemple de compilation globale :

```
csc /out:AnnuaireService.dll /t:library IAnnuaireService.cs CarteDeVisite.cs \
    AnnuaireDAL.cs AnnuaireService.cs
```

B.11 Utilisation depuis un client local

Nous allons maintenant, construire un client local capable d'ajouter une nouvelle fiche et de consulter les fiches présentes dans la base.

```

public class ClientLocal {
    public ClientLocal() { }
    static int Main() {
        try {
            IAnnuaireAdminService annuaire = new AnnuaireAdminService ();
            CarteDeVisite carte = annuaire.Rechercher ("RIVEILL");

            Console.WriteLine ("l'email de RIVEILL est {0}", carte.Email);
        } catch (Exception e) {
            Console.WriteLine (e.ToString());
        }
        return 0;
    }
}

```

B.12 Publications des services

Il s'agit d'exposer maintenant le service d'annuaire afin de le rendre accessible aux clients distants. Nous allons exposer ces services de deux manières différents. Le service d'ajout de carte de visite sera publié via le *.NET Remoting* et le service de consultation sera accessible au travers d'un Web Service. Dans le cas de cet exemple d'annuaire, il s'agit d'enrober la logique métier de l'application dans des conteneurs permettant l'accès distant au service. La figure suivante présente les différents objets qui seront créés dans le cas de l'approche *.Net Remoting* et dans le cas de l'approche par service web.

Les différents objets utilisés sont décrits dans la figure B.5.

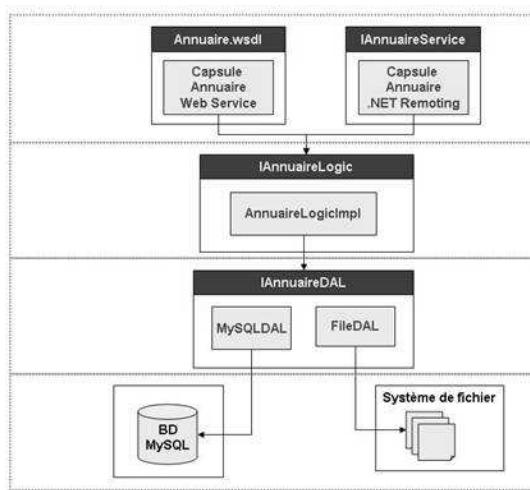


Figure B.5 – Objets utilisés pour les différents appels à distance

B.12.1 Publication .NET Remoting

Pour exposer un service via le *.NET Remoting* il faut créer un conteneur (objet servant dans la terminologie de l'OMG) capable de distribuer cet objet. Pour cela nous allons construire un exécutable qui publiera l'objet sur le port 7000 via un canal TCP en mode singleton (un seul objet serveur actif à la fois partagé par tous les clients).

AnnuaireServiceRemoting (AnnuaireServiceRemoting.cs)

```

using System;
using System.Runtime.Remoting;
using System.Runtime.Remoting.Channels;
using System.Runtime.Remoting.Channels.Tcp;

public class AnnuaireServiceRemoting: MarshalByRefObject, IAnnuaireService {
    IAnnuaireAdminService annuaire = new AnnuaireAdminService ();
    public CarteDeVisite Rechercher(string nom) {
        return annuaire.Rechercher(nom);
    }
}
  
```

```

    }
    public void Ajouter(CarteDeVisite carte) {
        annuaire.Ajouter(carte);
    }
}

```

Server (Server.cs)

```

using System;
using System.Runtime.Remoting;
using System.Runtime.Remoting.Channels;
using System.Runtime.Remoting.Channels.Tcp;

public class Server {
    [STAThread]
    static void Main(string[] args) {
        IChannel chan = new TcpChannel(7000);
        ChannelServices.RegisterChannel(chan, false);

        RemotingConfiguration.RegisterWellKnownServiceType(
            typeof(AnnuaireServiceRemoting),
            "TcpService",
            WellKnownObjectMode.Singleton);

        Console.WriteLine("Press Enter to disconnect the annuaire.");
        Console.ReadLine();
    }
}

```

B.12.2 Publication WEB Service

Pour pouvoir exécuter un service Web, il faut un serveur Web. Pour cela, nous allons utiliser Web Matrix qui permet d'activer un serveur Web interne. Le tutorial de Web Matrix (<http://rangiroa.essi.fr/cours/tutorial-webmatrix>) explique en détail la marche à suivre. Voici un cours résumé :

1. créer la classe `AnnuaireWebService` dans le fichier `AnnuaireWebService.asmx` (File, New File) puis 'General', 'XML Web Service', et finir de remplir la boîte de dialogue.
2. un canevas de service web est créé. Il vous reste à compléter le code pour qu'il accède à l'annuaire. Le code est presque le même que pour le servant dans le cadre du 'Remoting'. Seules les méthodes devant être accédées à distance sont précédées de l'attribut `[WebMethod]`.

```

<%@ WebService language="C#" class="AnnuaireWebService" Debug="true" %>

using System;
using System.Web.Services;
using System.Xml.Serialization;

public class AnnuaireWebService : System.Web.Services.WebService, IAnnuaireService {
    private IAnnuaireService service;

```

```

public AnnuaireWebService() {
    service=new AnnuaireService();
}

[WebMethod]
public CarteDeVisite Rechercher(string nom) {
    return service.Rechercher(nom);
}
}

```

3. Copier la dll dans un répertoire bin.
4. Sauvegarder et exécuter le service web (appuyer sur la touche F5). Une boîte de dialogue vous demande le port d'activation du Web Service (80 par défaut)
5. Un navigateur s'ouvre avec une page vous permettant d'interroger le service web.

`http://localhost:80/AnnuaireWebService.asmx pour avoir accès au service web
 http://localhost:8086/AnnuaireWebService.asmx?WSDL pour avoir accès à la description WSDL`

B.13 Utilisation de ces services

B.13.1 Par un client lourd

Dans le cas de .Net Remoting, le client n'a besoin de connaître que l'interface du service. Dans le cas de la consommation du service web il est nécessaire d'obtenir un proxy en C# sur ce service web. Ceci peut être fait par l'utilisation de l'utilitaire WSDL :

`wsdl.exe /out:proxy_webService.cs /n:proxy http://machine:port/URL?WSDL
 REM la directive /n permet de préciser l'espace de nom`

Voici de manière comparative les différentes manières de construire le client :

Client local

```

class Client {
    static void Main (string[] args) {
        // création de l'objet
        IAnnuaireService annuaire = new AnnuaireService ();

        CarteDeVisite c = annuaire.Recherche ("auteur");
    }
}

```

Client .NET Remoting

```

class Client {
    static void Main (string[] args) {
        // création d'une connection TCP
        TcpChannel channel = new TcpChannel ();
        ChannelServices.RegisterChannel (channel, false);
    }
}

```

```

// creation du proxy
IAnnuaireService annuaire = (IAnnuaireService) Activator.GetObject
    (typeof (IAnnuaireService),
    "tcp://127.0.0.1:7000/TcpService");

CarteDeVisite c = annuaire.Recherche ("auteur");
}
}

```

Client service web

```

class Client {
    static void Main (string[] args) {
        // creation du proxy
        proxy.AnnuaireWebService annuaire = new proxy.AnnuaireWebService ();

        proxy.cartedevisite c = annuaire.Recherche ("auteur");
    }
}

```

B.13.2 Par un client léger

A part l'architecture qui n'avait pas été respectée, nous l'avons déjà fait. Pour respecter celle-ci, nous avons deux possibilités.

Solution 1 Créer un client léger qui appelle le service précédemment créé, via le proxy généré précédemment. Voici par exemple, la partie code d'une telle page.

Pour pouvoir mettre au point le programme, l'entête de la page (onglet 'all') doit contenir la directive debug ainsi que le chargement du proxy généré :

```
<%@ Page Language="C#" Debug="true" %>
<%@ assembly Src="proxy_annuaireWS.cs" %>
```

La partie code est la suivante :

```

void Button_Click(object sender, EventArgs e) {
    // création du proxy et appel du service web
    proxy.AnnuaireWebService annuaire = new proxy.AnnuaireWebService ();
    proxy.cartedevisite carte = annuaire.Rechercher (nom.Text);

    // mise à jour des 'Label' contenu dans la page
    LabelPrenomValeur.Text = carte.Prenom;
    LabelEmailValeur.Text = carte.Email;

    // rendre visible les 'Label" de la page
    LabelPrenom.Visible = true;
    LabelPrenomValeur.Visible = true;
    LabelEmail.Visible = true;
    LabelEmailValeur.Visible = true;
}

```

La partie 'html' de la page contient un TextBox pour saisir le nom et quatre Label : deux pour afficher le texte "prénom" et "email" et deux autres pour afficher les valeurs associées.

```
...
<asp:TextBox id="nom" runat="server"></asp:TextBox>
...
<asp:Label id="LabelPrenom" runat="server" text="Prénom : " \
    visible="False"></asp:Label>
<asp:Label id="LabelPrenomValeur" runat="server" text="Label" \
    visible="False"></asp:Label>
...
<asp:Label id="LabelEmail" runat="server" text="Email : " \
    visible="False"></asp:Label>
<asp:Label id="LabelEmailValeur" runat="server" text="Label" \
    visible="False"></asp:Label>
```

Solution 2 Créer un client léger qui utilise directement la *dll* précédemment produite. Ceci peut être fait très facilement en déplaçant la *dll* dans un répertoire **bin**, puis en utilisant la directive 'import' pour charger le 'namespace' du TP :

```
<%@ import Namespace="TP" %>
```

Attention : si votre *dll* utilise des modules, il faut aussi déplacer dans ce répertoire les différents modules.

Le code est le suivant (très voisin de l'étape précédente... en l'absence de la directive import, il est aussi possible d'utiliser les noms de classe complet) :

```
void Button_Click(object sender, EventArgs e) {
    IAnnuaireService annuaire = new AnnuaireService ();
    CarteDeVisite carte = annuaire.Rechercher (nom.Text);

    LabelPrenomValeur.Text = carte.Prenom;
    LabelEmailValeur.Text = carte.Email;

    LabelPrenom.Visible = true;
    LabelPrenomValeur.Visible = true;
    LabelEmail.Visible = true;
    LabelEmailValeur.Visible = true;
}
```

B.14 Pour poursuivre le TP

- Utiliser le déploiement et la gestion de version afin de pouvoir faire évoluer le code et gérer de multiples versions de chaque composants.
- Nous avons construit très rapidement cet exemple, en particulier les cas d'erreur ne sont pas traités. Il faudrait en particulier quand des champs des cartes de visite sont vides.

- Il est possible de construire d'autres classes d'accès aux données afin d'utiliser une sauvegarde dans un fichier XML et non plus une base de données.
- Créer d'autres tables afin d'avoir par exemple pour chaque personne son compte bancaire afin d'effectuer des transferts entre comptes de manière transactionnelle.
- Mettre en place un mécanisme d'authentification afin que seule les personnes autorisées puissent accéder aux données sensibles.

Intergiciel et Construction d'Applications Réparties

©2006 D. Donsez (version du 19 janvier 2007 - 10:31)

Licence Creative Commons (<http://creativecommons.org/licenses/by-nc-nd/2.0/fr/deed.fr>)

Annexe C

OSGI en pratique

Le texte de cet atelier se trouve à l'URL :

<http://www-adele.imag.fr/users/Didier.Donsez/cours/exemplesosgi/tutorialosgi.htm>.

Annexe D

AOKell : une réalisation réflexive du modèle Fractal

L'objectif de cette annexe est d'illustrer le développement de contrôleur pour le canevas logiciel AOKell qui est une implémentation en Java du modèle de composants Fractal. Les prérequis pour suivre cet atelier sont de connaître le langage Java et d'avoir une connaissance basique du modèle de composants Fractal.

Cette annexe présente l'extension des membranes des composants Fractal primitifs avec un contrôleur permettant de journaliser (*logger*) des événements. Un nouveau contrôleur sera donc développé et enregistré dans la membrane des composants primitifs. L'interface et l'implémentation de ce contrôleur de journalisation sont identiques à celles définies dans Dream¹ qui est une bibliothèque de composants Fractal dédiée à la construction d'intergiciels orientés message.

Cette annexe débute par une présentation des outils à utiliser. Elle se poursuit par le développement du contrôleur de journalisation divisé en deux parties. La section D.2 présente le développement de ce contrôleur pour des membranes à base d'objets, tandis que la section D.3 reproduit cette démonstration pour des membranes à base de composants.

D.1 Préliminaires

D.1.1 Logiciels nécessaires

Les logiciels suivants sont nécessaires pour mettre en œuvre cet atelier :

- JDK >= 1.4
- Ant 1.6.x

Par ailleurs, il est nécessaire de se munir de l'archive aokell.zip et d'extraire les fichiers qu'elle contient. On y trouvera une version précompilée de AOKell 2.0 et la librairie de journalisation Monolog². L'archive contient deux sous-réertoires principaux : `oo` et `comp`. Le premier contient les fichiers pour la section D.2 dans laquelle nous illustrons l'écriture d'un

¹<http://dream.objectweb.org>

²<http://monolog.objectweb.org>

contrôleur pour des membranes à base d'objets, tandis que le sous-répertoire `comp` contient les fichiers pour la section D.3 dans laquelle nous illustrons l'écriture d'un contrôleur pour des membranes à base de composants.

D.1.2 AOKEll 2.0

AOKEll est une implémentation Java du modèle de composants Fractal. AOKEll respecte le niveau de conformité 3.3 ce qui fournit une compatibilité maximale avec les applications développées pour l'implémentation de référence, Julia. Deux points originaux distinguent AOKEll de Julia : le contrôle peut être développé à l'aide de composants et l'intégration du contrôle et de la partie métier des composants est réalisée à l'aide d'aspects.

Comme nous l'illustrons à la section D.3, les membranes de contrôle AOKEll peuvent être développées comme des assemblages de composants de contrôle. Ceux-ci, comme les composants applicatifs, sont des composants Fractal manipulables avec la même API. AOKEll peut ainsi être vu comme un système réflexif dans lequel des composants fournissent un niveau méta à une application elle-même développée à base de composants.

La seconde originalité d'AOKEll est d'utiliser un système de programmation par aspects pour intégrer les niveaux de contrôle et applicatif. La version 1.0 d'AOKEll utilisait pour cela AspectJ³. La version 2.0 a apporté en plus le support de Spoon⁴ est un compilateur ouvert pour le langage Java qui permet de transformer les programmes avant leur compilation. Pour cela, des mécanismes de processeurs et de gabarits (*templates*) de code ont été définis. Spoon permet de réaliser toutes sortes de transformation sur un programme source et en particulier, les transformations correspondant aux primitives des langages de programmation par tel que AspectJ. Spoon est architecturé comme un *back-end* du compilateur jdt (le compilateur Java d'IBM inclus dans Eclipse). Spoon transforme le code après que l'arbre syntaxique ait été construit par jdt. Spoon bénéficie ainsi des nombreuses optimisations qui ont été incluses dans cet outil et analyse et transforme le code Java bien plus rapidement qu'avec n'importe quel autre approche développée à l'aide d'une grammaire ad hoc et d'un analyseur syntaxique générique. Par ailleurs, les gabarits de code de Spoon ont été conçus de façon à être fortement typé ce qui permet de détecter les erreurs de transformation au plus tôt.

D.1.3 Test de l'installation

Les sous-répertoires de `oo/examples/` contiennent des exemples de programmes Java/-Fractal s'exécutant avec AOKEll. Afin de tester le bon fonctionnement de votre installation, vous pouvez vous déplacez dans un de ces sous-répertoires et taper : `ant execute`.

Nous allons illustrer cette démonstration avec l'exemple `helloworld` qui met en place l'architecture présentée figure D.1. Le composant `Client` invoque la méthode `print` fournie par l'interface `s` du composant `Serveur`. Cette méthode affiche le message transmis en paramètre. L'application peut être démarrée via l'interface `r` du composant `Client` exportée au niveau du composite.

En tapant `ant execute` dans le sous-répertoire `helloworld`, nous obtenons la trace d'exécution suivante :

³ <http://www.eclipse.org/aspectj>

⁴ <http://spoon.gforge.inria.fr>

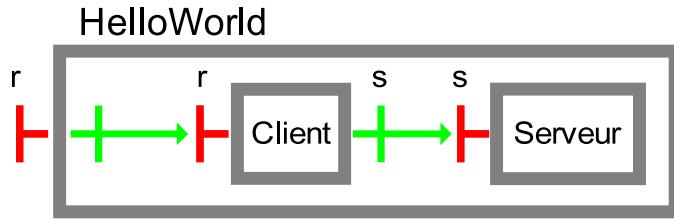


Figure D.1 – Architecture de l'exemple helloworld.

```

[java] =====
[java] === Hello World with the API ===
[java] =====
[java] CLIENT created
[java] SERVEUR created
[java] Interfaces du composant Client
[java] 0 name-controller
[java] 1 super-controller
[java] 2 component
[java] 3 r
[java] 4 binding-controller
[java] 5 lifecycle-controller
[java] 6 s
[java] Server: print method called
[java]     at cs.impl.ServerImpl.print(ServerImpl.java:31)
[java]     at aokell.generated.cs.impl.ServiceImplementedInterface.print
[java]     at cs.impl.ClientImpl.run(ClientImpl.java:33)
[java]     at aokell.generated.java.lang.RunnableImplementedInterface.run
[java]     at aokell.generated.java.lang.RunnableBoundableInterface.run
[java]     at cs.Main.main(Main.java:53)
[java]     at hw.Main.main(Main.java:33)
[java] Server: begin printing...
[java] ->hello world
[java] Server: print done.

```

Après les trois premières lignes contenant le message de bienvenue, nous trouvons les deux lignes indiquant la création des composants **Client** et **Serveur**, la liste des interfaces de contrôle du composant **Client** et l'exécution proprement dite de l'application avec l'appel de la méthode `print`. Notons en particulier, les sept interfaces du composant **Client** : cinq de contrôle et deux métier. Cette liste sera étendue par la suite avec une interface de contrôle de journalisation.

D.2 Contrôleur de journalisation orienté objet

Cette section décrit l'implémentation du contrôleur de journalisation pour des membranes à base d'objets. Nous utiliserons le sous-répertoire `oo` extrait de l'archive. Il y a deux façons de développer un contrôleur avec AOKell selon qu'il est faiblement ou fortement couplé avec la partie métier du composant.

Dans le premier cas (couplage faible), présenté section D.2.1, il n'y a pas de modification du code de la partie métier par le contrôleur qui se contente de fournir de nouvelles fonctionnalités. Il est “simplement” enregistré auprès de la membrane et on y accède via les méthodes d'introspection `getFcInterface` et `getFcInterfaces` de l'interface `Component`.

Dans le second cas (couplage fort), présenté section D.2.2, en plus d'être enregistré auprès de la membrane, le contrôleur peut modifier le comportement de la partie métier en y ajoutant du code ou en modifiant le code existant. Pour cela, il est nécessaire d'écrire un aspect.

D.2.1 Contrôleur objet faiblement couplé

Le développement d'un contrôleur objet faiblement couplé passe par les trois étapes suivantes :

1. Définition de l'interface de contrôle
2. Implémentation du contrôleur
3. Enregistrement du contrôleur

Étape 1 : Définition de l'interface de contrôle

L'interface de contrôle est le point d'entrée pour accéder à tout contrôleur. C'est une interface (au sens Fractal) serveur et externe. Elle est associée à une interface Java. Pour les besoins de cet atelier nous définissons l'interface `LoggerControllerItf` en agrégant deux interfaces existantes issues de Dream : `LoggerController` et `LoggerControllerRegister`.

L'interface `LoggerController` permet de déterminer le niveau de sévérité des messages à journaliser, tandis que l'interface `LoggerControllerRegister` permet d'enregistrer ou de retirer des producteurs d'événements à journaliser. Ceux-ci correspondent aux instances implémentant l'interface `Loggable`. Cette dernière définit une seule méthode `setLogger` qui fournit la référence du journal (instance implémentant l'interface `Logger`). En résumé, le principe de fonctionnement du contrôleur de journalisation sera le suivant :

1. Chaque composant souhaitant journaliser des événements s'enregistre auprès de son contrôleur de journalisation en utilisation la méthode `registerLogger` de l'interface `LoggerControllerRegister`. Pour cela, il est nécessaire de fournir une instance de type `Loggable`. Nous choisissons de faire en sorte que cela soit le composant lui-même qui implémente cette interface.
2. Le contrôleur de journalisation fournit un journal (instance de type `Logger`).
3. Le contrôleur de journalisation rappelle le composant via la méthode `setLogger`.
4. Le composant peut utiliser le journal pour enregistrer des événements avec la méthode `log`.

Notons que l'ensemble du processus précédent n'est en rien imposé par AOKell ni par Fractal. Il s'agit purement d'un choix de conception. De même, le fait d'agréger deux interfaces existantes pour définir l'interface de contrôle relève également d'un tel choix.

Au final, nous définissons l'interface suivante pour notre contrôleur de journalisation :

```

package org.objectweb.fractal.aokell.dream.lib.control.logger;

import org.objectweb.dream.control.logger.LoggerController;
import org.objectweb.dream.control.logger.LoggerControllerRegister;
import org.objectweb.fractal.aokell.lib.type.InterfaceTypeImpl;
import org.objectweb.fractal.api.type.InterfaceType;

/**
 * @author Lionel Seinturier <Lionel.Seinturier@lifl.fr>
 */
public interface LoggerControllerItf
    extends LoggerController, LoggerControllerRegister {

    final public static String NAME = "logger-controller";
    final public static InterfaceType TYPE =
        new InterfaceTypeImpl(
            NAME,
            LoggerControllerItf.class.getName(),
            false, false, false); }

```

Notons les deux constantes `NAME` et `TYPE` que nous ajoutons dans cette interface. Il s'agit d'une pratique qui n'a aucun caractère obligatoire mais qui permet de définir le nom et le type (au sens Fractal) associés à cette interface de contrôle.

Étape 2 : Implémentation du contrôleur

Les contrôleurs AOKell doivent implémenter l'interface `Controller` qui est utilisée par AOKell lors de l'initialisation d'une membrane de contrôle. Notre contrôleur de journalisation doit donc implémenter cette interface et son interface “métier”, c'est-à-dire `LoggerControllerItf`.

Pour l'implémentation de l'interface `LoggerControllerItf`, nous choisissons de réutiliser le code qui est fourni dans Dream pour le contrôleur de journalisation. Le fichier `BasicLoggerControllerImpl.java` du répertoire `examples/helloworld/src/org/objectweb/fractal/aokell/dream/lib/control/logger` reprend cette implémentation.

Il est nécessaire de compléter cette classe avec une implémentation de l'interface `LoggerControllerItf` qui définit les trois méthodes suivantes :

- `setFcCompCtrl` : cette méthode est utilisée par AOKell pour fournir au contrôleur la référence du contrôleur de type `Component` associé à la membrane courante.
- `initFcCtrl` : cette méthode permet d'initialiser le contrôleur. AOKell l'invoque après avoir invoqué `setFcCompCtrl` et après avoir créé tous les autres contrôleurs. Lors de l'exécution de `initFcCtrl` l'identité du contrôleur de type `Component` est donc connu et tout traitement d'initialisation nécessitant la présence des autres contrôleurs de la membrane peut être entrepris.
- `cloneFcCtrl` : cette méthode est invoquée par le contrôleur de type `Factory` pour cloner l'état du contrôleur courant. Cette méthode peut donc rester vide pour les contrôleurs sans état.

Pour résumer, l'ordre d'exécution des méthodes lors de l'initialisation d'un contrôleur AOKell est le suivant :

1. constructeur sans argument
2. méthode `setFcCompCtrl`
3. méthode `initFcCtrl`

Ouvrir le fichier `BasicLoggerControllerImpl.java`, ajouter l'interface `Controller` dans la clause `implements` et ajouter les trois méthodes suivantes :

```
public void setFcCompCtrl( Component compctrl ) {
    this.compctrl = compctrl;
}
private Component compctrl;
public void initFcCtrl() {}
public void cloneFcCtrl( Component dst, Object hints )
throws CloneCtrlException {}
```

Comme nous pouvons le constater, ce contrôleur se contente de sauvegarder la référence du contrôleur de type `Component`, ne fait aucune initialisation et est sans état.

Étape 3 : Enregistrement du contrôleur

Nous venons de créer une interface de contrôle et son implémentation. Il s'agit maintenant, soit de créer une nouvelle membrane pour accueillir ce contrôleur, soit de l'ajouter à une membrane existante. Dans les deux cas, le principe est identique. Nous illustrons le second en ajoutant le contrôleur de journalisation dans la définition des membranes primitives. Ainsi, tout nouveau composant créé avec cette membrane se verra associer en plus des cinq contrôleurs habituels (liaison, arrêt/démarrage, nommage, `Component`, accès au super-composant), un contrôleur de journalisation.

Pour cela, nous nous basons sur l'API d'AOKell qui fournit les éléments suivants :

- le singleton de la classe `Membranes` : il s'agit d'un dictionnaire des types de membranes enregistrés dans AOKell.
- la classe `ControllerDef` définit un type de contrôleur à l'aide de deux paramètres : le type (au sens Fractal) de l'interface de contrôle et la classe implémentant le contrôleur.
- la classe `MembraneDef` définit un type de membrane à l'aide de trois paramètres : un identifiant de membrane, un tableau de définitions de contrôleur (i.e. un tableau de `ControllerDef`) et un type marqueur (nous omettons pour l'instant cette notion sur laquelle nous reviendrons lors de la section D.2.2).

Nous ajoutons la définition du contrôleur de journalisation dans la définition des membranes primitives à l'aide des lignes de code suivantes. Il convient de placer ces lignes au début de la méthode `main` du fichier `Main.java` localisé dans le répertoire `oo/examples/helloworld/src/hw/`. Ce fichier est le point d'entrée de l'exemple `helloworld`.

```
Membranes membranes = Membranes.get();
MembraneDef prim = membranes.getMembraneDef("primitive");
ControllerDef cdef =
new ControllerDef(
    LoggerControllerItf.TYPE,
    BasicLoggerControllerImpl.class );
prim.registerControllerDef(cdef);
```

La première ligne récupère la référence du singleton correspondant au dictionnaire de définitions de membrane. La deuxième ligne récupère la définition de la membrane primitive. Les quatre lignes suivantes définissent un nouveau contrôleur à partir du type (au sens Fractal) défini dans l'interface `LoggerControllerItf` et de l'implémentation fournie par la classe `BasicLoggerControllerImpl`. Finalement, la dernière ligne enregistre ce contrôleur dans la définition des membranes primitives.

Tester le contrôleur de journalisation

Nous pouvons tester la nouvelle définition de la membrane primitive en relançant l'exemple `helloworld`. Une interface de contrôle supplémentaire apparaît dans la liste des interfaces implémentées par le composant `Client`.

Il s'agit, maintenant que le composant `Client` est équipé d'un contrôleur de journalisation, de l'utiliser. Nous allons donc ajouter au fichier `ClientImpl.java` localisé dans le répertoire `oo/examples/helloworld/src/cs/impl/`, les éléments suivants :

1. ajouter l'interface `Loggable` dans la clause `implements` de la classe `ClientImpl`,
2. ajouter le code de la méthode `setLogger` correspond à l'interface `Loggable` dans la classe `ClientImpl` :

```
private Logger logger;
public void setLogger(String name, Logger logger) {
    this.logger = logger;
}
```

3. enregistrer le composant `Client` auprès de son contrôleur de journalisation à l'aide du code suivant à placer en tête la méthode `run` :

```
try {
    LoggerControllerItf lci = (LoggerControllerItf)
        ((Component)this).getFcInterface("logger-controller");
    lci.register("myLogger",this);
}
catch( NoSuchInterfaceException nsie ) {
    System.err.println( nsie.getMessage() );
}
```

4. utiliser le journal, dont la référence est stockée dans la variable `logger`, pour enregistrer des événements. Nous allons par exemple, enregistrer l'appel du composant `Serveur` en ajoutant la ligne suivante juste avant l'appel à la méthode `print` :

```
logger.log(BasicLevel.INFO,"Component SERVEUR called");
```

Relancer l'exécution de l'exemple `helloworld`. La trace d'exécution suivante s'affiche :

```
[java] =====
[java] === Hello World with the API
[java] =====
[java] CLIENT created
[java] SERVER created
[java] Interfaces du composant Client
[java] 0 name-controller
[java] 1 super-controller
```

```
[java] 2 component
[java] 3 m
[java] 4 binding-controller
[java] 5 lifecycle-controller
[java] 6 logger-controller
[java] 7 s
[java] 20 août 2006 18:29:32 org.objectweb.util.monolog.wrapper.java
Log.Logger log
[java] INFO: Component SERVEUR called
[java] Server: print method called
[java]     at cs.impl.ServerImpl.print(ServerImpl.java:31)
[java]     at aokell.generated.cs.impl.ServiceImplementedInterface.print
[java]     at cs.impl.ClientImpl.run(ClientImpl.java:44)
[java]     at aokell.generated.java.lang.RunnableImplementedInterface.run
[java]     at aokell.generated.java.lang.RunnableBoundableInterface.run
[java]     at cs.Main.main(Main.java:53)
[java]     at hw.Main.main(Main.java:33)
[java] Server: begin printing...
[java] ->hello world
[java] Server: print done.
```

Comme nous le constatons, deux lignes supplémentaires apparaissent avec le message qui a été journalisé.

Bilan

Nous venons de définir un contrôleur de journalisation pour des membranes à base d'objets. Nous avons pour cela défini une interface de contrôle, une implémentation pour le contrôleur associé puis nous avons enregistré ce contrôleur dans la définition des membranes primitives. Ainsi, chaque nouveau composant primitif créé se verra équipé d'un contrôleur de journalisation.

Exercice D.1 *Plutôt que de modifier la définition de la membrane primitive, définir un nouveau type de membrane identifié par la chaîne `loggablePrimitive` contenant le contrôleur de journalisation et les contrôleurs associés aux membranes primitives.*

Exercice D.2 *Faire en sorte que les composants associés aux membranes `loggablePrimitive` soient automatiquement enregistrés auprès du contrôleur de journalisation (indice : méthode `initFcCtrl` et interface cachée `/content`).*

D.2.2 Contrôleur objet fortement couplé

Nous avons vu dans la section précédente, comment développer un contrôleur faiblement couplé pour des membranes à base d'objet. Cette section étend cette démonstration avec un contrôleur fortement couplé. Contrairement au cas précédent, un contrôleur fortement couplé peut modifier le code métier pour y ajouter de nouveaux éléments ou modifier le comportement des éléments existants. Ces modifications sont réalisées à l'aide du code dit glu. Ce code peut être développé soit avec AspectJ, soit avec Spoon. Pour les besoins

de cette démonstration, nous utilisons Spoon et allons injecter le code du contrôleur de journalisation dans le composant afin que celui-ci puisse y accéder directement.

Le développement d'un contrôleur objet fortement couplé passe par les cinq étapes suivantes :

1. *définition de l'interface de contrôle*
2. *implémentation du contrôleur*
3. définition d'une interface marqueur
4. écriture du code glu
5. *enregistrement du contrôleur*

Les étapes 1, 2 et 5 sont identiques respectivement aux étapes 1, 2 et 3 présentées précédemment. Dans la suite, nous nous focalisons sur les étapes 3 et 4 qui sont spécifiques aux contrôleurs fortement couplés.

Étape 3 : Définition d'une interface marqueur

De façon générale, une interface dite marqueur est une interface vide (i.e. sans méthode) qui qualifie les classes l'implémentant avec une certaine signification. C'est le cas par exemple de l'interface `java.io.Serializable` qui désigne en Java, les objets pouvant être transmis par copie dans un flux d'entrée/sortie.

Avec AOKell, chaque type de membrane est associé à une interface marqueur. Les composants dont on souhaite qu'ils soient fortement couplés avec leurs contrôleurs, doivent implémenter le marqueur correspondant à leur type de membrane. Le code glu (les aspects AspectJ ou les processeurs Spoon) se servent alors de la présence de ce marqueur pour modifier le code des composants ou y injecter des éléments additionnels. L'utilisation d'un marqueur est donc ce qui différencie les contrôleurs fortement couplés de ceux qui ne le sont que faiblement.

La définition du marqueur pour le contrôleur de journalisation comporte deux étapes :

1. Nous commençons par définir une interface vide `DreamLogType` comme étant le marqueur associé au contrôleur de journalisation. Par convention, nous lui attribuons un suffixe `Type`. Nous plaçons cette interface au même niveau que l'interface de contrôle et son implémentation, dans le package `org.objectweb.fractal.aokell.dream.lib.control.logger`.
2. Nous étendons le marqueur `PrimitiveType` associé aux membranes primitives avec la marqueur `DreamLogType`. Les marqueurs de membranes héritent des marqueurs associés à leur contrôleurs. Il suffit donc d'ajouter l'interface `DreamLogType` dans la clause `extends` de l'interface `PrimitiveType`.

Étape 4 : Écriture du code glu

Le code glu permet, soit d'injecter du code dans le composant, soit de modifier le code existant. Le contrôleur de cycle de vie fournit un exemple de modifications qui peuvent être entreprises : les appels sur un composant arrêté sont rejettés en ajoutant un bloc de code avant le début de chacune des méthodes métier du composant. Une telle modification n'est pas nécessaire pour le contrôleur de journalisation. Nous nous contentons d'injecter le

code de l'interface de contrôle `LoggerControllerItf` afin que le composant puisse y accéder directement sans avoir à passer par l'interface de contrôle `Component`.

Une description détaillée de Spoon est au delà de l'objectif de cet atelier. Nous nous contentons de fournir dans les figures D.2 et D.3 le code du processeur et du *template*. Le processeur transforme toutes les classes implémentant l'interface `DreamLogType`. Pour chacune d'entre elle, il insère le *template* `LoggerControllerTemplate`. Celui-ci insère les cinq méthodes implémentant l'interface `LoggerController`, les deux méthodes implémentant l'interface `LoggerControllerRegister` et la méthode `getLoggerC`.

```
package org.objectweb.fractal.aokell.glue;

import org.objectweb.fractal.aokell.dream.lib.control.logger.DreamLogType;
import org.objectweb.fractal.aokell.glue.SpoonHelper;
import spoon.processing.AbstractProcessor;
import spoon.reflect.declaration.CtClass;
import spoon.template.Substitution;
import spoon.template.Template;

/**
 * @author Lionel Seinturier <Lionel.Seinturier@lifl.fr>
 */
public class LoggerControllerProcessor
    extends AbstractProcessor<CtClass> {
    final private static Template t = new LoggerControllerTemplate();
    public void process(CtClass ct) {
        SpoonHelper.insert( ct, DreamLogType.class, t );
    }
}
```

Figure D.2 – Code du processeur Spoon pour le code glu du contrôleur de journalisation.

Comme nous pouvons le constater le code glu défini dans le *template* ne fournit pas directement le code de contrôle mais délègue sa réalisation à l'instance implémentant le contrôleur. Il s'agit d'une bonne pratique issue de la programmation par aspects qui consiste à découpler la logique d'intégration (l'aspect ou dans notre cas, le processeur et le *template* Spoon) de la réalisation concrète de la fonctionnalité à intégrer.

Tester le contrôleur de journalisation

Afin de tester le contrôleur de journalisation fortement couplé, il est nécessaire de :

1. ouvrir le fichier `glue.xml` localisé dans le répertoire `examples/helloworld/` et décommenter les deux lignes contenant les balises `templateSet` et `processor`. Ces deux lignes permettent de prendre en compte le processeur et le *template* Spoon que nous venons de présenter.
2. ouvrir le fichier `ClientImpl.java` localisé dans le répertoire `examples/helloworld/src/cs/impl` et remplacer le bloc `try/catch` de la méthode `run` (ce bloc a été introduit suite aux directives de la section D.2.1) correspondant à l'enregistrement du composant auprès de son contrôleur de journalisation par :

```
((LoggerControllerItf)this).register("myLogger",this);
```

```

package org.objectweb.fractal.aokell.glue;

import org.objectweb.dream.control.logger.Loggable;
import org.objectweb.fractal.aokell.dream.lib.control.logger.
LoggerControllerItf;
import org.objectweb.fractal.aokell.lib.control.component.ComponentItf;
import org.objectweb.fractal.aokell.lib.util.FractalHelper;
import spoon.template.Local;
import spoon.template.Template;

/**
 * @author Lionel Seinturier <Lionel.Seinturier@lifl.fr>
 */
public class LoggerControllerTemplate
    implements Template, LoggerControllerItf {

    @Local
    public LoggerControllerTemplate() {}

    @Local
    private ComponentItf _fcComp;
    private LoggerControllerItf getLoggerC() {
        return (LoggerControllerItf)
            FractalHelper.getFcInterface(_fcComp,"logger-controller"); }

    // LoggerController interface implementation
    public void setBaseName(String name) {
        getLoggerC().setBaseName(name); }
    public String getBaseName() {
        return getLoggerC().getBaseName(); }
    public int getLoggerLevel(String loggerName) {
        return getLoggerC().getLoggerLevel(loggerName); }
    public void setLoggerLevel(String loggerName, int level) {
        getLoggerC().setLoggerLevel(loggerName,level); }
    public String[] getLoggerNames() {
        return getLoggerC().getLoggerNames(); }

    // LoggerControllerRegister interface implementation
    public void register(String loggerName, Loggable loggable) {
        getLoggerC().register(loggerName,loggable); }
    public void unregister(String loggerName, Loggable loggable) {
        getLoggerC().unregister(loggerName,loggable); }
}

```

Figure D.3 – Code du *template* Spoon pour le code glu du contrôleur de journalisation.

Lancer l'exécution de l'exemple `helloworld`. La trace d'exécution est identique à celle obtenue précédemment. La différence se situe dans le fait que le composant peut être *casté* vers l'interface `LoggerControllerItf` qui a été introduite par le code `glu`. Le résultat de cette introduction peut être consulté en ouvrant le fichier `examples/helloworld/generated/glue/cs/impl/ClientImpl.java`.

Exercice D.3 *Faire en sorte que le code `glu` injecte également l'interface `Loggable`.*

D.3 Contrôleur de journalisation orienté composant

Cette section décrit l'implémentation du contrôleur de journalisation pour des membranes à base de composants. Nous utiliserons le sous-répertoire `comp` extrait de l'archive. Comme pour les membranes à base d'objet, le contrôleur peut être faiblement ou fortement couplé. Nous illustrerons le premier cas. Le couplage fort suit les mêmes principes que ceux énoncés en section D.2.2.

Le développement d'un contrôleur composant faiblement couplé passe par les quatre étapes suivantes :

1. *définition de l'interface de contrôle*
2. implémentation du contrôleur
3. définition de l'architecture de la membrane
4. enregistrement du contrôleur

La définition de l'interface de contrôle suit les mêmes principes que pour les contrôleurs objet.

Étape 2 : Implémentation du contrôleur

L'implémentation d'un contrôleur composant est identique à celle d'un composant objet. À titre d'exemple, avec AOKEll, le code des contrôleurs de base (liaison, cycle de vie, nommage, etc.) est identique que la membrane soit à base d'objets ou de composants. Deux différences existent néanmoins au niveau de la gestion des liens entre contrôleurs et de la sémantique de l'interface `Controller`.

La première différence entre contrôleurs objet et composant concerne la gestion des liens entre contrôleurs. Dans le cas des contrôleurs objet, les liens sont de simples références enregistrées dans le contrôleur `Component` : un contrôleur devant interagir avec un de ses pairs, récupère l'interface de contrôle correspondante auprès du contrôleur `Component`. Dans le cas des contrôleurs composant, les liens sont réifiés au sein d'une architecture. Les interactions entre contrôleurs doivent donc suivre les liens définis par cette architecture.

À titre d'exemple, les figures D.4 et D.5 illustrent l'architecture d'une membrane primitive et sa description en Fractal ADL. Le contrôleur de liaison (BC) peut interagir avec le contrôleur `Component` (Comp) avec lequel il a une liaison, mais pas avec le contrôleur de nommage (NC). Pour que cela soit possible, il est nécessaire de modifier cette architecture (statiquement ou dynamiquement) pour prendre en compte cette nouvelle dépendance.

La seconde différence entre contrôleurs objet et contrôleurs composant concerne la sémantique de l'interface `Controller`. Comme nous l'avons vu à l'étape 2 de la section D.2.1, l'interface `Controller` doit être implémentée par les composants. Au sein de

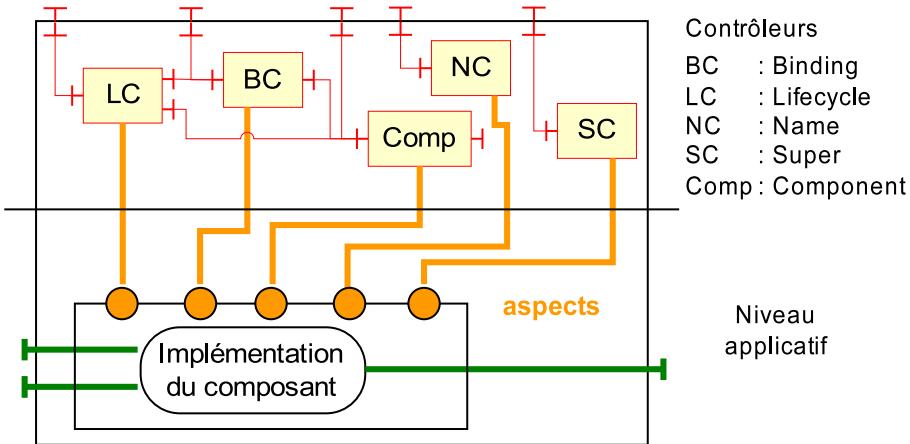


Figure D.4 – Membrane de contrôle pour les composants primitifs.

cette interface, la sémantique du paramètre de la méthode `setFcCompCtrl` diffère entre les membranes objet et les membranes composant. Bien que dans les deux cas, le type de ce paramètre soit `Component`, dans le premier cas il s'agit de la référence du contrôleur `Component` de la membrane, tandis que dans le second, il s'agit du contrôleur `Component` du composant de contrôle.

En effet, avec AOKell, les contrôleurs composants sont des composants Fractal. Ils sont donc équipés comme tous les composants Fractal, d'une membrane. Deux types de membranes pour les composants de contrôle sont définies dans AOKell : `mPrimitive` pour les composants de contrôle primitif et `mComposite` pour les composants de contrôle composite. Ces deux membranes fournissent un ensemble de contrôleurs qui gèrent le composant de contrôle. Parmi ces contrôleurs, on trouve notamment un contrôleur `Component`. Sa référence est transmise au composant de contrôle lors de l'appel de la méthode `setFcCompCtrl`.

Étape 3 : Définition de l'architecture de la membrane

Comme pour le contrôleur de journalisation objet, nous pouvons soit créer une nouvelle membrane pour y insérer le contrôleur composant de journalisation, soit l'insérer dans une membrane existante. Nous choisissons la seconde solution, et l'insérons dans la membrane primitive.

La description Fractal ADL de la figure D.5 est disponible dans le fichier `Primitive.fractal` du répertoire `features/membrane/comp/src/org/objectweb/fractal/aokell/lib/membrane/primitive/`. Nous allons y ajouter la définition du contrôleur de journalisation. Pour cela, il est nécessaire de :

1. définir le composant de journalisation en Fractal ADL. Nous allons pour cela suivre une bonne pratique qui consiste à séparer la définition du type de composant, de la définition de son implémentation. Dans le répertoire, `features/membrane/comp/src/org/objectweb/fractal/aokell/dream/lib/control/logger/`, nous définissons donc un fichier `LoggerControllerType.fractal` qui contient :

```
<definition name="org.objectweb.fractal.aokell.lib.membrane.primitive.Primitive">

    <!-- Composants de contrôle inclus dans une membrane primitive -->
    <component name="Comp" definition="ComponentController"/>
    <component name="NC" definition="NameController"/>
    <component name="LC" definition="NonCompositeLifeCycleController"/>
    <component name="BC" definition="PrimitiveBindingController"/>
    <component name="SC" definition="SuperController"/>

    <!-- Export des interfaces de contrôle -->
    <binding client="this://component" server="Comp://component"/>
    <binding client="this://name-controller" server="NC://name-controller"/>
    <binding client="this://lifecycle-controller"
            server="LC://lifecycle-controller"/>
    <binding client="this://binding-controller" server="BC://binding-controller"/>
    <binding client="this://super-controller" server="SC://super-controller"/>

    <!-- Liaisons entre composants de contrôle -->
    <binding client="BC://component" server="Comp://component"/>
    <binding client="LC://binding-controller" server="//BC.binding-controller"/>
    <binding client="LC://component" server="Comp://component"/>

    <controller desc="mComposite"/>
</definition>
```

Figure D.5 – Définition de l’architecture des membranes de contrôle pour les composants primitifs.

```
<definition name="org.objectweb.fractal.aokell.lib.control.logger.
LoggerControllerType">
    <interface
        name="//logger-controller"
        signature="org.objectweb.fractal.aokell.lib.control.logger.
                    LoggerControllerItf"
        role="server" />
</definition>
```

Le type du composant comprend une seule interface serveur de signature `LoggerControllerItf` et de nom `//logger-controller`. Nous pouvons maintenant définir son implémentation dans le fichier `LoggerController.fractal` en héritant de la définition de son type :

```
<definition name="org.objectweb.fractal.aokell.lib.control.logger.
LoggerController"
    extends="org.objectweb.fractal.aokell.dream.lib.control.logger.
LoggerControllerType" >
    <content class="org.objectweb.fractal.aokell.lib.control.logger.
BasicLoggerControllerImpl" />
    <controller desc="mPrimitive" />
</definition>
```

Le composant est implémenté par la classe `BasicLoggerControllerImpl` et est associé à une membrane `mPrimitive`.

2. insérer la définition du composant de journalisation dans le composite primitif à l'aide de la déclaration suivante :

```
<component
    name="LogC"
    definition="org.objectweb.fractal.aokell.lib.control.logger.
LoggerController" />
```

3. exporter l'interface de contrôle du composant de journalisation au niveau du composite :

```
<binding client="this://logger-controller"
server="LogC://logger-controller" />
```

4. faire en sorte que le type du composite soit étendu avec le type du composant de journalisation. Pour cela, il est nécessaire d'ajouter `org.objectweb.fractal.aokell.lib.control.logger.LoggerControllerType` dans la clause `extends` du fichier `Primitive.fractal`.

Étape 4 : Enregistrement du contrôleur

Dans notre cas, ayant choisi de modifier la membrane primitive existante, cette étape n'a pas lieu d'être.

Si par contre, le choix de définir une nouvelle membrane avait été fait, il aurait été nécessaire de modifier le fichier `AOKellMembranes.java` localisé dans le répertoire `features/comp/membrane/src/org/objectweb/fractal/aokell/`.

Tester le contrôleur de journalisation

Afin de tester le composant contrôleur de journalisation, il est nécessaire de :

1. Compiler et générer les exemples d'AOKell à l'aide de la commande `ant examples` à lancer dans le répertoire racine.
2. Se déplacer dans le répertoire `output/dist/examples/helloworld/`.
3. Lancer l'exécution de l'exemple à l'aide de la commande `ant execute`.

Comme dans les cas précédents, la trace d'exécution fait apparaître une nouvelle interface de contrôle implémentée par le composant `Client`.

Bilan

Nous venons de définir un contrôleur de journalisation pour des membranes à base de composants. La déclaration de ce nouveau contrôleur a été faite statiquement en modifiant la description Fractal ADL du composite associé aux membranes primitives. Tout nouveau composant primitif créé sera équipé de ce contrôleur.

Au delà de cette modification statique, AOKell permet d'introspecter dynamiquement une membrane existante pour en modifier l'architecture. Il aurait donc été envisageable d'y ajouter une instance du contrôleur de journalisation. Néanmoins, la difficulté d'une telle manipulation réside dans le type du composite correspondant à la membrane primitive. Comme nous l'avons vu, ce composite doit exporter toutes les interfaces de contrôle de la membrane. Donc, ajouter une nouvelle interface de contrôle dynamiquement revient à faire

évoluer dynamiquement le type de ce composite. À notre connaissance, la mutabilité des types de composants est un domaine encore peu étudié dans le cadre du modèle Fractal et qui nécessite de plus amples développements tant au niveau théorique qu'au niveau de la mise en œuvre dans des canevas logiciel comme Julia ou AOKell.

Annexe E

Joram : un intergiciel de communication asynchrone

Ce chapitre décrit les principes de conception et de réalisation de JORAM (*Java Open Reliable Asynchronous Messaging*), un intergiciel de communication asynchrone qui met en œuvre la spécification JMS (*Java Messaging Service*) pour la communication par messages entre applications Java.

La section 1 est un bref rappel sur les intergiciels à messages. La section 2 décrit les traits principaux du modèle de programmation défini par JMS. La section 3 présente les principes directeurs de conception de l'architecture interne de JORAM pour répondre aux objectifs d'une mise en œuvre distribuée et configurable. La section 4 décrit quelques fonctions avancées du système JORAM, pour la répartition de charge, la haute disponibilité et l'interopérabilité avec le monde extérieur. La section 5 présente rapidement le système d'agents distribué qui constitue la base technologique pour la mise en œuvre de JORAM. La section 6 conclut sur les utilisations de JORAM et ses perspectives d'évolution.

E.1 Introduction

Dans cette section, nous présentons les principales caractéristiques des systèmes à messages et les motivations pour leur usage, avant une brève introduction à JMS et à Joram, sujets qui seront développés dans les sections suivantes.

E.1.1 Les intergiciels à messages (MOM)

Les systèmes de communication asynchrones, fondés sur l'envoi de messages, connaissent aujourd'hui un regain d'intérêt dans le contexte des applications réparties sur Internet. En effet, on s'accorde à penser que les modèles de communication asynchrones sont mieux adaptés que les modèles synchrones (de type client-serveur) pour gérer les interactions entre systèmes faiblement couplés. Le couplage faible résulte de plusieurs facteurs de nature spatiale ou temporelle : l'éloignement géographique des entités commu-

nicantes, la possibilité de déconnexion temporaire d'un partenaire en raison d'une panne ou d'une interruption de la communication (pour les usages mobiles par exemple). Les modèles de communication asynchrones prennent en compte l'indépendance entre entités communicantes et sont donc mieux armés pour traiter ces problèmes. Aujourd'hui les systèmes de communication asynchrones, appelés MOM (*Message Oriented Middleware*), sont très largement répandus dans la mesure où ils constituent la base technologique pour la réalisation des classes d'applications suivantes :

- Intégration de données et intégration d'applications (EAI, B2B, ESB, etc.)
- Systèmes ubiquitaires et usages de la mobilité.
- Surveillance et contrôle d'équipements en réseau.

Du point de vue du modèle de communication les intergiciels à messages partagent les concepts suivants :

- Les entités communicantes sont découplées. L'émission (on dit aussi la production) d'un message est une opération non bloquante. D'autre part émetteur (producteur) et récepteur (consommateur) ne communiquent pas directement entre eux mais utilisent un objet de communication intermédiaire (boîte aux lettres).
- Deux modèles de communication sont fournis : un modèle point à point dans lequel un message produit est consommé par un destinataire unique ; un modèle multipoints dans lequel un message peut être adressé à une communauté de destinataires (communication de groupe).
- Le mode multipoints est généralement doublé d'un système de désignation associative dans lequel les destinataires du message sont identifiés par une propriété du message. Ce mode de communication est à l'origine du modèle Publication/Abonnement (*Publish/Subscribe*) largement répandu aujourd'hui.

Pendant de nombreuses années l'essor des systèmes de communication asynchrones a été retardé par l'absence de normalisation (au contraire de ce qui s'est passé pour les systèmes client-serveur avec CORBA). Les intergiciels à messages sont donc restés propriétaires à la fois du point de vue du modèle de programmation et de leur implémentation. A la fin des années 90, l'émergence de la spécification JMS (*Java Messaging Service*) dans le monde Java a partiellement comblé ce handicap, partiellement seulement car, comme nous le verrons plus loin, la norme définit le modèle de programmation et l'API correspondante. En revanche il n'y a toujours pas de tentative de normalisation sur l'intergiciel lui-même, ni même sur les mécanismes d'interopérabilité comme c'est le cas dans CORBA avec le protocole IIOP.

E.1.2 JMS (*Java Messaging Service*)

JMS (*Java Messaging Service*) est la spécification d'un service de messagerie en Java. Plus précisément JMS décrit l'interface de programmation pour utiliser un bus à messages asynchrone, c'est-à-dire la manière de programmer les échanges entre deux composants applicatifs.

Dans la structure d'une application JMS, on distingue les composants suivants (voir Figure 1).

- Le système de messagerie JMS (*JMS provider*). Dans la mise en œuvre du système de messagerie on distingue le service de base qui met en œuvre les abstractions du

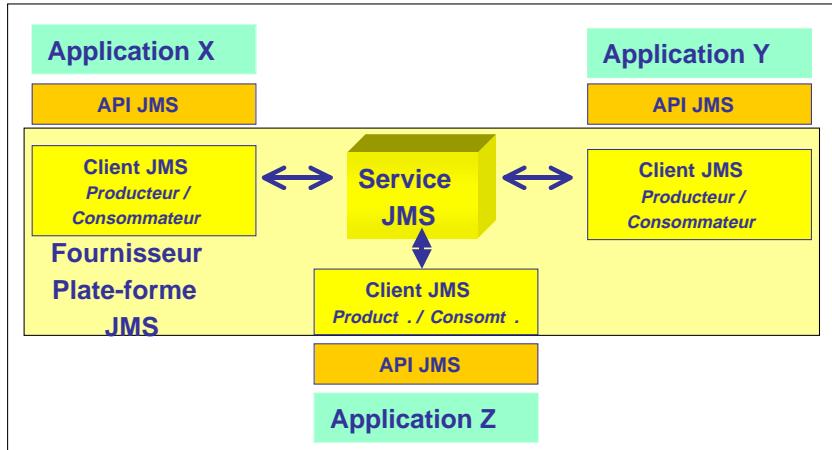


Figure E.1 – Application JMS

modèle de programmation JMS et une bibliothèque de fonctions liée aux programmes utilisateurs qui met en œuvre l’interface de programmation JMS.

- Les clients JMS (*JMS client*) sont les programmes (applications), écrits en langage Java, qui produisent (émettent) et consomment (reçoivent) des messages selon des protocoles spécifiés par l’API JMS.
- Les messages JMS sont des objets qui permettent de véhiculer des informations entre des clients JMS. Différents types de messages sont utilisables : texte structuré (par exemple un fichier XML), format binaire, objets Java (sérialisés), etc.

Deux modes de communication (*Messaging Domains*) sont fournis par la spécification JMS : La communication point à point est fondée sur des files de message (*Queues*). Un message est adressé à une queue par un client producteur (flot noté 1) d’où il est extrait par un client consommateur. Un message est consommé par un seul client. Le message est stocké dans la queue jusqu’à sa consommation ou jusqu’à l’expiration d’un délai d’existence. La consommation d’un message peut être synchrone (retrait explicite par le consommateur – flots 2 et 3) ou asynchrone (activation d’une procédure de veille préenregistrée chez le consommateur - non représenté dans la figure 2) par le gestionnaire de messages. La consommation du message est confirmée par un accusé de réception généré par le système ou le client.

La communication multipoints est fondée sur le modèle publication/abonnement (*Publish/Subscribe*) - voir figure 3. Un client producteur émet un message concernant un sujet prédéterminé (*Topic*) (flot noté 2 dans la figure 3). Tous les clients préalablement abonnés à ce *Topic* (flots 1a et 1b) reçoivent le message correspondant (flots 3a et 3b). Le modèle de consommation (implicite/explicite) est identique à celui du mode point à point.

La dernière version de la spécification JMS (dénotée JMS 1.1) unifie la manipulation des deux modes de communication au niveau du client JMS en introduisant la notion de **Destination**¹ pour représenter soit une queue de message, soit un *Topic*. Cette unification simplifie l’API (les primitives de production/consommation sont syntaxiquement fusionnées) et permet une optimisation des ressources de communication. Elle permet

¹Dans la suite ce terme « destination » sera utilisé pour faire référence indistinctement à une queue ou à un *Topic*.

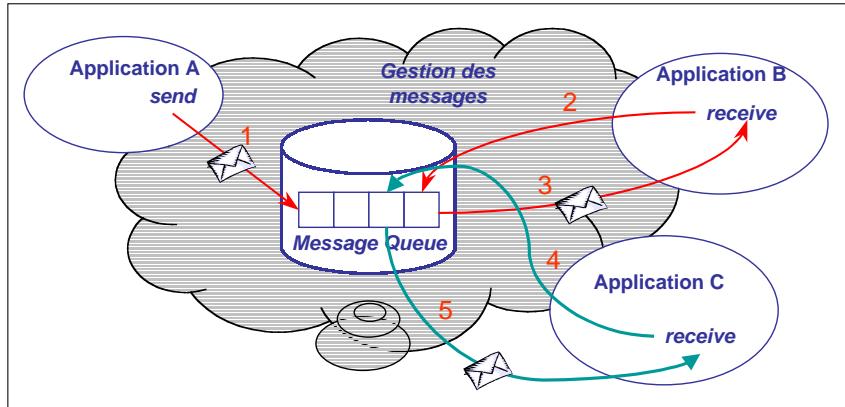


Figure E.2 – Modèle de communication Point à Point

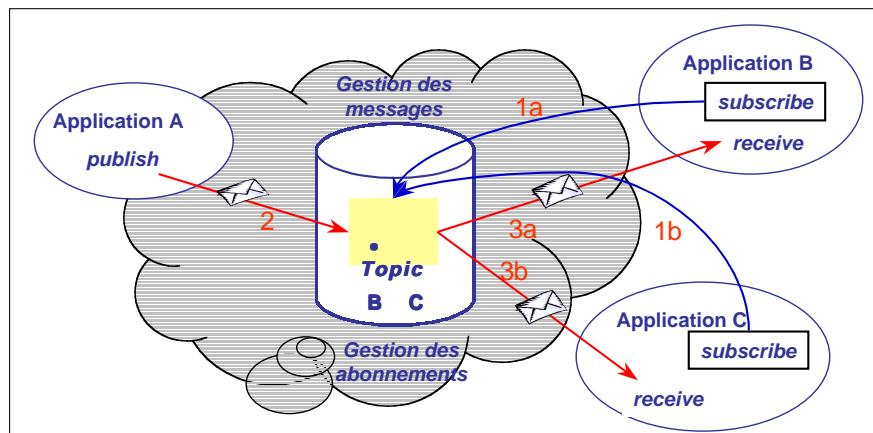


Figure E.3 – Modèle de communication Publish/Subscribe

également d'utiliser le mode transactionnel dans les deux modèles de communication. Bien entendu cela ne remet pas en cause la sémantique propre à chacun de ces modes de communication, qui doit être prise en compte dans la programmation du client JMS. Enfin, la spécification JMS définit des options de qualité de service, en particulier pour ce qui concerne les abonnements (temporaires ou durables) et la garantie de délivrance des messages (propriété de persistance).

E.1.3 Ce que JMS ne dit pas et ne fait pas

JMS définit un protocole d'échange entre des producteurs et des consommateurs de messages. JMS ne donne aucune indication sur la mise en œuvre du service de messagerie qui est donc une implantation propriétaire chez tous les fournisseurs de plates-formes JMS. JMS définissant l'API d'accès au service de messagerie, une application JMS est supposée être indépendante d'une plate-forme JMS. Cette indépendance répond à un objectif de portabilité. Dans la réalité la portabilité est réduite du fait que les fonctions d'administration sont propres à chaque plate-forme. Par ailleurs l'interopérabilité entre deux clients JMS implantés sur des plates-formes différentes n'est pas garantie car elle requiert l'existence d'une passerelle spécifique pour mettre en correspondance les mécanismes des deux plates-formes. Cette fonction est disponible dans certaines offres du marché pour un nombre restreint de plates-formes. JORAM fournit également cette fonction (voir 4.3). La spécification JMS n'est pas complète. Certaines fonctions essentielles au fonctionnement d'une plate-forme JMS ne sont pas décrites dans la spécification et font donc l'objet d'implantations propriétaires. C'est le cas en particulier pour la configuration et l'administration du service de messagerie, pour la sécurité (intégrité et confidentialité des messages) et pour certains paramètres de qualité de service. À l'opposé, la plupart des plates-formes proposent des fonctions additionnelles qui se présentent comme des valeurs ajoutées aux offres concurrentes (par exemple les *Topics* hiérarchisés, des fonctions de sécurité et des mécanismes de haute disponibilité, etc.). Sur ces deux points JORAM n'échappe pas à la règle. En ce qui concerne l'administration, la spécification JMS est limitée à la définition d'un certain nombre d'objets d'administration qui sont gérés par le service de messagerie et qui sont utilisés par les clients JMS au travers de fonctions spécifiques de l'API JMS.

E.1.4 Joram : une mise en œuvre d'un service de messagerie JMS

JORAM (*Java Open Reliable Asynchronous Messaging*) est une implémentation 100% Java de la spécification JMS. JORAM est conforme aux dernières spécifications JMS 1.1 (elles-mêmes étant partie intégrante de la spécification J2EE 1.4). JORAM est un composant *open source* disponible sur la base de code ObjectWeb sous licence LGPL - <http://joram.objectweb.org>. JORAM est aujourd'hui en exploitation dans de nombreux environnements opérationnels où il est utilisé de deux façons complémentaires :

- Comme un système de messagerie Java autonome entre des applications JMS développées pour des environnements variés (de J2EE à J2ME).
- Comme un composant de messagerie intégré dans un serveur d'application J2EE. A ce titre, JORAM est une brique essentielle du serveur J2EE JOnAS, également disponible sur ObjectWeb - <http://jonas.objectweb.org>

Comme toutes les autres réalisations de plates-formes JMS, JORAM est structuré en deux parties : une partie «serveur JORAM» qui gère les objets JMS (Queues, Topics , connexions, etc.) et une partie « client JORAM » qui est liée à l'application cliente JMS. Comme nous le verrons plus loin en détaillant l'architecture de JORAM, le serveur JORAM peut être mis en œuvre de façon centralisée ou de façon distribuée. La communication entre un client JORAM et un serveur JORAM s'appuie sur le protocole TCP/IP. Une variante, présentée plus loin, consiste à utiliser le protocole HTTP/SOAP pour les clients JMS développés dans un environnement J2ME. La communication entre deux serveurs JORAM peut utiliser différents types de protocoles selon les besoins (TCP/IP, HTTP, SOAP, communication sécurisée via SSL). Client et serveurs peuvent être sur des machines physiques différentes ; ils peuvent partager la même machine et s'exécuter dans des processus différents ou partager le même processus.

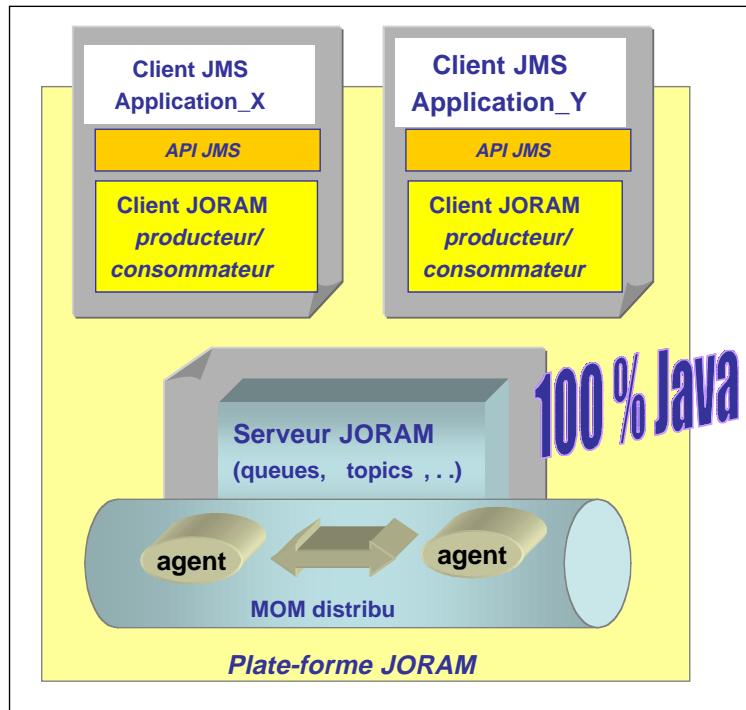


Figure E.4 – La plate-forme JORAM

En résumé, JORAM est une plate-forme JMS disponible en *open source*, dont la réalisation s'appuie sur un intergiciel à messages qui exploite une technologie d'agents distribués, présentée brièvement en section 4. La structure générale de la plate-forme JORAM est représentée sur la figure 4.

E.2 Le modèle de programmation JMS

Cette section présente les principes directeurs du modèle de programmation JMS. La description détaillée de l'API JMS n'est pas un objectif de ce papier (à cet effet le lecteur pourra se référer au tutoriel disponible en ligne). Nous exposons ici uniquement les éléments

nécessaires à la compréhension des aspects architecturaux développés plus loin.

E.2.1 Les abstractions de JMS

JMS définit un ensemble de concepts communs aux deux modèles de communication Point-à-Point et *Publish/Subscribe* :

- **ConnectionFactory** : un objet d'administration utilisé par le client JMS pour créer une connexion avec le système de messagerie.
- **Connection** : une connexion active avec le système de messagerie.
- **Destination** : l'objet de communication entre deux clients JMS. Cet objet désigne la destination des messages pour un producteur et la source des messages attendus pour un consommateur. La destination désigne une **Queue** de messages dans le modèle de communication point à point et un **Topic** dans le modèle de communication *Publish/Subscribe*.
- **Session** : un contexte (mono-thread) pour émettre et recevoir des messages.
- **MessageProducer** : un objet créé par une session et utilisé pour émettre des messages à un objet **Destination**.
- **MessageConsumer** : un objet créé par une session et utilisé pour recevoir les messages déposés dans un objet **Destination**.

Ces objets se déclinent selon le modèle de communication choisi, comme le montre le tableau de la figure 5.

<i>Interface "parent"</i>	<i>Point- -point</i>	<i>Publish/Subscribe</i>
Destination	Queue	Topic
ConnectionFactory	QueueConnectionFactory	TopicConnectionFactory
Connection	QueueConnection	TopicConnection
Session	QueueSession	TopicSession
MessageProducer	QueueSender	TopicPublisher
MessageConsumer	QueueReceiver	TopicSubscriber

Figure E.5 – Correspondance des interfaces *Point à Point* et *Publish/Subscribe*

E.2.2 Principe de fonctionnement

Un client JMS exécute la séquence d'opérations suivante :

- Il recherche un objet **ConnectionFactory** dans un répertoire en utilisant l'API JNDI (*Java Naming and Directory Interface*)
- Il utilise l'objet **ConnectionFactory** pour créer une connexion JMS, il obtient alors un objet **Connection**.
- Il utilise l'objet **Connection** pour créer une ou plusieurs sessions JMS, il obtient alors des objets **Session**.
- Il utilise le répertoire pour trouver un ou plusieurs objets **Destination**.
- Il peut alors, à partir d'un objet **Session** et des objets **Destination**, créer les objets **MessageProducer** et **MessageConsumer** pour émettre et recevoir des messages.

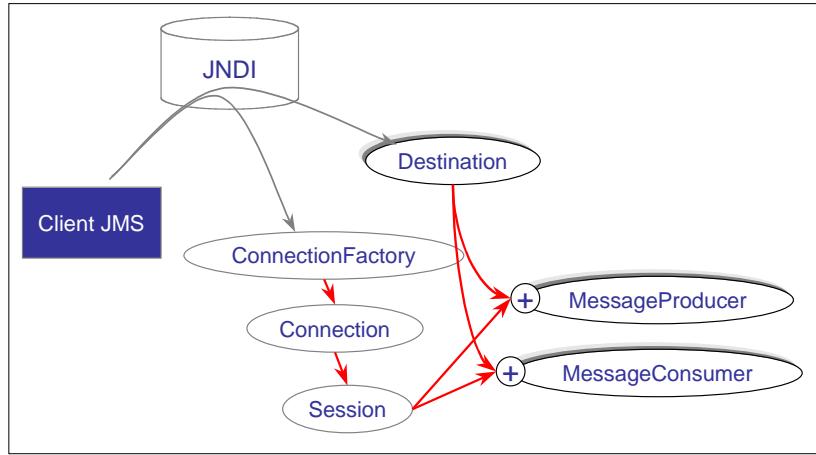


Figure E.6 – Architecture d'une application JMS

Les étapes et les objets caractérisant une application JMS sont résumés dans la figure 6.

E.2.3 Messages

Les systèmes de messagerie manipulent les messages comme des entités comportant un en-tête et un corps. L'en-tête contient l'information utile pour l'identification et le routage du message ; le corps contient les données utiles à l'application. Le modèle de message JMS répond aux critères suivants :

- Il fournit une interface de message unique.
- Il supporte les messages contenant du texte, des objets Java ou des données XML.

Structure des messages

Un message JMS est composé de trois parties :

- Un en-tête : qui contient l'ensemble des données utilisées à la fois par le client et le système pour l'identification et l'acheminement du message.
- Des propriétés : en plus des champs standard de l'en-tête, les messages permettent d'ajouter des champs optionnels sous forme de propriétés normalisées, ou de propriétés spécifiques à l'application ou au système de messagerie.
- Un corps : JMS définit différents types de corps de message afin de répondre à l'hétérogénéité des systèmes de messagerie.

En-tête des messages

Un en-tête de message JMS est composé des champs suivants :

- **JMSDestination** : ce champs contient la destination du message, il est fixé par la méthode d'envoi de message en fonction de l'objet **Destination** spécifié.
- **JMSDeliveryMode** : ce champ définit le mode de délivrance du message (persistant

ou non²) ; il est fixé par la méthode d'envoi de message en fonction des paramètres spécifiés.

- **JMSMessageId** : ce champs contient un identificateur qui identifie de manière unique chaque message envoyé par un système de messagerie. Il est fixé par la méthode d'envoi de message et peut-être consulté après envoi par l'émetteur.
- **JMSTimeStamp** : ce champs contient l'heure de prise en compte du message par le système de messagerie, il est fixé par la méthode d'envoi de message.
- **JMSReplyTo** : ce champs contient la **Destination** à laquelle le client peut éventuellement émettre une réponse. Il est fixé par le client dans le message.
- **JMSExpiration** : ce champs est calculé comme la somme de l'heure courante (GMT) et de la durée de vie d'un message (time-to-live). Lorsqu'un message n'est pas délivrée avant sa date d'expiration il est détruit ; aucune notification n'est définie pour prévenir de l'expiration d'un message.
- **JMSCorrelationId**, **JMSPriority**, **JMSRedelivered**, **JMSType**, etc.

Propriétés

Les propriétés permettent à un client JMS de sélectionner les messages en fonction de critères applicatifs. Un nom de propriété doit être de type **String** ; une valeur peut être : **null**, **boolean**, **byte**, **short**, **int**, **long**, **float**, **double** et **String**. Un client peut ainsi définir des filtres en réception dans l'objet **MessageConsumer** à l'aide d'une chaîne de caractère dont la syntaxe est basée sur un sous-ensemble de la syntaxe d'expression de conditions du langage SQL.

Corps du message

JMS définit cinq formes de corps de messages :

- **StreamMessage** : un message dont le corps contient un flot de valeurs de types primaires Java ; il est rempli et lu séquentiellement.
- **MapMessage** : un message dont le corps est composé d'un ensemble de couples noms-valeurs.
- **TextMessage** : un message dont le corps est une chaîne de caractère (**String**).
- **ObjectMessage** : un message dont le corps contient un objet Java sérialisable.
- **BytesMessage** : un message dont le corps est composé d'un flot d'octets ; ce type de message permet de coder un message conformément à une application existante.

E.2.4 Objets JMS

Objets d'administration

JMS ne définit pas une syntaxe d'adressage standard. En lieu et place, il définit l'objet **Destination** qui encapsule les différents formats d'adresses des systèmes de messagerie. L'objet **ConnectionFactory** encapsule l'ensemble des paramètres de configuration définit par l'administrateur, il est utilisé par le client pour initialiser une connexion avec le système de messagerie.

²Le mode non persistant correspond à une sémantique au plus une fois, tandis que le mode persistant correspond à une sémantique exactement une fois : le message ne être ni perdu ni dupliqué.

Objet Connection

Un objet **Connection** représente une connexion active avec le système de messagerie. Cet objet supporte le parallélisme, c'est-à-dire qu'il fournit plusieurs voies de communication logiques entre le client et le serveur. Lors de sa création, le client peut devoir s'authentifier. L'objet **Connection** permet de créer une ou plusieurs sessions. Lors de sa création une connexion est dans l'état stoppé³. Elle doit être explicitement démarrée pour recevoir des messages. Une connexion peut être temporairement stoppée puis redémarrée ; après son arrêt, il peut y avoir encore quelques messages délivrés ; arrêter une connexion n'affecte pas sa capacité à transmettre des messages. Du fait qu'une connexion nécessite l'allocation de nombreuses ressources dans le système de messagerie, il est recommandé de la fermer lorsqu'elle n'est plus utile.

Objet Session

Un objet JMS **Session** est un contexte *mono-thread* pour produire et consommer des messages. Il répond à plusieurs besoins :

- Il construit les objets **MessageProducer** et **MessageConsumer**.
- Il crée les objets **Destination** et **Message**.
- Il supporte les transactions et permet de grouper plusieurs opérations de réception et d'émission dans une unité atomique qui peut être validée (resp. invalidée) par la méthode **Commit** (resp. **Rollback**).
- Il réalise l'ordonnancement des messages reçus et envoyés.
- Il gère l'acquittement des messages.

Bien qu'une session permette la création de multiples objets **MessageProducer** et **MessageConsumer**, ils ne doivent être utilisés que par un flot d'exécution à la fois (contexte mono-threadé). Pour accroître le parallélisme un client JMS peut créer plusieurs sessions, chaque session étant indépendante. Dans le mode *Publish/Subscribe*, si deux sessions s'abonnent à un même sujet, chaque client abonné (**TopicSubscriber**) reçoit tous les messages émis sur le **Topic**. Du fait qu'une session nécessite l'allocation de ressources dans le système de messagerie, il est recommandé de la fermer lorsqu'elle n'est plus utilisée.

Objet MessageConsumer

Un client utilise un objet **MessageConsumer** pour recevoir les messages envoyés à une destination particulière. Un objet **MessageConsumer** est créé en appelant la méthode **CreateConsumer** de l'objet **Session** avec un objet **Destination** en paramètre. Un objet **MessageConsumer** peut être créé avec un sélecteur de message pour filtrer les messages à consommer. JMS propose deux modèles de consommation (synchrone et synchrone) pour les messages. Dans le modèle synchrone un client demande le prochain message en utilisant la méthode **Receive** de l'objet **MessageConsumer** (mode de consommation *Pull*). Dans le mode asynchrone il enregistre au préalable un objet qui implémente la classe **MessageListener** dans l'objet **MessageConsumer** ; les messages sont alors délivrés lors de leur arrivée par appel de la méthode **onMessage** sur cet objet (mode de consommation *Push*).

³afin de ne pas perturber l'initialisation de l'application par l'arrivée de messages.

Objet MessageProducer

Un client utilise un objet **MessageProducer** pour envoyer des messages à une destination. Un objet **MessageProducer** est créé appelant la méthode **CreateProducer** de l'objet **Session** avec un objet **Destination** en paramètre. Si aucune destination n'est spécifiée un objet **Destination** doit être passé à chaque envoi de message (en paramètre de la méthode **Send**). Un client peut spécifier le mode de délivrance, la priorité et la durée de vie par défaut pour l'ensemble des messages envoyés par un objet **MessageProducer**. Il peut aussi les spécifier pour chaque message.

E.2.5 Communication en mode Point à point

Un objet **Queue** encapsule un nom de file de message du système de messagerie. Rappelons que JMS ne définit pas de fonctions pour créer, administrer ou détruire des queues. Dans les faits, ces fonctions sont réalisées par des outils d'administration propres à chaque plate-forme JMS. Un client utilise un objet **QueueConnectionFactory** pour créer une connexion active (objet **QueueConnection**) avec le système de messagerie. L'objet **QueueConnection** permet au client d'ouvrir une ou plusieurs sessions (objets **QueueSession**) pour envoyer et recevoir des messages. L'interface des objets **QueueSession** fournit les méthodes permettant de créer les objets **QueueReceiver**, **QueueSender** et **QueueBrowser**.

E.2.6 Communication en mode *Publish/Subscribe*

Le modèle JMS *Publish/Subscribe* définit comment un client JMS publie vers et s'abonne à un noeud dans une hiérarchie de sujets. JMS nomme ces nœuds des **Topics**. Un objet **Topic** encapsule un nom de sujet du système de messagerie. JMS ne met aucune restriction sur la sémantique d'un objet **Topic**, il peut s'agir d'une feuille, ou d'une partie plus importante de la hiérarchie⁴. Un client utilise un objet **TopicConnectionFactory** pour créer un objet **TopicConnection** qui représente une connexion active avec le système de messagerie. Un objet **TopicConnection** permet de créer une ou plusieurs sessions (objet **TopicSession**) pour produire et consommer des messages. L'interface des objets **TopicSession** fournit les méthodes permettant de créer les objets **TopicPublisher** et **TopicSubscriber**. Un client utilise un objet **TopicPublisher** pour publier des messages concernant un sujet donné. Il utilise un objet **TopicSubscriber** pour recevoir les messages publiés sur un sujet donné. Les objets **TopicSubscriber** classiques ne reçoivent que les messages publiés pendant qu'ils sont actifs. Les messages filtrés par un sélecteur de message ne seront jamais délivrés ; dans certains cas une connexion peut à la fois émettre et recevoir sur un sujet. L'attribut **NoLocal** sur un objet **TopicSubscriber** permet de ne pas recevoir les messages émis par sa propre connexion.

E.2.7 JMS par l'exemple

Cette section présente quelques exemples de séquences de code associées aux principales opérations de l'API JMS.

⁴Pour s'abonner à une large classe de sujets par exemple.

Initialisation d'une session "Point-to-Point"

```

// We need a pre-configured ConnectionFactory.
// Get it by looking it up using JNDI.
Context messaging = new InitialContext();
QueueConnectionFactory connectionFactory =
        (QueueConnectionFactory) messaging.lookup(" &");

// Gets Destination objects using JNDI.
Queue queue1 = (Queue) messaging.lookup(" &");
Queue queue2 = (Queue) messaging.lookup(" &");

// Creates QueueConnection, then use it to create a session.
QueueConnection connection = connectionFactory.createQueueConnection();
QueueSession session = connection.createQueueSession(
        false, // not transacted
        Session.AUTO_ACKNOWLEDGE); // acknowledge on receipt

// Getting a QueueSender and a QueueReceiver.
QueueSender sender = session.createSender(queue1);
QueueReceiver receiver = session.createReceiver(queue2);

// Getting a QueueReceiver with a selector message.
Queue queue3 = (Queue) messaging.lookup(" &");
String selector = new String("(name = 'Bull') or (name = 'IBM'))");
QueueReceiver receiver2 = session.createReceiver(queue3, selector);

```

Initialisation d'une session "Publish/Subscribe"

```

// We need a pre-configured ConnectionFactory.
// Get it by looking it up using JNDI.
Context messaging = new InitialContext();
TopicConnectionFactory connectionFactory =
        (TopicConnectionFactory) messaging.lookup(" &");

// Gets Destination object using JNDI.
Topic Topic = (Topic) messaging.lookup(" &");

// Creates TopicConnection, then use it to create a session.
TopicConnection connection = connectionFactory.createTopicConnection();
TopicSession session =
        connection.createTopicSession(false, Session.CLIENT_ACKNOWLEDGE);

// Getting a TopicPublisher &
TopicPublisher publisher = session.createPublisher(Topic);
// & then a TopicSubscriber.
TopicSubscriber subscriber = session.createSubscriber(Topic);
Subscriber.setMessageListener(listener);

```

Construction de messages

Les messages sont créés au moyen de primitives de l'interface `Session` en fonction du type de message désiré : `createBytesMessage`, `createTextMessage`, `createMapMessage`, `createStreamMessage`, et `createObjectMessage`.

Utilisation d'un BytesMessage

```
byte[] data
BytesMessage message = session.createByteMessage();
message.writeBytes(data);
Utilisation d'un TextMessage
StringBuffer data
TextMessage message = session.createTextMessage();
message.setText(data);
```

Utilisation d'un MapMessage

Chaque message est constitué d'un ensemble de paires noms/valeur. Le client reçoit la totalité du message, il peut n'extraire que la partie utile du message et ignorer le reste ; les paires sont positionnées au moyen de méthodes sur l'objet `Message`, l'ordre des mises à jour est quelconque.

```
MapMessage message = session.createMapMessage();
message.setString("Name", " &");
message.setDouble("Value", doubleValue);
message.setLong("Time", longValue);
```

Utilisation d'un createStreamMessage

De la même manière, le serveur peut envoyer un message contenant les différents champs en séquence, chacun écrit au moyen de sa propre primitive. Si le client n'est intéressé que par une partie du message il doit tout de même le lire entièrement. Les champs doivent être écrits dans l'ordre de lecture.

```
StreamMessage message = session.createStreamMessage();

message.writeString(" &");
```

```
message.writeDouble(doubleValue);
```

```
message.writeLong(longValue);
```

Utilisation d'un ObjectMessage

L'information peut être envoyée sous la forme d'un objet sérialisable contenant l'ensemble des informations utiles. Le client utilise alors l'interface de cet objet pour obtenir l'information utile.

```
ObjectMessage message = session.createObjectMessage();
message.setObject(obj);
```

Envoi et reception de messages

Point-to-Point

```
// Sending of all of these message types is done in the same way:  
sender.send(message);  
// Receiving of all of these message types is done in the same way. Here is how to  
// receive the next message in the queue.  
// Note that this call will block indefinitely until a message arrives on the queue.  
StreamMessage message;  
message = (StreamMessage)receiver.receive();
```

Publish/Subscribe

```
// Sending (publishing) of all of these message types is done in the same way:  
publisher.publish(message);  
// Receiving of all of these message types is done in the same way. When the  
// client subscribed to the Topic, it registered a message listener. This listener  
// will be asynchronously notified whenever a message has been published to  
// the Topic. This is done via the onMessage() method in that listener class.  
// It is up to the client to process the message there.  
void  
onMessage(Message message) throws JMSEException {  
    // unpack and handle the messages we receive.  
    ...  
}
```

Lecture d'un message

BytesMessage

```
byte[] data;  
int length;  
length = message.readBytes(data);  
TextMessage  
StringBuffer data;  
data = message.getText();  
MapMessage           // Note : l'ordre de lecture des champs est quelconque.  
String name = message.getString("Name");  
double value = message.getDouble("Value");  
long time = message.getLong("Time");  
StreamMessage      // Note : l'ordre de lecture des champs  
                   // doit etre identique a l'ordre d'ecriture.  
String name = message.readString();  
double value = message.readDouble();  
long time = message.readLong();  
ObjectMessage  
obj = message.getObject();
```

E.3 Architecture de Joram

Cette section approfondit quelques aspects fondamentaux de l'architecture de la plate-forme JORAM qui lui confèrent des propriétés uniques en matière de flexibilité et de scalabilité.

E.3.1 Principes directeurs et choix de conception

La caractéristique essentielle de la plate-forme JORAM est son architecture distribuée et configurable. L'architecture de base de JORAM est de type *snowflake*, c'est-à-dire qu'une plate-forme JORAM est constituée d'un ensemble de serveurs JORAM distribués, interconnectés par un bus à message. Chaque serveur gère un nombre variable de clients JMS. La répartition des serveurs et la distribution des clients sur les serveurs sont de la responsabilité de l'administrateur de la plate-forme JORAM en fonction des besoins de l'application. Ce choix constitue donc un premier niveau de configuration. Un deuxième niveau réside dans la capacité de localiser les destinations (**Queues** et **Topics**) en fonction des besoins (nous verrons plus loin que ce choix d'implantation a des conséquences importantes sur les propriétés de l'application : performance, évolutivité, etc.). Un dernier niveau de configuration concerne les paramètres de qualité de service associés au bus à messages (protocole de communication, sécurité, persistance, etc.). Le choix de mécanismes appropriés est ici le résultat d'un compromis entre le niveau de qualité de service souhaité et le coût de la solution correspondante.

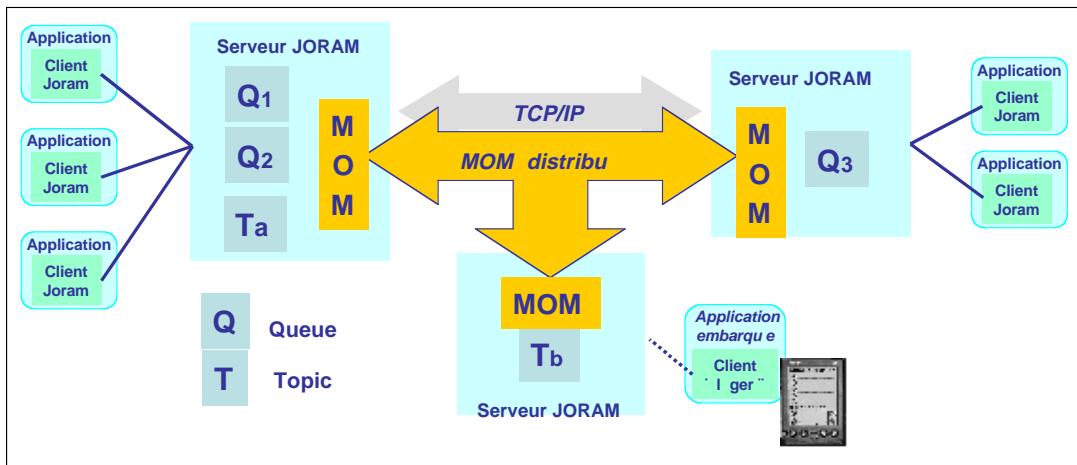


Figure E.7 – Architecture de la plate-forme JORAM

L'architecture générale d'une plate-forme JORAM est illustrée dans la figure 7.

E.3.2 Architecture logique

Le principe de fonctionnement d'une communication entre deux clients JMS est illustré dans la figure 8 (pour le cas particulier d'une communication point à point). Cette description permet de mettre en évidence les objets de communication et les flots de contrôle et de données. Les objets **Connection**, **Session** et **Sender** (respectivement **Receiver**) sont des

objets temporaires JMS créés par le client JORAM lors de l'établissement d'une connexion logique entre le client et le serveur JORAM. Sur le serveur, chaque client est représenté par un objet **Proxy**. Cet objet persistant est créé par le serveur lors de la création d'un utilisateur. Il remplit deux fonctions essentielles :

- La gestion des communications entre le client et le serveur.
- L'acheminement des messages vers / depuis la destination (ici une queue de messages).

Modèle de communication point à point

A ce stade de la présentation, la distribution n'est pas représentée. Il s'agit d'un schéma purement fonctionnel (i.e. dans les faits, les objets **Proxy** et **Queue** peuvent être implantés sur un ensemble de serveurs coopérants). Une communication entre un client producteur et un client consommateur met en œuvre les échanges suivants (représentés par les flèches rouges) :

1. L'interprétation d'une primitive **Send** d'un client JMS producteur se traduit par l'envoi d'un message sur la connexion entre le client JORAM et l'objet **Proxy** (noté ici **Proxy-P**) associé au client dans le serveur JORAM. Cette interaction est représentée par la flèche pleine 1.
2. L'objet **Proxy-P** encapsule le message JMS dans un message destiné à être véhiculé par le bus (noté ici **Message MOM**). Ce message est sauvegardé dans un support persistant local. Un accusé de réception est envoyé au client JORAM qui le remonte ensuite au client JMS (flèche pointillée 2). Du point de vue du client JMS, l'opération **Send** est terminée.
3. Le message construit par l'objet **Proxy** est véhiculé par le **MOM** vers le site de résidence de la queue de messages. Cette interaction est représentée par la flèche pleine 3. Sur le site où se trouve la queue de messages, le message est enregistré sur un support persistant local.
4. L'interprétation d'une primitive **Receive** par le client consommateur génère un message de contrôle vers l'objet **Proxy** associé (noté ici **Proxy-C**) qui le fait suivre au site où se trouve la file de message (objet **Queue**). Le message applicatif correspondant est extrait de la queue de messages et est envoyé vers l'objet **Proxy-C**. Celui-ci extrait le message JMS de son enveloppe véhiculée par le **MOM** et l'envoie au client JMS. Cette suite d'échanges est représentée par les flèches pleines 4. La requête **Receive** du client consommateur est terminée.
5. Un accusé de réception est envoyé vers le site de la localisation de l'objet **Queue** afin de retirer le message de la queue et de libérer les ressources correspondantes (flèches pointillées). L'accusé de réception peut être généré par le client JMS ou par le système.

Dans l'étape N°2 le message **MOM** est sauvegardé avant d'être envoyé vers le site de l'objet **Queue**. En cas de problème pendant l'étape N°3, la sauvegarde réalisée par l'objet **Proxy** permet de ré-exécuter cette étape jusqu'à son aboutissement. Cette fonction, communément appelée *Store and Forward*, permet d'assurer la garantie de délivrance des messages au niveau du serveur JMS. Notons que nous ne décrivons pas ici les échanges au niveau **MOM**.

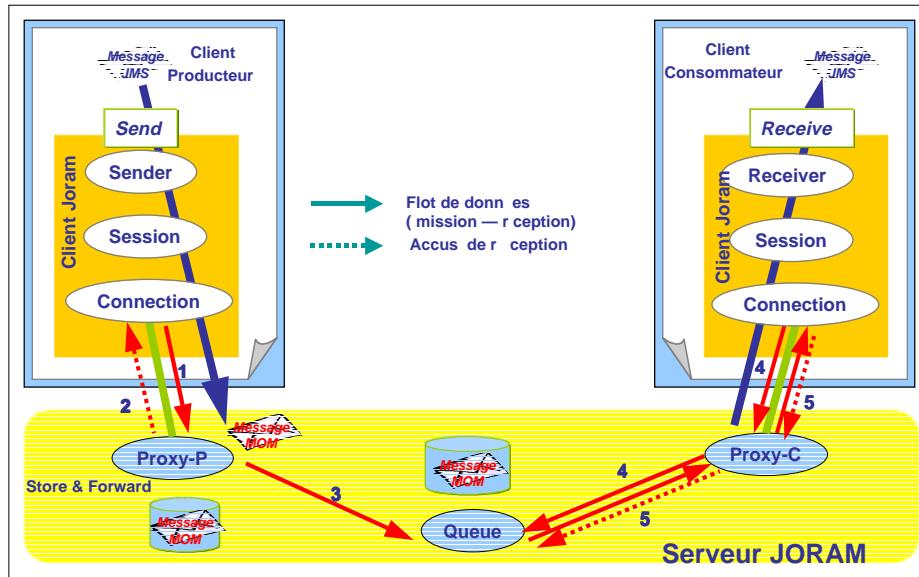


Figure E.8 – Communication Point à Point

Modèle de communication *Publish/Subscribe*

La figure 9 décrit les objets et flots de contrôle pour un mode de communication de type *Publish/Subscribe*.

1. Les étapes 1 et 2 sont similaires à celles du modèle de communication point à point.
2. Dans l'étape 3, le message construit par le Proxy du producteur est acheminé directement vers l'ensemble des objets Proxy des consommateurs où ils sont enregistrés. Notons ici une différence essentielle avec le schéma précédent dans la mesure où l'objet **Topic** est utilisé, non pas comme une destination finale, mais plutôt comme un aiguillage vers un ensemble de destinations finales représentées par les objets Proxy des consommateurs.
3. L'opération de consommation du message se traduit par un échange entre le client et l'objet Proxy-C (flèches pleines 4). L'accusé de réception est ensuite envoyé vers l'objet Proxy-C (flèche pointillée 5). Par rapport au schéma précédent on notera que le protocole de consommation se limite à un dialogue entre le client et l'objet Proxy. Il n'y a pas de dialogue entre les objets Proxy et le Topic .

Dans la suite, on étudie comment ce schéma d'architecture logique est mis en œuvre dans le cadre d'une configuration centralisée (serveur Joram unique) dans un premier temps, puis dans une configuration distribuée (serveurs coopérants).

E.3.3 Architecture centralisée

Cette option correspond à une configuration simple dans laquelle tous les objets **Destination** sont centralisés sur un seul serveur auquel sont connectés des clients par l'intermédiaire de l'objet **Proxy** correspondant. Cette configuration est illustrée dans la figure 10.

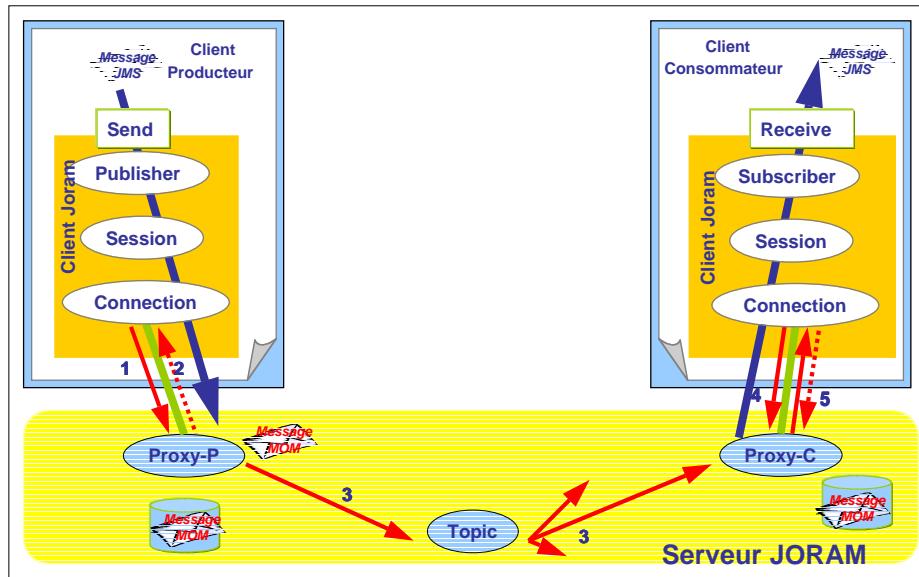


Figure E.9 – Modèle de communication *Publish/Subscribe*

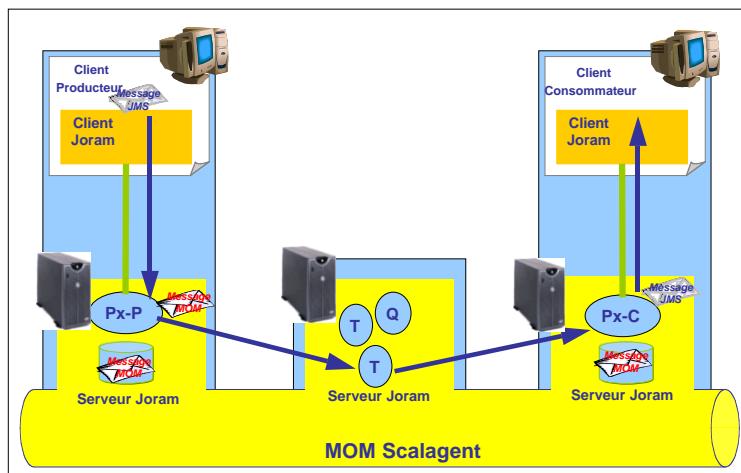


Figure E.10 – Architecture centralisée

Dans cette figure, les flots liés à l'opération de production sont représentés par des flèches pleines. Les flots de l'opération de consommation sont représentés par des flèches pointillées. On ne représente pas les flots liés aux accusés de réception. Dans le cas d'une communication de type *Publish/Subscribe* le dialogue est réalisé directement entre les deux objets **Proxy**.

Dans cette configuration la simplicité des échanges est une conséquence directe de la co-localisation de l'objet **Queue** et des objets **Proxy**. Un autre élément de simplicité est lié aux opérations d'administration (i.e. création des utilisateurs, des destinations, etc.) qui sont regroupées sur un site unique. L'inconvénient majeur de cette solution est son manque de disponibilité et une capacité d'extension réduite. Une défaillance du serveur signifie un arrêt du système global. Par ailleurs le nombre de requêtes et d'objets gérés par le serveur est limité par sa capacité de calcul et de stockage.

E.3.4 Architecture distribuée

Dans une architecture distribuée, plusieurs serveurs JORAM coopèrent pour assurer la communication des messages entre des clients connectés à ces divers serveurs. La Figure 11 décrit la structure des échanges pour une architecture répartie sur trois sites géographiques composés chacun d'un serveur et d'un client. Le schéma représenté utilise l'exemple d'une communication point à point dans lequel chaque serveur est responsable de la queue de messages destinés aux clients JMS connectés à ce serveur⁵.

Le message JMS, émis par le client JMS *Client* à destination de la queue de messages *Q3* est envoyé vers le proxy *Px1* sur le serveur *S1*. Le message MOM qui encapsule le message JMS est généré. La fonction *Store and Forward* sauve le message localement avant de l'envoyer vers le serveur *S3*, où se trouve localisée la queue *Q3*. Le message stocké dans la queue *Q3* peut être consommé par la suite par un client JMS connecté au serveur *S3*. De façon identique un message produit par le client JMS *Client-2* à destination de la queue de messages *Q1* est transmis via l'objet *Px2* du serveur *S2*, puis est ensuite consommé par le client JMS *Client* à partir du serveur local *S1*.

La communication entre les serveurs est mise en œuvre par le bus à messages sous-jacent qui garantit l'acheminement des messages quelles que soit les pannes de sites et de réseau.

E.3.5 Joram : une plate-forme JMS configurable

La construction d'une plate-forme JMS adaptée à un contexte applicatif donné est un exercice difficile dans la mesure où l'architecture de la plate-forme est le résultat de compromis délicats entre de nombreux critères souvent antinomiques, tels que performance, disponibilité, scalabilité, flexibilité et capacité d'évolution, sécurité, coûts de développement et d'exploitation, etc. Le rôle de l'architecte dans la définition d'une plate-forme est essentiel. À partir d'un cahier des charges il doit prendre en compte l'ensemble des critères d'évaluation et leur fixer un poids dans la perspective de définir l'architecture « la mieux adaptée ». Ce travail n'est possible que si le service de messagerie fournit des capacités de configuration suffisantes pour traduire les options d'architecture retenues. C'est le cas de

⁵Cette configuration particulière est prise à titre d'exemple. Dans la réalité la configuration des **Queues** et **Topics** sur les serveurs est laissée à l'initiative de l'administrateur.

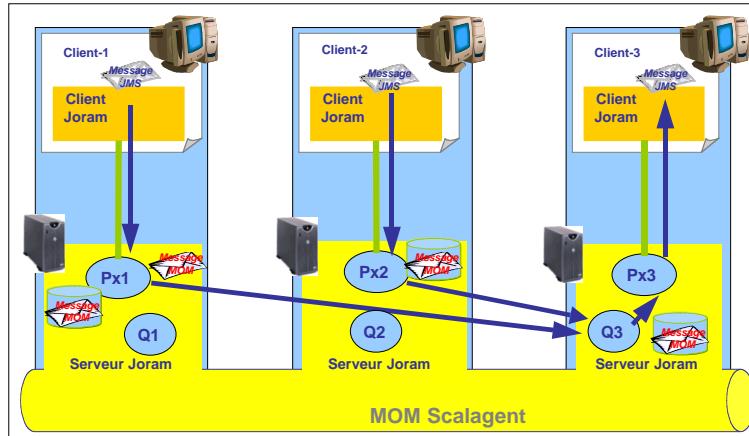


Figure E.11 – Architecture distribuée

la plate-forme JORAM qui permet, à plusieurs niveaux, de définir les choix de conception appropriés :

- Organisation des serveurs et clients JMS, et placement des objets JMS. Comme cela est illustré dans la figure 7, La structure de la plate-forme JORAM est de type *snowflake*, ce qui donne à l'architecte du système une grande liberté pour décider de l'emplacement des serveurs et de la répartition des clients sur les serveurs pour répondre aux besoins de l'application (par exemple pour servir un ensemble de clients géographiquement proches dans une approche de type serveur de proximité, ou bien pour respecter certaines contraintes de sécurité). Multiplier le nombre de serveurs pour une meilleure couverture géographique des clients distribués a un coût d'exploitation (en terme de machines) qui doit être mis en balance avec la performance et la fiabilité des communications pour des clients très éloignés d'un serveur. Le dimensionnement des serveurs (calcul, mémoire, stockage) est également un point clé pour la performance et la disponibilité de l'ensemble du système.
- Placement des objets de communications (**Queues** et **Topics**). Sauf dans des cas très particuliers, on peut noter qu'il est souhaitable de rapprocher les objets Destination des clients consommateurs. Cette stratégie a un impact positif sur les performances et la disponibilité des clients.
- Evolution du système et passage à l'échelle. Cette propriété fait référence à la capacité de faire évoluer le système pour répondre aux évolutions de l'application. JORAM fournit ainsi des fonctions d'administration qui permettent d'ajouter et/ou de retirer un serveur depuis un point central d'administration.
- Protocoles de communication. Plusieurs protocoles de transport sont disponibles pour la connexion client - serveur et pour les connexions entre serveurs : TCP/IP, HTTP, SOAP, SSL, etc.
- Paramètres de qualité de service (persistance, sécurité). La fiabilité et la mise en sécurité des communications a un coût. C'est pourquoi JORAM donne la possibilité de retenir ou non ces options selon les besoins.
- Niveau de disponibilité grâce à la clustérisation et à la réPLICATION (voir plus loin en section 4).

Peu de systèmes aujourd’hui permettent d’agir simultanément sur ces divers paramètres et de construire ainsi la plate-forme de messagerie adaptée à un environnement particulier. Le niveau de flexibilité de JORAM est de ce point de vue un atout incontestable par rapport aux produits concurrents.

E.4 Joram : fonctions avancées

Cette section décrit quelques fonctions avancées de JORAM (au sens où elles ne sont pas définies dans la spécification JMS 1.1).

E.4.1 Répartition de charge

L’architecture distribuée de JORAM est exploitée pour mettre en oeuvre des mécanismes de répartition de charge avec le double objectif suivant : accroître la disponibilité grâce à la réplication des objets de communication ; optimiser le flux des messages entre les serveurs. La répartition de charge s’applique de façon différente aux **Topics** et aux files de messages (**Queues**).

Topic répliqué

Un « Topic clustérisé » est répliqué sur plusieurs serveurs. Notons que cette forme de réplication s’applique aussi bien à des serveurs fortement couplés (grappe de machines) qu’à des serveurs géographiquement distribués. Le principe de fonctionnement du Topic clustérisé est illustré dans la figure 12.

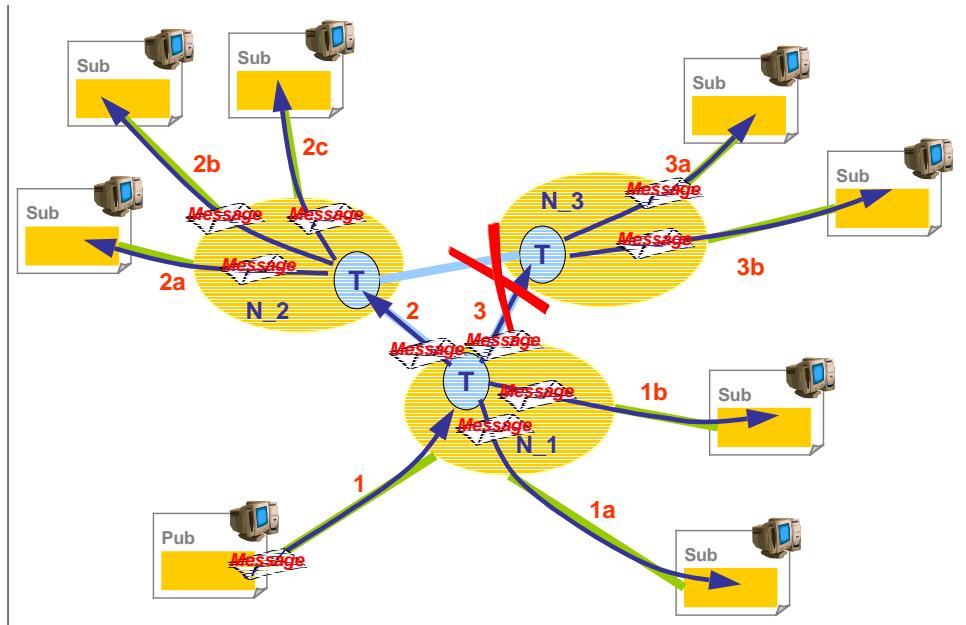


Figure E.12 – Topic "clustérisé"

Dans cet exemple un **Topic** (noté T) est répliqué sur trois serveurs N_1 , N_2 et N_3 . Des clients JMS connectés à ces serveurs sont supposés s'être abonnés au **Topic** T. Chaque serveur est responsable de la gestion des abonnements réalisés par ses propres clients. Une application cliente sur le noeud N_1 publie un message correspondant au **Topic** T (flot noté 1 dans la figure 12). Le noeud N_1 diffuse le message à ses abonnés locaux (flots 1a et 1b) et fait suivre le message aux noeuds N_2 et N_3 (flots 2 et 3). Par la suite, chacun d'entre eux diffuse le message reçu à ses clients locaux (flots 2i et 3j).

La mutualisation des messages entre les noeuds permet de réduire le trafic. Par ailleurs, une panne d'un serveur n'affecte que les clients connectés à ce serveur. Le reste de l'application continue à fonctionner. L'utilisation d'un **Topic** clustérisé est transparent du point de vue du programmeur d'application mais requiert beaucoup d'attention de la part de l'administrateur du système pour être efficace.

Queue répliquée

Le principe des « queues clustérisées » est un peu différent (voir figure 13). Plusieurs exemplaires du même objet **Queue** sont localisés sur des serveurs indépendants. Chacune de ces copies est accessible uniquement aux clients connectés au serveur correspondant. Si la charge sur une copie dépasse un certain seuil, les messages reçus ensuite sur cette copie sont redirigés vers un autre exemplaire de la même queue géré par un autre serveur. Le seuil est un paramètre configurable qui peut prendre en compte divers critères tels que : nombre de messages en attente, nombre de requêtes de lecture en attente, délai d'attente dépassé pour une requête en lecture, etc. Il est important de noter que la sémantique des queues de message n'est pas modifiée, à savoir qu'un message donné n'est consommé que par un seul client JMS).

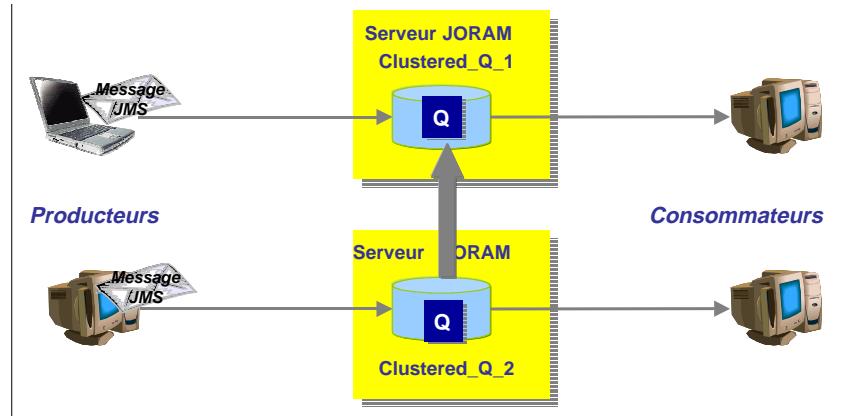


Figure E.13 – Clustered Queue

Comme dans le cas des **Topic**, le concept de queue clustérisée permet d'améliorer les performances et le niveau de disponibilité sans impact sur la programmation de l'application.

E.4.2 Fiabilité et haute disponibilité

Le terme fiabilité fait référence ici à la capacité de transporter un message de bout en bout entre un client producteur et un client consommateur malgré les incidents pouvant affecter de façon temporaire le réseau et les serveurs. La fiabilité dans JORAM est la résultante de plusieurs mécanismes complémentaires :

- Un accusé de réception entre un client JMS et son représentant dans le serveur (objet *proxy*) permet de fiabiliser la communication entre client et serveur (sémantique « au moins une fois »).
- La fonction Store and Forward réalisée par le proxy permet de fiabiliser les échanges entre le *proxy* et la destination (sémantique « exactement une fois »).
- Enfin le bus à messages assure des échanges fiabilisés entre serveurs (voir 5.2).

Pour répondre aux besoins de haute disponibilité une version spécifique du serveur JORAM (notée JORAM HA pour *High Availability*) a été conçue en suivant une approche de réplication active en mode Maître - Esclave. Le schéma de principe du serveur JORAM HA est représenté sur la figure 14.

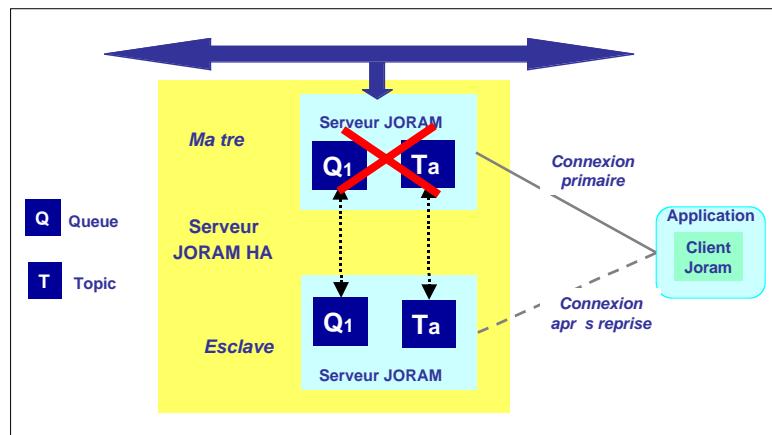


Figure E.14 – Serveur JORAM HA

Queues et Topics sont répliqués sur des serveurs JORAM s'exécutant sur une grappe de machine. Le serveur maître exécute les requêtes des clients et propage les opérations vers le serveur esclave qui réplique le traitement localement. En cas de panne du serveur maître, les clients établissent une nouvelle connexion vers le serveur esclave et continuent leur travail sans interruption. Ce fonctionnement permet un haut niveau de continuité de service au prix d'une redondance du serveur JORAM. La version actuelle du serveur JORAM HA utilise les mécanismes de JGroups.

E.4.3 Connectivité élargie

Cette section décrit plusieurs fonctions qui permettent d'enrichir la connectivité entre une plate-forme JORAM et le monde extérieur.

- Passerelle JMS pour l'interopérabilité avec d'autres plates-formes JMS,
- Utilisation du protocole SOAP
- Support de l'architecture de connectivité JCA 1.5

- Passerelles vers les protocoles SMTP et FTP

Passerelle JMS

La passerelle JMS permet à une application JMS gérée par JORAM de communiquer avec une destination gérée par une autre plate-forme JMS (appelée xMQ dans la figure 15) de façon transparente du point de vue du programmeur d'application.

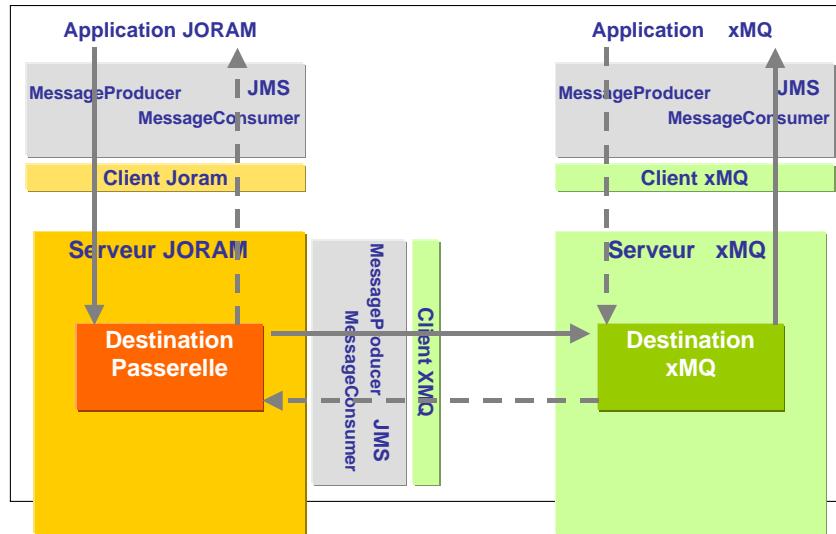


Figure E.15 – Passerelle JMS

Le lien entre une plate-forme JORAM et une plate-forme xMQ est réalisé par le biais d'un objet destination JORAM spécifique, appelé destination passerelle (qui peut être une Queue ou un Topic), qui est le représentant de la destination finale. L'objet « destination passerelle » joue deux rôles complémentaires :

- En tant que destination gérée par JORAM, il reçoit les messages produits par les clients producteurs et les requêtes des clients consommateurs gérés par la plate-forme JORAM.
- En tant que représentant d'un objet destination externe, il se comporte comme un client JMS géré par la plate-forme xMQ pour propager les messages et requêtes vers la destination finale.

Utilisation du protocole SOAP

L'utilisation du protocole HTTP-SOAP fournit un moyen normalisé pour accéder à des services distants en échangeant des messages XML sur des connexions HTTP. Dans certains cas il peut être utile d'utiliser ce protocole pour accéder depuis un client aux services d'un serveur JORAM, en particulier pour répondre aux besoins suivants :

- Contraintes de sécurité imposées par la gestion de pare-feux,
- Prise en compte de clients s'exécutant dans un environnement J2ME pour lequel l'API JMS complète ne peut pas être fournie (par exemple J2ME ne fournit pas de

fonction de sérialisation des objets).

L'usage du protocole SOAP dans JORAM est illustré dans la figure 16. Sur le serveur, un objet proxy particulier, construit comme un service SOAP, fournit un accès à des clients SOAP. Le service SOAP utilisé par JORAM est la version développée par la fondation Apache (<http://ws.apache.org/soap>). Dans cette approche un serveur JORAM est encapsulé dans un conteneur Tomcat et agit comme une passerelle entre Tomcat et les autres services de JORAM. Du côté client, une bibliothèque spécifique met en œuvre une connexion basée sur le protocole HTTP et des messages au format XML/SOAP. Dans les environnements J2ME, une version particulière de cette bibliothèque, appelée KJORAM, a été adaptée pour prendre en compte les restrictions imposées par J2ME.

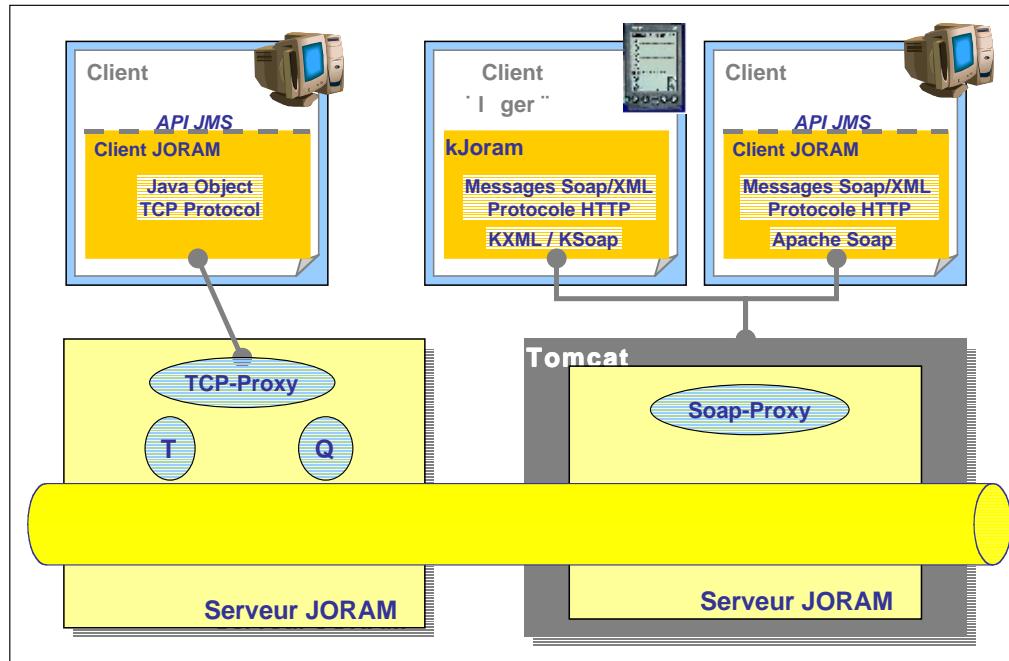


Figure E.16 – Utilisation de SOAP et KJORAM

La disponibilité de KJORAM étend les usages de la plate-forme JORAM à de nouveaux domaines d'application, marqués en particulier par l'utilisation d'équipements disposant de ressources limitées (par exemple les PDA, les téléphones et de façon plus générale tous les terminaux Java). Les applications J2ME s'exécutant sur ces équipements sont maintenant capables de coopérer avec des applications JMS s'exécutant sur des serveurs d'applications.

JCA 1.5

JORAM est conforme aux spécifications de l'architecture JCA 1.5 (*J2EE Connector Architecture*), qui décrit la manière d'intégrer des ressources externes dans un serveur d'application J2EE. Les fonctions suivantes sont disponibles :

- Gestion du cycle de vie de la ressource : création, démarrage, arrêt.
- Gestion des connexions avec les composants EJB

- Gestion transactionnelle conforme à l'interface XA.

L'utilisation de l'API JCA permet d'intégrer JORAM avec tout serveur d'application J2EE qui met en oeuvre cette spécification. C'est en particulier la voie d'intégration classique avec le serveur JOnAS (<http://jonas.objectweb.org>).

Connecteurs Joram

Les connecteurs JORAM offrent des passerelles pour réaliser l'interopérabilité avec des applications externes en utilisant des protocoles de transport normalisés. Deux connecteurs sont disponibles aujourd'hui :

- Passerelle *mail* : cette fonction permet d'émettre et recevoir des messages JMS en utilisant le protocole SMTP. La fonction est mise en oeuvre par le biais d'objets **Queues** et **Topics** spécialisés pour réaliser cette interopérabilité. Ces objets sont configurés et installés comme les destinations normales.
- Passerelle FTP : cette fonction est identique à la précédente pour le protocole FTP. C'est un dispositif utile lorsque les messages JMS à transporter sont de très grande taille.

E.4.4 Sécurité

Pour sécuriser les échanges (entre serveurs et entre un serveur et ses clients), JORAM utilise, à la demande, des connexions SSL afin d'authentifier les acteurs et de chiffrer les messages. La gestion des pare-feux pose des problèmes particuliers. La stratégie recommandée consiste à configurer les pare-feux pour autoriser l'usage des ports requis par JORAM. Cette solution s'applique autant pour les communications client - serveur que pour les communications entre serveurs. Une solution alternative consiste à utiliser, dans la plate-forme JORAM, des protocoles couramment acceptés par les pare-feux (HTTP et SOAP). La section 4.3.2 a montré comment SOAP peut être utilisé pour sécuriser la liaison entre clients et serveurs. Les serveurs JORAM disposent également d'un module de communication fondé sur HTTP pour permettre les communications entre serveurs traversant un pare-feu.

E.5 Joram : bases technologiques

Les propriétés de JORAM sont une conséquence directe de la technologie utilisée pour la réalisation de la plate-forme. Cette technologie et l'usage qui en est fait dans JORAM sont brièvement présentés dans cette section.

E.5.1 Les agents distribués

L'expérience de nombreuses années d'étude des architectures réparties nous a conduit à adopter une approche de type « intelligence distribuée » pour la conception des applications réparties. Rapprocher les traitements des sources de données permet, d'une part de répartir la charge de travail sur les ressources de calcul disponibles sur le réseau, et d'autre part de limiter la bande passante en ne véhiculant que l'information pertinente.

La mise en œuvre de ce principe fondateur s'appuie sur un modèle de programmation, un environnement d'exécution, et des outils de développement présentés dans la suite.

- Modèle de programmation à base d'agents communicants. Les agents sont des objets Java distribués communiquant par messages. Le contenu d'un agent est défini par une classe Java héritant d'une classe pré-définie « agent » qui définit le comportement générique d'un objet agent. Les agents se conforment à un modèle de programmation de type « événement-réaction ». Un événement correspond à la notification d'un message typé qui va se traduire par l'exécution d'une méthode de l'objet. Cette exécution peut, à son tour, provoquer la production d'événements auxquels un ou plusieurs agents vont réagir. Par défaut l'exécution d'une réaction est atomique (i.e. propriété du "tout ou rien") et l'état d'un agent est persistant.
- Environnement d'exécution : serveurs d'agents. Les agents s'exécutent au sein d'une structure d'accueil nommée « serveur d'agents », dont l'organisation est représentée sur la figure 17.

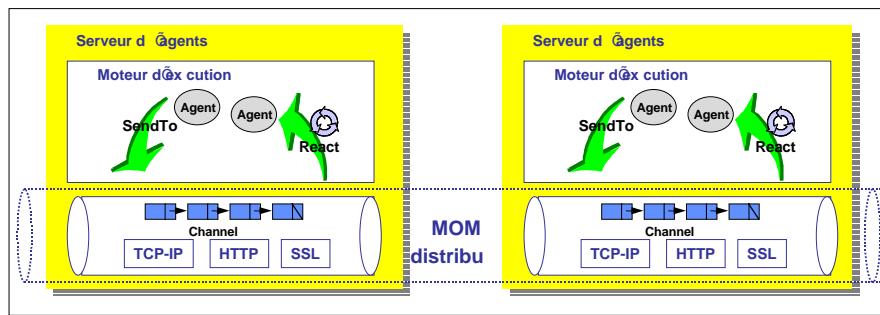


Figure E.17 – Structure d'un serveur d'agents

Le cœur du serveur est un moteur d'exécution qui contrôle le flot d'exécution des agents. Ce flot est unique et prend la forme d'une boucle qui consiste, pour chaque notification, à exécuter la méthode associée à la réaction de l'agent destinataire. Le flot exécute aussi le code des modules de persistance et d'atomicité. Le serveur comporte également des flots d'exécution associés à la gestion des communications au sein d'un sous-système appelé le bus local. Les bus locaux des différents serveurs coopèrent pour fournir une mise en œuvre distribuée d'un bus à message distribué. Ce bus à messages global permet la communication entre les différents agents, qu'ils soient hébergés par le même serveur ou non. Un bus local est constitué de deux types de composants : un « canal » et un ensemble de « composants réseau ».

- Le canal est chargé de distribuer les messages en provenance du moteur d'exécution vers les composants réseau et vice-versa.
- Les composants réseau sont responsables de l'émission de messages provenant du canal à destination des serveurs distants. Il existe différents composants réseau correspondant à différents protocoles de communication (par exemple TCP/IP, HTTP, SOAP, etc.). Par ailleurs, il est possible d'ajouter à un composant réseau des modules logiciels pour la mise en œuvre de propriétés complémentaires (par exemple sécurité, ordonnancement causal, etc.).

La structure d'accueil est configurable et fournit différentes politiques de fonctionnement des agents hébergés, en particulier l'atomicité des réactions (i.e. stratégie du « tout

ou rien »), et la persistance de l'état des agents (i.e. un changement d'état correspond à la complétion d'une réaction) Un ensemble d'opérations d'administrations élémentaires permettent de créer et de configurer des serveurs d'agents dans un environnement Internet.

E.5.2 Joram : une plate-forme JMS à base d'agents

La technologie à base d'agents distribués décrite ci-dessus a été largement utilisée pour construire des systèmes répartis dans des domaines d'application très variés. La plate-forme JORAM est un système distribué particulier mis en œuvre à l'aide de cette technologie. Un serveur JORAM, c'est-à-dire la partie de la plate-forme JORAM qui implante les objets JMS est un serveur d'agents structuré de la façon suivante :

- **Queues** et **Topics** sont représentés par des agents persistants.
- Sur chaque serveur un agent **ConnectionManager** gère les connexions avec les clients JMS gérés par ce serveur.
- Un agent Proxy persistant est créé pour chaque utilisateur reconnu par le système. Par la suite, cet agent a la charge de gérer la communication pour le compte des clients JMS (producteur ou consommateur) associés à cet utilisateur. Rappelons qu'une de ses fonctions essentielles est de mettre en œuvre la fonction *Store and Forward*.

La persistance des abonnements et des messages est mise en œuvre directement par la persistance de l'état des agents correspondants. Notons que cette propriété est « débrayable » si elle n'est pas requise par l'application, ce qui se traduit par un gain de performance. La flexibilité de la plate-forme JORAM bénéficie directement de la capacité de configuration des serveurs d'agents, en particulier pour ce qui concerne les éléments suivants :

- Le choix des protocoles de communication (TCP/IP, HTTP, SOAP, etc.) et des paramètres de QoS (persistance et garantie de délivrance, sécurité, etc.).
- Un choix d'implantation des serveurs JORAM pour définir une configuration distribuée de type « snowflake » qui réponde aux besoins de l'application cible.
- Un choix d'implantation des objets (agents) **Destination** et des objets (agents) **Proxy**.

Les services d'administration permettent, par la suite, de faire évoluer ces paramètres de configuration pour adapter la structure d'un système JORAM à de nouveaux besoins (implantation d'un nouveau serveur, migration des **Queues** et des **Topics**, changement de paramètres de sécurité, etc.).

E.6 Conclusion

JORAM est un système de messagerie caractérisé par deux propriétés majeures :

- Il est conforme à la spécification JMS1.1. Cette conformité a été démontrée dans le cadre de la campagne de certification J2EE 1.4 réalisée en liaison avec le serveur d'application JOnAS⁶. Il est donc possible d'utiliser JORAM comme environnement d'exécution de toute application distribuée qui suit l'interface de programmation définie par JMS.

⁶La spécification JMS est un élément de la spécification J2EE.

- L'architecture de JORAM est distribuée et hautement configurable. Cette propriété permet de concevoir une plate-forme d'exécution adaptée aux besoins d'une application donnée.

La figure 18 synthétise les divers usages de JORAM pour la mise en œuvre d'applications distribuées.

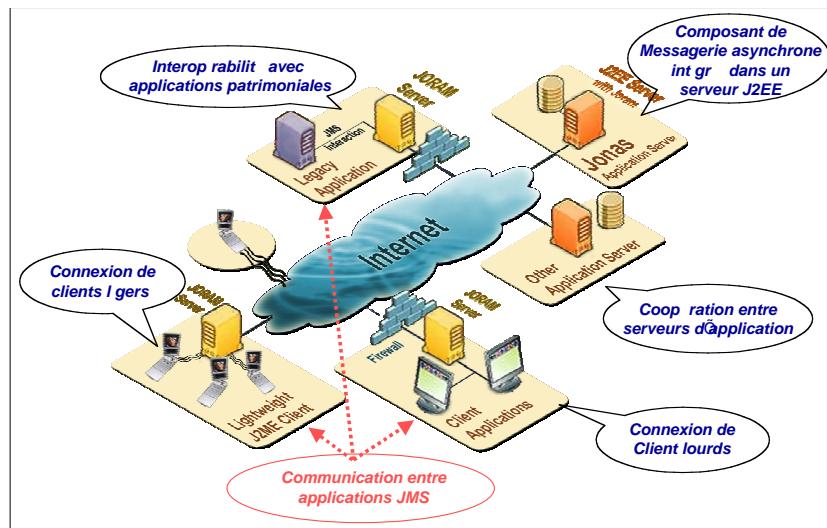


Figure E.18 – Les usages de JORAM

Ces usages sont les suivants.

- Système de communication asynchrone fiable entre applications JMS s'exécutant dans des environnements Java variés (J2EE, J2SE, J2ME), y compris la possibilité d'établir une passerelle vers des applications s'exécutant sur des plates-formes JMS externes.
- Composant de messagerie asynchrone intégré dans un serveur d'application J2EE. L'intégration dans le serveur JOnAS (<http://jonas.objectweb.org>) est la plus accomplie (administration globale unique). JORAM est également utilisé en liaison avec d'autres serveurs J2EE, en particulier JBoss.
- La combinaison des deux usages précédents permet d'élargir le champ d'action des serveurs d'application à des clients s'exécutant sur des dispositifs à faibles ressources (téléphones mobiles, équipements industriels, etc.).
- Interopérabilité avec des applications patrimoniales (legacy applications). JMS est aujourd'hui le canal de communication privilégié pour l'intégration d'application dans les plates-formes EAI (*Enterprise Application Integration*) et ESB (*Enterprise Service Bus*). En conséquence JORAM peut être utilisé comme base de mise en œuvre d'une telle plate-forme d'intégration.
- Coopération entre serveurs d'application J2EE. L'utilisation de JORAM permet ainsi de réaliser des versions distribuées du serveur d'application JOnAS.

La plate-forme JORAM est aujourd'hui en exploitation opérationnelle dans le monde entier pour des contextes applicatifs très variés : santé, banque, transport, logistique,

télécommunications, etc. Parmi les usages connus⁷ on peut citer l'EDI, l'intégration de données, l'administration de systèmes et d'applications répartis.

Les travaux en cours sur JORAM visent deux objectifs complémentaires :

- Etendre son champ d'application, en particulier dans le domaine des systèmes embarqués. Cet objectif requiert de pouvoir réaliser une version « légère » du serveur JORAM pour répondre aux contraintes de ressources de nombreuses classes d'équipements tels que carte à puce, lecteur RFID, contrôleur industriel, etc (en particulier pour ce qui concerne la mémoire RAM et la mémoire persistante de type mémoire flash) . Une approche, parmi d'autres, consiste à déporter une fonction store and forward dans la librairie client. Cette nouvelle structure permettrait également de mettre en œuvre, sous certaines conditions, des liaisons pair à pair (*peer to peer*) entre clients JORAM sans avoir recours à un serveur tiers.
- Améliorer les performances et les fonctions d'administration de la plate-forme JORAM. Ces deux aspects sont présentés globalement dans la mesure où ils présentent des dépendances fortes. La performance est un paramètre difficile à maîtriser car il dépend de critères multiples, tels que volume et taille des messages, nombre de clients, nombre de connexions, etc. L'optimisation d'une plate-forme JORAM relève à la fois de l'évolution de certains mécanismes internes (gestion de la concurrence, de la mémoire et de la persistance pour ne citer que ceux là), et du bon dimensionnement de la plate-forme pour un usage déterminé (par exemple les ressources des serveurs, le placement des objets de communication, etc.). Parmi les pistes explorées, l'usage d'un système de gestion de base de données pour gérer la persistance des messages est envisagé. Une piste à plus long terme concerne l'usage de mécanismes auto-adaptatifs pour permettre à une plate-forme JORAM de recueillir des informations sur son comportement afin de s'adapter « spontanément » à des défaillances (auto-réparation), à des baisses de performance (auto-optimisation), ou plus simplement à des changements de configuration (connexion / déconnexion d'un serveur).

Un axe de travail complémentaire concerne l'utilisation de JORAM comme moteur d'un bus de services d'entreprise (ou ESB pour *Enterprise Service Bus*). L'objectif est d'enrichir JORAM avec des fonctions de transformation de données (fondées sur une représentation XML) et de routage par le contenu pour répondre aux besoins des utilisateurs en matière d'intégration d'applications.

Pour conclure, notons que JORAM est une brique d'une plate-forme technologique plus large qui vise à fournir les fondations techniques pour la construction et le déploiement de solutions d'intégration dans le monde Internet/Java (voir figure 19).

Comme cela a été exposé dans la section 5, JORAM a été réalisé en s'appuyant sur la technologie à base d'agents développée par ScalAgent. En d'autres termes on peut considérer que JORAM constitue une personnalité d'un système réparti à agents qui met en œuvre l'API JMS. Notons que l'ensemble de c'est l'ensemble du produit « agents + JORAM » qui est disponible en logiciel libre, comme indiqué dans la figure 19.

Sur la même base technologique, ScalAgent a développé des briques d'intégration complémentaires qui s'appuient sur deux technologies émergentes : les composants et les architectures logicielles. L'intérêt pour les composants est une réalité aujourd'hui comme

⁷Comme la plupart des produits diffusés en logiciel libre, les retours connus sur les usages de JORAM sont peu nombreux et, en règle générale, peu détaillés.

le montre l'usage croissant des architectures à base d'EJB ou .NET. Par ailleurs les composants constituent des unités de programmation à gros grain qui facilitent la réutilisation et la construction d'applications par assemblage (i.e. sans recours à un langage de programmation). Cet aspect, qui vise à simplifier et raccourcir le cycle de développement des applications, n'est pas encore intégré dans les modèles tels que EJB et .NET. Cette capacité de construction d'applications par assemblage de composants est exploitée par les approches fondées sur les architectures logicielles. La définition des composants, de leurs interfaces et des leurs dépendances fonctionnelles est décrite dans un langage déclaratif, appelé ADL (pour *Architecture Description Language*) qui prend la forme d'un IDL étendu. La compilation d'une description ADL fournit une représentation logique de l'application distribuée. Ce référentiel est ensuite exploité dans toutes les étapes du cycle de vie de l'application pour automatiser et contrôler les opérations d'administration telles que le déploiement, la surveillance et la reconfiguration de l'application.

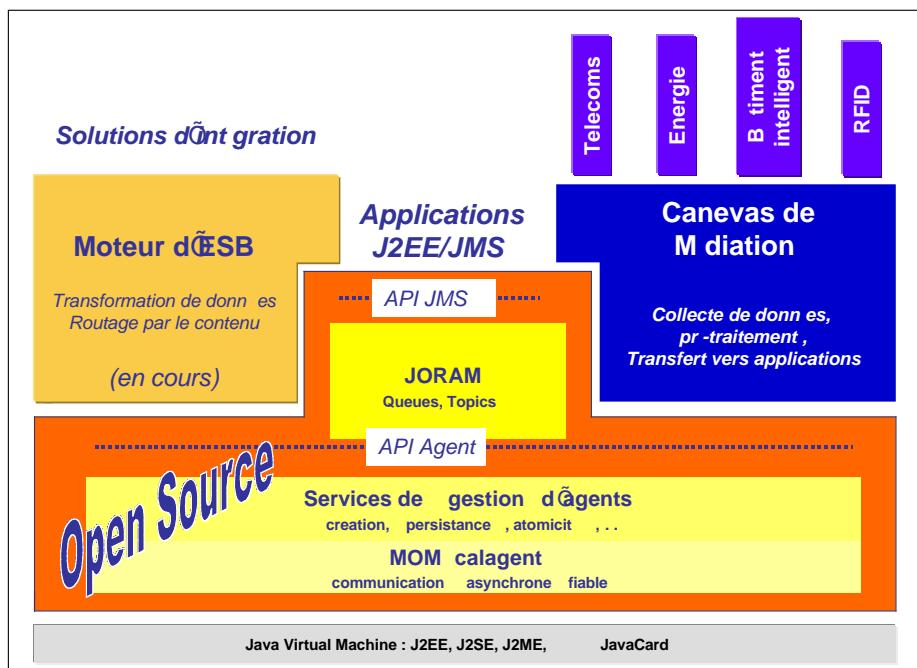


Figure E.19 – Joram et la médiation

Cette approche a été suivie par ScalAgent pour développer une infrastructure d'intégration de données d'entreprise (désignée globalement sous le terme « infrastructure de médiation »). Cette offre regroupe les couches hautes de la figure 19 et comprend les éléments suivants :

- un ensemble de composants de médiation pour la collecte, le traitement et la remontée d'informations d'usage produites par des équipements, services ou applications délocalisés. Ces composants peuvent être personnalisés et assemblés pour décrire des flots de données entre des équipements et des applications métiers. A l'exécution, les composants sont mis en œuvre par des agents exécutés sous le contrôle d'un ensemble de serveurs d'agents.
- Un ensemble d'outils graphiques pour construire, déployer et administrer des solu-

tions d'intégration par personnalisation et assemblage de ces composants. Ces outils s'appuient sur une représentation de type ADL.

- un service de déploiement pour automatiser le processus de déploiement d'une solution d'intégration sur un ensemble de sites.

Cet environnement utilise la même base technologique que JORAM pour gérer la distribution des données et du contrôle. JORAM est également utilisé comme vecteur de communication avec les applications patrimoniales et avec un serveur d'application J2EE où est implantée la logique de traitement des données collectées et remontées par l'infrastructure de médiation. Cette infrastructure de médiation est utilisée pour développer ensuite des solutions verticales adaptées à un usage donné, par exemple : la facturation télécom, la gestion intelligente d'énergie, ou encore la tracabilité à partir de données RFID.

E.7 Références

Documentation en ligne sur JORAM

- Page d'accueil : <http://joram.objectweb.org>
- Manuel de l'utilisateur : <http://joram.objectweb.org/doc/index.html>
- Wiki JORAM : <https://wiki.objectweb.org/joram/>

Publications concernant les bases technologiques de JORAM

Vivien Quéma, Roland Balter, Luc Bellissard, David Féliot, André Freyssinet and Serge Lacourte. Asynchronous, Hierarchical and Scalable Deployment of Component-Based Applications. In Proceedings of the 2nd International Working Conference on Component Deployment (CD'2004), Edinburgh, Scotland, may 2004

Roland Balter, Luc Belissard and Vivien Quéma. A Scalable and Flexible Operation Support System for Networked Smart Objects. In Proceedings of the 2nd Smart Objects Conference (SOC), Grenoble, France, May 15-17, 2003

Vivien Quéma, Roland Balter, Luc Bellissard, David Féliot, André Freyssinet and Serge Lacourte. Administration and Deployment Tools in a Message-Oriented Middleware. In the 4th ACM/IFIP/USENIX International Middleware Conference (Middleware), Poster session, Rio de Janeiro, Brazil, June 16-20, 2003.

Vivien Quéma, Luc Bellissard and Philippe Laumay. Application-Driven Customization of Message-Oriented Middleware for Consumer Devices. In Proceedings of the Workshop on Software Infrastructures for Component-Based Applications on Consumer Devices, in association with EDOC'02, Lausanne (Switzerland), September 16th, 2002.

L. Bellissard, N. De Palma, A. Freyssinet, M. Herrmann and S. Lacourte. An Agent Platform for Reliable Asynchronous Distributed Programming. Symposium on Reliable Distributed Systems (SRDS'99), Lausanne (Suisse), October 20th-22th, 1999.

R. Balter, L. Bellissard, F. Boyer, M. Riveill and J.Y. Vion-Dury. Architecturing and Configuring Distributed Applications with Olan. In Proceedings of the IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware'98), The Lake District (England), September 15th-18th, 1998.

L. Bellissard, N. de Palma and M. Riveill. Dynamic Reconfiguration of Agent-Based Applications. ACM European SIGOPS Workshop, Sintra (Portugal), September 7th-10th 1998.

Documentation relative à JMS

- Page d'accueil JMS : <http://java.sun.com/products/jms/>
- Tutorial JMS : <http://java.sun.com/products/jms/tutorial/index.html>
- Forum JMS : <http://forum.java.sun.com/forum.jsp?forum=29>
- Introduction à JMS (dans IBM developerwork) : "Enterprise messaging with JMS"
- Nommage et administration (dans IBM developerwork) : "Implementing vendor-independent JMS solutions"
- JMS 1.1 (dans IBM developerwork) : "JMS 1.1 simplifies messaging with unified domains"
- Les architectures applicatives JMS (dans *TheServerSide*) : "JMS Application Architectures"

Annexe F

Speedo : un système de gestion de la persistance

F.1 Introduction

Cette annexe décrit le fonctionnement et l'utilisation de Speedo, un système de gestion de la persistance d'objets Java. Cette description s'appuie sur le modèle de persistance proposé dans le cadre des spécifications JDO (*Java Data Objects*) [JSR-000012 2004, JSR-000243 2006], mises en œuvre par Speedo.

Le sujet de la persistance est au cœur des applications informatiques pratiquement depuis l'origine de la discipline. Pérenniser les informations manipulées par une application par delà son exécution ou les problèmes de pannes afférents à son environnement d'exécution matériel et logiciel est une propriété essentielle. Les disques durs représentent le support le plus courant pour gérer les informations pérennes, avec des abstractions logicielles de plus ou moins haut niveau pour les manipuler, que ce soit des systèmes de fichiers ou des systèmes de gestion de bases de données.

Il se trouve que les formats des données manipulées dans les langages de programmation et leur représentation dans le support pérenne sont dans tous les cas différents. Le couplage entre l'espace mémoire lié à l'environnement d'exécution d'un langage et l'espace mémoire pérenne est central dans le problème de la persistance. Dans le seul contexte du langage Java, plusieurs propositions ont été faites pour proposer des solutions le plus transparente possible du point de vue du programmeur, que ce soit au travers de spécifications standard [Sun JSR-000153 2003, JSR-000220 2006, JSR-000012 2004, JSR-000243 2006] ou de produits spécifiques [Oracle 1994, Hibernate 2002, Enhydra 2002, Castor 1999]. Le produit Speedo [Chassande-Barrioz 2003] proposé dans le cadre du consortium Objectweb [ObjectWeb 1999] se situe dans cette mouvance.

F.1.1 Qu'est-ce que Speedo ?

Speedo est un canevas de gestion de la persistance dans le contexte du langage Java. Il est utilisé pour implanter les deux standards prépondérants sur le sujet dans le cadre

de Java : JDO2 [JSR-000243 2006] et EJB3 [JSR-000220 2006]. Ces standards ont eu des évolutions différentes :

- JDO : l'objectif de JDO est de fournir un standard de persistance indépendant des supports de stockage (c'est à dire non lié aux seules bases de données relationnelles), et qui s'applique aussi bien dans le cadre de Java de base [J2SE 2005] que de Java pour l'entreprise [J2EE 2005]. Deux versions 1 [JSR-000012 2004] et 2 [JSR-000243 2006] de la spécification se sont succédées, sachant que les évolutions concernent soit des ajouts fonctionnels, soit des différences dans la spécification des informations de projection vers les bases de données relationnelles. Le modèle de programmation n'est notamment pas remis en cause entre ces deux versions.
- EJB : les spécifications EJB font partie de l'environnement Java pour l'entreprise [J2EE 2005]. Elles incluent un support de la persistance à travers les *entity beans*. Ces spécifications ont eu trois évolutions majeures, sachant qu'à chaque changement de version, le modèle de programmation a été complètement changé. Par ailleurs, la dernière mouture de ces spécifications propose un cadre de gestion de la persistance Java valide aussi bien pour Java de base [J2SE 2005] que pour Java pour l'entreprise [J2EE 2005]. Ce standard reste fortement lié aux bases de données relationnelles.

Fonctionnellement, Speedo s'apparente aux *clients lourds* des bases de données à objets puisqu'il met en œuvre des techniques de gestion de cache évoluées ou encore de gestion de concurrence. Il s'appuie sur un certain nombre d'autres canevas comme le montre la figure F.1, au nombre desquels on retrouve :

- Perseus [Chassande-Barrioz and Dechamboux 2003] : c'est un canevas qui identifie différents composants concourants à la mise en œuvre de systèmes d'échange de données entre un support de stockage et une mémoire d'exécution d'application, prenant en compte des comportements transactionnels ou encore des contextes d'exécution concurrents. Il définit aussi les liens qui régissent les interactions entre ces différents composants. Parmi ces composants, nous pouvons citer le gestionnaire de concurrence, le système de journalisation ou encore le gestionnaire d'E/S.
- MEDOR [Team 2000] : c'est un canevas qui permet de manipuler des requêtes ensemblistes à travers des arbres combinant des opérations algébriques. Il permet de s'adapter à différents moteurs d'exécution de requêtes, comme par exemple un moteur SQL.
- JORM [Team 1998] : ce canevas offre un moyen de projeter des objets vers différentes structures de stockage. Il permet notamment de supporter les références entre objets et la navigation à travers ces références pour ce qui concerne le déclenchement d'E/S.

Par ailleurs, l'architecture de Speedo et d'autres canevas de l'écosystème technique s'appuie sur Fractal [Team 2003] ce qui permet d'expliquer l'architecture logicielle de l'ensemble. L'intérêt est qu'il est alors possible d'adapter l'environnement Speedo pour changer certains comportements, et notamment amener de nouvelles politiques de gestion de la concurrence, de gestion de remplacement des objets dans le cache, etc.

F.1.2 Principales fonctions de Speedo

Au niveau des modèles programmatiques, Speedo a fait le choix d'implanter des standards car ils sont garants de pérennité pour l'utilisateur et par là de stabilité pour les solutions fournies. Outre le support des standard JDO2 et EJB3, Speedo offre des spécificités

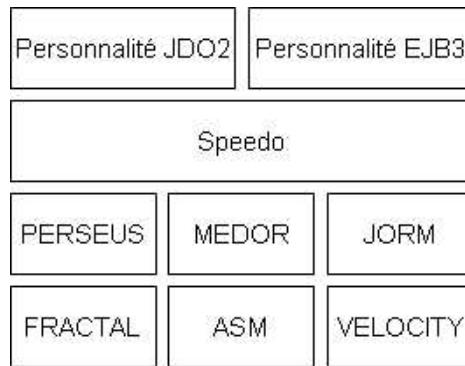


Figure F.1 – Décomposition fonctionnelle de Speedo

qui ne mettent pas en cause la portabilité des applications et qui sont autant d'atouts vis-à-vis d'autres solutions ou produits de persistance équivalents.

Gestion des caches

Speedo propose deux niveaux de gestion de cache. Il gère dans un premier temps l'ensemble des objets actifs dans le contexte d'exécution d'une transaction. Lorsqu'un nouvel objet doit être attaché à un tel contexte, on vérifie qu'il n'y est pas déjà. S'il n'y est pas, un deuxième niveau de cache est mis en œuvre. C'est un cache global qui contient les objets Java persistants présents en mémoire. Le fait de recycler les objets d'un contexte à un autre permet d'alléger la charge d'exécution liée à la création/destruction d'objets (allégement de la charge du ramasse-miettes).

Pour la gestion de l'encombrement mémoire par le cache global, Speedo supporte différentes politiques de remplacement telles que LRU (*Least Recently Used*). Par ailleurs, il est possible d'implanter de nouvelles politiques si nécessaire.

Gestion de la concurrence

Les solutions de persistance actuelles délèguent la gestion de la concurrence d'accès au système de gestion de base de données sous-jacent. Cela a l'avantage de permettre à d'autres applications n'utilisant pas Speedo d'accéder à la base de données en cohérence avec les applications Speedo.

Si cette contrainte est levée, Speedo supporte la gestion de la concurrence directement en mémoire au niveau des objets Java, ce qui permet d'éviter nombre d'accès inutiles à la base de données pour effectuer des prises de verrou. Là aussi différentes politiques sont proposées comme des politiques pessimiste ou optimiste, avec gestion des situations de *dead locks*.

Chargement d'objet anticipé

Dans bien des cas, il peut être intéressant de ramener les objets qui vont être instanciés en mémoire, notamment lorsqu'il s'agit de collection d'objets. En effet, traverser une collection d'objets implique de ramener en mémoire des objets représentant une collection

(cas de relations multi-valuées), ou la collection résultant d'une requête ensembliste, puis d'instancier chaque objet de cette collection. Cela nécessite $n + 1$ requêtes vers la base. Speedo permet de remonter les objets de la collection en même temps, ce qui ne nécessite plus qu'une requête vers la base au lieu des $n + 1$. L'impact de cette approche sur les performances est donc majeur.

F.1.3 Restrictions actuelles

F.1.4 Organisation du cours Speedo

Ce cours décrit la mise en œuvre de Speedo à travers sa personnalité JDO. Dans le même temps, il couvre aussi une description du standard JDO lui-même.

F.2 Gestion d'objets persistants avec JDO : les principes

En 1999, un groupe d'experts travaillant sur la problématique de persistance des objets Java a décidé d'apporter l'ensemble de ses travaux à SUN, dans le cadre du JSR 12 ([JSR-000012 2004]) : spécifications dites *Java Data Objects* (JDO). Ambitieux, l'objectif de cette spécification est *write once, persist anywhere* (coder une fois, persister n'importe où). Ce standard conçu à l'origine par des éditeurs de bases de données à objets se transforme en une solution de persistance universelle pour Java. En effet, n'importe quel support de stockage, un SGBDR, un SGBDO, de simples fichiers, peut recevoir une interface JDO. Réciproquement, une application développée dans un cadre JDO peut persister sans aucune modification sur tout support de persistance doté d'une interface JDO. Plusieurs solutions commerciales ou open source sont apparues très rapidement. Face aux retours des utilisateurs et aux évolutions de la spécification EJB (2.x puis la prochaine 3.0), le même groupe d'experts, dans le cadre du JSR 243, a travaillé sur l'évolution de la spécification JDO. Cette nouvelle version 2.0 apporte son lot de nouvelles fonctions et de standardisation, en particulier la projection objet/relationnel (O/R dans la suite).

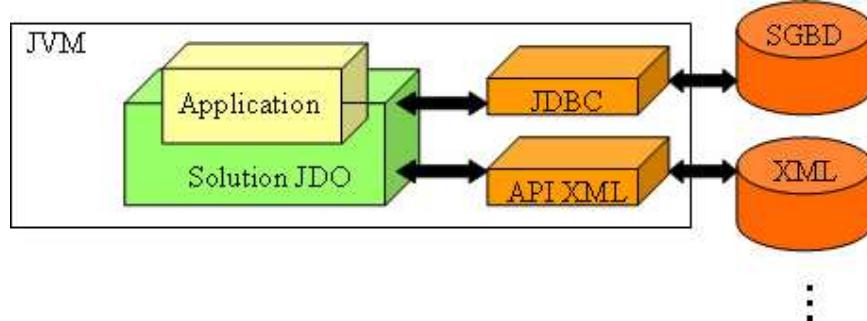


Figure F.2 – Position de JDO dans l'architecture d'application

JDO est une spécification qui définit un modèle de programmation transparent pour la persistance d'objets Java vers un système de stockage potentiellement transactionnel. Les produits JDO sont donc des outils intergiciels de projection d'objets langage vers diverses structures de stockage. Les avantages de JDO sont les suivants :

- Modélisation objet sans contraintes : supporte de l'héritage, des collections, des interfaces, etc.
- Persistance transparente : permet d'avoir des objets métiers complètement indépendants de la structure de stockage (par exemple, pas de dépendance vers les APIs JDBC).
- Réduction des temps de développement : en effet tout le code dédié à la gestion de la structure de stockage, qui représente un effort non négligeable (évaluée à 40% de l'effort de développement par certaines études), est prise en charge par JDO.
- Persistance universelle : JDBC est limitée au SGBDR, JDO est capable, au déploiement, d'assurer la persistance aussi bien dans un SGBDR que dans un SGBDO, un système de type CICS, des fichiers plats ou au format XML.
- JDO est disponible sur toutes versions de Java : J2SE, J2ME et J2EE, en architecture client/serveur ou multi étages.

Comme le montre la figure F.2, une solution JDO est un canevas logiciel qui prend en charge la persistance des objets applicatifs nécessitant cette propriété.

F.2.1 Classes et objets persistants

Les objets persistants sont des objets Java comme les autres. Simplement, le programmeur doit indiquer les classes de son application qui peuvent produire des objets persistants : nous les appelons classes persistantes. En effet, le principe de JDO est d'ajouter le code nécessaire au support de la persistance dans la classe en question (cette phase est aussi appelée phase d'*enhancement* en anglais). Pratiquement toutes les classes java peuvent être traitées afin que leurs instances puissent persister.

L'unité de persistance est donc l'objet d'une classe persistante. Néanmoins, la persistance d'un tel objet n'est pas lié à sa classe. C'est le programmeur qui décide du moment où un objet devient persistant. Il le fait soit explicitement grâce à la fonction `makePersistent` de l'API JDO, soit implicitement par rattachement à un objet déjà persistant.

Une instance persistante est toujours identifiée de manière unique par un identifiant permettant de désigner l'objet dans la structure de stockage. Cet identifiant permet donc de faire le lien entre l'objet Java présent en mémoire sa projection dans la structure de stockage sous-jacente. L'identifiant peut être basé sur des champs de la classe ou calculé par ailleurs (par la solution JDO, par le support de stockage, etc).

Rendre un objet persistant signifie que l'ensemble des variables ou des champs désignés comme persistants, composant ainsi l'état de cet objet persistant, est enregistré sur le support de stockage. Dès lors qu'une de ces variables est modifiée en mémoire, cette modification est propagée vers le support de stockage. L'instant de synchronisation dépend du contexte d'exécution : par exemple, les modifications effectuées dans un contexte transactionnel sont généralement propagées à la validation de la transaction.

JDO définit les types de champs qui peuvent être persistants. Nous pouvons les regrouper suivant différentes catégories :

- Types primitifs : il s'agit de `boolean`, `byte`, `char`, `short`, `int`, `long`, `float`, `double`, `byte[]` et `char[]`.
- Types "objet" simples : il s'agit des classes Java `java.lang.Boolean`, `java.lang.Byte`, `java.lang.Character`, `java.lang.Short`, `java.lang.Integer`, `java.lang.Long`, `java.lang.Float`, `java.lang.Double`, `java.lang.String`, `java.lang.Number`, `java.math.BigDecimal`,

`java.math.BigInteger`, `java.util.Date`, `java.util.Locales`, `java.sql.Date`, `java.sql.Time` et `java.sql.Timestamp`

- Types "objet" multivalués : il s'agit des classes ou interfaces Java `java.util.Collection`, `java.util.Set`, `java.util.HashSet`, `java.util.List`, `java.util.LinkedList`, `java.util.ArrayList`, `java.util.Vector`, `java.util.Map`, `java.util.HashMap`, `java.util.TreeMap`, `java.util.HashTable` et `java.util.Properties`.
- Types "objet" persistants : les références vers des objets persistants peuvent aussi être persistantes. Elles sont définies par les champs dont les types sont soit des classes persistantes, soit des interfaces persistantes. Comme nous l'avons précisé auparavant, ces classes ou ces interfaces sont spécifiées par le programmeur.

Enfin, la transparence proposée par JDO est telle qu'il n'est pas nécessaire d'utiliser des méthodes d'accès du type `getXXXX/setXXXX` permettant de manipuler le champ `xxxx` (même patron que celui proposé par les JavaBeans [B. Stearns 2000a, B. Stearns 2000b]). Ce type de patron est en effet imposé dans d'autres solutions comme EJB2 ou Hibernate.

F.2.2 Déclinaison Speedo des principes JDO

L'implantation JDO de Speedo garantit le niveau de transparence le plus élevé même si cela n'est pas requis par la spécification. En effet, Speedo garantit qu'il y a une seul objet Java représentant un objet persistant en mémoire. Typiquement, dans un environnement multi-programmé (plusieurs contexte d'exécution ou *threads* se déroulant en parallèle), tous ces contextes d'exécution partagent les mêmes objets. Cela signifie que ces objets sont soumis aux mêmes contraintes de réentrance que n'importe quel objet Java.

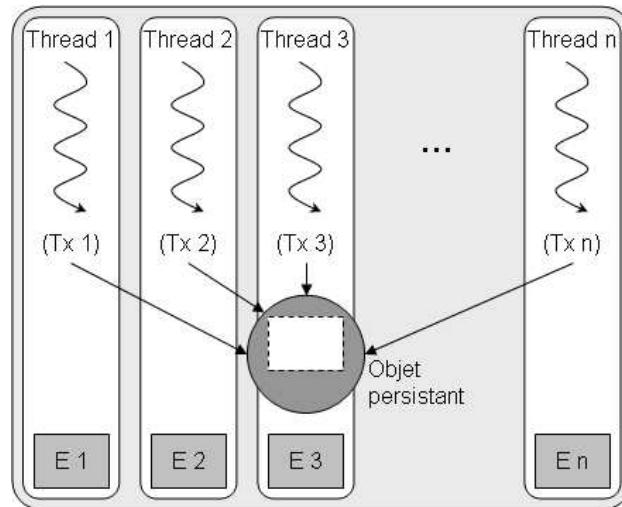


Figure F.3 – Découplage objet persistant/état

Les différents modes possibles de synchronisation avec le support de stockage (les modes transactionnels nécessitent de gérer l'isolation de contextes) font qu'il est nécessaire dans certains cas de pouvoir gérer en mémoire plusieurs replicas de l'état d'un objet persistant. Pour cela, Speedo met en œuvre le patron *état* (en anglais *state pattern*) [Gamma et al. 1994], les champs d'un objet persistant étant alors gérés en mémoire dans

un autre objet représentant cet état comme le montre la figure F.3. Cette démarche permet à Speedo de mettre en œuvre des politiques de concurrence en mémoire indépendantes de celles fournies par le support de stockage sous-jacent. Evidemment, ces choix d'architecture sont globalement transparents pour le programmeur, sauf dans la phase de mise au point de programme où le programmeur a accès aux champs persistant indirectement dans l'objet qu'il a défini : il lui faut les chercher dans l'objet state référencé par son objet persistant. Enfin, certaines politiques de concurrence sont incompatibles avec le processus de mise au point (pas de référence vers l'objet état) : c'est le cas de la politique optimiste gérée en mémoire par Speedo.

F.3 Chaine de production de programme

Cette section a pour objectif de montrer quels sont les éléments qui interviennent dans la chaîne de production de programmes utilisant JDO, et plus particulièrement la solution Speedo. Elle détaille aussi les différentes étapes du processus de production de programme Speedo.

F.3.1 Phases de la production de programme

Avant de s'intéresser à la chaîne de production de programme dans son ensemble, il est important de comprendre le processus d'*enhancement* proposé par Speedo et d'autres implémentations JDO, sachant que cette approche de mise en œuvre de JDO n'est pas imposée par la spécification.

Principe de l'*enhancement*

JDO permet d'utiliser des classes Java et de rendre leurs instances persistantes. Cependant l'ajout de la propriété de persistance à une classe nécessite de lui appliquer un traitement post compilation appelée *enhancement*. Lors de cette étape, le fichier `.class` subit des transformations dépendant de sa persistance décrite dans un descripteur JDO (fichier `.jdo`). Après cette opération les classes ainsi transformées peuvent être utilisées dans une application JDO ou dans une application n'utilisant pas JDO.

Toute compilation (initiale ou recompilation suite à une modification des sources) des classes à rendre persistantes nécessite de lancer l'*enhancer* (programme modifiant les classes pour prendre en compte la persistance). L'étape d'*enhancement* peut être standard ou particulière à une implantation JDO. Afin d'obtenir de meilleures performances et offrir un modèle de programmation plus intéressant, Speedo a son propre *enhancer*. Celui-ci s'utilise via une tâche Ant (cf. <http://ant.apache.org/>).

Eléments de la chaîne de production de programmes

La figure F.4 représente les différents éléments de la chaîne de production de programmes. Les éléments initiaux de cette chaîne sont les fichiers sources (`.java`) et les descripteurs de persistance JDO (`.jdo`). La première étape est la compilation des sources en byte code Java (génération des `.class`). La deuxième étape est l'*enhancement*. Cette

étape est réalisée par l'*enhancer* de Speedo. Durant cette étape, les classes persistantes sont modifiées et de nouvelles classes sont générées par Speedo afin de gérer la persistance.

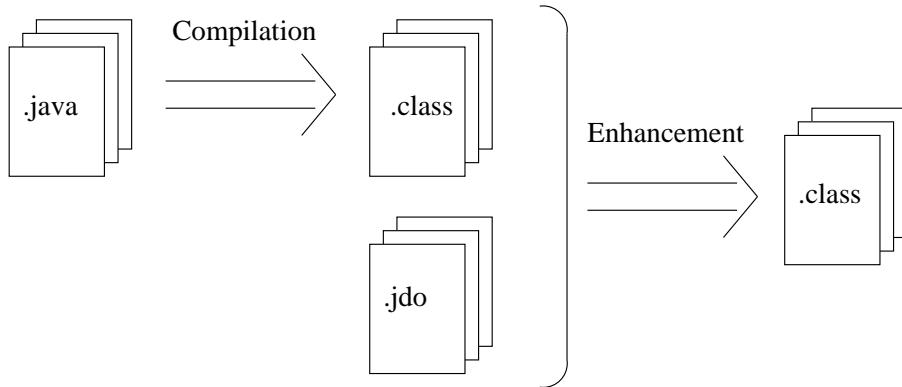


Figure F.4 – Chaîne de production de programmes Speedo

La stratégie de Speedo est de générer au moment de la compilation tout le code nécessaire à la persistance. Ce choix est motivé par l'objectif de performance. La génération de code est assurée par deux techniques : la génération de source Java via Velocity (cf. <http://jakarta.apache.org/velocity/index.html>) et la génération de byte code Java directement via ASM [Bruneton et al. 2002]. La génération de byte code Java directe se prête difficilement à la génération de classes longues et complexes.

F.3.2 Utilisation de Ant

Speedo fournit une tâche Ant pour le lancement de l'enhancement. Comme pour toute tâche Ant, il faut d'abord la définir avant de l'utiliser. La définition d'une tâche Ant se fait en utilisant la tâche `taskdef`. Ci dessous, un exemple de `target` d'initialisation définissant les tâches Speedo :

```

<target="init">
    <!-- repertoire contenant les sources des classes persistantes -->
    <property name="src" value="${basedir}/src"/>
    <!-- repertoire où sont mis les classes compilées ou enhancées -->
    <property name="build" value="${basedir}/build"/>
    <!-- Repertoire de la distribution Speedo -->
    <property name="speedoDist" value="${basedir}/../speedo"/>

    <!-- definition d'un classpath -->
    <path id="classpath">
        <pathelement location="${build}"/>
        <pathelement location="${speedoDist}/etc"/>
        <fileset dir="${speedoDist}">
            <include name="speedo-jdo.jar"/>
            <include name="lib/log/log4j.jar"/>
        </fileset>
    </path>
    <taskdef resource="speedo-jdo.tasks" classpathref="classpath"/>
</target>
.

```

Le lancement de l'*enhancer* est fait immédiatement après la compilation en appelant la tâche `speedo-jdo` :

```

<target="compile" depends="init">
    <mkdir dir="${build}"/>
    <!-- compilation -->
    <javac srcdir="${src}" destdir="${build}" debug="on">
        <classpath refid="classpath"/>
        <include name="**/*.java"/>
    </javac>
    <!-- enhancement -->
    <speedo-jdo src="${src}" output="${build}" />
</target>
.

```

La sortie produite par l'*enhancer* Speedo est généralement de la forme suivante :

```
{prompt} ant compile

init:

compile:
    [mkdir] Created dir: /usr/local/home/developper/workspace/speedo/output/test
/jdo
[javac] Compiling 94 source files to /usr/local/home/developper/workspace/sp
eedo/output/test/jdo
[speedo-jdo] [INFO] Computing classes requiring enhancement ...
[speedo-jdo] [INFO] Completing the meta data...
[speedo-jdo] [INFO] Auto O/R Mapping...
[speedo-jdo] [INFO] Building Jorm Meta Information...
[speedo-jdo] [INFO] Generating Jorm files...
[speedo-jdo] [INFO] Generating Speedo files...
[speedo-jdo] [INFO] Serializing Meta Info...
[speedo-jdo] [INFO] Enhancing Persistent classes...
[speedo-jdo] [INFO] Enhancing Persistent aware classes...
[speedo-jdo] [INFO] Compiling Java files...
[speedo-jdo] [INFO] All Done in 8s 282ms

BUILD SUCCESSFUL
Total time: 34 seconds
{prompt}
```

Cette trace montre notamment les différentes étapes de l'*enhancer* Speedo. Le processus d'*enhancement* Speedo étant gourmand en mémoire, il peut être nécessaire de gonfler la mémoire allouée au processus Ant l'exécutant (problème de *Out of memory* à l'*enhancement*). Pour cela, il faut définir ou modifier la variable d'environnement `ANT_OPTS` pour qu'elle contienne les paramètres suivant : `-Xms130m -Xmx130m -Xss256k`. Dans ce cas, la mémoire mise à disposition de la tâche Ant est de 130 Mo. On peut augmenter ce chiffre suivant le besoin.

F.4 Programmation d'objets Java persistants

Cette section fait le tour des fonctions qui permettent de gérer des objets Java persistants grâce à JDO. Elle s'attache notamment à voir de quelle manière le modèle de programmation Java est impacté lorsqu'on ajoute le support de la persistance.

F.4.1 Exécuter un programme utilisant Speedo

Avant de commencer à programmer des objets Java persistant, nous montrons comment installer Speedo et exécuter un programme JDO à l'aide de Speedo.

Installation de Speedo

Pour pouvoir utiliser Speedo, la première étape est de récupérer le logiciel. Pour télécharger le logiciel, il faut aller sur le site Web de Speedo : <http://speedo.objectweb.org>. Ensuite, il faut naviguer par le lien `download` du menu `Project Links` présent sur la gauche

de la page du site. Cliquer ensuite sur la livraison à télécharger. Par exemple, on peut choisir la livraison binaire `Speedo-jdo_1.4.5.bin.zip` de la personnalité JDO de Speedo.

Il faut ensuite décompresser le fichier téléchargé dans un répertoire d'installation (nous l'appellerons `/root`). Attention, nous utilisons le format de nommage Unix pour désigner des chemins d'accès à des fichiers ou à des répertoires. Nous obtenons un répertoire nommé `/root/Speedo-jdo_1.4.5` avec la structure présenté dans la figure F.5.

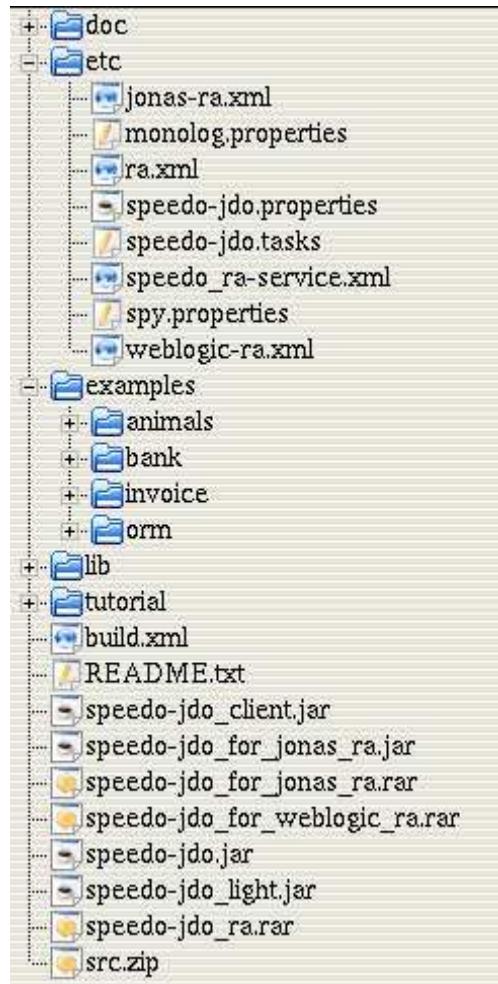


Figure F.5 – Distribution Speedo pour JDO

A partir de cette arborescence, on va pouvoir construire la librairie complète permettant d'utiliser Speedo. En se plaçant dans ce répertoire, nous lançons la construction de ces librairies. Le pré-requis est que `ant` (cf. <http://ant.apache.org>) soit installé :

```

{prompt} cd /root/Speedo-jdo_1.4.5
{prompt} ant
Buildfile: build.xml

archives:

archives-speedo-jdo.jar:
[speedo-jdo.jar] Building zip: /root/Speedo-jdo_1.4.5/speedo-jdo.jar

archives-speedo-jdo_light.jar:
[speedo-jdo.jar] Building zip: /root/Speedo-jdo_1.4.5/speedo-jdo_light.jar

archives-speedo-jdo_client.jar:
[speedo_client.jar] Building zip: /root/Speedo-jdo_1.4.5/speedo-jdo_client.jar

archives-speedo-jdo_ra.rar:
[speedo_ra.rar] Building jar: /root/Speedo-jdo_1.4.5/speedo-jdo_ra.rar

archives-jonas:
[speedo-jdo_for_jonas_ra.rar] Building zip: /root/Speedo-jdo_1.4.5/speedo-jdo_for_jonas_ra.jar
[speedo-jdo_for_jonas_ra.rar] Building jar: /root/Speedo-jdo_1.4.5/speedo-jdo_for_jonas_ra.rar

archives-weblogic:
[speedo-jdo_for_weblogic_ra.rar] Building jar: /root/Speedo-jdo_1.4.5/speedo-jdo_for_weblogic_ra.rar

BUILD SUCCESSFUL
Total time: 36 seconds

{prompt}

```

Cela permet de construire les librairies permettant d'utiliser Speedo. Ces librairies contiennent des fichiers de configuration correspondant à des configurations par défaut dédiées à l'environnement d'exécution cible (cf. section F.4.1). Les librairies générées se retrouvent dans le répertoire `/root/Speedo-jdo_1.4.5`.

Librairies nécessaires à l'exécution d'une application Speedo

Suivant l'environnement d'exécution dans lequel Speedo est utilisé, différentes librairies sont proposées. En fait, principalement deux environnements sont considérés :

- L'environnement Java de base (J2SE) : dans cet environnement, seule la librairie `speedo-jdo.jar` est nécessaire. Elle contient un fichier de configuration par défaut qui est probablement non opérationnel pour l'utilisateur de Speedo. Il est possible de modifier le fichier correspondant dans l'arborescence de la figure F.5. Il s'agit du fichier `speedo-jdo.properties` du répertoire `/root/Speedo-jdo_1.4.5/etc`.
- L'environnement Java pour entreprise (J2EE) : dans cet environnement, plusieurs intégrations de Speedo sont possibles. Dans tous les cas, il s'agit d'intégration sous la forme de connecteur JCA. La manière standard est d'intégrer tout le code Speedo, le

code du connecteur JCA Speedo, ainsi que tous les fichiers de configuration dans une archive .rar. C'est ce que propose le fichier `speedo-jdo_ra.rar`. Ce fichier contient notamment un fichier de configuration spécifique au serveur d'application. Deux fichiers spécifiques sont proposés : un pour Jonas et l'autre pour Weblogic de BEA. Les autres modes d'intégration ne sont pas décrits ici. Là encore, les fichiers de configuration peuvent être modifiés dans l'arborescence avant création des librairies. Il s'agit respectivement des fichiers `jonas-ra.xml` et `weblogic-ra.xml` dans le répertoire `/root/Speedo-jdo_1.4.5/etc`. Il permettent principalement de définir le nom JNDI sous lequel le connecteur Speedo est enregistré dans le serveur correspondant.

Le fichier de configuration de Speedo

Le fichier de configuration de Speedo pour JDO s'appelle `speedo-jdo.properties`. Il est utilisé dans les deux contextes d'exécution décrits précédemment, même si toutes les informations de configuration ne sont pas communes. Pour les informations communes, il s'agit à la fois d'options standard proposées par JDO et d'options spécifiques à Speedo. Le fichier est dument commenté et les valeurs proposées par défaut permettent à Speedo de fonctionner tout à fait correctement sans autres modifications.

Dans le cas d'un contexte d'exécution Java standard, il est nécessaire de configurer l'accès à la base de données. C'est généralement un accès à travers JDBC, qui requiert de spécifier la classe du pilote JDBC, ainsi que l'URL de connexion, le nom et le mot de passe de l'utilisateur permettant d'accéder à la base.

Il est bien sûr possible de gérer le fichier de configuration à l'extérieur de la librairie. Il suffit que le fichier soit accessible avant celui de la librairie au niveau du *classpath* de l'application.

Lancer une application

Une fois les librairies disponibles et les fichiers de configuration définis, il est possible de lancer une application.

```
{prompt} java -cp app_path:/root/Speedo-jdo_1.4.5/speedo-jdo.jar com.org.AppClass
```

Cette commande permet d'activer l'application Java dont le point d'entrée est défini par la classe `com.org.AppClass`. On suppose que le code de cette application est accessible par le chemin défini par `app_path`. Il peut s'agir d'un répertoire ou d'une autre archive Java.

F.4.2 Utiliser le système de persistance JDO

Comme présenté dans la section F.2.1, un objet persistant est un objet Java appartenant à une classe persistante, et qui a une représentation dans un support de stockage. Etant donné qu'il est difficile d'utiliser l'adresse de l'objet Java en mémoire pour désigner son représentant de l'espace de stockage, la notion d'identifiant est introduite pour désigner de manière unique le représentant de l'espace de stockage. Les deux principales différences

entre une instance persistante et une instance non persistante sont la possession d'un identifiant et la connectivité à la solution JDO.

L'identifiant permet de construire la liaison entre l'objet Java persistant et son représentant, autorisant ainsi les échanges de données (E/S) entre les deux espaces. L'objet Java est alors dit *connecté* au système de stockage. En effet pour le manipuler, une connexion vers le système de persistance est requise. Cette connexion permet au système de persistance (le pilote JDO, en l'occurrence Speedo) de fournir des données à jour, de gérer la concurrence d'accès et de synchroniser les modifications avec le support de stockage.

Pour la connexion à l'espace de stockage, la spécification JDO définit deux concepts importants :

- La **PersistenceManagerFactory** (PMF dans la suite) : elle représente l'espace de stockage dans lequel les objets Java persistants sont projetés (par exemple une base de données relationnelle, un répertoire de stockage de fichiers xml, etc). Une PMF est une instance d'un objet Java implémentant l'interface `javax.jdo.PersistenceManagerFactory`. Elle est fournie par l'implémentation JDO. Lorsqu'on veut transférer des données d'un espace de stockage vers un autre il faut récupérer une PMF différente pour chacun d'eux.
- Le **PersistenceManager** (PM dans la suite) : il représente une connexion avec le système de persistance. Ainsi pour manipuler des instances persistantes, il est nécessaire qu'un PM soit ouvert, ce qui permet au *thread* dans lequel le PM a été ouvert de s'exécuter dans un contexte de persistance. Un PM est une instance implantant l'interface `javax.jdo.PersistenceManager`, fournie par l'implémentation JDO. Les PMs sont gérés par et attachés à une PMF. C'est le PM qui permet d'effectuer l'essentiel des actions sur les instances de classes persistantes (par exemple, rendre persistant, récupérer, effacer ou détacher des instances).

Pour résumer, au niveau du modèle de programmation de la persistance, les deux interfaces des PMFs et des PMs correspondent aux principales APIs que doit connaître et maîtriser le programmeur. Pour le reste, il s'agit de programmation Java standard, pratiquement sans contrainte.

Récupération d'une PMF

La façon de récupérer une PMF dépend du cadre d'utilisation du pilote JDO. Dans un serveur d'application J2EE et si le driver JDO est installé comme un *resource adapter* (cf. spécification JCA) alors la PMF est récupérée par l'application en utilisant le service JNDI. Pour plus de détails sur l'intégration J2EE / JDO, il faut se référer au chapitre F.7.

Dans une application Java standard, la classe `JDOHelper` (inclus dans les APIs JDO) doit être utilisée comme dans l'exemple suivant :

```

import javax.jdo.PersistenceManagerFactory;
import javax.jdo.JDOHelper;
...
Properties p = new Properties();
ClassLoader cl = getClass().getClassLoader();
InputStream is = cl.getResourceAsStream("myJdoConf.properties");
if (is == null) {
    throw MyException("fichier de configuration introuvable");
}
p.load(is);
PersistenceManagerFactory pmf = JDOHelper.getPersistenceManagerFactory(p); .

```

Le principe de récupération d'une PMF consiste à demander au `JDOHelper` une PMF correspondant à des caractéristiques précises définies par un ensemble de propriétés. Dans l'exemple ci-dessus, les propriétés sont définies dans un fichier de configuration `myJdoConf.properties` qui est chargé via le *classpath*, en utilisant du *classloader* courant. Le `JDOHelper` maintient les associations entre les propriétés définies et la référence vers la PMF correspondante. Cela implique que plusieurs appels au `JDOHelper` avec le même ensemble de propriétés retournent la même PMF. Cependant, il est conseillé au développeur de conserver dans son application la référence à la ou les PMFs utilisées, de façon à minimiser la mise en œuvre de ce processus coûteux.

Cycle de vie des PM

Un PM est récupéré de manière très simple en utilisant une usine PMF :

```

import javax.jdo.PersistenceManager;
...
PersistenceManager pm = pmf.getPersistenceManager();
try {
    ...
} catch (...) {
    ...
} finally {
    if (pm != null && !pm.isClosed()) {
        pm.close();
    }
}
.
```

Tout PM récupéré doit absolument être fermé après utilisation et ceci quels que soient les événements survenus au cours de l'exécution (exceptions, annulation de transaction, ...). En effet, le PM mobilise généralement des ressources physiques ou des ressources critiques qui, si elles ne sont pas libérées, sont une source d'ennuis majeurs pour la suite de l'exécution de l'application (c'est une des sources majeure de problème lorsqu'on utilise JDO).

Il est donc important que les PMs soient libérés. La fermeture d'un PM se fait en appelant la méthode `close`. L'exemple précédent illustre le type de séquence de code permettant de libérer les ressources dans tous les cas.

F.4.3 Identifiants d'objet persistant

L'identifiant d'un objet persistant permet de le désigner de manière unique. Les formats d'identifiant supportés par JDO sont divisés en deux grandes catégories :

- Identifiant applicatif : Cette première catégorie contient tous les formats d'identifiant dont les valeurs sont fournies par l'application elle-même. Ces identifiant sont donc composés de données persistantes de l'objet dont les valeurs sont effectivement définies par l'application.
- Identifiant système : Cette deuxième catégorie contient tous les formats d'identifiants dont les valeurs sont indépendantes des données applicatives. Elles sont donc définies soit par l'implémentation JDO soit par le support de stockage, de manière transparente vis-à-vis du code de l'application.

La catégorie est choisie dans le descripteur de persistance JDO qui spécifie pour chaque classe persistante la catégorie d'identifiant choisie.

Identifiant applicatif

Pour un identifiant applicatif, une classe spécifique doit être définie. Elle doit être composée du même ou des mêmes champs de la classe persistante composant l'identifiant. Dans la classe identifiant, les champs doivent être déclarés `public`. Les champs composant l'identifiant peuvent être automatiquement calculés (cf. section F.4.3).

Dans l'exemple suivant, la classe `ClientId`, contenant les champs `public nom` et `prenom`, est la classe identifiant associée à la classe persistante `Client`. Il est clair qu'on retrouve les mêmes champs dans les deux classes associées.

```
class Client {
    String nom;
    String prenom;
    String adresse;
    Collection<Compte> comptes;
    Agence agence;
    ...
}

class ClientId {
    public String nom;
    public String prenom;
}
```

Dans le descripteur JDO, le nom de la classe d'identifiant associée est spécifié par l'attribut `objectid-class` au niveau de la balise de la classe. Chacun des champs persistants utilisés dans l'identifiant doit par ailleurs être marqué comme `primary-key` :

```
<class name="Client" identity-type="application" objectid-class="ClientId">
    <field name="nom" primary-key="true"/>
    <field name="prenom" primary-key="true"/>
</class>
```

Lorsque l'identifiant d'une classe persistante est composé d'un seul champ applicatif, JDO propose une simplification au développeur en fournissant des classes pré écrites pour certains types de champ. Dans ce cas là, il n'est pas nécessaire de spécifier la classe d'identifiant associée à la classe persistante car elle est déduite par le type du champ marqué unique comme `primary-key` :

```
<class name="Banque" identity-type="application">
    <field name="nom" primary-key="true"/>
</class>
```

Dans cette exemple la classe servant d'identifiant est la classe `javax.jdo.identity.StringIdentity` car le champ unique composant l'identifiant est de type `String`. Voici la définition de la classe `StringIdentity` à titre d'exemple :

```
package javax.jdo.identity;

public class StringIdentity extends SingleFieldIdentity {
    public String getKey();
    public StringIdentity(Class pcClass, String key);
}
```

Toutes les classes pouvant servir d'identifiant sont définies dans le package `javax.jdo.identity`. Le type de l'unique champ peut être : `byte`, `char`, `short`, `int`, `long` ou tout type objet. Cependant, pour des raisons de conception, il est tout à fait possible de définir sa propre classe contenant le champ unique.

Les classes définies par le développeur et servant d'identifiant pour une classe persistante sont soumises à un certain nombre de contraintes :

- Un constructeur `public` sans argument est obligatoire.
- Un constructeur avec un unique argument de type `String` est obligatoire. Le paramètre de type `String` est le résultat de la méthode `toString` et doit permettre de reconstituer un identifiant à partir de sa forme `String`.
- Les méthodes `equals` et `hashCode` doivent être implémentées en utilisant les champs de l'identifiant.
- La méthode `toString` doit être surchargée afin de renvoyer une image sous la forme d'un chaîne de caractères du contenu de l'identifiant. Le résultat de cette méthode `toString` doit pouvoir être donné au constructeur avec le paramètre `String` décrit précédemment.
- Les champs composant l'identifiant doivent être définis de la même manière dans la classe persistante et dans la classe d'identifiant. Dans la classe d'identifiant ces champs doivent être `public`.
- Pour les classes persistantes avec héritage, toutes les classes concrètes d'un arbre doivent avoir le même identifiant que la classe concrète la plus générale.
Par exemple, prenons 3 classes persistantes A, B et C telles que B et C sont des sous classes de A et `AId` est la classe d'identifiant de A :
 - Si A est concrète alors B et C doivent utiliser `AId` comme classe d'identifiant. Cependant, certaines implémentations conseillent d'utiliser des classes `BId` et `CId` pour respectivement les classes B et C où `BId` et `CId` sont des sous classes de `AId` ne contenant pas de nouveaux champs.
 - Si la classe A est abstraite, et B et C sont concrètes, alors une sous classe de A (B ou C) peut avoir une classe d'identifiant différente de `AId`, c'est à dire `BId` ou `CId`. Cette classe d'identifiant doit alors être une sous classe de `AId` et peut contenir de nouveaux champs pour composer l'identifiant.

Cette section montre bien que la manipulation d'identifiant applicatif est laborieuse. Par ailleurs, l'utilisation de tels identifiants reste difficile car le programmeur n'est pas à l'abri d'un changement des informations d'un identifiant, ce qui pose de nombreux problèmes, comme par exemple la perte de validité de toutes les références vers l'objet changeant d'identité.

Identifiant système

Les identifiants systèmes, définis par la catégorie *datastore*, regroupent tous les identifiants dont les valeurs ne sont pas attribuées par l'application. La valeur de l'identifiant n'est pas exposée dans un champ de la classe persistante. Il faut utiliser les APIs de JDO, notamment le PM pour récupérer un tel identifiant. La stratégie de calcul de l'identifiant se définit dans le descripteur JDO au moyen de la balise **datastore-identity** comme dans l'exemple suivant :

```
<class name="Compte" identity-type="datastore">
    <datastore-identity strategy="uuid-hex"/>
</class>
```

Dans cet exemple, la stratégie est *uuid-hex*, c'est à dire un identifiant de type UUID sous la forme hexadécimale. Les stratégies possibles sont détaillées dans la section qui suit. Il est clair que l'utilisation de type d'identifiant est beaucoup plus simple pour le programmeur car elle offre une transparence totale : pas de champ d'identifiant à déclaré, pas de valeurs à gérer. Par ailleurs, le système garantit l'immutabilité des identifiants de ce type, évitant les problèmes d'incohérence comme celui de la validité des références.

Stratégie de génération de valeur d'identifiant

Les champs persistants d'une classe d'identifiant ou les identifiants système peuvent recevoir une valeur générée. La génération de la valeur peut suivre sept stratégies possibles :

- **native** : L'identifiant est choisi par l'implémentation JDO.
- **identity** : L'identifiant est choisi par le support de stockage lui-même. Par exemple, on peut vouloir utiliser directement l'identifiant fourni par une base de données à objets.
- **sequence** : L'identifiant est calculé en utilisant une séquence. La séquence peut être une séquence générée par le support de stockage (par exemple une séquence SQL) ou par le pilote JDO lui-même.
- **autoassign** : L'identifiant est projeté sur une colonne SQL auto incrémentée.
- **increment** : L'identifiant est une valeur entière calculée à partir de la valeur maximum des identifiants des autres instances d'une classe. Cet identifiant ne doit être utilisé que lorsque l'application JDO est la seule à insérer des nouvelles données en base. En effet, pour des raisons de performance, l'implémentation JDO ne calcule que la valeur maximum qu'au démarrage. Elle gère cette valeur en mémoire par la suite.
- **uuid-string** : L'identifiant est une chaîne de 16 caractères générés par l'implémentation JDO. La valeur générée est unique et est au format UUID.

- **uuid-hex** : L'identifiant épouse le même principe que `uuid-string` mais sa longueur est de 32 caractères (chaque caractère de la chaîne UUID est encodé sous la forme de deux caractères hexadécimaux).

L'exemple suivant montre comment spécifier une stratégie pour le calcul automatique de la valeur d'un champ d'identifiant :

```
<class name="Compte" identity-type="application">
    <field primary-key="true" value-strategy="uuid-hex"/>
</class>
```

Notons que l'utilisation de ces générateurs peut se faire en dehors de la gestion de valeurs d'identifiant. Elle peut se faire sur des champs persistants normaux, c'est à dire n'appartenant pas à un identifiant.

F.4.4 Rendre un objet persistant

Un objet Java appartenant à une classe persistante est non persistant à sa création par le programmeur. Deux situations différentes vont amener à ce qu'il devienne effectivement persistant :

- Le programmeur affecte dans un objet qui est déjà persistant une référence (ou relation) vers un objet qui ne l'est pas. L'objet ainsi référencé devient alors persistant et par transitivité tous les objets qu'il référence lui-même. Nous parlons alors de *persistiance par attachement*.
- Le programmeur utilise les APIs JDO pour rendre l'objet persistant. Nous parlons dans ce cas de *persistiance explicite*. Dans ce cas, le même principe de transitivité que décrit précédemment s'applique.

Lorsqu'une instance devient persistante, elle obtient un identifiant. Dans le cas d'identifiant applicatif, il est nécessaire que le ou les champs composant l'identifiant aient été préalablement affectés par l'application. Les champs objet doivent notamment être non nul. Dans le cas d'un identifiant système, l'implémentation JDO calcule la valeur de l'identifiant au moment où l'objet devient persistant. Selon la catégorie d'identifiant système, ce processus peut mettre en œuvre un accès au support de stockage (par exemple lors de l'utilisation d'une séquence SQL).

Persistiance explicite

La persistiance explicite la méthode `makePersistent` de l'interface `PersistenceManager`. L'exemple ci-dessous montre comment rendre explicitement persistant une instance :

```

import javax.jdo.PersistenceManager;
...
PersistenceManager pm = pmf.getPersistenceManager();
Banque b = new Banque("Banque de France");
// ici 'b' n'est pas persistant : c'est un objet Java normal
pm.currentTransaction().begin();
pm.makePersistent(b);
// ici 'b' est devenu persistant
pm.currentTransaction().commit();
// ici 'b' est persistant de manière "durable"
pm.close();
.

```

Dans l'exemple, l'instance `b` est devenue persistante suite à l'appel à la méthode `makePersistent`. Cependant, elle peut être persistante, avoir un identifiant et ne pas être écrite sur le support de stockage. JDO impose seulement que toutes les actions aient été propagées sur le support au plus tard à la validation de la transaction. Si la transaction est annulée (`pm.currentTransaction.rollback()`) alors l'instance n'est plus persistante et aucune donnée relatif à cet objet n'existe dans le support de stockage. L'exemple ci-dessus montre la forme la plus simple pour rendre persistant une instance. L'interface `PersistenceManager` propose d'autres méthodes pour rendre persistant des objets par paquets : il s'agit des différentes déclinaisons de `makePersistentAll`.

Persistance par attachement

La persistance par transitivité, ou par attachement n'utilise pas l'API JDO. Comme on l'a dit, le principe est d'affecter une instance non encore persistante à un champ d'une instance déjà persistante (cas d'une relation monovaluée entre objets persistants). Cela peut aussi se faire de façon indirecte dans le cas de relations multi-valuées. En effet, dans ce cas, l'objet référençant référence d'abord un objet *collection* qui référence à son tour les objets qui deviennent persistants. L'objet *collection* intermédiaire n'est pas considéré comme un objet persistant (légère perte d'orthogonalité vis-à-vis du langage Java). L'exemple suivant montre un tel cas de propagation de la persistance à travers la relation multi-valuée `agences` qui est une collection qui référence toutes les agences d'une banque (définie par la classe `Banque`) :

```

import javax.jdo.PersistenceManager;
...
PersistenceManager pm = pmf.getPersistenceManager();
Banque b = new Banque("Banque de France");
pm.currentTransaction().begin();
pm.makePersistent(b);
// ici 'b' est persistant
Agence a = new Agence("agence principale");
// ici 'a' n'est pas persistant
b.getAgences().add(a);
// ici 'a' est devenu persistant
pm.currentTransaction().commit();
pm.close();
.

```

L'instance `a` est devenue persistante au moment où elle a été ajoutée dans la collection `agences` possédée par l'instance persistante `b`. Le fonctionnement est identique pour une référence simple.

La transitivité de la propriété de persistance est aussi valable pour les objets référencés par une instance qui devient persistante de manière explicite ou par attachement. Ainsi lorsqu'une instance devient persistante, c'est toute la grappe d'objets atteignables par cet objet, qui le devient :

```
import javax.jdo.PersistenceManager;
...
PersistenceManager pm = pmf.getPersistenceManager();
Banque b = new Banque("Banque de France");
Agence a = new Agence("agence principale");
b.getAgences().add(a);
// ici ni 'b' ni 'a' ne sont persistants
pm.currentTransaction().begin();
pm.makePersistent(b);
// ici 'b' et 'a' sont devenus persistants
pm.currentTransaction().commit()
pm.close();
```

Le modèle de persistance est donc relativement orthogonal par rapport au langage Java. La seule entorse au modèle est que les collections d'objets persistants ne sont pas des objets partageables comme les autres objets Java dans le cadre de la gestion de relations multi-valuées persistantes.

F.4.5 Détruire un objet persistant

La destruction d'objet persistant est toujours explicite dans le cadre de JDO. En effet, il n'y a pas au niveau de l'espace de stockage de ramasse-miettes comme le gère Java en mémoire.

L'API JDO fournit deux moyens pour détruire des objets persistants :

- La destruction explicite : L'API JDO fournit des méthodes permettant de détruire un objet persistant ou une collection d'objets persistant et cela en désignant de manière explicite les objets en question. Le résultat d'un telle opération est généralement de produire autant de requêtes de destruction vers la base de données que d'objets à détruire.
- La destruction ensembliste : Pour des raisons d'efficacité, JDO propose aussi un mécanisme de destruction d'objets persistants basé sur des requêtes associatives. Comme pour les requêtes de sélection d'objets (cf. section F.5), il s'agit de définir les objets à détruire à base d'une requête ensembliste associative. Cette approche permet de détruire des ensembles d'objets persistants en une seule requête vers la base de données.

Une fois supprimé, un objet persistant n'est plus connecté au système et perd son identifiant. La suppression doit être réalisée dans une transaction. De plus, la suppression est concrètement effectuée dans le support de persistance au moment de la validation de la transaction ou lors de l'éviction de l'objet de la mémoire.

La suppression en cascade, c'est à dire propager la destruction à travers les relations, n'est pas couverte par la spécification. Mais la suppression en cascade peut être réalisée par l'utilisation d'un *Listener* ou d'un *CallBack* permettant sur surveiller le cycle de vie d'un objet. De plus, certaines implémentations offrent la fonction de 'cascade delete' au moyen d'une extension dans le descripteur de persistance.

Destruction explicite

La suppression explicite repose sur l'utilisation des méthodes `deletePersistent` et `deletePersistentAll` de l'interface `javax.jdo.PersistenceManager`. Ces méthodes prennent en paramètre la ou les instances persistantes à supprimer. Voici un exemple simple :

```
import javax.jdo.PersistenceManager;
...
PersistenceManager pm = pmf.getPersistenceManager();
Banque b = new Banque("Banque de France");
pm.currentTransaction().begin();
pm.makePersistent(b);
pm.currentTransaction().commit();
pm.close();
// ici b designe un objet persistant (present dans l'espace de stockage)
...
pm = pmf.getPersistenceManager();
pm.currentTransaction().begin();
pm.deletePersistent(b);
pm.currentTransaction().commit();
pm.close();
// ici l'objet designe par b n'a plus de representant dans l'espace de stockage
// en memoire b reference un objet Java non persistant representant une banque
// avec les valeurs qu'avait l'objet persistant avant sa destruction
```

Les méthodes `deletePersistentAll` sont l'équivalent des méthodes `makePersistentAll` pour la destruction et prennent en paramètre des collections d'objets persistants à détruire.

Destruction ensembliste

La suppression à l'aide des requêtes consiste à définir une requête JDO désignant l'ensemble des objets persistants à supprimer. Cette fonction de la spécification JDO 2 n'est actuellement pas supportée par Speedo.

F.4.6 Relations et cohérence

Ayant passé en revue le modèle de programmation de la persistance, nous observons qu'il n'y a pas une transparence totale de la persistance par rapport à Java. Deux points doivent être bien intégré par le programmeur JDO car il représente un danger du point de vue de la vision de la cohérence des données.

Référencement d'objets persistants détruits

Le premier point concerne les références vers des objets détruits. Contrairement à Java, les destructions d'objets persistants sont pilotées directement par le programmeur. Il est ainsi possible de détruire un objet persistant alors qu'il est référencé par d'autres objets persistants. Deux cas se présentent :

- L'objet détruit n'est référencé que par des objets persistants ayant une image dans le support de stockage mais n'étant pas représentés en mémoire. Lors de la prochaine activation de ces objets en mémoire, les références devenues invalides à la destruction de l'objet référencé sont transformées en pointeurs nul. On se retrouve alors avec les mêmes comportement que lorsqu'un programmeur utilise une référence Java nulle (par exemple, génération d'une `NullPointerException` lors d'une tentative de déréférencement).
- L'objet détruit est référencé en mémoire par des objets Java persistants. Dans ce cas, les liens vers l'objet détruit reste valide jusqu'à la validation de la transaction dans laquelle la destruction intervient. Dans tout autre contexte transactionnel (notamment les subséquents), on retombe sur la sémantique décrite dans le point précédent.

Même si ce comportement n'est pas en ligne avec Java, la sémantique reste néanmoins claire. Il reste que ce comportement doit être bien compris par le programmeur JDO.

Cohérence des relations bidirectionnelles

Lorsque des objets se réfèrent mutuellement dans le cadre de relations bidirectionnelles, dans un cadre Java normal, c'est au programmeur de se soucier de gérer la cohérence de celles-ci. Il se trouve que dans le cadre de projection vers le support de stockage, notamment les bases de données relationnelles, cette cohérence est souvent implicite car les modes de projection (cf. section F.6) permettent de représenter les deux liens associées au travers d'un seule information.

Dans la première version de la spécification JDO, il était possible de définir une option qui impliquait que cette cohérence en mémoire était gérée par la solution JDO. Speedo support cette possibilité mais elle n'est dorénavant plus standard. C'est bien au programmeur de garantir cette cohérence et il ne paraît pas judicieux de s'appuyer sur une telle possibilité offerte par une solution JDO si elle n'est plus standard, puisqu'elle impacte directement la portabilité du code JDO.

F.4.7 Fonctions avancées

Dans un modèle de données, les classes persistantes sont reliées par des références. Le graphe peut être connexe ou partiellement connexe. Lorsque l'application requiert une instance persistante, elle peut souvent nécessiter en plus de ce simple objet, un graphe d'objet connexe à cet objet. La spécification JDO 2 permet de préciser ce graphe par la notion de *fetch-plan*, afin par exemple d'optimiser le chargement des données. Un *fetch-plan* se définit par un ensemble de *fetch-groups*.

Les *fetch-groups*

Un *fetch-group* est toujours associé à une classe et définit l'ensemble des champs qui doivent être chargés. Un *fetch-group* peut aussi se définir à partir d'un autre *fetch-group*. Deux *fetch-groups* existent par défaut : `all` et `default`. Le *fetch-group* `default` charge les champs primitifs seulement, tandis que le *fetch-group* `all` charge tous les champs. Les *fetch-groups* se définissent dans le descripteur de persistance JDO. Un *fetch-group* est nommé. La portée de ce nom est globale. Il est donc important de choisir des noms de *fetch-groups* complètement qualifiés.

Voici le modèle de donnée utilisé pour les exemples de *fetch-group* :

```
public class Personne {
    String nom;
    Adresse adresse;
    Set enfants;
}

public class Adresse {
    String street;
    String ville;
    Pays pays;
}

public class Pays {
    String code;
    String nom;
}
```

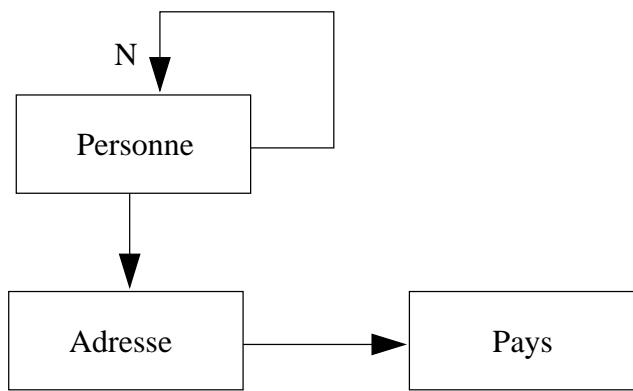


Figure F.6 – Modèle de données pour les exemples de *fetch-group*

Exemple : *fetch-group* de base

```
<class name="Personne" ...>
  ...
  <fetch-group name="detail">
    <fetch-group name="default">
      <field name="adresse"/>
      <field name="adresse.pays.code"/>
    </fetch-group name="detail">
  </class>
```

Ce *fetch-group* s'appelle *detail* et est associé à la classe *Personne*. Il désigne :

- le champ primitif *nom* via le *fetch-group default*,
- la référence vers l'adresse de la personne,
- le champ *code* du pays correspondant à l'adresse (cela inclus aussi l'objet *Pays*).

L'exemple ci-dessus montre que les clauses *field* peuvent désigner un champ de classe, mais aussi un chemin dans le graphe. Le chemin est défini par la succession de noms de champ référence séparés par le caractères `'.'`. L'accès aux éléments d'un champ multi-valué (Collection, Set, List, Map, ..) se fait en utilisant des mots clés :

- `#element` pour accéder aux éléments d'une collection,
- `#key` et `#value` pour accéder à une *Map*.

Exemple : *fetch-group* avec une collection

```
<fetch-group name="monfg">
  <fetch-group name="detail">
    <field name="enfants#element.nom"/>
  </fetch-group name="detail">
```

Ce *fetch-group* s'appelle *monfg*. Il comprend le *fetch-group detail* et le champ *nom* des éléments de l'ensemble des enfants de la personne.

Afin de traiter les cas de récursivité dans les modèles de données, l'attribut *fetch-depth* définit la profondeur à laquelle le *fetch-group* doit s'appliquer.

Exemple : *fetch-group* avec une collection

```
<fetch-group name="monfg2">
  <fetch-group name="detail">
    <field name="enfants" fetch-depth="1"/>
  </fetch-group name="detail">
```

Ce *fetch-group* s'appelle *monfg2*. Il comprend le *fetch-group detail* et les enfants de la personne. Les petits enfants ne seront pas pris en compte.

Les *fetch-plans*

Un unique *fetch-plan* est associé à chaque *PersistenceManager*. Il est utilisé pour les requêtes, les méthodes *getObjectById*, *refresh* et *detach*. Le *fetch-plan* peut être redéfini à tout moment par l'application. Un *fetch-plan* peut contenir des *fetch-groups* de différentes classes. Les *fetch-groups* peuvent se composer ou plus exactement s'additionner.

L'interface *javax.jdo.FetchPlan* offre les méthodes suivantes :

```

FetchPlan addGroup(String fetchGroupName);
FetchPlan removeGroup(String fetchGroupName);
Set<String> getGroups();
FetchPlan setGroups(Collection<String> fetchGroupNames);
FetchPlan setGroups(String[] fetchGroupNames);
.

```

Ces cinq méthodes permettent de définir le contenu du *fetch-plan* en listant les noms des fetch-groups qui le composent.

L'attribut `fetchsize` définit le nombre maximal d'instances qui seront récupérées par le *fetch-plan*. Deux valeurs particulières sont définies par la spécification. `FETCH_SIZE_GREEDY` indique que toutes les instances désignées par le *fetch-plan* doivent être récupérées immédiatement. `FETCH_SIZE_OPTIMAL` indique que l'implémentation est libre d'optimiser le chargement des instances désignées.

L'attribut `DetachementOptions` permet de définir le comportement du détachement vis-à-vis des champs persistants. `DETACH_LOAD_FIELDS` indique que les champs non encore chargés mais inclus dans le *fetch-plan* doivent être chargés. `DETACH_UNLOAD_FIELDS` indique que les champs non inclus dans le *fetch-plan* doivent être déchargés.

F.5 Requêtes ensemblistes

F.5.1 Introduction

Les requêtes JDO permettent de récupérer des données du support de persistance, structurées sous forme d'objets. Dans le cas général, les requêtes renvoient des objets du modèle de données défini par les descripteurs de persistance. Cependant, il est possible de spécifier de nouveaux objets pour accueillir le résultat.

Les requêtes sont définies dans le code de l'application ou dans le descripteur de persistance. Lorsqu'elles sont définies dans le code de l'application, il est possible de construire dynamiquement des requêtes, ce qui apporte une grande puissance d'expression.

Voici un exemple de requête JDO définie et exécutée dans le code de l'application :

```

import javax.jdo.PersistenceManager;
import javax.jdo.Query;
import java.util.Iterator;
import java.util.List;
...
PersistenceManager pm = pmf.getPersistenceManager();
pm.currentTransaction().begin();

Query q = pm.newQuery(Client.class);
q.setFilter("this.adresse.endsWith(suffix)");
q.declareParameters("String suffix");

try {
    List l = (List) q.execute("Paris");
    for (Iterator it = l.iterator(); it.hasNext();) {
        Client c = (Client) it.next();
        System.out.println("client: " + c);
        c.setAdresse(c.getAdresse() + " France");
    }
} finally {
    q.closeAll();
}

pm.currentTransaction().commit();
pm.close();
.
.
```

La requête sélectionne les instances de la classe `Client` (et ses sous-classes) dont l'`adresse` se termine par une chaîne de caractères spécifiée en paramètre. La valeur du paramètre est "Paris". L'utilisation d'un bloc `try finally` permet de s'assurer de la fermeture des itérateurs ouverts sur la collection résultante. En effet, c'est le moyen d'indiquer au pilote JDO qu'il peut relâcher les ressources liées à cette requête (`resultSet`, `connection`, ...).

Une requête JDO est un objet qui implémente l'interface `javax.jdo.Query`. Une instance de `Query` se récupère par l'interface `PersistanceManager` via les méthodes `newQuery`. La requête est à définir/personnaliser en utilisant les méthodes `setXXX`. Les méthodes `execute` lancent l'évaluation de la requête et retournent par défaut une liste d'objet (objet de type `java.util.List`). Les objets rentrés par la requête sont sensibles au *fetch plan* courant afin de récupérer des graphes d'objets.

Les requêtes JDO peuvent être utilisées classiquement dans une transaction, mais aussi en dehors d'une transaction si la propriété `nonTransactionalRead` est activée (voir la section sur les transactions).

Les attributs d'une requête JDO sont les suivants :

Méthode	Description
setClass(Class)	La classe servant de domaine initial d'interrogation. Par défaut, c'est aussi la classe résultat. Par défaut les sous classes sont incluses.
setFilter(String)	String définissant une expression booléenne comme filtre de sélection. C'est un peu l'équivalent d'une clause WHERE en SQL.
declareParameters(String)	String déclarant le nom et le type des paramètres utilisés dans le filtre. Le séparateur de paramètre est la virgule. L'ordre de déclaration correspond exactement à l'ordre des paramètres dans des méthodes 'execute'.
declareVariables(String)	String déclarant les variables utilisées dans le filtre. Les variables servent principalement à désigner le contenu d'une collection. Le séparateur entre les déclarations est le point virgule
declareImports(String)	String déclarant le nom complet des classes. Cela permet d'utiliser le nom court des classes pour la déclaration des variables, des paramètres ou lors d'opération de cast. Le séparateur d'import est le point-virgule.
setResultClass	La classe accueillant la projection. Par défaut, le résultat est la classe initiale.
setOrdering(String)	Ordonnancement des résultats
setResult(String)	La projection de la requête. Par défaut, la projection est composée des instances de la classe initiale ("DISTINCT this"). Cette chaîne de caractère permet de définir une projection plus complexe.
setGroupBy(String)	Définit la clause comment les résultats doivent être groupés pour les opérateurs d'agrégation.
setUnique(boolean)	Indique si le résultat est composé d'un seul élément. Si oui alors l'objet renvoyé par la méthode execute est directement la valeur. Si non alors la méthode execute retourne une java.util.List des résultats.
setRange(long pre, long der)	Permet de limiter la cardinalité du résultat et de faire de la pagination. Le premier entier un indique l'index du premier élément à sélectionner. Le deuxième entier indique la position du premier élément non sélectionné.
setIgnoreCache(boolean)	Indique si les données modifiées dans le cache L1 doivent être prises en compte dans la requête.
setUnmodifiable()	Permet de rendre la requête non modifiable.

F.5.2 Requêtes prédefinies

Les requêtes peuvent être définies dans le descripteur de persistance sous forme d'une seule chaîne de caractères dans une balise `query`. Toute requête définie dans le descripteur de persistance est nommée afin de pouvoir être utilisée depuis l'application. Voici un exemple de déclaration :

```
<class name="Client" identity-type="application" objected-class="ClientId"
      table="T_CLIENT">
    <field name="nom" primary-key="true" column="C_NOM"/>
    <field name="prenom" primary-key="true" column="C_PRENOM"/>
    <field name="adresse" column="C_ADRESSE"/>
    <query name="maRequete1">
        [&!]DATA[
            SELECT WHERE adresse.endsWith(:prefix)
        ]
    </query>
</class>
```

Et son utilisation dans un bout d'application via la méthode `newNamedQuery` :

```
PersistenceManager pm = pmf.getPersistenceManager();
pm.currentTransaction().begin();
Query q = pm.newNamedQuery(Client.class, "maRequete1");
try {
    List l = (List) q.execute("Paris");
    for(Iterator it = l.iterator(); it.hasNext();) {
        Client c = (Client) it.next();
    }
} finally {
    q.closeAll();
}
pm.currentTransaction().commit();
pm.close();
```

Voici la grammaire exacte pour la chaîne de caractères représentant une requête au format chaîne de caractères :

```
SELECT [ UNIQUE ] [<résultat>] [ INTO <nom de classe résultat> ]
[ FROM <nom de classe> [exclude subclasses]
[ WHERE <filtre> ]
[ VARIABLES <clause variables> ]
[ PARAMTERS <clause paramètres> ]
[ IMPORTS <clause imports> ]
[ GROUP BY <clause group by> ]
[ ORDER BY <clause order by> ]
[ RANGE <index premier>, <index dernier> ] .
```

Tous les mots clés doivent être écrits soit tout en majuscule, soit tout en minuscule. L'ordre des clauses défini par la grammaire doit être respecté. La définition d'une requête sous forme d'une chaîne de caractères est identique à la définition d'une requête en utilisant

les méthodes `set...` de l'interface `Query`. Cela signifie que les valeurs des différentes clauses de la grammaire ci-dessus, correspondent exactement aux valeurs mises par les méthodes `set...` de l'interface `Query`.

F.5.3 Filtre de requête

Le filtre de requête représente une expression booléenne définissant un filtre de sélection du résultat. Le langage d'expression des filtres est très proche de la syntaxe Java. Les opérateurs utilisables sont les suivants :

Comparateur	<code>==, !=, <, <=, >, >=</code>
Opérateurs arithmétiques	<code>+, -, *, /, %</code>
Opérateurs booléens	<p><code>!</code> est le NON logique <code>&</code> est un ET logique (et non un ET bit à bit) <code>&&</code> est un ET conditionnel <code> </code> est un OU logique (et non un OU bit à bit) <code> </code> est un OU conditionnel</p>

Les opérandes de l'expression sont des chemins dans les graphes de données ou des paramètres de la requête. Les chemins débutent par la classe initiale ou par une variable et se terminent par un champ persistant. La navigation prend en compte le polymorphisme des classes. Pour les champs multi-valués (collection, list, map, ...), il faut utiliser une variable afin de nommer le contenu. Il est aussi possible d'appeler certaines méthodes :

Méthode	Description
contains(Object)	S'applique aux types Collection pour définir des variables ou des conditions d'appartenance.
– get(Object), – containsKey(Object), – containsValue(Object),	S'applique aux types Map
isEmpty()	S'applique aux types Collection et Map afin de définir une condition
size()	S'applique aux types Collection et Map afin d'accéder à la taille
– toLowerCase(), – toUpperCase(), – indexOf(String), – indexOf(String, int), – substring(int), – substring(int,int), – startsWith(String), – endsWith(String), – matches(String)	S'applique au String. Pour la méthode match, il est possible d'utiliser des expressions régulières mais réduites aux symboles suivants : – '.' représente un caractère indéfini – ".*" représente une chaîne de caractère indéfinie – "?i" indique que la comparaison ne doit pas être sensible à la casse.
Math.abs(numeric)	Valeur absolue
Math.sqrt(numeric)	Racine carré

Ci-dessous, une série d'exemples de filtre permettant de mieux comprendre. Voici le modèle de données sur lequel sont basés ces exemples :

```
package com.xyz.hr;
class Employee {
    String name;
    float salary;
    Department dept;
    Employee boss;
}

package com.xyz.hr;
class Department {
    String name;
    Collection emps;
}
.
```

Exemple 1 : Comparaison simple de float

Sélection des employées dont le salaire est supérieur à 30000 :

```
Query q = pm.newQuery (Employee.class, "salary > 30000");
Collection emps = (Collection) q.execute();
.
```

La requête correspondante sous la forme d'une chaîne de caractères est la suivante :
select where salary > 30000.

Exemple 2 : Comparaison à un paramètre de type float

Sélection des employées dont le salaire est supérieur à un paramètre implicite :

```
Query q = pm.newQuery (Employee.class, "salary > :sal");
Collection emps = (Collection) q.execute(new Float(30000));
.
```

La requête correspondante sous la forme d'une chaîne de caractères est la suivante :
`select where salary > :sal.`

Exemple 3 : Comparaison à un paramètre de type Department

Sélection des employés appartenant à un département :

```
Query q = pm.newQuery (Employee.class, "dept == :d");
Department d = ... ;
Collection emps = (Collection) q.execute(d);
.
```

La requête correspondante sous la forme d'une chaîne de caractères est la suivante :
`select where dept == :d.`

Exemple 4 : Navigation simple

Sélection des employées appartenant à un département :

```
Query q = pm.newQuery (Employee.class, "dept.name == :dn");
Collection emps = (Collection) q.execute("R&D");
.
```

La requête correspondante sous la forme d'une chaîne de caractères est la suivante :
`select where dept.name == :dn.`

Exemple 5 : Navigation sur un champ collection

Sélection des départements dont au moins un des employés a un salaire supérieur à 30000 :

```
Query q = pm.newQuery(Department.class,
                      "this.emps.contains(e) && e.salary > :sal");
Collection emps = (Collection) q.execute(new Float(30000));
.
```

La requête correspondante sous la forme d'une chaîne de caractères est la suivante :
`select where emps.contains(e) && e.salary > :sal.`

On note ici l'utilisation de la fonction `contains` sur la collection `emps` afin de définir la variable `e`. Le type de la variable `e` est le type des éléments de la collection `emps`, c'est à dire `Employee`. La requête renvoie une liste d'instance de `Department` sans doublon. En effet, la sémantique de `contains` est *il existe*. Cela signifie qu'un `contains` avec une variable en paramètre se traduit généralement par un *IN* SQL avec une requête imbriquée.

Exemple 6 : Appartenance à une collection

Sélection des départements dont le nom est contenu dans un paramètre de type collection.

```
Query q = pm.newQuery (Department.class, ":depts.contains(name)");
List depts = Arrays.asList(new String [] {"R&D", "Sales", "Marketing"});
Collection deps = (Collection) q.execute(depts); .
```

La requête correspondante sous la forme d'une chaîne de caractères est la suivante :
`select where :depts.contains(name).`

On note ici l'utilisation de la fonction `contains` sur le paramètre `depts` de type collection de `String`. Cette requête est en fait un raccourci syntaxique évitant l'écriture fastidieuse d'une comparaison entre le champ `name` et les différents éléments de la collection.

F.5.4 Cardinalité du résultat

Les requêtes JDO peuvent renvoyer un ou plusieurs objets. Les APIs JDO permettent à l'utilisateur de spécifier la cardinalité du résultat grâce à la méthode `setUnique`. Ainsi, lorsque la requête est marquée avec `setUnique(true)`, cela signifie qu'il y a un seul résultat. Par conséquent, afin de simplifier l'utilisation de l'API, la méthode `execute` renvoie directement la valeur attendue ou `null` si la requête ne produit aucun résultat. S'il y a plus d'une valeur alors une exception `JDOUserException` est générée.

Voici un exemple sélectionnant un employé par son nom. L'application s'attend à trouver un seul employé pour un nom donné même si ce n'est pas l'identifiant :

```
Query q = pm.newQuery (Employee.class, "name == :n");
q.setUnique(true);
Employee e = (Employee) q.execute("Smith"); .
```

Une requête JDO peut renvoyer un très grand nombre de résultats. Une quantité de données trop importante peut réduire considérablement les performances. Pour éviter cela, les APIs JDO permettent de limiter la taille du résultat en spécifiant une fenêtre sur les éléments requis du résultat. Cette fenêtre est définie par l'index du premier élément requis et l'index du dernier. Ce mécanisme permet en particulier de faire de la pagination de résultats en exécutant la requête pour chaque page avec les index correspondants.

Pour définir les limites du résultat il faut spécifier l'attribut 'range' de la requête : `setRange(long from, long toExcl)`. La première valeur est l'index du premier élément à retourner, tandis que la deuxième valeur est l'index du premier élément à ne pas sélectionner. Ces index suivent exactement les principes des index des tableaux en Java. Cependant les limites minimum et maximum sont respectivement 0 et `Long.MAX_VALUE`.

Voici un exemple de requête affichant la liste des noms des employés par page :

```

public List getNomsEmployes(int indexPage, int taillePage) {
    int premier = indexPage * taillePage;
    PersistenceManager pm = pmf.getPersistenceManager();

    Query q = pm.newQuery (Employee.class);
    q.setResult("this.name");
    q.setRange(premier, premier + taillePage);
    List noms = new ArrayList((List) q.execute());
    q.closeAll();
    pm.close();
    return noms;
}

```

F.5.5 Résultat complexe

Depuis la version 2.0 de la spécification JDO, il est possible d'écrire des requêtes dont le résultat ne contient pas forcément des instances de classes persistantes. Il est donc possible de spécifier une clause SELECT plus évoluée que simplement "DISTINCT this". La méthode `setResult` de l'interface `Query` permet de spécifier cette clause select.

L'exemple ci-dessous montre comment sélectionner une seule valeur, ici le nom de l'employé. La requête renvoie donc la liste des noms des employés :

```

Query q = pm.newQuery (Employee.class);
q.setResult("name");
Collection names = (Collection) q.execute();
for(Iterator it = names.iterator(); it.hasNext();) {
    String name = (String) it.next();
}

```

Lorsque le résultat contient des doublons, il est possible de les éliminer en ajoutant le mot clé `DISTINCT` de la manière suivante : `q.setResult("DISTINCT name")`.

Quand le résultat de la requête est composé d'une seule colonne, La collection renvoyée par la méthode `execute` contient directement les valeurs. En revanche, si le résultat est un n-uplet composé de plusieurs colonnes, la collection renvoyée contient soit un tableau d'objets, soit une instance de classe spécifiée par la méthode `setResultClass`.

Voici un premier exemple ne spécifiant pas de classe de résultat :

```

Query q = pm.newQuery (Employee.class, "dept.name == deptName");
q.declareParameters ("String deptName");
q.setResult("name, salary, boss");
Collection names = (Collection) q.execute("R&D");
Iterator it = names.iterator();
while (it.hasNext()) {
    Object[] os = (Object[]) it.next();
    String name = (String) os[0];
    Float sal = (Float) os[1];
    Employee boss = (Employee) os[2];
    ...
}

```

La collection contient des tableaux d'objets. L'ordre dans le tableau correspond à l'ordre dans la clause result.

Voici le même exemple spécifiant la classe `Info` comme conteneur de résultat :

```
class Info {
    public String name;
    public Float salary;
    public Employee reportsTo;
}

Query q = pm.newQuery (Employee.class, "dept.name == deptName");
q.declareParameters ("String deptName");
q.setResult("name, salary, boss as reportsTo");
q.setResultClass(Info.class);
Collection names = (Collection) q.execute("R&D");
Iterator it = names.iterator();
while (it.hasNext()) {
    Info info = (Info) it.next();
    String name = info.name;
    Employee boss = info.reportsTo;
    ...
}
q.closeAll(); .
```

On note la correspondance entre les noms des colonnes dans le résultat avec les noms des champs dans la classe résultat. Dans l'exemple, afin de garantir cette correspondance, l'utilisation d'un alias est nécessaire pour la troisième colonne.

Une variante de cette solution consiste à définir un constructeur dans la classe résultat `Info` et de l'utiliser dans la clause result :

```
class Info {
    public String name;
    public Float salary;
    public Employee reportsTo;
    public Info(String n, Float s, Employee e) {
        this.name = n ; this.salary = s ; this.reportsTo = e;
    }
}
...
q.setResult("new Info(name, salary, boss)");
q.setResultClass(Info.class); .
```

En résumé, les classes résultantes respectent l'une des deux contraintes suivantes :

- Il existe un constructeur dont la signature est compatible avec les types des colonnes du résultat.
- Pour chaque nom de colonne du résultat, une des trois sous-contraintes qui suit est vérifiée :
 - Il existe un champ `public` de même nom et de même type.
 - Il une méthode `setter` correspondante.

- Il existe une méthode `put(Object name, Object value)` traitant ce champ. Ce dernier cas permet d'utiliser une `Map` comme conteneur de résultat.

Voici un tableau résumant la forme du résultat en fonction des attributs `result`, `resultClass` et `unique` :

setResult	SetResultClass	setUnique	Forme du résultat
-----------	----------------	-----------	-------------------

F.5.6 Agrégats et *group by*

Depuis la version 2.0 de la spécification JDO, il est possible d'écrire des requêtes utilisant des opérateurs d'agrégation comme le calcul de somme ou de moyenne. Ces opérateurs peuvent être utilisés dans la clause 'result'. Les opérateurs possibles sont `COUNT`, `MIN`, `MAX`, `SUM` et `AVG`.

Voici un exemple calculant la moyenne des salaires d'un département :

```
Query q = pm.newQuery(Employee.class, "dept.name == deptName");
q.declareParameters("String deptName");
q.setResult("avg(salary)");
Float avgSalary = (Float) q.execute("R&D");
.
```

Dans cet exemple, l'agrégation agissant sur tous les employés, la méthode `execute` renvoie automatiquement la valeur sans passer par une liste car l'unicité du résultat est garantie.

Il est possible d'utiliser des opérateurs d'agrégation dans plusieurs expressions de la clause `result`, comme dans l'exemple suivant :

```
Query q = pm.newQuery(Employee.class, "dept.name == deptName");
q.declareParameters("String deptName");
q.setResult("avg(salary), sum(salary)");
Object[] avgSum = (Object[]) q.execute("R&D");
Float average = (Float) avgSum[0];
Float sum = (Float) avgSum[1];
.
```

Comme dans SQL, il est possible de grouper les résultats pour les calculs d'agrégats. Ainsi voici un exemple donnant la moyenne des salaires par département :

```
Query q = pm.newQuery(Employee.class);
q.setResult("avg(salary), dept.name");
q.setGrouping("dept.name");
Collection results = (Collection) q.execute();
Iterator it = results.iterator(); it.hasNext();) {
    Object[] info = (Object[]) it.next();
    Float average = (Float) info[0];
    String deptName = (String) info[1];
    ...
}
.
```

Il est possible de filtrer le résultat obtenu après le `group by`. Voici un exemple renvoyant les moyennes des salaires par département ayant plus d'un employé :

```

Query q = pm.newQuery(Employee.class);
q.setResult("avg(salary), dept.name");
q.setGrouping("dept.name having count(dept.name) > 1");
Collection results = (Collection)q.execute();
for (Iterator it = results.iterator(); it.hasNext();) {
    Object[] info = (Object[]) it.next();
    Float average = (Float) info[0];
    String deptName = (String) info[1];
    ...
}
.
.
```

Cette section donne un aperçu de l'étendue des possibilités offertes par le langage de requête de JDO (et les APIs programmatiques associées) pour effectuer des traitements ensemblistes. La puissance est équivalente à SQL mais si cela ne suffit pas, il est aussi possible d'exécuter des requêtes SQL natives dans le cas où une base relationnelle est utilisée comme support de stockage. Il est clair que cette possibilité doit être utilisée avec parcimonie car elle rompt une propriété essentielle de l'approche middleware : l'indépendance et la portabilité vis-à-vis du support de stockage.

F.6 Projection vers une base de données relationnelle

La projection Objet/Relationnel (en anglais *O/R mapping* définit la translation entre les structures de données mémoires représentées par des classes et leurs champs, et les structures de données sur une base relationnelle représentées par des tables et des colonnes. Ce chapitre décrit pour chacun des éléments mémoires comment déclarer sa projection dans une base de données relationnelle.

F.6.1 Projection des identifiants d'objet persistant

La catégorie d'identifiant est choisie dans le descripteur au niveau de la balise `class` par l'attribut `identity-type`. Les valeurs possibles sont "application" ou "datastore" correspondant respectivement aux catégories applicative et système.

Les sections suivantes présentent en détail les différents formats d'identifiant supportés par Speedo.

Identifiant applicatif basé sur un seul champ persistant

L'identifiant est un composé d'un seul champ de la classe persistante. Ce champ doit être de type primitif : `char`, `byte`, `short`, `int`, `long`, `java.lang.Character`, `java.lang.Byte`, `java.lang.Short`, `java.lang.Integer` ou `java.lang.Long`. Pour utiliser ce type d'identifiant il faut définir l'attribut `identity-type` avec la valeur "application" et indiquer le champ qui représente l'identifiant en définissant l'attribut `primary-key` à "true". Voici un exemple simple, d'une classe `Invoice` ayant pour identifiant, le champ `number` :

```

<class name="Invoice" identity-type="application">
    <field name="number" primary-key="true"/>
</class>
.
```

Avec ce mode d'identifiant, toutes les formes de projection de l'héritage ne sont pas supportées par Speedo. De plus amples explications sont données dans la section F.6.6 concernant la projection d'un arbre d'héritage.

Identifiant applicatif basé sur plusieurs champs persistants

L'identifiant est un composé de plusieurs champs de la classe persistante. Ces champs doivent être de type primitif comme ceux définis dans la section précédente. Pour utiliser ce type d'identifiant il faut :

- Définir l'attribut `identity-type` avec la valeur "application".
- Définir l'attribut `primary-key` à "true" pour chaque champ composant l'identifiant.
- Spécifier une classe d'identifiant contenant tous les champs composants l'identifiant. Dans cette classe servant d'identifiant doit se conformer aux règles définies dans la section F.4.3.

Ci-dessous, un exemple de déclaration de l'identifiant pour la classe persistante `Adresse` :

```
<class name="Adresse" identity-type="application" identity-type="application"
      objectid-class="AdresseId">
    <field name="rue" primary-key="true"/>
    <field name="ville" primary-key="true"/>
</class>
```

Ci-dessous, le code de la classe `AdresseId` servant d'identifiant :

```
class AdresseId {
    public String rue;
    public String ville;
    public AdresseId(String str) {
        int idx = s.indexOf('$');
        this.rue = s.substring(0, idx);
        this.ville = s.substring(idx + 1);
    }
    public String toString() {
        return rue + "$" + ville;
    }
}
```

Identifiant géré par Speedo

L'identifiant géré par Speedo est une valeur entière de type `java.lang.Long` ou `long`. Les valeurs sont calculées par un générateur lui-même persistant. Afin de supporter le polymorphisme, ce nombre entier est décomposé en deux parties. Les 20 bits de poids fort servent à identifier la classe (soit plus d'un million de possibilités), tandis que les 44 bits de poids faible servent à identifier l'instance dans la classe (soit plus de 8 000 milliards de possibilités). Pour utiliser cet identifiant, il suffit de ne rien définir car c'est le mode par défaut, comme dans l'exemple suivant :

```
<class name="A">
  <field name="f1"/>
  <field name="f2"/>
</class>
```

Ce type d'identifiant est particulièrement intéressant dans le cas de l'héritage ou lorsque le support de stockage ne supporte pas les séquences. Comme cet identifiant est basé sur un générateur persistant, des structures particulières sont nécessaires :

- La table `jf_longgen` contient les générateurs utilisés pour les identifiants.
- La table `jf_classid` contient l'association entre le générateur et la classe persistante.

Bien que l'identifiant soit géré par Speedo, la valeur de l'identifiant peut être disponible dans un champ de la classe. Le type de ce champ doit être `long` ou `java.lang.Long`. Pour cela, il suffit de déclarer le champ comme `primary-key`. Attention, si le type du champ est `long`, c'est la valeur `-1` qui représente une référence nulle. Si le type du champ est `java.lang.Long`, c'est la valeur `null` qui en base de données représente une référence nulle.

Identifiant basé sur une séquence

L'identifiant peut être basé sur une séquence de la base de données. Voici un exemple où la classe `Myclass` a un identifiant basé sur une séquence. L'identifiant est le champ persistant `id` :

```
<class name="MyClass" identity-type="application">
  <field name="id" primary-key="true" value-strategy="sequence"
        sequence="my_seq"/>
  <field name="other_field"/>
</class>
```

F.6.2 Projection de la classe

Une classe persistante est de manière conventionnelle stockée dans une table dont les colonnes correspondent aux champs persistants de la classe. L'attribut `table` de la balise `class` permet de définir la table principale de stockage des instances de la classe. Si aucun nom de table n'est spécifié, Speedo choisit par défaut le nom de la classe :

```
<class name="Banque" identity-type="application" table="T_BANQUE">
  ...
</class>
```

Cependant, il est tout à fait possible de stocker une classe sur plusieurs tables. On distingue alors une table principale et des tables secondaires. Chaque table secondaire doit définir une jointure avec la table principale.

```
<class name="A" identity-type="application" table="A_TP">
  <join table="A_TS" column="FK_ID"/>
  <field name="id" primary-key="true" column="ID"/>
  <field name="f1" column="F1"/>
  <field name="f2" column="F2" table="A_TS"/>
  <field name="f3" column="F3" table="A_TS"/>
</class>
```

L'exemple ci-dessus décrit une classe A ayant trois champs persistants. La classe est projetée sur 2 tables A_TP (A Table Principale) et A_TS (A Table Secondaire). La table secondaire A_TS est liée à la table principale par la jointure A_TP.ID = A_TS.FK_ID.

F.6.3 Projection d'attributs simples

Un champ simple est un champ dont le type est primitif (`int`, `long`, `char`, ...) ou un objet simple (`java.lang.Boolean`, `java.sql.Date`, `java.math.BigDecimal`, ...). Un champ simple est projeté sur une seule colonne d'une seule table. La colonne peut appartenir à la table principale ou à une table secondaire. Pour définir le nom de la colonne dans la table principale, il suffit de spécifier l'attribut `column` dans la balise `field` :

```
<class ...>
  <field name="f1" column="M\_F1"/>
  ...
</class>
```

Si aucun nom de colonne n'est spécifiée Speedo nomme automatiquement la colonne. Pour définir le nom de la colonne dans une table secondaire, il faut spécifier le nom de la colonne et le nom de la table :

```
<class ...>
  <field name="f2" column="TS1_F2" table="A_TS1"/>
  ...
</class>
```

Lorsqu'une classe est projetée sur plusieurs tables, il faut définir la jointure entre les tables secondaires et la table principale. Il est possible de spécifier plus précisément des informations sur la colonne comme dans l'exemple suivant :

```
<class>
  <field name="f2">
    <column="TS1_F2" table="A_TS1" sql-type="VARCHAR(32)">
  </field>
  ...
</class>
```

D'autres attributs peuvent être spécifiés :

- `length` : la longueur de la chaîne de caractères
- `scale` : la précision des types numériques
- `default-value` : la valeur par défaut de la colonne

Ces informations concernant les colonnes SQL, ne sont pas exploitées par le runtime de Speedo. En revanche, elles peuvent être utilisées par des outils permettant la génération du schéma de la base de données.

F.6.4 Projection de relations mono-valuées

Pour stocker une relation un-un dans une base de données relationnelle, il faut au moins une clé étrangère dans une table. Le modèle de données ci-dessous est utilisé comme exemple pour les relations unidirectionnelles un-un :

```

class A {
    long aid; //identifiant de la classe A
    B unB;    // référence vers la classe B
}

class B {
    long bid; //identifiant de la classe B
}
.
```

Relation 1-1 unidirectionnelle

La clé étrangère peut être dans la table correspondant à la classe ayant la référence, c'est à dire le champ `unB` :

TA		TB	
IDA	FK_IDB	IDB	

Figure F.7 – Projection de relation 1-1 unidirectionnelle

La déclaration de ce mapping est extrêmement simple. Il suffit d'indiquer le nom de colonne dans laquelle le champ `unB` est projeté.

```

<class name="A" identity-type="application" table="TA">
    <field name="aid" primary-key="true" column="IDA"/>
    <field name="unB" column="FK_IDB"/>
</class>

<class name="B" identity-type="application" table="TB">
    <field name="bid" primary-key="true" column="IDB"/>
</class>
.
```

Dans le cas où la classe référencée possède un identifiant composite (avec plusieurs champs), le schéma de base est celui décrit dans la figure F.8.

TA			TB	
IDA	FK_IDB1	FK_IDB2	IDB1	IDB2

Figure F.8 – titre ??

La déclaration de la projection est la suivante :

```

<class name="A" identity-type="application" table="TA">
    <field name="aid" primary-key="true" column="IDA"/>
    <field name="unB">
        <column name="FK_IDB1" target="IDB1"/>
        <column name="FK_IDB2" target="IDB2"/>
    </field>
</class>

<class name="B" identity-type="application" table="TB"
      objectid-class="BId">
    <field name="bid1" primary-key="true" column="IDB1"/>
    <field name="bid2" primary-key="true" column="IDB2"/>
</class>
.
.
```

Pour chacune des colonnes composant l'identifiant, il faut définir le nom de la colonne correspondant dans la table TA. L'attribut `name` définit le nom de la colonne dans TA (soit le nom de la clé étrangère) alors que l'attribut `target` définit le nom de la colonne dans la table TB (soit le nom de la clé primaire).

Relation 1-1 bidirectionnelle

Pour les relations bidirectionnelles, le modèle suivant est utilisé comme exemple :

```

class A {
    long aid; //identifiant de la classe A
    B unB;    // référence vers la classe B
}

class B {
    long bid; //identifiant de la class B
    A unA;    //référence vers la classe A
}
.
```

TA		TB	
IDA	FK_IDB	IDB	

Figure F.9 – Projection d'une relation 1-1 bidirectionnelle

Dans le cas d'une relation bidirectionnelle, afin de ne pas alourdir de manière inutile la description, la projection est déclarée d'un côté seulement. De l'autre, il suffit d'indiquer à l'aide de l'attribut `mapped-by` le nom du champ de la référence opposée (attribut `inverse` dans le descripteur JDO). La déclaration de la projection est la suivante :

```

<class name="A" identity-type="application" table="TA">
    <field name="aid" primary-key="true" column="IDA"/>
    <field name="unB" column="FK_IDB"/>
</class>

<class name="B" identity-type="application" table="TB">
    <field name="bid" primary-key="true" column="IDB"/>
    <field name="unA" mapped-by="unB"/>
</class> .

```

F.6.5 Projection de relations multi-valuées

Comme défini auparavant, une relation multi-valuée correspond à un objet persistant référençant un ensemble d'objets persistants généralement à travers une collection. Il y en a plusieurs types dont la projection est décrite dans les sous-sections qui suivent.

Relation 1-n unidirectionnelle

Pour ce type de relation, le modèle de données ci-dessous est utilisé :

```

class A {
    long aid;           //identifiant de la classe A
    Collection<B> mesB; //référence des B
}

class B {
    long bid;           //identifiant de la classe B
}

```

Pour une projection sans table de jointure, la déclaration est la suivante :

```

<class name="A" identity-type="application" table="TA">
    <field name="aid" primary-key="true" column="IDA"/>
    <field name="mesB">
        <collection element-type="B"/>
        <element column="FK_IDA"/>
    </field>
</class>

<class name="B" identity-type="application" table="TB">
    <field name="bid" primary-key="true" column="IDB"/>
</class> .

```

La déclaration consiste à définir le nom de la colonne pour les éléments. Par convention ce nom de colonne est dans la table de l'élément référencé (ici B, soit la table TB).

Pour une projection avec table de jointure (cf. figure F.10), la déclaration est la suivante :

TA		TA		TB	
IDA		FK_IDA	FK_IDB	IDB	

Figure F.10 – Projection d'une relation 1-n unidirectionnelle

```

<class name="A" identity-type="application" table="TA">
    <field name="aid" primary-key="true" column="IDA"/>
    <field name="mesB" table="TA2B">
        <collection element-type="B"/>
        <join column="FK_IDA"/>
        <element column="FK_IDA"/>
    </field>
</class>

<class name="B" identity-type="application" table="TB">
    <field name="bid" primary-key="true" column="IDB"/>
</class>
.
```

Le principe est de déclarer une table et sa jointure vers la table principale. Cette table secondaire peut être déclarée au niveau de la classe ou seulement pour un champ comme ici. De la même manière que le cas sans jointure, il faut définir le nom de la colonne correspondant à l'élément.

Relation 1-n bidirectionnelle

Pour ce type de relation, le modèle de données ci-dessous est utilisé :

```

class A {
    long aid;           //identifiant de la classe A
    Collection<B> mesB; //référence des B
}

class B {
    long bid;           //identifiant de la class B
    A unA;              //référence un A
}
.
```

Le principe général des relations bidirectionnelles est de ne définir la projection que dans un seul sens de la relation. Pour l'autre sens (l'autre référence), il suffit d'indiquer le nom du champ de la référence inverse. Pour une projection sans table de jointure, la déclaration est la suivante :

```

<class name="A" identity-type="application" table="TA">
    <field name="aid" primary-key="true" column="IDA"/>
    <field name="mesB" mapped-by="unA">
        <collection element-type="B"/>
    </field>
</class>

<class name="B" identity-type="application" table="TB">
    <field name="bid" primary-key="true" column="IDB"/>
    <field name="unA" column="FK_IDA"/>
</class>.

```

Ici c'est la déclaration la plus simple qui a été choisie en définissant la projection de la référence mono valuée (champ `unA`) et en indiquant que le champ inverse (champ `mesB`) était à déduire de la projection du premier champ. Pour une projection avec table de jointure, la déclaration est la suivante :

```

<class name="A" identity-type="application" table="TA">
    <field name="aid" primary-key="true" column="IDA"/>
    <field name="mesB" table="TA2B">
        <collection element-type="B"/>
        <join column="FK_IDA"/>
        <element column="FK_IDA"/>
    </field>
</class>

<class name="B" identity-type="application" table="TB">
    <field name="bid" primary-key="true" column="IDB"/>
    <field name="unA" mapped-by="mesB"/>
</class>.

```

Ici, il a été choisi de définir la projection de la référence multi-valuée (champ `mesB`) et d'indiquer que la projection du champ inverse (champ `unA`) est déduite de celle du champ multi-valué.

Relation m-n bidirectionnelle

Le modèle de données ci-dessous est utilisé comme exemple :

```

class A {
    long aid;           //identifiant de la classe A
    Collection<B> mesB; //référence des B
}

class B {
    long bid;           //identifiant de la classe B
    Collection<A> mesA; //référence des A
}.

```

Le principe général des relations bidirectionnelles est de définir la projection d'un seul sens de la relation. Pour l'autre sens (l'autre référence), il suffit d'indiquer le nom du champ

de la référence inverse. La déclaration de la projection d'une relation m-n unidirectionnelle est la suivante :

```
<class name="A" identity-type="application" table="TA">
    <field name="aid" primary-key="true" column="IDA"/>
    <field name="mesB" table="TA2B">
        <collection element-type="B"/>
        <join column="FK_IDA"/>
        <element column="FK_IDB"/>
    </field>
</class>

<class name="B" identity-type="application" table="TB">
    <field name="bid" primary-key="true" column="IDB"/>
    <field name="mesA" mapped-by="mesB"/>
</class>
```

Ici, il a été choisi de définir la projection de la référence multi-valuée (champ `mesB`) et d'indiquer que la projection du champ inverse (champ `mesA`) est déduit de la projection du premier champ.

F.6.6 Projection de liens d'héritage

Dans un arbre d'héritage, à chaque définition d'un nouveau lien d'héritage entre deux classes, il y a généralement ajout de nouveaux champs dans la sous-classe. Le problème lorsqu'on doit projeter ce type de relation est le compromis à trouver entre l'efficacité du stockage (pas de perte d'espace), la dynamicité de la gestion du schéma de l'espace de stockage, et l'efficacité de l'interrogation ainsi que de la mise à jour des objets dans l'arbre d'héritage. Il n'y a pas de solution idéale dans tous les cas. Pour pouvoir faire le bon compromis, un arbre d'héritage peut être projeté selon trois stratégies : filtrée, horizontale, verticale. Voici un exemple d'arbre d'héritage très simple :

```
class A {
    String id;
    String f1;
}

class B extends A {
    String f2;
}
```

Pour ce modèle, voici les différentes projections possibles en fonction de la stratégie :

TA		
ID	F1	F2

Figure F.11 – Projection de l'héritage en mode filtré

- Filtrée : Une table contient tous les champs (ceux de la classe mère et ceux de la classe fille). L’unique table contient des instances de la classe mère et des instance de la classe fille. Ce mode de projection est illustré dans la figure F.11.

TB		
ID	F1	F2

Figure F.12 – Projection de l’héritage en mode horizontal

- Horizontale : Chaque classe a sa propre table contenant tout les champs herités et les champs de la classe courante. La table ne contient que des instances de la classe qui lui est associée. Ce mode de projection est illustré dans la figure F.12.

TA		TB	
ID	F1	ID	F2

Figure F.13 – Projection de l’héritage en mode vertical

- Verticale : Chaque classe a sa propre table contenant uniquement les champs de la classe courante. Cette table a en plus une jointure avec la table de la classe mère. Ce mode de projection est illustré dans la figure F.13.

Le nom de la classe mère et la stratégie doivent être indiqués dans le descripteur JDO comme dans l’exemple suivant :

```
<class name="B" persistence-capable-superclass="A">
    <inheritance strategy="superclass-table"/>
    ...
</class>
```

Les sections suivantes décrivent en détail les différentes projections possibles.

Projection filtrée

Avec une projection par filtrage, la classe fille et la classe mère sont stockées dans la même table. La table est définie au niveau de la classe parente. La projection des champs n’est faite qu’une seule fois. Pour distinguer les classes des instances, un filtre est nécessaire. Speedo sait supporter des filtres évolués. Cependant, la majorité des utilisateurs utilise un simple discriminant. Pour le discriminant trois cas sont possibles :

- Le discriminant est inclus dans l’identifiant de la classe. Cette stratégie est la plus performante. En effet, une référence vers un objet d’une classe de l’arbre d’héritage inclue le type exact de l’instance désignée. Ceci évite des requêtes éventuellement complexes. Quand l’identifiant est constitué de champs persistants, le discriminant est l’un d’entre eux. En revanche, l’identifiant est géré par Speedo, incluant l’identifiant de la classe. Voici un exemple où le champ `id2` est le discriminant. La valeur

"discrim_A" désigne les instances de la classe A, tandis que la valeur "discrim_B" désigne les instances de la classe B :

```
<class name="A" table="TA" identity-type="application">
    <inheritance strategy="new-table">
        <discriminator column="ID2" stategy="value-map" value="discrim_A"/>
    </inheritance>
    <field name="id1" primary-key="true" column="ID1"/>
    <field name="id2" primary-key="true" column="ID2"/>
    <field name="f1" column="F1"/>
</class>

<class name="B" persistence-capable-superclass="A">
    <inheritance strategy="superclass-table">
        <discriminator column="ID2" value="discrim_B"/>
    </inheritance>
    <field name="f2" column="F2"/>
</class>
```

- Le discriminant n'est pas inclus dans l'identifiant de la classe. Cette solution est moins performante. Voici un exemple :

```
<class name="A" table="TA" identity-type="application">
    <inheritance strategy="new-table">
        <discriminator column="ID2" stategy="value-map" value="discrim_A"/>
    </inheritance>
    <field name="id1" primary-key="true" column="ID1"/>
    <field name="id2" column="ID2"/>
    <field name="f1" column="F1"/>
</class>

<class name="B" persistence-capable-superclass="A">
    <inheritance strategy="superclass-table">
        <discriminator column="ID2" value="discrim_B"/>
    </inheritance>
    <field name="f2" column="F2"/>
</class>
```

- Le discriminant n'est pas un champ persistant mais une colonne de la table. Cette colonne ne correspond pas à un champ persistant. Sa valeur est constante pour une classe donnée. C'est la valeur de cette constante qui discrimine les classes des instances. Cette stratégie est intéressante si on ne veut pas se préoccuper de la valeur du discriminant en ayant un identifiant applicatif. Souvent la valeur du discriminant est le nom de la classe persistante. Voici un exemple :

```

<class name="A" table="TA">
    <inheritance strategy="new-table">
        <discriminator column="MY_COL" stategy="class-name"/>
    </inheritance>
    <field name="id1" primary-key="true" column="ID1"/>
    <field name="f1" column="F1"/>
</class>

<class name="B" persistence-capable-superclass="A">
    <inheritance strategy="superclass-table">
        <discriminator column="MY_COL"/>
    </inheritance>
    <field name="f2" column="F2"/>
</class>
.

```

L'identifiant géré par Speedo inclut un discriminant. C'est le meilleur choix quand l'identifiant n'est pas basée sur des valeurs applicatives.

Projection horizontale

Avec une stratégie de projection horizontale, chaque classe a sa propre table contenant tous les champs hérités. En effet, dans le descripteur de la classe fille, il faut redéfinir la projection des champs hérités. Ils doivent être projetés sur des structures de la table courante. Actuellement, Speedo supporte ce type de projection uniquement dans le cas où l'identifiant contient un discriminant. C'est le cas de l'identifiant geré par Speedo. Voici un exemple :

```

<class name="A" table="TA">
    <inheritance strategy="new-table"/>
    <field name="id" column="ID"/>
    <field name="f1" column="F1"/>
</class>

<class name="B" persistence-capable-superclass="A" table="TB">
    <inheritance strategy="new-table"/>
    <field name="id" column="ID"/>
    <field name="A.f1" column="F1"/>
    <field name="f2" column="F2"/>
</class>
.

```

On notera qu'afin de distinguer les champs hérités des champs de la classe courante, les noms des champs hérités sont préfixés par le nom de la classe.

Projection verticale

Avec la stratégie de projection verticale, chaque classe a sa propre table contenant uniquement les champs de la classe courante. Cette table a en plus une jointure avec la table de la classe mère. Actuellement Speedo supporte uniquement le cas où l'identifiant inclut un discriminant. Ceci est le cas le plus performant. En effet, cela évite de faire des jointures avec toutes les classes filles pour déterminer l'instance exacte. Voici un exemple :

```

<class name="A" table="TA">
  <inheritance strategy="new-table"/>
  <field name="id" column="ID"/>
  <field name="f1" column="F1"/>
</class>

<class name="B" persistence-capable-superclass="A" table="TB">
  <inheritance strategy="new-table">
    <join column="ID"/>
  </inheritance>
  <field name="f2" column="F2"/>
</class>
.
.
```

F.7 Intégration dans J2EE

L'intégration d'une solution JDO dans un serveur J2EE concerne l'aspect transactionnel et la manière dont le produit JDO est ajouté au serveur d'application et récupéré par l'application. Pour cela on distingue trois modes d'intégration décrits dans les sections qui suivent.

F.7.1 Aucune intégration

Le produit JDO est utilisé dans une application J2EE sans le lier au serveur d'application. Ce cas est le plus simple puisqu'il correspond à une application Java classique. Les transactions sont locales et doivent être démarquées avec l'API JDO (`javax.jdo.Transaction`). Le pilote JDO est instancié en utilisant le `JDOHelper` comme expliqué précédemment. Les librairies de la solution JDO peuvent être intégrées à l'application ou au serveur d'application. Ce mode d'intégration est très classique pour de petites applications Web ayant un besoin de connectivité à la base.

F.7.2 Intégration JTA

Le produit JDO est intégré au système de gestion des transactions fournis par le serveur d'application. Ce mode d'intégration est souvent appelé "intégration JTA".

Dans ce mode, la démarcation transactionnelle de l'application est réalisée en utilisant les APIs JTA (`javax.transaction.UserTransaction`), ou par le conteneur EJB (attribut transactionnel des méthodes des EJB). L'API JDO de démarcation des transactions (`javax.jdo.Transaction`) ne doit pas être utilisée sous peine d'effectuer des transactions locales désynchronisées des contextes transactionnels gérés par le serveur d'application. La solution JDO se raccroche automatiquement à la transaction JTA courante.

Le pilote JDO est instancié en utilisant le `JDOHelper`. Dans ce type d'intégration, la solution JDO doit retrouver le gestionnaire de transaction du serveur d'application. Ceci n'étant décrit dans aucune spécification, l'intégration est spécifique à chaque serveur d'application. Heureusement, la plupart des solutions JDO savent s'intégrer aux serveurs d'applications les plus courants. Cependant il peut s'avérer nécessaire d'indiquer le serveur d'application utilisé dans le fichier de configuration (propriétés de la PMF) de la solution JDO.

Les librairies de la solution JDO peuvent être intégrées à l'application ou au serveur d'application.

F.7.3 Intégration JCA / JTA

Le produit JDO est intégré au serveur d'application comme un *connecteur* (ou un *resource adapter*). Ce mode d'intégration est souvent appelé "intégration JCA/JTA".

Dans ce mode l'intégration, la solution JDO suit la spécification JCA (Java Connector Architecture). Le produit est empaqueté sous la forme d'une archive .rar. Cet archive peut être incluse dans l'application directement (c'est à dire dans un .ear) ou au niveau du serveur.

L'intégration sous forme d'un connecteur implique en principe l'intégration JTA. La PersistenceManagerFactory est instanciée par le connecteur JCA fourni avec la solution JDO. La PMF est automatiquement enregistrée dans l'annuaire JNDI du serveur. Il est ensuite très facile de faire la liaison entre l'application J2EE et la ressource enregistrée dans JNDI.

Enfin les propriétés de la PMF peuvent être spécifiées directement dans les fichiers de configuration du connecteur ra.xml ou xxx-ra.xml (où xxx est le nom du serveur d'application), ou dans le fichier de propriétés de la solution JDO. Dans ce dernier cas les fichiers de configuration du connecteur permettent de définir le nom du fichier de configuration à charger à partir du *classpath*.

F.8 Conclusion et perspectives

La persistance d'objet Java a trouvé un très bon support avec JDO. Il reste que c'est un sujet stratégique pour bon nombre de grands éditeurs logiciels et qu'il s'agit d'une spécification définie en dehors du cadre de J2EE. La spécification EJB3 qui émerge actuellement dans le cadre de J2EE est fortement inspirée des résultats de JDO 2 même si elle reste en deça fonctionnellement. Elle est néanmoins appuyée par des acteurs majeurs de l'industrie logicielle ce qui lui donne une chance importante de s'imposer. On peut penser que ces deux spécifications concurrentes vont converger dans l'avenir car elles sont bâties sur les mêmes principes. C'est tellement vrai que Speedo devrait supporter ces deux spécifications dans un avenir très proche.

Intergiciel et Construction d'Applications Réparties

©2006 R. Balter (version du 19 janvier 2007 - 10:31)

Licence Creative Commons (<http://creativecommons.org/licenses/by-nc-nd/2.0/fr/deed.fr>)

Annexe G

Infrastructure M2M pour la gestion de l'énergie

Annexe provisoirement indisponible.

Glossaire

Ce glossaire, destiné à regrouper les définitions de termes et acronymes associés à l'intergiciel, est encore très incomplet.

[Axis] implantation de SOAP (voir page 90).

[BPEL] Business Process Language (voir page 97).

[Beehive] implantation de SOAP (voir page 90).

[Bundle] unité de conditionnement et de déploiement des applications OSGi

[Chorégraphie] ensemble d'interactions qui peuvent ou doivent avoir lieu entre un ensemble de services (voir page 79).

[JSR] *Java Specifications Request*.

[Orchestration] ensemble d'interactions et d'actions internes dans lesquelles un service donné peut ou doit s'engager ainsi que les dépendances entre ces actions (voir page 79).

[OSGi Alliance] consortium industriel ayant pour vocation la définition et la promotion de la spécification OSGi.

[OSGi] spécification ouverte d'une plate-forme de déploiement et d'exécution de services Java.

[POJO] (*Plain Old Java Object*) bon vieux objets Java n'obéissant pas un modèle de programmation particulier (JavaBean, EJB, MBean, SCR, Fractal...).

[REST] REpresentational State Transfer (voir page 94).

[RFE] *Request For Extension*.

[SCR] (*Service Component Runtime*) modèle à composants dynamiques orienté service introduit dans la version 4 de la spécification OSGi.

[SGML] Standard Generalized Markup Language (voir page 84).

[SOAP] Simple Object Access Protocol (voir page 91).

[UPnP] (*Universal Plug and Play*) pile de protocoles de découverte et d'utilisation d'équipements connectés par un réseau ad-hoc.

[UPnP Forum] consortium industriel ayant pour vocation la définition et la promotion d'une pile de protocoles UPnP. Ce consortium spécifie également des interfaces standard d'équipements principalement dans le domaine de la domotique.

[WSDL] Web service Description Language (voir page 84).

[XML] eXtended Markup Language (voir page 84).

Bibliographie

- [Aalst et al. 2005] Aalst, W., Dumas, M., Ouyang, C., Rozinat, A., and Verbeek, H. (2005). Choreography Conformance Checking: An Approach Based on BPEL and Petri Nets. BPMCenter Report BPM-05-25, BPMcenter.org.
- [Aalst et al. 2003] Aalst, W., van Dongen, B., Herbst, J., Maruster, L., Schimm, G., and Weijters, A. (2003). Workflow Mining: A Survey of Issues and Approaches. *Data and Knowledge Engineering*, 47(2):237–267.
- [Agha 1986] Agha, G. A. (1986). Actors: A Model of Concurrent Computation in Distributed Systems. In *The MIT Press, ISBN 0-262-01092-5*, Cambridge, MA.
- [Aldrich et al. 2002] Aldrich, J., Chambers, C., and Notkin, D. (2002). ArchJava: Connecting software architecture to implementation. In *Proceedings of the 24th International Conference on Software Engineering (ICSE'02)*, pages 187–197. ACM Press.
- [Alexander 1964] Alexander, C. (1964). *Notes on the Synthesis of Form*. Harvard University Press.
- [Alexander et al. 1977] Alexander, C., Ishikawa, S., and Silverstein, M. (1977). *A Pattern Language: Towns, Buildings, Construction*. Oxford University Press. 1216 pp.
- [Almes et al. 1985] Almes, G. T., Black, A. P., Lazowska, E. D., and Noe, J. D. (1985). The Eden system: A technical review. *IEEE Transactions on Software Engineering*, SE-11(1):43–59.
- [Alonso et al. 2003] Alonso, G., Casati, F., Kuno, H., and Machiraju, V. (2003). *Web Services. Concepts, Architectures and Applications*. Springer Verlag.
- [Altenhofen et al. 2005] Altenhofen, M., Boerger, E., and Lemcke, J. (2005). An execution semantics for mediation patterns. In *In Proceedings of the BPM'2005 Workshops: Workshop on Choreography and Orchestration for Business Process Management*, Nancy, France.
- [Alur et al. 2003] Alur, D., Malks, D., and Crupi, J. (2003). *Core J2EE Patterns: Best Practices and Design Strategies, Second Edition*. Prentice Hall. 650 pp.
- [Ancona et al. 2000] Ancona, D., Lagorio, G., and Zucca, E. (2000). A Smooth Extension of Java with Mixins. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP'00)*, pages 154–178, Sophia Antipolis and Cannes, France.
- [Andrews et al. 2003] Andrews, T., Curbera, F., Dholakia, H., Goland, Y., Klein, J., Leymann, F., Liu, K., Roller, D., Smith, D., Thatte, S., Trickovic, I., and Weerawarana, S. (2003). Business Process Execution Language for Web Services, Version 1.1. Standards proposal by BEA Systems, IBM Corporation and Microsoft Corporation.
- [ANSA] ANSA. Advanced networked systems architecture. <http://www.ansa.co.uk/>.
- [Apache] Apache. The Apache Foundation Software. <http://www.apache.org>.

- [Arregui et al. 2000] Arregui, D., Pacull, F., and Rivière, M. (2000). Heterogeneous component coordination: The CLF approach. In *EDOC*, pages 194–2003. 4th International Enterprise Distributed Object Computing Conference (EDOC 2000), Makuhari, Japan, IEEE Computer Society.
- [ASM 2002] ASM (2002). ASM: a Java Byte-Code Manipulation Framework. The ObjectWeb Consortium, <http://www.objectweb.org/asm/>.
- [Avalon] Avalon. The Apache Avalon Project. <http://avalon.apache.org>.
- [Axis] Axis. Projet Web Services - Apache - Axis. <http://ws.apache.org/axis/>.
- [B. Stearns 2000a] B. Stearns (2000a). JavaBeans 101 Tutorial, Part I. <http://java.sun.com/developer/onlineTraining/Beans/bean01/index.html>.
- [B. Stearns 2000b] B. Stearns (2000b). JavaBeans 101 Tutorial, Part II. <http://java.sun.com/developer/onlineTraining/Beans/bean02/index.html>.
- [Baïna et al. 2004] Baïna, K., Benatallah, B., Casati, F., and Toumani, F. (2004). Model-Driven Web Services Development. In *Proceedings of the 16th International Conference on Advanced Information Systems Engineering (CAiSE'04)*, Riga, Latvia. Springer.
- [Balter et al. 1991] Balter, R., Bernadat, J., Decouchant, D., Duda, A., Freyssinet, A., Krakowiak, S., Meysembourg, M., Le Dot, P., Nguyen Van, H., Paire, E., Riveill, M., Roisin, C., Rousset de Pina, X., Scioville, R., and Vandôme, G. (1991). Architecture and implementation of Guide, an object-oriented distributed system. *Computing Systems*, 4(1):31–67.
- [Banavar et al. 1999] Banavar, G., Chandra, T., Strom, R., and Sturman, D. (1999). A Case for Message Oriented Middleware. In *13th Int. Symp. on Distributed Computing (DISC)*, LNCS 1693.
- [Banâtre and Banâtre 1991] Banâtre, J.-P. and Banâtre, M., editors (1991). *Les systèmes distribués : expérience du projet Gothic*. InterÉditions.
- [Baresi et al. 2004] Baresi, L., Ghezzi, C., and Guinea, S. (2004). Smart monitors for composed services. In *Proceedings of the 2nd International Conference on Service-Oriented Computing (IC-SOC)*, pages 193–202, New York, NY, USA.
- [Barros et al. 2005a] Barros, A., Dumas, M., and Hofstede, A. (2005a). Service interaction patterns. In *Proceedings of the 3rd International Conference on Business Process Management*, Nancy, France. Springer. Extended version available at: <http://www.serviceinteraction.com>.
- [Barros et al. 2005b] Barros, A., Dumas, M., and Oaks, P. (2005b). A critical overview of the web services choreography description language (ws-cdl). *BPTrends Newsletter*, 3(3).
- [Baude et al. 2003] Baude, F., Caromel, D., and Morel, M. (2003). From distributed objects to hierarchical grid components. In *Proceedings of the International Symposium on Distributed Objects and Applications (DOA '03)*.
- [BCEL] BCEL. Byte Code Engineering Library. <http://jakarta.apache.org/bcel>.
- [Beehive] Beehive. Projet Apache - Beehive. <http://beehive.apache.org>.
- [Benatallah et al. 2005a] Benatallah, B., Casati, F., Grigori, D., H.R. Motahari-Nezhad, and Toumani, F. (2005a). Developing Adapters for Web Services Integration. In *Proceedings of the 17th International Conference on Advanced Information System Engineering, CAiSE 2005, Porto, Portugal*, pages 415–429.
- [Benatallah et al. 2005b] Benatallah, B., Dumas, M., and Sheng, Q.-Z. (2005b). Facilitating the Rapid Development and Scalable Orchestration of Composite Web Services. *Distributed and Parallel Database*, 17(1).

- [Ben Albahari et al. 2002] Ben Albahari, Peter Drayton, and Brad Merrill (2002). *C# Essentials*. O'Reilly. ISBN: 0-596-00315-3, 216 pages.
- [Bertino et al. 2004] Bertino, E., Guerrini, G., and Mesiti, M. (2004). A matching algorithm for measuring the structural similarity between an xml document and a dtd and its applications. *Information Systems*, 29(1):23–46.
- [Beugnard 2005] Beugnard, A. (2005). Contribution à l'étude de l'assemblage de logiciels. Habilitation à diriger des recherches, Université de Bretagne Sud.
- [Beugnard 2006] Beugnard, A. (2006). Method overloading and overriding cause encapsulation flaw. In *Object-Oriented Programming Languages and Systems, 21st ACM Symposium on Applied Computing (SAC)*, Dijon, France.
- [Beugnard et al. 1999] Beugnard, A., Jézéquel, J.-M., Plouzeau, N., and Watkins, D. (1999). Making Components Contract Aware. *IEEE Computer*, 32(7):38–45.
- [Bieber and Carpenter 2002] Bieber, G. and Carpenter, J. (2002). Introduction to Service-Oriented Programming. <http://www.openwings.org>.
- [Birrell and Nelson 1984] Birrell, A. D. and Nelson, B. J. (1984). Implementing remote procedure calls. *ACM Transactions on Computer Systems*, 2(1):39–59.
- [Birrell et al. 1995] Birrell, A. D., Nelson, G., Owicki, S., and Wobber, E. (1995). Network objects. *Software—Practice and Experience*, 25(S4):87–130.
- [Blakeley et al. 1995] Blakeley, B., Harris, H., and Lewis, J. (1995). *Messaging and Queueing Using the MQI: Concepts and Analysis, Design and Development*. McGraw-Hill.
- [Blow et al. 2004] Blow, M., Goland, Y., Kloppmann, M., Leymann, F., Pfau, G., Roller, D., and Rowley, M. (2004). BPELJ: BPEL for Java. White paper.
- [Bouraqadi 2005] Bouraqadi, N. (2005). *FracTalk: Fractal Components in Smalltalk*. csl.ensm-douai.fr/FracTalk.
- [Bracha and Cook 1990] Bracha, G. and Cook, W. (1990). Mixin-based inheritance. In *Proceedings of the Conference on Object-Oriented Programming: Systems, Languages and Applications (ECOOP/OOPSLA '90)*, volume 25 of *SIGPLAN Notices*, pages 303–311. ACM Press.
- [Brinch Hansen 1978] Brinch Hansen, P. (1978). Distributed Processes: a concurrent programming concept. *Communications of the ACM*, 21(11):934–941.
- [Brownbridge et al. 1982] Brownbridge, D. R., Marshall, L. F., and Randell, B. (1982). The Newcastle Connection — or UNIXes of the World Unite! *Software – Practice and Experience*, 12(12):1147–1162.
- [Bruneton 2001] Bruneton, É. (2001). *Un support d'exécution pour l'adaptation des aspects non-fonctionnels des applications réparties*. PhD thesis, Institut National Polytechnique de Grenoble.
- [Bruneton et al. 2003] Bruneton, E., Coupaye, T., and Stefani, J. (2003). The Fractal Component Model. Technical report, Specification v2, ObjectWeb Consortium, <http://www.objectweb.org/fractal>.
- [Bruneton et al. 2002] Bruneton, E., Lenglet, R., and Coupaye, T. (2002). ASM: A code manipulation tool to implement adaptable systems. In *Journées Composants 2002 (JC'02)*. asm.objectweb.org/current/asm-eng.pdf.
- [Burke 2003] Burke, B. (2003). It's the aspects. Java's Developer's Journal. www.sys-con.com/story/?storyid=38104&DE=1.

- [Buschmann et al. 1995] Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., and Stal, M. (1995). *Pattern-Oriented Software Architecture, Volume 1: A System of Patterns*. John Wiley & Sons. 467 pp.
- [Cabrera et al. 2002a] Cabrera, F., Copeland, G., Cox, B., Freund, T., Klein, J., Storey, T., Langworthy, D., and Orchard, D. (2002a). Web Service Coordination, WS-Coordination. <http://www.ibm.com/developerworks/library/ws-coor/>. IBM, Microsoft, BEA.
- [Cabrera et al. 2002b] Cabrera, F., Copeland, G., Cox, B., Freund, T., Klein, J., Storey, T., and Thatte, S. (2002b). Web service transaction, ws-transaction. <http://www.ibm.com/developerworks/library/ws-transpec/>. IBM, Microsoft, BEA.
- [CACM 1993] CACM (1993). *Communications of the ACM*, special issue on concurrent object-oriented programming. 36(9).
- [Cahill et al. 1994] Cahill, V., Balter, R., Harris, N., and Rousset de Pina, X., editors (1994). *The COMANDOS Distributed Application Platform*. ESPRIT Research Reports. Springer-Verlag. 312 pp.
- [Campbell and Habermann 1974] Campbell, R. H. and Habermann, A. N. (1974). The specification of process synchronization by path expressions. In Gelenbe, E. and Kaiser, C., editors, *Operating Systems, an International Symposium*, volume 16 of *LNCS*, pages 89–102. Springer Verlag.
- [Carbon 2004] Carbon (2004). *The Carbon Project*. carbon.sourceforge.net.
- [Carzaniga et al. 1998] Carzaniga, A., Fuggetta, A., Hall, R. S., van der Hoek, A., Heimbigner, D., and Wolf, A. L. (1998). A characterization framework for software deployment technologies. Technical Report CU-CS-857-98, Dept. of Computer Science, University of Colorado.
- [Castor 1999] Castor (1999). The castor project. <http://www.castor.org/>.
- [Cervantes 2004] Cervantes, H. (2004). *Vers un modèle à composants orienté services pour supporter la disponibilité dynamique*. PhD thesis, Université Joseph Fourier (Grenoble 1), Grenoble, France.
- [Cervantes and Hall 2004] Cervantes, H. and Hall, R. S. (2004). A framework for constructing adaptive component-based applications: Concepts and experiences. In Crnkovic, I., Stafford, J. A., Schmidt, H. W., and Wallnau, K. C., editors, *CBSE*, volume 3054 of *Lecture Notes in Computer Science*, pages 130–137. Springer.
- [Chakrabarti et al. 2002] Chakrabarti, A., de Alfaro, L., Henzinger, T. A., Jurdzinski, M., and Mang, F. Y. (2002). Interface Compatibility Checking for Software Modules. In *Proceedings of the 14th International Conference on Computer-Aided Verification (CAV)*, volume 2404 of *LNCS*, pages 428–441. Springer-Verlag.
- [Chassande-Barrio 2003] Chassande-Barrio, S. (2003). *The Speedo Project*. ObjectWeb. speedo.objectweb.org.
- [Chassande-Barrio and Dechamboux 2003] Chassande-Barrio, S. and Dechamboux, P. (2003). *The Perseus Project*. ObjectWeb. perseus.objectweb.org.
- [Clarke et al. 2001] Clarke, M., Blair, G., Coulson, G., and Parlavantzas, N. (2001). An efficient component model for the construction of adaptive middleware. In *Proceedings of Middleware'01*.
- [Coulson et al. 2004] Coulson, G., Blair, G., Grace, P., Joolia, A., Lee, K., and Uyema, J. (2004). A component model for building systems software. In *Proceedings of the IASTED Software Engineering and Applications (SEA '04)*.

- [Dahl et al. 1970] Dahl, O.-J., Myhrhaug, B., and Nygaard, K. (1970). The SIMULA 67 common base language. Technical Report S-22, Norwegian Computing Center, Oslo, Norway.
- [Dasgupta et al. 1989] Dasgupta, P., Chen, R. C., Menon, S., Pearson, M. P., Ananthanarayanan, R., Ramachandran, U., Ahamad, M., LeBlanc, R. J., Appelbe, W. F., Bernabéu-Aubán, J. M., Hutto, P. W., Khalidi, M. Y. A., and Wilkenloh, C. J. (1989). The design and implementation of the Clouds distributed operating system. *Computing Systems*, 3(1):11–46.
- [Dijkman and Dumas 2004] Dijkman, R. and Dumas, M. (2004). Service-oriented design: A multi-viewpoint approach. *International Journal of Cooperative Information Systems*, 13(4):337–368.
- [Dijkstra 1968] Dijkstra, E. W. (1968). The Structure of the THE Multiprogramming System. *Communications of the ACM*, 11(5):341–346.
- [Dillenseger 2004] Dillenseger, B. (2004). *The CLIF Project*. ObjectWeb. clif.objectweb.org.
- [Donsez 2006] Donsez, D. (2006). Courtage et déploiement dynamiques de composants pour l'infrastructure d'équipements UPnP. In *UbiMob'06: 3èmes Journées Francophones Mobilité et Ubiquité*, New York, NY, USA. ACM Press.
- [Donsez et al. 2004] Donsez, D., Cervantes, H., and Désertot, M. (2004). FROGi : Déploiement de composants fractal sur OSGi. In *Conférence Francophone sur le Déploiement et la Reconfiguration, Grenoble, France, 28-29 Octobre 2004*, pages 147–158.
- [Dowling and Cahill 2001] Dowling, J. and Cahill, V. (2001). The K-Component architecture meta-model for self-adaptative software. In *Proceedings of Reflection'01*, volume 2192 of *Lecture Notes in Computer Science*, pages 81–88. Springer-Verlag.
- [Dudney et al. 2003] Dudney, B., Asbury, S., Krozak, J. K., and Wittkopf, K. (2003). *J2EE AntiPatterns*. Wiley. 600 pp.
- [Dumont et al. 1998] Dumont, B., Horn, F., Tran, F. D., and Stefani, J.-B. (1998). Jonathan: an Open Distributed Processing Environment in Java. In Davies, R., Raymond, K., and Seitz, J., editors, *Proceedings of Middleware'98: IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing*, pages 175–190, The Lake District, UK. Springer-Verlag.
- [Dumas et al. 2005] Dumas, M., Aalst, W., and Hofstede, A. (2005). *Process-Aware Information Systems: Bridging People and Software through Process Technology*. John Wiley & Sons.
- [Dustdar et al. 2004] Dustdar, S., Gombotz, R., and Baino, K. (2004). Web Services Interaction Mining. Technical Report TUV-1841-2004-16, Information Systems Institute, Vienna University of Technology, Wien, Austria.
- [Désertot et al. 2006] Désertot, M., Donsez, D., and Lalanda, P. (2006). A dynamic service-oriented implementation for java EE servers. In *2006 IEEE International Conference on Services Computing (SCC 2006)*, New York, NY, USA. IEEE Computer Society.
- [Edd Dumbill and M. Bornstein 2004] Edd Dumbill and M. Bornstein, N. (2004). *Mono: A Developer's Notebook*. O'Reilly. ISBN: 0-596-00792-2, 304 pages.
- [Elmagarmid 1990] Elmagarmid, A. K., editor (1990). *Database Transactions Models for Advanced Applications*. Morgan Kaufmann Publishers.
- [Enhydra 2002] Enhydra (2002). Data object design studio.
<http://www.enhydra.org/tech/dods/index.html>.
- [Escoffier and Donsez 2005] Escoffier, C. and Donsez, D. (2005). FractNet: An implementation of the Fractal component model for .NET. In *2ème Journée Francophone sur Développement de Logiciels par Aspects (JFDLPA '05)*. www-adele.imag.fr/fractnet/.

- [Escoffier et al. 2006] Escoffier, C., Donsez, D., and Hall, R. S. (2006). Developing an osgi-like service platform for .net. In *IEEE Consumer Communications and Networking Conference (CCNC'06) 2006*.
- [Fassino et al. 2002] Fassino, J.-P., Stefani, J.-B., Lawall, J., and Muller, G. (2002). Think: A software framework for component-based operating system kernels. In *Proceedings of the USENIX Annual Technical Conference*, pages 73–86.
- [Fassino 2001] Fassino, J.-Ph. (2001). *THINK : vers une architecture de systèmes flexibles*. PhD thesis, École Nationale Supérieure des Télécommunications, Paris.
- [Fauvet and Ait-Bachir 2006] Fauvet, M.-C. and Ait-Bachir, A. (2006). An automaton-based approach for web service mediation. In *Workshop on Web Services. 13th International Conference on Concurrent Engineering*, Antibes, France. Invited paper.
- [Fauvet et al. 2005] Fauvet, M.-C., Duarte, H., Dumas, M., and Benatallah, B. (2005). Handling transactional properties in Web service composition. In *Proceeding of Web Information Systems Engineering (WISE)*, number 3806 in LNCS, New York City. Springer 2005, ISBN 3-540-30017-1.
- [Fielding 2000] Fielding, R. (2000). *Architectural Styles and the Design of Network-based Software Architectures*. PhD thesis, University of California, Irvine, USA.
- [Fleury and Reverbel 2003] Fleury, M. and Reverbel, F. (2003). The JBoss extensible server. In *Proceedings of the 4th ACM/IFIP/USENIX International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware'03)*, volume 2672 of *Lecture Notes in Computer Science*, pages 344–373. Springer-Verlag.
- [Fox et al. 1998] Fox, A., Gribble, S. D., Chawathe, Y., and Brewer, E. A. (1998). Adapting to network and client variation using infrastructural proxies: Lessons and perspectives. *IEEE Personal Communications*, pages 10–19.
- [Frénot and Stefan 2004] Frénot, S. and Stefan, D. (2004). Open-service-platform instrumentation: JMX management over OSGi. In *UbiMob'04: 1ères Journées Francophones Mobilité et Ubiquité*, pages 199–202, New York, NY, USA. ACM Press.
- [Fu et al. 2004] Fu, X., Bultan, T., and Su, J. (2004). Analysis of interacting BPEL web services. In *Proceedings of the 13th International Conference on World Wide Web (WWW)*, pages 621–630, New York, NY, USA.
- [Gaaloul et al. 2004] Gaaloul, W., Bhiri, S., and Godart, C. (2004). Discovering Workflow Transactional Behavior from Event-Based Log. In *Proceedings of the OTM Confederated International Conferences, CoopIS, DOA, and ODBASE 2004*, pages 3–18.
- [Gamma et al. 1994] Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (1994). *Design Patterns: Elements of Reusable Object Oriented Software*. Addison-Wesley. 416 pp.
- [Goldfarb 1990] Goldfarb, C.-F. (1990). *The SGML Handbook*. Oxford Clarendon Press.
- [Gong 2001] Gong, L. (2001). A Software Architecture for Open Service Gateways. *IEEE Internet Computing*, 6:64–70.
- [Gray and Reuter 1993] Gray, J. and Reuter, A. (1993). *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann.
- [Grimes 1997] Grimes, R. (1997). *Professional DCOM Programming*. Wrox Press. 592 pp.
- [Gruber et al. 2005] Gruber, O., Hargrave, B. J., McAffer, J., Rapicault, P., and Watson, T. (2005). The Eclipse 3.0 platform: Adopting OSGi technology. *IBM Systems Journal*, 44(2):289–300.

- [Gunnerson and Wienholt] Gunnerson, E. and Wienholt, N. *A Programmer's Introduction to C# 2.0*. Apress. ISBN 1590595017, 568 pages.
- [Hagen and Alonso 2000] Hagen, C. and Alonso, G. (2000). Exception handling in workflow management systems. *IEEE Transactions on Software Engineering*, 26(10):943–958.
- [Hagimont and De Palma 2002] Hagimont, D. and De Palma, N. (2002). Removing Indirection Objects for Non-functional Properties. In *Proceedings of the 2002 International Conference on Parallel and Distributed Processing Techniques and Applications*.
- [Hamilton et al. 1993] Hamilton, G., Powell, M. L., and Mitchell, J. G. (1993). Subcontract: A flexible base for distributed programming. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles*, volume 27 of *Operating Systems Review*, pages 69–79, Asheville, NC (USA).
- [Handley et al. 2000] Handley, M., Schulzrinne, H., Schooler, E., and Rosenberg, J. (2000). SIP: Session Initiation Protocol.
<http://www.ietf.org/internet-drafts/draft-ietf-sip-rfc2543bis-02.txt>.
- [Hibernate 2002] Hibernate (2002). Relational persistence for java and .net.
<http://www.hibernate.org/>.
- [Hivemind 2004] Hivemind (2004). *The Hivemind Project*. jakarta.apache.org/hivemind.
- [Hoare 1969] Hoare, C. A. R. (1969). An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–585.
- [J2EE 2005] J2EE (2005). Java 2 Enterprise Edition. <http://java.sun.com/products/j2ee>.
- [J2SE 2005] J2SE (2005). Java 2 Standard Edition. <http://java.sun.com/products/j2se>.
- [JBoss Group 2003] JBoss Group (2003). JBoss Application Server.
<http://www.jboss.com/products/jbossas>.
- [JDO 2002] JDO (2002). *Java Data Objects*. Sun Microsystems. java.sun.com/products/jdo/.
- [Jones 1993] Jones, M. B. (1993). Interposition agents: Transparently interposing user code at the system interface. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles*, pages 80–93, Asheville, NC (USA).
- [joram 2002] joram (2002). JORAM. ObjectWeb, <http://www.objectweb.org/joram/>.
- [JSR-000012 2004] JSR-000012, S. (2004). Java data objects (jdo) specification.
<http://www.jcp.org/en/jsr/detail?id=12>.
- [JSR-000220 2006] JSR-000220, S. (2006). Enterprise javabeans 3.0 specification.
<http://jcp.org/aboutJava/communityprocess/final/jsr220/index.html>.
- [JSR-000243 2006] JSR-000243, S. (2006). Java data objects 2.0 - an extension to the jdo specification.
<http://www.jcp.org/en/jsr/detail?id=243>.
- [JSR 181] JSR 181. Web Services Metadata for the *JavaTM* Platform.
<http://jcp.org/en/jsr/detail?id=181>.
- [Jul et al. 1988] Jul, E., Levy, H., Hutchinson, N., and Black, A. (1988). Fine-grained mobility in the Emerald system. *ACM Transactions on Computer Systems*, 6(1):109–133.
- [Kaler and Nadalin] Kaler, C. and Nadalin, A. Web services security policy language (ws-securitypolicy) version 1.1. Standards proposal by IBM Corporation, Microsoft Corporation, RSA Security and VeriSign.

- [Kavantzas et al. 2005] Kavantzas, N., Burdett, D., Ritzinger, G., Fletcher, T., Lafon, Y., and Barreto, C. (2005). Web Services Choreography Description Language, Version 1.0. W3C Candidate Recommendation.
- [Kiczales 1996] Kiczales, G. (1996). Aspect-Oriented Programming. *ACM Computing Surveys*, 28(4):154.
- [Kiczales et al. 1991] Kiczales, G., des Rivières, J., and Bobrow, D. G. (1991). *The Art of the Metaobject Protocol*. MIT Press. 345 pp.
- [Kiczales et al. 2001] Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., and Griswold, W. G. (2001). An overview of AspectJ. In *Proceedings of ECOOP 2001*, volume 2072 of *LNCS*, pages 327–355, Budapest, Hungary. Springer-Verlag.
- [Kon et al. 2002] Kon, F., Costa, F., Blair, G., and Campbell, R. (2002). The case for reflective middleware. *Communications of the ACM*, 45(6):33–38.
- [Koutsonikola and Vakali 2004] Koutsonikola, V. and Vakali, A. (2004). LDAP: Framework, practices, and trends. *IEEE Internet Computing*, 8(5):66–72.
- [Kramer 1998] Kramer, R. (1998). iContract - The Java Design by Contract Tool. In *Proceedings of the Technology of Object-Oriented Languages and Systems (TOOLS) Conference*, pages 295–307.
- [Layaida et al. 2004] Layaida, O., Atallah, S. B., and Hagimont, D. (2004). A framework for dynamically configurable and reconfigurable network-based multimedia adaptations. *Journal of Internet Technology*, 5(4):57–66. Special Issue on Real Time Media Delivery over the Internet.
- [Lea 1999] Lea, D. (1999). *Concurrent Programming in Java*. The Java Series. Addison-Wesley, 2nd edition. 412 pp.
- [Lea et al. 1993] Lea, R., Jacquemot, C., and Pillevesse, E. (1993). COOL: System Support for Distributed Object-oriented Programming. *Communications of the ACM (special issue, Concurrent Object-Oriented Programming, B. Meyer, editor)*, 36(9):37–47.
- [Leclercq et al. 2005a] Leclercq, M., Quéma, V., and Stefani, J.-B. (2005a). DREAM : un canevas logiciel à composants pour la construction d'intergiciels orientés messages dynamiquement configurables. In *4ème Conférence Francophone autour des Composants Logiciels (avec CFSE-RENPAR 2005)*, Le Croisic, France.
- [Leclercq et al. 2005b] Leclercq, M., Quema, V., and Stefani, J.-B. (2005b). DREAM: a component framework for the construction of resource-aware, configurable middleware. *IEEE Distributed Systems Online*, 6(9).
- [Lendenmann 1996] Lendenmann, R. (1996). *Understanding OSF DCE 1.1 for AIX and OS/2*. Prentice Hall. 312 pp.
- [Levin et al. 1975] Levin, R., Cohen, E. S., Corwin, W. M., Pollack, F. J., and Wulf, W. A. (1975). Policy/mechanism separation in hydra. In *Proceedings of the Fifth ACM Symposium on Operating Systems Principles*, pages 132–140.
- [Liang and Bracha 1998] Liang, S. and Bracha, G. (1998). Dynamic class loading in the java virtual machine. In *Conference on Object Oriented Programming, Systems, Languages, and Applications (OOPSLA '98)*.
- [Limthanmaphon and Zhang 2004] Limthanmaphon, B. and Zhang, Y. (2004). Web service composition transaction management. In Schewe, K.-D. and Williams, H., editors, *ADC*, volume 27 of *CRPIT*. Database Technologies 2004, Proceedings of the Fifteenth Australian Database Conference, ADC 2004, Dunedin, New Zealand, Australian Computer Society.

- [Lindholm and Yellin 1996] Lindholm, T. and Yellin, F. (1996). *The Java Virtual Machine Specification*. Addison-Wesley. 475 pp.
- [Liskov 1988] Liskov, B. (1988). Distributed programming in Argus. *Communications of the ACM*, 31(3):300–312.
- [Maes 1987] Maes, P. (1987). Concepts and Experiments in Computational Reflection. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '87)*, pages 147–155, Orlando, Florida, USA.
- [Martens 2005] Martens, A. (2005). Analyzing Web Service Based Business Processes. In Cerioli, M., editor, *Proceedings of the 8th International Conference on Fundamental Approaches to Software Engineering (FASE 2005)*, pages 19–33. Springer.
- [McIlroy 1968] McIlroy, M. (1968). Mass produced software components. In Naur, P. and Randell, B., editors, *Software Engineering: A Report On a Conference Sponsored by the NATO Science Committee*, pages 138–155, Garmisch, Germany.
- [Merle et al. 2004] Merle, P., Moroy, J., Rouvoy, R., and Contreras, C. (2004). *Fractal Explorer*. ObjectWeb.
fractal.objectweb.org/tutorials/explorer/index.html.
- [Meyer 1992] Meyer, B. (1992). Applying Design by Contract. *IEEE Computer*, 25(10):40–52.
- [Middleware 1998] Middleware (1998). IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing. September 15–18 1998, The Lake District, England.
- [Mitchell et al. 1994] Mitchell, J. G., Gibbons, J., Hamilton, G., Kessler, P. B., Khalidi, Y. Y. A., Kougiouris, P., Madany, P., Nelson, M. N., Powell, M. L., and Radia, S. R. (1994). An overview of the Spring system. In *Proceedings of COMPCON*, pages 122–131.
- [Monson-Haefel 2002] Monson-Haefel, R. (2002). *Enterprise JavaBeans*. O'Reilly & Associates, Inc., 3rd edition. 550 pp.
- [msmq 2002] msmq (2002). Microsoft Message Queuing (MSMQ). Microsoft,
<http://www.microsoft.com/msmq/>.
- [Mullender et al. 1990] Mullender, S. J., van Rossum, G., Tanenbaum, A. S., van Renesse, R., and van Staveren, H. (1990). Amoeba: A distributed operating system for the 1990s. *IEEE Computer*, 23(5):44–53.
- [Naur and Randell 1969] Naur, P. and Randell, B., editors (1969). *Software Engineering: A Report On a Conference Sponsored by the NATO Science Committee, 7-11 Oct. 1968*. Scientific Affairs Division, NATO. 231 pp.
- [.NET] .NET. Microsoft Corp. <http://www.microsoft.com/net>.
- [OASIS UDDI 2005] OASIS UDDI (2005). Universal Description, Discovery and Integration version 3.0. <http://www.uddi.org>.
- [ObjectWeb 1999] ObjectWeb (1999). Open Source Middleware. <http://www.objectweb.org>.
- [ObjectWeb 2004] ObjectWeb (2004). *The Kilim Project*. kilim.objectweb.org.
- [Objectweb Enhydra 2005] Objectweb Enhydra (2005). Enhydra XMLC.
<http://xmlc.objectweb.org/>.
- [ODMG] ODMG. The Object Data Management Group. <http://www.odmg.org>.
- [ODP 1995a] ODP (1995a). ITU-T & ISO/IEC, Recommendation X.902 & International Standard 10746-2: “ODP Reference Model: Foundations”.
http://archive.dstc.edu.au/AU/research_news/odp/ref_model/standards.html.

- [ODP 1995b] ODP (1995b). ITU-T & ISO/IEC, Recommendation X.903 & International Standard 10746-3: "ODP Reference Model: Architecture".
http://archive.dstc.edu.au/AU/research_news/odp/ref_model/standards.html.
- [OMG] OMG. The Object Management Group. <http://www.omg.org>.
- [Open Group] Open Group. <http://www.opengroup.org/>.
- [Oracle 1994] Oracle (1994). Oracle topLink.
<http://www.oracle.com/technology/products/ias/toplink/index.html>.
- [O'Sullivan et al. 2002] O'Sullivan, J., Edmond, D., and ter Hofstede, A. (2002). What's in a Service? *Distributed and Parallel Databases*, 12(2-3):117-133.
- [Papazoglou 2003] Papazoglou, M. (2003). Web services and business transactions. Technical Report 6, Infolab, Tilburg University, Netherlands.
- [Parrington et al. 1995] Parrington, G. D., Shrivastava, S. K., Wheater, S. M., and Little, M. C. (1995). The design and implementation of Arjuna. *Computing Systems*, 8(2):255-308.
- [Pawlak et al. 2001] Pawlak, R., Duchien, L., Florin, G., and Seinturier, L. (2001). JAC : a flexible solution for aspect oriented programming in Java. In Yonezawa, A. and Matsuoka, S., editors, *Proceedings of Reflection 2001, the Third International Conference on Metalevel Architectures and Separation of Crosscutting Concerns*, volume 2192 of *LNCS*, pages 1-24, Kyoto, Japan. Springer-Verlag.
- [Peltz 2003] Peltz, C. (2003). Web services orchestration and choreography. *IEEE Computer*, 36(10):46-52.
- [PicoContainer 2004] PicoContainer (2004). *The PicoContainer Project*. www.picocontainer.org.
- [Platt 1999] Platt, D. S. (1999). *Understanding COM+*. Microsoft Press. 256 pp.
- [Plexus 2004] Plexus (2004). *The Plexus Project*. plexus.codehaus.org.
- [PLoP] PLoP. The Pattern Languages of Programs (PLoP) Conference Series.
<http://www.hillside.net/conferences/plop.htm>.
- [Quéma et al. 2004] Quéma, V., Balter, R., Bellissard, L., Féliot, D., Freyssinet, A., and Lacourte, S. (2004). Scalagent : une plate-forme à composants pour applications asynchrones. *Technique et Science Informatiques*, 23(2).
- [Quéma 2005] Quéma, V. (2005). Vers l'exogiciel - une approche de la construction d'infrastructures logicielles radicalement configurables. Thèse de Doctorat de l'Institut National Polytechnique de Grenoble, décembre 2005.
- [RM 2000] RM (2000). *Workshop on Reflective Middleware*. Held in conjunction with Middleware 2000, 7-8 April 2000. <http://www.comp.lancs.ac.uk/computing/RM2000/>.
- [Rodriguez 2005] Rodriguez, E. (2005). The Apache Directory Server and the OSGi Service Platform. OSGi World Congress, 12-15 October 2005, Paris, France.
- [Rogerson 1997] Rogerson, D. (1997). *Inside COM*. Microsoft Press, Redmond, USA.
- [Rouvoy 2004] Rouvoy, R. (2004). *The GoTM Project*. ObjectWeb. gotm.objectweb.org.
- [Rozinat and Aalst 2005] Rozinat, A. and Aalst, W. (2005). Conformance Testing: Measuring the Alignment Between Event Logs and Process Models. BETA Working Paper Series, WP 144, Eindhoven University of Technology, Eindhoven.

- [SCA 2005] SCA (2005). *Service Component Architecture Assembly Model Specification*. www-128.ibm.com/developerworks/library/specification/ws-sca/.
- [Schantz et al. 1986] Schantz, R., Thomas, R., and Bono, G. (1986). The architecture of the Cronus distributed operating system. In *Proceedings of the 6th International Conference on Distributed Computing Systems*, pages 250–259. IEEE.
- [Schmidt et al. 2000] Schmidt, D. C., Stal, M., Rohnert, H., and Buschmann, F. (2000). *Pattern-Oriented Software Architecture, Volume 2: Patterns for Concurrent and Networked Objects*. John Wiley & Sons. 666 pp.
- [Seinturier et al. 2006] Seinturier, L., Pessemier, N., Duchien, L., and Coupaye, T. (2006). A component model engineered with components and aspects. In *Proceedings of the 9th International SIGSOFT Symposium on Component-Based Software Engineering (CBSE'06)*, volume 4063 of *Lecture Notes in Computer Science*, pages 139–153. Springer.
- [Shapiro 1986] Shapiro, M. (1986). Structure and encapsulation in distributed systems: The proxy principle. In *Proc. of the 6th International Conference on Distributed Computing Systems*, pages 198–204, Cambridge, Mass. (USA). IEEE.
- [Shapiro et al. 1989] Shapiro, M., Gourhant, Y., Habert, S., Mosseri, L., Ruffin, M., and Valot, C. (1989). SOS: An object-oriented operating system - assessment and perspectives. *Computing Systems*, 2(4):287–337.
- [Smith 1982] Smith, B. C. (1982). *Reflection And Semantics In A Procedural Language*. PhD thesis, Massachusetts Institute of Technology. MIT/LCS/TR-272.
- [Spring 2004] Spring (2004). *The Spring Framework*. www.springframework.org.
- [Strom et al. 1998] Strom, R., Banavar, G., Chandra, T., Kaplan, M., Miller, K., Mukherjee, B., Sturman, D., and Ward, M. (1998). Gryphon: An Information Flow Based Approach to Message Brokering. In *Proceedings of ISSRE'98*.
- [Stutz et al. 2003] Stutz, D., Ted Neward, and Geoff Shilling (2003). *Shared Source CLI Essentials*. O'Reilly. ISBN 0-596-00351-x, 378 pages.
- [Sun 2005] Sun (2005). J2EE Application Validation Kit. <http://java.sun.com/j2ee/avk/>.
- [Sun JSR-000116 2003] Sun JSR-000116 (2003). SIP Servlet API. <http://jcp.org/aboutJava/communityprocess/final/jsr116/index.html>.
- [Sun JSR-000127 2004] Sun JSR-000127 (2004). JavaServer Faces Specification. <http://jcp.org/aboutJava/communityprocess/final/jsr127/index2.html>.
- [Sun JSR-000152 2003] Sun JSR-000152 (2003). JavaServer Pages 2.0 Specification. <http://jcp.org/aboutJava/communityprocess/final/jsr152/index.html>.
- [Sun JSR-000153 2003] Sun JSR-000153 (2003). Java Entreprise Bean 2.1 Specification. <http://www.jcp.org/aboutJava/communityprocess/final/jsr153/>.
- [Sun JSR-000154 2003] Sun JSR-000154 (2003). Java Servlet 2.4 Specification. <http://jcp.org/aboutJava/communityprocess/final/jsr154/index.html>.
- [Sun JSR-000904 2000] Sun JSR-000904 (2000). JavaMail Specification. <http://jcp.org/aboutJava/communityprocess/maintenance/jsr904/index.html>.
- [Sun Microsystems 2003] Sun Microsystems (2003). Java Remote Method Invocation (RMI). <http://java.sun.com/products/jdk/rmi/>.

- [Suvée et al. 2005] Suvée, D., Vanderperren, W., and Jonckers, V. (2005). FuseJ: An architectural description language for unifying aspects and components. In *Workshop Software-engineering Properties of Languages and Aspect Technologies (SPLAT) at AOSD'05*. ssel.vub.ac.be/Members/dsuvee/papers/splatsuvee2.pdf.
- [Szyperski 2002] Szyperski, C. (2002). *Component Software - Beyond Object-Oriented Programming*. Addison-Wesley. 2nd ed., 589 pp.
- [Szyperski and Pfister 1997] Szyperski, C. and Pfister, C. (1997). Component-oriented programming: WCOP'96 workshop report. In Mühlhäuser, M., editor, *Workshop Reader of 10th Eur. Conf. on Object Oriented Programming ECOOP'96, Linz, July 8-12*, Special Issues in Object-Oriented Programming, pages 127–130. Dpunkt, Heidelberg.
- [Tatsumi et al. 2001] Tatsumi, M., Sasaki, T., Chiba, S., and Itano, K. (2001). A Bytecode Translator for Distributed Execution of “Legacy” Java Software. In *ECOOP 2001 – Object-Oriented Programming*, volume 2072 of *LNCs*, pages 236–255. Springer Verlag.
- [Team 2003] Team, F. (2003). Fractal: a modular and extensible component model. <http://fractal.objectweb.org/>.
- [Team 1998] Team, J. (1998). Jorm: a framework for mapping typed objects to various storage systems. <http://jorm.objectweb.org/>.
- [Team 2000] Team, M. (2000). Medor: a middleware framework for enabling distributed object requests. <http://medor.objectweb.org/>.
- [The Apache Software Foundation 2005] The Apache Software Foundation (2005). The Struts Framework. <http://struts.apache.org/>.
- [The Apache Software Foundation 2006] The Apache Software Foundation (2006). Apache Geronimo. <http://geronimo.apache.org/>.
- [The Objectweb Consortium 2000] The Objectweb Consortium (2000). JOnAS: Java (TM) Open Application Server. <http://jonas.objectweb.org/>.
- [Tuscany 2006] Tuscany (2006). *The Tuscany Project*. incubator.apache.org/projects/tuscany.html.
- [UN/CEFACT and OASIS 1999] UN/CEFACT and OASIS (1999). Electronic Business XML. <http://www.ebxml.org>.
- [UPnP Forum 2005] UPnP Forum (2005). Universal Plug and Play Forum. www.upnp.org.
- [van Renesse et al. 2003] van Renesse, R., Birman, K., and Vogels, W. (2003). Astrolabe: A robust and scalable technology for distributed system monitoring, management, and data mining. *ACM Transactions on Computer Systems*, 21(2).
- [Vidyasankar and Vossen 2003] Vidyasankar, K. and Vossen, G. (2003). A multi-level model for web service composition. Technical report, Dept. of Information Systems, University of Muenster. Tech. Report No. 100.
- [Völter et al. 2002] Völter, M., Schmid, A., and Wolff, E. (2002). *Server Component Patterns*. John Wiley & Sons. 462 pp.

- [W3C-DOM 2005] W3C-DOM (2005). The Document Object Model (DOM), W3C Consortium. <http://www.w3.org/DOM>.
- [W3C-WSA-Group 2004] W3C-WSA-Group (2004). W3C Web Service Architecture Group. Web Services Architecture. <http://www.w3.org/TR/ws-arch>.
- [W3C-WSD-Group] W3C-WSD-Group. Web Services Description Working Group. <http://www.w3.org/2002/ws/desc/>.
- [W3C-XML] W3C-XML. XML coordination group. <http://www.w3.org/XML/>.
- [W3C-XMLP-Group] W3C-XMLP-Group. XML Protocol working group. <http://www.w3.org/2000/xp/Group/>.
- [W3C-XMLSchema] W3C-XMLSchema. XML coordination group. <http://www.w3.org/XML/Schema>.
- [Waldo 1999] Waldo, J. (1999). The Jini architecture for network-centric computing. *Communications of the ACM*, 42(7):76–82.
- [Waldo et al. 1997] Waldo, J., Wyant, G., Wollrath, A., and Kendall, S. (1997). A Note on Distributed Computing. In Vitek, J. and Tschudin, C., editors, *Mobile Object Systems: Towards the Programmable Internet*, volume 1222 of *LNCS*, pages 49–64. Springer-Verlag.
- [Weatherley and Gopal 2003] Weatherley, R. and Gopal, V. (2003). Design of the Portable.Net Interpreter DotGNU. In *Linux.conf.au, Perth (Australia)*, 22-25 January.
- [Weerawarana et al. 2005] Weerawarana, S., Curbera, F., Leymann, F., Story, T., and Ferguson, D. (2005). *Web Services Platform Architecture: SOAP, WSDL, WS-Policy, WS-Addressing, WS-BPEL, WS-Reliable-Messaging, and More*. Prentice Hall.
- [Weinand et al. 1988] Weinand, A., Gamma, E., and Marty, R. (1988). ET++ - An Object-Oriented Application Framework in C++. In *Proceedings of OOPSLA 1988*, pages 46–57.
- [Weiser 1993] Weiser, M. (1993). Some computer science issues in ubiquitous computing. *Communications of the ACM*, 36(7):74–84.
- [White 1976] White, J. E. (1976). A high-level framework for network-based resource sharing. In *National Computer Conference*, pages 561–570.
- [Winer 1999] Winer, D. (1999). XML-RPC specification. <http://www.xmlrpc.com/spec>.
- [Wohed et al. 2003] Wohed, P., Aalst, W., Dumas, M., and Hofstede, A. (2003). Analysis of Web Services Composition Languages: The Case of BPEL4WS. In *22nd International Conference on Conceptual Modeling (ER 2003)*, pages 200–215. Springer.
- [Wollrath et al. 1996] Wollrath, A., Riggs, R., and Waldo, J. (1996). A Distributed Object Model for the Java System. *Computing Systems*, 9(4):265–290.
- [Yellin and Strom 1997] Yellin, D. and Strom, R. (1997). Protocol specifications and component adaptors. 19(2):292–333.