ÉTUDES, DÉVELOPPEMENT & INTÉGRATION

ARCHITECTURES RÉPARTIES EN JAVA

Middleware Java, services web, messagerie instantanée, transfert de données



RESSOURCES NUMÉRIQUES

Annick Fron

3e édition

DUNOD

Toutes les marques citées dans cet ouvrage sont des marques déposées par leurs propriétaires respectifs.

Illustration de couverture :
© iStock.com / ispyfriend

Le pictogramme qui figure ci-contre mérite une explication. Son objet est d'alerter le lecteur sur la menace que représente pour l'avenir de l'écrit, particulièrement dans le domaine de l'édition technique et universitaire, le développement massif du photocopillage.
Le Code de la propriété intellectuelle du 1er juillet 1992 interdit en effet expressément la photocopie à usage collectif sans autorisation des ayants droit. Or, cette pratique s'est généralisée dans les établissements d'enseignement supérieur, provoquant une baisse brutale des achats de livres et de revues, au point que la possibilité même pour les auteurs de créer des œuvres nouvelles et de les faire éditer correctement est aujourd'hui menacée.
Nous rappelons donc que toute reproduction, partielle ou totale, de la présente publication est interdite sans autorisation de l'auteur, de son éditeur ou du Centre français d'exploitation du droit de copie (CFC, 20, rue des Grands-Augustins, 75006 Paris).



© Dunod, 2007, 2012, 2015
5 rue Laromiguière, 75005 Paris
www.dunod.com

ISBN 978-2-10-074403-9

Le Code de la propriété intellectuelle n'autorisant, aux termes de l'article L. 122-5, 2° et 3° a), d'une part, que les « copies ou reproductions strictement réservées à l'usage privé du copiste et non destinées à une utilisation collective » et, d'autre part, que les analyses et les courtes citations dans un but d'exemple et d'illustration, « toute représentation ou reproduction intégrale ou partielle faite sans le consentement de l'auteur ou de ses ayants droit ou ayants cause est illicite » (art. L. 122-4).
Cette représentation ou reproduction, par quelque procédé que ce soit, constituerait donc une contrefaçon sanctionnée par les articles L. 335-2 et suivants du Code de la propriété intellectuelle.

Copyright © 2015 Dunod.

Table des matières

Avant-propos XV

Chapitre 1 – Problématique des applications réparties 1

1.1 Applications réparties 1

1.1.1 Communication entre programmes 1

1.1.2 Répartition et parallélisme 2

1.1.3 Les huit erreurs conceptuelles des applications réparties 3

1.2 Concepts d'applications réparties 4

1.2.1 Grain des applications 4

1.2.2 Couplage des applications 4

1.2.3 MOM et RPC 5

1.2.4 Mode synchrone et asynchrone 6

1.2.5 Sémantique de communication et qualité de service 7

1.3 Impact sur le génie logiciel 7

1.3.1 Modification de la conception 7

1.3.2 Modification du processus de développement 8

Chapitre 2 – Rappels sur Java 11

2.1 Sérialisation 11

2.1.1 Sérialisation d'un objet 11

2.1.2 Graphes d'objets 13

2.1.3 Personnalisation de la sérialisation 15

Copyright © 2015 Dunod.

© Dunod – Toute reproduction non autorisée est un délit.

2.1.4 *Gestion des versions* 15
2.2 JSON, JavaScript Object Notation : une autre technique de sérialisation 16
2.2.1 *Comparaison de JSON et XML* 16
2.2.2 *La syntaxe* 17
2.2.3 *Exemple de fichier au format JSON* 17
2.2.4 *Mappings en Java* 19
2.2.5 *Mise en œuvre* 20
2.2.6 *Conclusion* 21
2.3 Protocol Buffers : une technique de sérialisation performante et déclinée sur plusieurs langages 21
2.3.1 *Une sérialisation performante et versatile* 21
2.3.2 *Mise en œuvre* 21
2.4 Utilisation des threads : activités en parallèle 25
2.4.1 *Thread et processus* 25
2.4.2 *Simulation du parallélisme* 26
2.4.3 *Création d'un thread* 27
2.4.4 *Partage de mémoire entre threads* 28
2.4.5 *Utilisation de ExecutorService et sémantique d'exécution* 29
2.5 Utilisation des futurs 30
2.5.1 *Programmation asynchrone* 30
2.5.2 *Synchronisation sur attente du résultat* 30
2.5.3 *Utilisation de la puissance de calcul des multiprocesseurs avec les tâches Fork-Join en Java 7* 33
2.5.4 *Nouvelle API asynchrone en Java 8* 35
2.6 Design patterns 45
2.6.1 *Pattern de la fabrique* 46
2.6.2 *Pattern du mandataire* 47
2.6.3 *Proxies dynamiques* 49
2.7 Propriétés en Java 50
2.8 Réflexion 50
2.8.1 *Classe d'un objet* 51
2.8.2 *Méthodes d'une classe* 52

Copyright © 2015 Dunod.

Chapitre 3 – Sockets et Java NIO 55

3.1 Communication en réseau 55

3.1.1 Couches basses du système 55

3.1.2 Adresses en IPv4, IPv6 et interfaces réseau 56

3.1.3 TCP et UDP 57

3.1.4 Port de communication 57

3.1.5 Télécharger un fichier web à partir d'une URL 58

3.2 Communication avec TCP 59

3.2.1 Mode client serveur 59

3.2.2 Code d'une socket serveur et d'une socket cliente 60

3.2.3 Réception de l'information par le serveur 60

3.2.4 Utilisation de TCP 61

3.3 Utilisation de threads dans un serveur 61

3.3.1 Connexion de plusieurs clients 61

3.3.2 Utilisation de threads pour le traitement de requêtes en parallèle 62

3.3.3 Limitations du nombre de threads : pool de threads 63

3.4 Autres types de sockets 64

3.4.1 Sockets avec UDP 64

3.4.2 Multicast 67

3.5 Utilisation de Java NIO 68

3.5.1 Une nouvelle bibliothèque 68

3.5.2 Canaux et tampons 69

3.5.3 Sockets avec NIO 71

3.6 Nouvelle socket asynchrone en Java 7 : pour plus de performance 74

3.6.1 Intérêt des sockets asynchrones 74

3.6.2 Programme du serveur 75

3.6.3 Programme du client 76

Chapitre 4 – Interfaces et pliage des arguments 79

4.1 L'interface, un contrat entre client et serveur 79

4.1.1 Insuffisance des sockets 79

4.1.2 Spécification des interfaces 80

4.1.3 Partager des types de données entre les applications 82

Copyright © 2015 Dunod.

© Dunod – Toute reproduction non autorisée est un délit.

4.2 Pliage des arguments (marshalling) 83
4.2.1 Représentation des données 83
4.2.2 Passage des arguments par valeur ou par référence 84
4.3 Interface, implémentation et déploiement 88
4.3.1 Une implémentation pour plusieurs interfaces 88
4.3.2 Plusieurs implémentations de la même interface 89

Chapitre 5 – RMI 91
5.1 Comment utiliser RMI 91
5.1.1 Positionnement de RMI 91
5.1.2 Définition des objets distants 92
5.2 Mise en œuvre de RMI 93
5.2.1 Définition des interfaces distantes 93
5.2.2 Implémentation de l'interface 94
5.2.3 La génération de la souche 96
5.2.4 Enregistrement de l'objet dans l'annuaire et code serveur 97
5.2.5 Appel distant depuis le client 98
5.2.6 Arrêt d'un objet serveur 98
5.3 Passage des arguments 100
5.3.1 Passage des arguments par référence 100
5.3.2 Passage des arguments par valeur 101
5.3.3 Exceptions 102
5.3.4 Création d'objets à distance 103
5.4 RMI en multithread 103
5.4.1 Exclusion mutuelle pour éviter des accès concurrents à une variable 103
5.4.2 Synchronisation sur le client 104
5.5 Activation avec RMI 105
5.5.1 Modifications de l'implémentation du serveur pour l'activation 105
5.5.2 Activation de l'objet distant 106
5.5.3 Code du client 107
5.5.4 Mise en œuvre 108
5.6 Autres aspects de RMI 109
5.6.1 Gestion du ramasse-miettes 109
5.6.2 Modification du protocole de RMI 110

Copyright © 2015 Dunod.

5.7 RMI IIOP 110
5.7.1 *Premières étapes de mise en œuvre* 111
5.7.2 *Nouvel annuaire CORBA* 111

Chapitre 6 – CORBA 113

6.1 Évolution de CORBA 113
6.1.1 *L'OMG* 113
6.1.2 *CORBA et Java* 115
6.1.3 *Pourquoi utiliser CORBA ?* 116
6.2 Architecture de CORBA 116
6.2.1 *Appels distants* 116
6.2.2 *L'ORB* 117
6.2.3 *Souches et POA* 118
6.2.4 *L'IDL* 122
6.2.5 *Les IOR* 124
6.2.6 *Les services CORBA* 125
6.2.7 *Utilisation de CORBA en Java* 126
6.3 Un exemple statique simple : client et serveur en Java 127
6.3.1 *IDL et classes générées* 127
6.3.2 *Implémentation des classes* 130
6.4 Variantes de code 132
6.4.1 *Implémentation par délégation et non par héritage* 132
6.4.2 *Appel dynamique d'un client* 133
6.5 Utilisation d'un annuaire 134
6.5.1 *Utilisation de l'annuaire côté serveur* 135
6.5.2 *Hiérarchie de contextes* 136
6.5.3 *Code du client* 136
6.5.4 *Service d'annuaire interopérable* 137
6.6 Passage d'arguments 137
6.6.1 *Passage d'arguments par référence* 137
6.6.2 *Passage des arguments par valeur* 140
6.7 Utilisation des politiques du POA 141
6.8 Introduction à DDS 144

Copyright © 2015 Dunod.
© Dunod – Toute reproduction non autorisée est un délit.

Chapitre 7 – Annuaires-LDAP-JNDI 147

7.1 Références d'applications et annuaires 147

7.1.1 Gestion des références en réseau 147

7.1.2 Références distantes 148

7.1.3 Points d'entrée dans le système 149

7.2 Enregistrement d'un objet dans un annuaire 149

7.2.1 Unicité du nom 150

7.2.2 Mise à disposition d'un objet 150

7.2.3 Impact sur les performances 151

7.2.4 Modification de référence d'un objet 152

7.3 Conception d'un service d'annuaire 152

7.3.1 Quels objets stocker dans le service de nommage ? 152

7.3.2 Utilisation de plusieurs annuaires en parallèle 153

7.3.3 Utilisation de pointeurs intelligents pour référencer un ensemble d'objets 154

7.4 JNDI 156

7.4.1 Service de noms et annuaire 156

7.4.2 Les multiples implémentations de JNDI 157

7.4.3 Paramètres d'implémentation 157

7.5 Utilisation de JNDI 159

7.5.1 Opérations de nommage 159

7.5.2 Opérations d'annuaire sur les attributs 160

7.6 Introduction à LDAP 161

7.6.1 Annuaire d'entreprise 161

7.6.2 La hiérarchie de noms en LDAP 163

7.6.3 LDAP comme fédérateur d'annuaires 163

7.7 Utilisation de JNDI pour les accès LDAP 164

7.7.1 Stockage d'un objet sérialisé dans un annuaire LDAP 165

7.7.2 Stockage d'un objet CORBA dans LDAP 166

7.7.3 Stockage d'une référence à un objet RMI dans un annuaire LDAP 166

7.7.4 Stockage d'une ConnectionFactory JMS dans LDAP 167

7.8 Recherche d'éléments dans LDAP 168

Copyright © 2015 Dunod.

Chapitre 8 – JMS 169

8.1 Java Messaging Service 169

8.1.1 Une API standard 169

8.1.2 Communication à plusieurs 170

8.1.3 Le modèle de producteur/consommateur - les destinations 171

8.1.4 Structure des messages 174

8.1.5 Connexion et session 174

8.2 Exemple de code pour envoyer et recevoir un message en JMS 176

8.2.1 Envoi d'un message 176

8.2.2 Réception d'un message synchrone 177

8.2.3 Réception d'un message asynchrone 178

8.2.4 Retour sur la création des connexions et des sessions 180

8.2.5 Utilisation des hiérarchies de Topic 182

8.3 Nature des messages et codage de l'information 183

8.3.1 Le corps du message 183

8.3.2 Propriétés des messages : filtrage et routage 185

8.3.3 En-tête des messages et qualité de service 187

8.4 Fiabilité de la communication 188

8.4.1 Abonnés durables 188

8.4.2 Séquence des messages 189

8.4.3 Acquittements de messages 190

8.4.4 Utilisation des transactions 193

8.5 JMS 2.0 194

8.5.1 Utilisation de l'API simplifiée pour envoyer un message 195

8.5.2 Utilisation de JMS 2.0 pour la réception synchrone des messages 196

8.5.3 Utilisation de JMS 2.0 pour la réception asynchrone des messages 196

8.5.4 Autres simplifications en JMS 2.0 197

8.5.5 Nouvelle fonctionnalité : plusieurs consommateurs sur le même abonnement 197

8.5.6 Nouvelle fonctionnalité : délai de livraison 198

8.5.7 Nouvelle fonctionnalité : envoi asynchrone 198

8.5.8 Nouvelle fonctionnalité : comptage de livraison 199

Copyright © 2015 Dunod.

© Dunod – Toute reproduction non autorisée est un délit.

Chapitre 9 – SOAP et services web 201

9.1 Communication entre applications sur Internet 201

9.1.1 Enjeux du « B to B » 201

9.1.2 Intérêt d'un mode RPC sur Internet 202

9.1.3 Service et… qualité de service 203

9.1.4 Standardisation des services web 204

9.2 Architecture de SOAP 204

9.2.1 Trois composants fondamentaux 204

9.2.2 Requêtes HTTP et messages SOAP 205

9.2.3 Versalitité de SOAP 207

9.2.4 Interopérabilité des protocoles : les attaches 207

9.3 WSDL 208

9.3.1 Un langage d'interface 208

9.3.2 Structure d'un document WSDL 209

9.3.3 Plusieurs styles WSDL : lien avec l'interopérabilité 212

9.4 Bijections entre Java et XML 215

9.4.1 Modèle objet ou modèle de document 215

9.4.2 Bibliothèques Java-XML 216

9.5 Client SOAP en Java 217

9.5.1 Choix d'implémentation 217

9.5.2 Apache Axis 217

9.5.3 Téléchargement du fichier d'interface 218

9.6 Création d'un service web avec Java 6 219

9.7 Création d'un service web avec JAX-WS 221

9.7.1 JAX-WS 221

9.7.2 Téléchargement du logiciel nécessaire 222

9.7.3 Lancement du serveur web Tomcat et arborescence de fichiers 223

9.7.4 Code de l'application serveur 226

9.7.5 Passage des paramètres 228

9.7.6 Mise en œuvre 230

9.7.7 Test de l'application et code du client 235

9.7.8 Variations d'implémentation 236

9.8 Création d'un service web avec Axis 239

Copyright © 2015 Dunod.

9.9 Comparaison de déploiement ... 244
9.10 REST – Representational State Transfer ... 246
9.10.1 *Les méthodes HTTP et les services REST* ... 246
9.10.2 *Mise en œuvre sous Eclipse* ... 247
9.10.3 *Implémentation du code du domaine* ... 249
9.10.4 *Mise en œuvre du serveur* ... 250
9.10.5 *Déploiement et test* ... 253
9.10.6 *Client Java de test* ... 255

Chapitre 10 – JMX, test et déploiement ... 259
10.1 Architecture de JMX ... 259
10.1.1 *JMX est une API de J2SE* ... 259
10.1.2 *Motivation de JMX* ... 260
10.1.3 *Architecture de JMX* ... 261
10.2 Mise en œuvre de JMX ... 263
10.2.1 *Plusieurs types de MBeans* ... 263
10.2.2 *Nommage des MBeans* ... 265
10.2.3 *Mise à disposition d'un MBean dans un serveur* ... 265
10.2.4 *Ajout d'un connecteur dans le serveur* ... 266
10.2.5 *Notifications* ... 269
10.2.6 *Requêtes d'un serveur MBeanServer* ... 270
10.2.7 *Découvertes d'agents JMX et annuaires* ... 271
10.3 Journalisation ... 271
10.3.1 *La journalisation partie prenante du développement et de l'administration* ... 271
10.3.2 *La bibliothèque Log4J* ... 272
10.3.3 *Les différents types de composants* ... 272
10.3.4 *Utilisation de Log4J* ... 274

Chapitre 11 – Courriels et messagerie ... 277
11.1 Utilisation de courriels en Java ... 277
11.1.1 *Introduction aux serveurs de courriel* ... 277
11.1.2 *L'interface JavaMail* ... 278
11.1.3 *Les principales classes* ... 278
11.1.4 *Le code d'envoi d'un message* ... 278

Copyright © 2015 Dunod.
© Dunod – Toute reproduction non autorisée est un délit.

11.1.5 Le code de réception des messages 279
11.1.6 Pièces attachées 280
11.1.7 Autres fonctionnalités 281
11.2 XMPP 282
11.2.1 Une norme 282
11.2.2 OpenFire 284

Bibliographie 291

Index 295

Copyright © 2015 Dunod.

Avant-propos

Java réparti est partout

Ce livre sur les architectures réparties en Java vise à démystifier les applications réparties : réparties, ou distribuées (de l'anglais *distributed*), elles sont omniprésentes, et ne sont plus réservées à de grands comptes et des équipes de pointe. Java est désormais intégré dans de très nombreux appareils, depuis les téléphones mobiles jusqu'aux serveurs d'entreprise. On parle d'architectures réparties chaque fois que deux appareils communiquent entre eux, et le champ d'action embrasse un vaste domaine d'applications : collecte de données d'automates vers la gestion de production, surveillance par vidéo, applications interactives sur les mobiles, communication entre applications sur un même ordinateur, lien entre centres de contrôle aérien, communication avec un satellite...On emploie de plus en plus le terme *middleware*, qui désigne en anglais les logiciels d'intermédiation : le *middleware* a un rôle important puisqu'il sert de messager entre les composants de l'application, mais il est souvent négligé car peu visible par l'utilisateur final.

Java simplifie le développement des applications réparties

Très longtemps le terme « applications réparties » a rimé avec CORBA (*Common Object Request Broker Architecture*), et repoussait le plus hardi développeur par une complexité de développement peu intuitive et des savoirs jalousement gardés par des gourous.

Java a facilité le processus de développement en faisant des applications réparties une commodité intégrée au langage – RMI (*Remote Method Invocation*) est un package de la version standard de Java. Plus besoin d'acheter des intergiciels (*middleware*) onéreux et coûteux en formation. Par contre le panorama de technologies disponibles (RMI, CORBA, SOAP, XML-RPC, JMS, JXTA) ne cesse de se diversifier et d'offrir de nouvelles fonctionnalités : sécurité, activation automatique, transactions...

Le langage a également des atouts pour faciliter la communication entre machines : téléchargement de code, gestion automatique de la mémoire, gestion des fils d'exécution (*threads*). Tous ces éléments sont autant d'avantages pour un développement bien plus facile qu'en C++.

Copyright © 2015 Dunod.
© Dunod – Toute reproduction non autorisée est un délit.

Java a créé une communauté qui a permis une large diffusion des concepts à mettre en œuvre pour définir des architectures, et qui plus est des architectures réparties : la facilité de développement a permis de se concentrer non plus sur les couches de bas niveau mais sur la vision complexe de systèmes complets depuis l'automate de production jusqu'aux commandes web en ligne au travers de serveurs d'applications.

Java a enfin développé des processus d'innovation dans les groupes de travail JSR (*Java Specification Requests*), qui ont permis une intégration continue de nouvelles fonctionnalités et d'optimisations, en particulier sur les techniques de concurrence (synchronisation et gestin des activités des programmes) et de communication en réseau.

Théorie et expérience : un livre pragmatique

Ce livre a mûri au cours de mon expérience d'enseignement de deux ans et demi en école d'ingénieurs, mais aussi au travers de nombreuses missions de consultant en entreprise pour répondre à des problèmes toujours différents. Il est le fruit de très nombreuses heures d'expérimentation avec du code, et de recherches incessantes sur Internet car ce monde est toujours en mouvement.

Il est destiné à la fois aux architectes et aux développeurs, car rien ne vaut une compréhension du code pour comprendre l'architecture globale et les concepts mis en œuvre. Il n'a aucun but exhaustif, mais au contraire se veut assez large et il a pour vocation d'aiguiller le lecteur vers des pistes d'approfondissement plus spécialisées.

Il cherche à faire ressortir les concepts clés des applications réparties en essayant de stopper pour un moment l'actualité Java afin de fixer les idées. Il vise à simplifier les concepts afin de ne retenir que l'essentiel, quitte à renvoyer le lecteur sur des références Internet pour plus de détails. Ce livre veut également tenir à distance le langage marketing des concepts à la mode tels que SOA (*Service Oriented Architecture* ou encore architecture fondée sur les services) pour extraire la substantifique moelle de la manière de faire communiquer deux machines entre elles.

Cet ouvrage peut être lu de deux manières : d'abord comme un **manuel pratique** permettant de retrouver la mise en œuvre des techniques classiques des applications réparties. La version 5 de Java a apporté de nombreuses simplifications qui justifient une mise à jour par rapport à des ouvrages précédents. De plus, peu d'ouvrages font une synthèse aussi œcuménique des diverses technologies. Au long des chapitres, le livre tentera de dégager des concepts communs à plusieurs technologies, afin de fournir des éléments de choix et de comparaison. Le livre identifiera les concepts de synchrone/asynchrone, le pliage des données (*marshalling*) et la sérialisation, la recherche des références, etc.

La deuxième piste de lecture répond à des soucis de **conception et d'architecture**, toutefois les problèmes de qualité de service et d'aspects non fonctionnels (sécurité, transactions...) ne seront qu'ébauchés.

L'ouvrage fait enfin référence à du code disponible en ligne permettant en quelques clics de jouer les exemples proposés, afin de permettre une appropriation rapide par le

Copyright © 2015 Dunod.

lecteur. Le code a été volontairement simplifié pour une lecture synthétique, notamment en omettant les packages et les exceptions qui sont proposées automatiquement par les environnements de développement classique. Des liens sur l'actualité en ligne ou des thèmes de recherche seront également proposés. La bibliographie se résume à l'essentiel, mais les références internet sont par contre nombreuses.

L'anglais cohabite avec le français sur la plupart des mots-clés, afin de permettre une recherche externe plus facile. Les noms de classes qui représentent des objets seront en général sans pluriel.

Ressources en ligne

Téléchargez le code des exemples
– sur www.dunod.com/contenus-complementaires/9782100738700
– ou sur www.afceurope.com.
Il a été volontairement simplifié pour une lecture synthétique – notamment en omettant les packages et les exceptions qui sont proposées automatiquement par les environnements de développement classique.



Communiquez également en direct avec l'auteur à l'adresse info@afceurope.com et visitez son site web www.afceurope.com.

Mise à jour de la deuxième édition

Cinq années se sont écoulées depuis la première édition, et naturellement le paysage Java a beaucoup changé. Oracle a racheté Sun d'une part, d'autre part des initiatives de Java en Open Source ont vu le jour : on notera l'apparition du Open JDK 7 en parallèle avec Java 7. Pour simplifier, ce livre conserve les notations et les classes du Java d'Oracle, et la compatibilité du code avec Open JDK n'est pas mentionnée.

Les machines et les architectures ont également évolué : les machines multiprocesseurs sont devenues très répandues, et les adresses IPv6 commencent à apparaître sur les réseaux. Les services web se sont petit à petit normalisés autour de l'API JAX-WS. La mise à jour prend en compte ces évolutions, et met à jour également toutes les références sur le web, en assurant une compatibilité du code avec les nouvelles versions, notamment Java 6 et 7.

Mise à jour de la troisième édition

Trois années se sont écoulées depuis la deuxième édition, et Java 8 est apparu avec de nouveaux concepts tels que les *lambdas*. Ces concepts n'ont toutefois que peu modifié les fonctionnalités réseau. Par contre les acteurs des nouvelles technologies tels que Facebook et Google sont montés en puissance, et certains chapitres vont esquisser

Copyright © 2015 Dunod.
© Dunod – Toute reproduction non autorisée est un délit.

de loin les technologies utilisées pour le Big Data, les messageries instantanées et les architectures REST. Quelques mises à jour mineures ont été rajoutées sur JMS.

Remerciements

Je tiens enfin à remercier les personnes qui m'ont aidé à relire ce livre, notamment Alexis MOUSSINE-POUCHKINE, Éric DUBUIS, Khanh MAUDOUX, Jean-François PLOTEAU ainsi que ceux qui en ont inspiré la matière comme Didier DONSEZ, Jon SIEGEL. Je remercie enfin les équipes de Schneider Electric et France Telecom avec lesquelles j'ai travaillé sur ces sujets d'applications.

Copyright © 2015 Dunod.

1

Problématique des applications réparties

Objectif

Le but de ce court chapitre est de présenter la problématique des applications réparties et d'introduire les concepts fondamentaux qui seront utilisés tout au long de ce livre

1.1 APPLICATIONS RÉPARTIES

1.1.1 Communication entre programmes

Les applications réparties vont permettre de faire communiquer des programmes entre eux, soit sur des machines différentes au travers du réseau, soit sur la même machine. Toutefois, elles ne se résument pas à l'échange de données en réseau sur le Web, comme c'est le cas lorsque vous utilisez un navigateur Internet, mais mettent en œuvre réellement une synchronisation des programmes entre eux.

L'avènement du B to B (*Business to Business*) permettant par exemple d'interconnecter des systèmes de réservation d'avion et d'hôtel a rendu ce concept populaire, mais il s'étend bien au-delà des échanges sur Internet et se retrouve à l'intérieur des réseaux d'entreprise et des outils de production industrielle. Un nouveau mot à la mode est l'Internet des objets, ou la vision de petits appareils communiquant directement entre eux à travers un réseau. Ce livre décrit les protocoles de communication fondamentaux permettant à des appareils de communiquer entre eux en Java.

Copyright © 2015 Dunod.
© Dunod – Toute reproduction non autorisée est un délit.

Un **intergiciel** (*middleware*) est un programme qui s'insère entre deux applications et qui va permettre de faire communiquer des machines entre elles, indépendamment de la nature du processeur, du système d'exploitation, voire du langage. Dans ce livre, on ne traitera que du langage Java, mais on donnera également des pistes de lecture pour s'interconnecter avec d'autres langages.

Il existe d'autres technologies permettant à deux machines de communiquer mettant en œuvre des systèmes d'exploitation spécifiques ou des architectures de machines spécifiques. Ce livre ne s'adresse qu'aux architectures réparties logicielles s'appuyant sur des systèmes d'exploitation et des processeurs usuels.

1.1.2 Répartition et parallélisme

On fait souvent la confusion entre **répartition** et **parallélisme**. Très souvent, le parallélisme implique des tâches nombreuses mais répétitives, s'effectuant généralement en boucle. Le problème est alors de paralléliser les tâches, ou de paralléliser les données, et il existe des optimisations et des algorithmes nombreux dans ce domaine dont il ne sera pas question ici. Les architectures parallèles exploitent des régularités de traitement ou de données qui limitent leur champ d'application.

Une **architecture répartie** (*distributed*) se définira par des données et des tâches différentes sur les machines. En particulier des applications réparties ne partagent pas de mémoire, contrairement aux applications multiprocesseur. Le problème reste alors la transmission des données entre tâches et la synchronisation entre celles-ci.

Il peut y avoir plusieurs raisons d'utiliser des applications réparties :

- répartir les données **géographiquement** ou topologiquement (*distributed applications*) ;
- permettre la **redondance** d'une application afin de diminuer le taux de pannes et augmenter la fiabilité ;
- permettre la **montée en charge** (*scalability*) ;
- **intégrer des applications existantes** qui cohabitent dans l'entreprise.

Il existe un vaste champ d'application des architectures réparties, depuis les navigateurs web jusqu'aux communications embarquées dans un téléphone ou un satellite. Naturellement, les architectures à mettre en œuvre dépendent des contraintes et du type d'application. Très souvent, une architecture prendra en compte une combinaison de solutions différentes pour résoudre localement telle ou telle contrainte. Cet ouvrage propose des clés de lecture pour effectuer les choix les plus appropriés selon la situation, voire d'interconnecter plusieurs solutions de nature différente : on pourra trouver couramment des passerelles entre web services et RMI, entre JMS et le service de notification de CORBA, etc.

Copyright © 2015 Dunod.

1.1.3 Les huit erreurs conceptuelles des applications réparties

La conception des applications réparties doit souvent lutter contre des intuitions fausses ou des hypothèses ruineuses. Peter Deutsch puis plus tard James Gosling pour la huitième ont gravé dans le marbre les erreurs suivantes [1]:

1. **Le réseau est fiable** : un utilisateur trébuchant dans un câble peut suffire à rendre l'application inutilisable, sans compter les nombreux équipements réseau qui peuvent tomber en panne. Les pertes de connexion sont difficiles à détecter. Ceci sera encore plus crucial pour des architectures ouvertes telles que les services web où on n'a aucun droit de regard sur l'application distante. Concrètement il faudra rajouter des mécanismes de renvoi et d'accusés de réception applicatifs afin de s'assurer de la fiabilité de la transmission.
2. **La latence est nulle** : le temps de propagation de l'information est négligeable sur le réseau. Même à la vitesse de la lumière il faudra toujours 30 ms pour faire un ping entre l'Europe et les États-Unis. Alors que la bande passante a été multipliée par un facteur 1 500 environ en 10 ans, la latence n'a augmenté que d'un facteur 10. Ceci se traduit la plupart du temps par une exigence de grouper les informations plutôt que d'accéder *n* fois au serveur.
3. **La bande passante du réseau est infinie** : ceci est heureusement le facteur qui est de moins en moins critique. Toutefois, il est fortement couplé à la latence lorsque l'on veut obtenir un réseau fiable : il faut alors mettre en place des accusés de réception. Le taux d'erreur va donc naturellement limiter la bande passante surtout sur longues distances.
4. **Le réseau est sécurisé :** ce problème fait à présent la une de tous les journaux. On se souviendra seulement que la sécurité globale est directement liée à celle du maillon le plus faible, et que le compromis à trouver entre coût, ergonomie et sécurité dépend d'une analyse réaliste et personnalisée de la menace et des contre-mesures à appliquer.
5. **La topologie de l'application est statique** : hélas, ceci est rarement vrai. Même si cela le reste pour un temps, la tentation est toujours grande de rajouter un équipement, ou un outil de surveillance, de connecter un nouveau système ou un appareil... On pensera pour cela à imaginer dès le départ au minimum des services d'annuaires, voire des mécanismes de découverte de services ou de machines. Ceci sera traité dans le chapitre 7 sur les Annuaires. La sécurité peut par ailleurs imposer une modification de la topologie.
6. **Il y a un seul administrateur du réseau** : si cela est vrai, il faut lui fournir des outils puissants pour gérer l'ensemble des applications et identifier les dysfonctionnements, mais le plus souvent le système complet dépendra toujours d'un élément externe (routeur IP, base de données, annuaire d'entreprise...) qui imposera ses propres contraintes.

1. http://en.wikipedia.org/wiki/Fallacies_of_Distributed_Computing

© Dunod – Toute reproduction non autorisée est un délit.
Copyright © 2015 Dunod.

7. **Le coût du transport est nul** : il y a toujours un coût pour plier les données sur le réseau et gérer les ressources.
8. **Le réseau est homogène** : bien que le protocole IP soit prédominant, on peut avoir affaire à des mobiles sur Bluetooth, des réseaux de stockage sur Inifiniband, etc. On peut aussi mélanger les couches de protocoles : du RMI sur un protocole IIOP de CORBA, du SOAP sur du JMS, etc. On essaiera le plus possible de respecter des standards (par exemple JMS 1.1, SOAP 1.2, etc.) tout en prenant garde qu'un standard garantisse parfois une interchangeabilité entre applications mais pas toujours une interopérabilité.

1.2 CONCEPTS D'APPLICATIONS RÉPARTIES

1.2.1 Grain des applications

On parlera de **grain** des applications pour évaluer la taille des éléments mis en présence. Typiquement un ERP (*Entreprise Resource Planning*) faisant la comptabilité et la paye de l'entreprise sera un composant à *gros-grain*, un objet Java communiquant à travers le réseau avec d'autres objets individuels constituera un élément à *grain fin*. Dans le transport aérien par exemple, on envisagera des solutions différentes si l'on veut connecter des centres de contrôle aérien selon le grain d'information partagé : partage d'information sur les plans de vol journaliers, échanges des pistes radar à la seconde, etc. Nous verrons que les solutions avec des services web sont plutôt adaptées aux architectures gros grains, alors que les architectures de type RPC (*Remote Procedure Call* ou appel de méthodes à distance) seront réservées à la communication à grain fin.

En parallélisme, le grain fin pourra descendre au niveau de chaque instruction de programme, ce qui ne sera pas traité ici.

1.2.2 Couplage des applications

On parle de **couplage lâche** lorsque deux applications échangent peu de données, ou peu souvent, ou avec peu de types de messages : de grosses applications autonomes échangeant peu de données, avec un temps de réponse important, et selon un modèle de dialogue simple ont un couplage lâche. On aura typiquement un couplage lâche avec des composants de gros grain, et un couplage étroit entre composants de grain fin. Le grain fin nécessite typiquement davantage de synchronisation et d'harmonisation des données.

La première étape de conception d'une architecture répartie sera d'évaluer les couplages entre systèmes ou entre objets. On peut envisager d'utiliser pour cela les diagrammes de séquence UML ou les diagrammes d'interaction.

Le couplage peut également s'analyser selon d'autres axes :

- **L'axe de sécurité** : Quelles sont les applications qui partageront le même contexte d'authentification ? Quelles sont les applications regroupées derrière les mêmes zones de pare-feu ?

Copyright © 2015 Dunod.

- **L'axe de persistance** : Quelles applications vont partager les mêmes données ?
- **L'axe de déploiement** : Quelles applications devront être lancées en premier ? Quelles sont les dépendances de configuration ?
- **L'axe de gestion des versions** : Quelles sont les dépendances de versions entre les différents composants ?
- **L'axe du couplage d'horloges** : Quelle est la séquence des messages, quels sont les points de synchronisation, l'ordre temporel doit-il être préservé ?

Cette liste pourrait bien sûr être étendue. On analysera également les **goulets d'étranglement** (SPOF, *Single Point Of Failure*) qui vont fragiliser l'architecture : points de stockage de données, équipements réseau, serveurs uniques, etc.

1.2.3 MOM et RPC

La communication entre applications a toujours été un problème majeur des systèmes distribués. Il existe de nombreuses solutions à ce problème que l'on peut regrouper en deux grandes familles :

- **Les technologies d'appel de procédure à distance** (RPC, *Remote Procedure Call*) regroupant les standards tels que CORBA voir chapitre 6), RMI (voir chapitre 5), DCOM (Microsoft DNA) ou encore .NET Remoting (Microsoft .NET) et plus récemment SOAP (voir chapitre 9). Leur principe réside dans l'invocation d'un service (i.e. d'une procédure ou d'une méthode d'un objet) situé sur une machine distante indépendamment de sa localisation ou de son implémentation. Le concept est le même que l'appel de sous-procédure, avec un déroutement du contexte d'appel et une attente bloquante des résultats (figure 1.1).

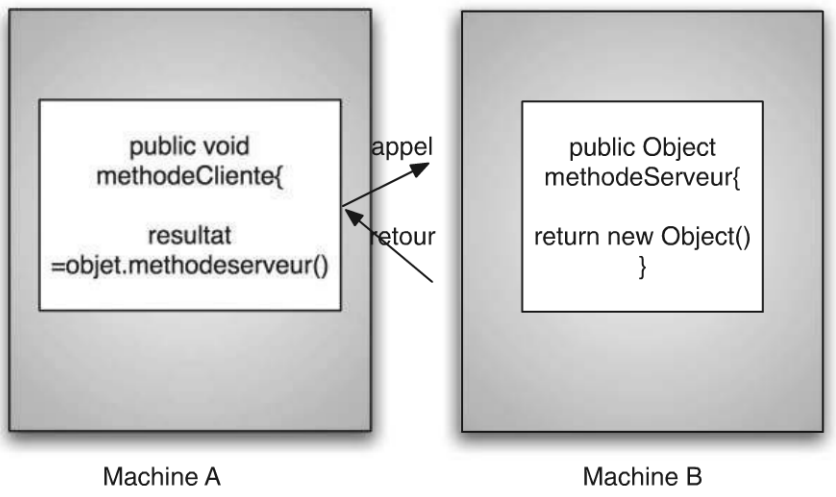


Figure 1.1 — Communication par mode RPC

Copyright © 2015 Dunod.
© Dunod – Toute reproduction non autorisée est un délit.

- **Les technologies d'échanges de messages** : les applications voulant communiquer entre elles s'échangent des messages véhiculés par l'infrastructure MOM (*Message Oriented Middleware*) selon différents modes de communication que nous détaillerons par la suite (figure 1.2). Ces messages ont une nature complètement générique, ils peuvent représenter tous types de contenu aussi bien du binaire (image, objets sérialisés) que du texte (document XML). Le chapitre 8 illustre ces techniques avec JMS.

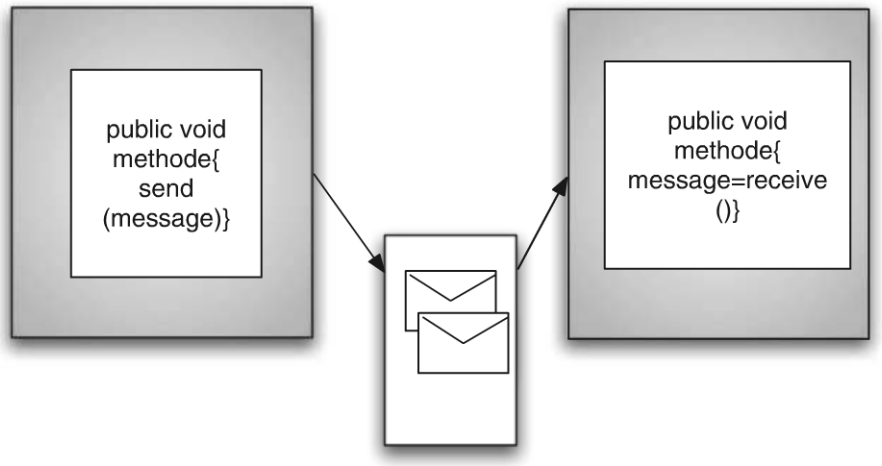


Figure 1.2 — Communication par messages

1.2.4 Mode synchrone et asynchrone

Les termes synchrone et asynchrone recouvrent souvent des concepts variés en informatique. Lorsque l'on parle d'applications réparties, on parlera le plus souvent de **mode synchrone** lorsque l'application cliente est bloquée en attente d'une réponse à une requête au serveur, et de **mode asynchrone** lorsque l'appel n'est pas bloquant – l'application continue son déroulement sans attendre la réponse. Un appel synchrone nécessite la présence simultanée du client et du serveur, alors que l'appel asynchrone peut décorréler dans le temps la requête et sa prise en charge.

Typiquement l'analogie du mode **synchrone** est le téléphone : les deux correspondants sont présents au moment de l'appel et chaque interlocuteur attend la réponse de l'autre avant de parler. Le mode **asynchrone** est semblable à la communication par courrier, où l'épistolier va envoyer un message qui sera lu ultérieurement par son correspondant, sans en attendre de réponse immédiate.

Le mode **synchrone** va permettre de s'assurer que le message a été envoyé et que l'on a bien reçu une réponse : on obtient plus de fiabilité. Par contre, le mode **asynchrone** permet de libérer du temps de calcul pour d'autres actions afin de paralléliser les tâches : on obtient plus de performance.

Copyright © 2015 Dunod.

1.2.5 Sémantique de communication et qualité de service

Lorsque l'on envoie un message ou que l'on appelle une fonction sur un réseau, il peut arriver que la communication effectue l'opération zéro, une fois ou plus si l'on implémente un mécanisme de renvoi automatique sur levée d'exception. Être sûr que l'opération a été effectuée une fois et une seule implique une qualité de service qui repose en bonne partie sur le bus logiciel. Il existe naturellement un compromis entre qualité de service et performance, et ceci fera partie des choix fondamentaux d'une architecture répartie.

Par exemple pour un transfert bancaire, il est très important que l'opération ait lieu une fois et une seule, sinon un des comptes peut être débiteur ou créditeur. Par contre pour l'affichage d'une page web, on peut réitérer la requête sans souci la plupart du temps. Nous verrons au paragraphe 8.4 sur JMS comment assurer une communication fiable de cette manière.

1.3 IMPACT SUR LE GÉNIE LOGICIEL

Les impacts sur le génie logiciel sont nombreux, et il faut d'emblée repenser les processus de conception et de développement en vue d'une architecture répartie. Les méthodes dédiées au logiciel réparti sont naturellement moins mûres que pour le logiciel traditionnel.

De plus, la programmation répartie va ajouter des degrés de complexité nombreux :

- Une réflexion nécessaire sur la topologie de l'application et sur les interfaces entre systèmes.
- Pour des applications ouvertes, la prise en compte de connecteurs externes à identifier.
- La prise en compte d'environnements non homogènes avec des langages différents.
- La qualité de service de la communication.
- La prise en compte de toutes les couches de communication, de réseau, de systèmes d'exploitation.
- Des problématiques non fonctionnelles supplémentaires : les problèmes de sécurité et d'identification, les problèmes de répartition de charge, de reprise sur panne, de persistance des données, de reprise sur erreur.

1.3.1 Modification de la conception

Si l'on suit la démarche d'UML, il faudra non seulement identifier les classes et leurs opérations, mais aussi identifier la **synchronisation** de tous ces objets sur le réseau et les fils parallèles d'exécution.

Le diagramme de **déploiement** va impacter fortement la conception des objets, et bouleverser le cycle classique de génie logiciel. De plus, UML ne permettra pas

Copyright © 2015 Dunod.
© Dunod – Toute reproduction non autorisée est un délit.

de représenter la migration de processus entre machines, fort heureusement ces techniques sont encore rares.

1.3.2 Modification du processus de développement

Aux primitives de base de Java vont s'ajouter **de nombreuses autres bibliothèques et API** pour prendre en compte la synchronisation, les références distantes, le pliage des données sur le réseau, les bus logiciels, la sécurité, les protocoles réseau. La prise en compte de nombreux outils externes, parfois en Open Source, signifie que l'on ne maîtrise plus le processus de développement en entier, sauf à réinventer la roue et dépenser des sommes colossales pour obtenir le même degré de développement. **La compilation n'est plus la seule étape** dans le processus de développement : en effet, un compilateur ne gère que la mémoire d'un processeur local, il n'existe pas de compilateur de compilateurs adressant plusieurs machines simultanément dans un langage classique. Alors qu'une application classique se contente des outils `javac` et `java` pour la compilation et le lancement d'un programme Java, le processus de développement d'une application répartie va prendre en compte de nombreux **outils**, soit pour la génération d'adaptateurs de communications (`rmic`, `java2idl`, `wsdl2java`...), soit pour le lancement de serveurs de toutes sortes (`orbd`, `rmid`...), ou pour la gestion de la sécurité (`policytool`, `keytool`...).

Le déploiement des applications est un problème important : on parlera de déploiement pour définir l'installation, la configuration et le paramétrage d'une application. Le déploiement peut parfois devenir une tâche aussi importante que la programmation si le problème n'a pas été pris en compte dès le départ. Imaginez configurer mille automates Java dans une salle blanche de production de composants en silicium, ou un réseau d'un million de livebox réparties chez les utilisateurs : il faut prévoir dès le départ qu'il sera difficile ou fastidieux d'accéder séparément à chacun des éléments et que la configuration et la mise à jour doivent être automatisées (voir le chapitre 10).

La surveillance ou l'administration de l'application (*monitoring*) et la traçabilité (journaux ou *logs*) vont devenir une part importante de l'application à ne pas négliger (voir également chapitre 10 sur JMX).

De plus, l'intergiciel est par définition enfoui entre les applications, et il sera souvent difficile de mesurer les progrès d'avancement du projet, car bien sûr la communication entre applications n'apporte aucun point de fonctionnalité métier ! Les métriques de projet doivent donc être revues.

Enfin mentionnons que le **debug** va devenir un cauchemar, que les tests devront être fortement renforcés pour tester à la fois chacun des composants séparément mais aussi toutes les combinaisons de composants, et qu'il faudra également imaginer des **tests** spécifiques :

- **tests de temps de réponse** du middleware : temps de réponse individuel ou global au système, régime transitoire ou établi ;
- **test de charge** en fonction du nombre de composants et de connexions ;

Copyright © 2015 Dunod.

- **tests de synchronisation** des opérations ;
- **tests d'accès concurrents** aux ressources et de pose de verrous.

La simulation d'un environnement réaliste peut également coûter très cher : il n'est pas possible de mettre un nombre de JVM important sur une machine, et il est parfois trompeur de simuler en local un comportement en réseau car la communication peut passer par des pipes, ou court-circuiter les interfaces réseau. Et il n'existe pas de nombre de Reynolds comme en thermodynamique pour extrapoler le fonctionnement global à partir d'une maquette ! En aéronautique on sait mettre en œuvre un modèle réduit en soufflerie, pour le logiciel on n'a pas encore trouvé de modèle générique pour passer à l'échelle.

En résumé

Les applications réparties sont des applications qui font communiquer plusieurs ordinateurs sans mémoire partagée. On distingue les modes de communication RPC et par messages. Le mode synchrone est bloquant mais fiabilise les appels, à l'inverse du mode asynchrone, qui permet souvent des gains de performance.

Mentionnons enfin que les applications réparties imposent un mode de développement spécifique, et alourdissent le test et le déploiement du logiciel.

Copyright © 2015 Dunod.

Copyright © 2015 Dunod.

2

Rappels sur Java

Objectif

La programmation d'applications réparties en Java utilise des notions qui ne sont pas propres aux applications réparties, et que nous rappellerons dans ce chapitre. Les nouvelles notions introduites par Java 5, 6 et 7 qui seront utiles pour comprendre l'évolution des applications réparties en Java seront également présentées.

2.1 SÉRIALISATION

Le but de la **sérialisation** en Java est de permettre de transférer un objet vers un support binaire. Pour les types primitifs, ceci se fait de manière canonique (entiers, chaînes de caractères, etc.). Pour les objets structurés, Java définit un mécanisme appelé *sérialisation* permettant de lire et d'écrire un objet de et vers une représentation en binaire de manière récursive.

2.1.1 Sérialisation d'un objet

Afin de pouvoir sérialiser un objet, il suffit que celui-ci implémente l'interface `Serializable`, comme dans l'exemple suivant :

```
public class Fleur implements Serializable {
  private String variete;
  private String couleur;
  public Fleur(String uneCouleur) {couleur=uneCouleur; }
}
```

© Dunod – Toute reproduction non autorisée est un délit.
Copyright © 2015 Dunod.

Le mécanisme est alors automatique et s'effectue récursivement sur les variables d'instances : ici l'objet `Fleur` va sérialiser tour à tour `variete` et `couleur`, qui sont des types primitifs.

```
// Écriture d'un objet dans un fichier
FileOutputStream fos = new FileOutputStream("fleur.ser");
ObjectOutputStream out = new ObjectOutputStream(fos);
out.writeObject(new Fleur("rouge"));
out.close();
fos.close();
```

Les classes `ObjectOutputStream` et `ObjectInputStream` permettent de relire les objets de manière standard.

```
// Lecture et écriture d'un objet dans un buffer
ByteArrayOutputStream bytes = new ByteArrayOutputStream();
ObjectOutputStream outStream= new ObjectOutputStream(bytes);
outStream.writeObject(new Fleur("rouge");
ObjectInputStream inputStream= new ObjectInputStream(new
  ByteArrayInputStream(bytes.toByteArray()));
Fleur fleur= (Fleur)inputStream.readObject();
```

L'objet peut également hériter d'une classe `Serializable`.

Tous les objets ne sont pas Serializable

Certains objets ne sont pas `Serializable` : soit parce qu'ils sont liés à des primitives du système, soit parce qu'ils sont liés à un contexte d'exécution en mémoire tel que les `Thread`. Les objets fichiers sont devenus `Serializable` dans les versions récentes de Java.

Par contre, si vous utilisez l'outil Java `serialver` sur la classe `java.awt.Graphics` il répondra que la classe n'est pas `Serializable` :

```
serialver java.awt.Graphics
Class java.awt.Graphics is not Serializable.
```

Il est possible alors d'isoler les variables d'instance non `Serializable` ou les variables d'instance devant rester secrètes à l'aide du mot-clé `transient` : ainsi ces variables d'instances ne seront pas sérialisées.

```
public class Fleur implements Serializable {
private transient Thread activite;
private String couleur;
}
```

La sérisalisation ne nécessite rien d'autre que l'implémentation de l'interface `Serializable` sans aucune autre méthode, toutefois il est recommandé de surcharger les méthodes suivantes [Bloch2002] : `hashCode()`, `toString()`, `equals()`.

Copyright © 2015 Dunod.

Nous avons vu que toutes les classes n'étaient pas sérialisables. Par exemple, supposons que l'on souhaite développer un jeu avec sauvegarde des cartes de jeu. Si l'on utilise une classe `BufferedImage`, il ne sera pas possible de la sérialiser, car l'état d'un objet `BufferedImage` dépend d'un objet de classe `Graphics`, non sérialisable.

Il est nécessaire de bien vérifier la sérialisation de manière **récursive**, car l'objet peut se révéler impossible à sérialiser, ou bien la taille de l'empreinte binaire totale peut s'alourdir considérablement par le jeu des variables d'instance. On doit alors définir des points d'élagage du graphe d'objet, soit en définissant une méthode spécifique, soit en utilisant le mot-clé `transient`.

2.1.2 Graphes d'objets

Les objets sont rarement isolés : leurs variables d'instances sont à leur tour des objets, et le tout constitue un graphe. Par exemple, un objet `Personne` sera lié à un objet `Adresse`, et ainsi de suite (figure 2.1).

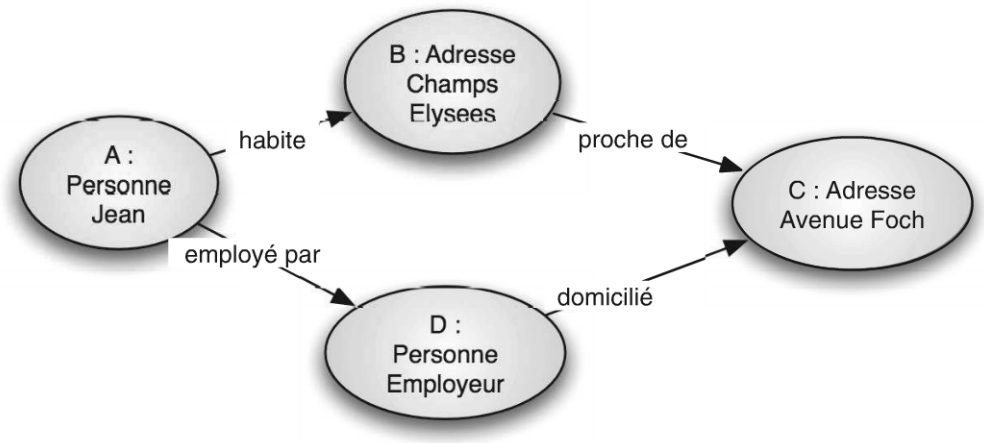


Figure 2.1 — Un graphe d'objets : A pointe vers B et D, D et B vers C

La sérialisation est récursive. Par exemple dans le graphe (A,B,C,D) de la figure 2.1, la sérialisation de A va sérialiser également B, C et D en tenant compte des cycles (C ne sera sérialisé qu'une seule fois et non une fois à travers de B et une fois à travers de D).

Les graphes d'objets obtenus peuvent avoir des cycles : A pointe vers B qui pointe vers A.

Lors de la désérialisation de l'objet, il faut pouvoir reconstituer le graphe exact. Notamment il faut pouvoir identifier les cycles afin de reboucler les objets de même identifiant : par exemple si A pointe vers B et B vers A, l'image de B par transmission doit pointer vers A et non une copie de A. Demandez à votre banquier ce qui pourrait advenir si vous versiez de l'argent sur un avatar de votre compte au lieu du compte

Copyright © 2015 Dunod.
© Dunod – Toute reproduction non autorisée est un délit.

réel... Cette fonction est prise en compte dans les nouvelles versions de Java, mais ce n'était pas vrai dans la 1.1 !

On peut écrire un petit programme de test avec `JUnit`[1], où chaque fleur pointe vers un objet variété, et où la variété comporte une liste de fleurs. On complète la classe `Fleur` ainsi :

```
public class Fleur implements Serializable {
  private Variete variete;
  private String couleur;
  public Fleur(String uneCouleur) {couleur=uneCouleur;}
  public String getCouleur() {return couleur; }
  public void setCouleur(String couleur) {this.couleur = couleur; }
  public Variete getVariete() {return variete; }
  public void setVariete(Variete variete) {this.variete = variete; }}
```

Et on ajoute la classe `Variete` :

```
public class Variete implements Serializable{
  private List<Fleur> liste= new ArrayList();
  public void ajout(Fleur fleur) {
    fleur.setVariete(this);
    liste.add(fleur); }
  public Fleur premiere() {return liste.get(0); }}
```

La méthode `premiere` renvoie la première fleur de la liste. On vérifie alors que la variété de la première fleur de la variété est bien égale (`equals`) à l'objet variété :

```
ByteArrayOutputStream bytes = new ByteArrayOutputStream();
ObjectOutputStream outStream= new ObjectOutputStream(bytes);
outStream.writeObject(variete);
ObjectInputStream inputStream= new ObjectInputStream(
new ByteArrayInputStream(bytes.toByteArray()));
Variete var= (Variete)inputStream.readObject();
assertEquals(var.premiere().getVariete(),var);
```

Le pliage des données structurées se prête à l'analogie suivante (figure 2.2) : on considère une Tour Eiffel en allumettes que l'on doit stocker dans une petite boîte à plat. On numérote ainsi les allumettes en notant leurs voisines, afin de pouvoir reconstruire la structure à l'identique !

On considère une Tour Eiffel en allumettes que l'on doit stocker dans une petite boîte à plat comme le montre la figure 2.2. On numérote ainsi les allumettes en notant leurs voisines, afin de pouvoir reconstruire la structure à l'identique !

1. *http://junit.org*

Copyright © 2015 Dunod.

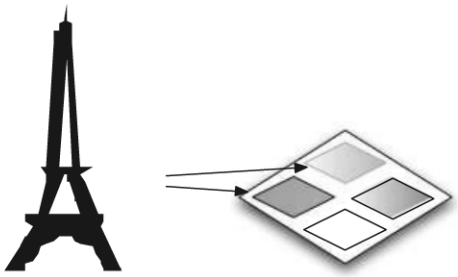


Figure 2.2 — Pliage d'une structure d'objets

2.1.3 Personnalisation de la sérialisation

Pour des besoins spécifiques, il est possible de personnaliser la manière dont s'effectuera la sérialisation. On utilise alors l'interface `Externalizable`, et on implémente les méthodes `readExternal` et `writeExternal` en mettant à la suite les champs de l'objet. La lecture et l'écriture s'effectuent alors comme dans l'exemple précédent.

```
public class UserInfo implements Externalizable {
private String login ;
private int id ;
public void writeExternal(ObjectOutput out) throws IOException {
       out.writeObject(crypte(login));
       out.writeInt(id) ;}
public void readExternal(ObjectInput in) throws IOException,
ClassNotFoundException {
       login= decrypte((String) in.readObject());
       id=in.readInt() ;}
```

2.1.4 Gestion des versions

Lorsqu'une classe change de version, soit par modification des variables d'instance, soit par modification de la version de compilation, les données sérialisées ne sont plus compatibles, et le plus souvent même plus lisibles. Il est possible de calculer un numéro de sérialisation : le SUID, pour la classe afin d'indiquer au compilateur les versions de classes qui seront compatibles. On peut utiliser l'outil `serialver` pour cela.

```
serialver java.io.File
static final long serialVersionUID = 301077366599181567L
```

Copyright © 2015 Dunod.
© Dunod – Toute reproduction non autorisée est un délit.

2.2 JSON, JAVASCRIPT OBJECT NOTATION : UNE AUTRE TECHNIQUE DE SÉRIALISATION

Il existe d'autres techniques de sérialisation que la sérialisation native de Java. Dans ce qui va suivre, nous allons en voir deux, JSON, qui est largement utilisé pour les applications web, et Protocol Buffers, qui est utilisé pour la communication de données entre applications.

JSON (*jison* ou *jayson*, *jason* en anglais) est un format de sérialisation reconnu par JavaScript, après que l'ECMA ait défini la fonction *eval* qui parse le format, dans le standard ECMAScript, en 1999.

Il est défini dans deux normes concurrentes, RFC 7159 de l'IETF et ECMA-404.

Il a été popularisé par le développement d'Ajax, un mécanisme pour rendre les pages web dynamiques. Le terme JSON revient souvent lorsque l'on parle d'Ajax. On sait qu'il s'agit d'un format de fichier alternatif à XML, mais plus compact et plus rapide à analyser.

La plupart des navigateurs web supportent JSON par les commandes JavaScript `JSON.parse()` et `JSON.stringify()`.

2.2.1 Comparaison de JSON et XML

Les avantages de JSON :

- La vitesse de traitement.
- La simplicité de mise en œuvre.On n'a pas besoin de parser un fichier XML pour extraire des informations à travers le net, car JSON est reconnu nativement par JavaScript.

Les contenus binaires peuvent être intégrés et échangés sur le net avec une représentation textuelle spéciale utilisant une commande comme : `new Buffer(file).toString('base64')`.

Les avantages de XML :

- XML est extensible et sert de support à des formats tels que RSS, SVG.
- Il est largement utilisé et reconnu par tous les langages de programmation.
- Il est plus facile à lire et convient mieux pour les fichiers destinés aux non-programmeurs.

Notons que XML aussi bien que JSON ne conviennent pas pour stocker directement des données binaires de taille importante.

Copyright © 2015 Dunod.

2.2.2 La syntaxe

Les éléments de JSON sont :

- **Un objet** – Il contient un membre ou une liste de membres, chaque membre étant de la forme :

```
"nom" : "valeur"
```

La syntaxe de l'objet est :

```
{ membre, membre, .... }
```

- **Un tableau** – Il contient une ou plusieurs valeurs séparées par des virgules.

```
[ valeur, valeur, ....]
```

- **Des valeurs** – Une valeur peut être un objet, un tableau, un litéral (chaîne, nombre, true, false, null).

Et il n'y a besoin de rien de plus pour créer un fichier JSON !

2.2.3 Exemple de fichier au format JSON

Un exemple simple : la définition d'un menu. Il s'agit d'un objet composé de membres qui sont un attribut et un tableau lequel contient d'autres objets, les lignes du menu.

```
{
  "menu": "Fichier",
  "commandes": [
      {
          "titre": "Nouveau",
          "action":"CreateDoc"
      },
      {
          "titre": "Ouvrir",
          "action": "OpenDoc"
      },
      {
          "titre": "Fermer",
          "action": "CloseDoc"
      }
  ]
}
```

L'équivalent XML :

```
<?xml version="1.0" ?>
<racine>
  <menu>Fichier</menu>
```

Copyright © 2015 Dunod.
© Dunod – Toute reproduction non autorisée est un délit.

```
  <commandes>
     <item>
         <titre>Nouveau</titre>
         <action>CreateDoc</action>
     </item>
     <item>
         <titre>Ouvrir</titre>
         <action>OpenDoc</action>
     </item>
     <item>
         <titre>Fermer</titre>
         <action>CloseDoc</action>
     </item>
  </commandes>
</racine>
```

Comment utiliser le format

Le fichier permet de charger de l'information stockée dans ce format à partir du serveur ou de transmettre au serveur de l'information dans un fichier de ce format : par exemple, le contenu d'un formulaire qui vient d'être rempli. Il y a donc trois aspects : le traitement par le navigateur, par le serveur, et la transmission des données entre les deux.

Côté client

C'est particulièrement simple JSON faisant partie de la norme JavaScript. Le contenu d'un fichier JSON ou la définition de données dans ce format sont assignés à une variable, laquelle devient un objet du programme.

Côté serveur

Les fichiers au format JSON s'utilisent dans différents langages de programmation, notamment PHP et Java. Des parseurs permettent de générer les liens entre les structures de données décrites en JSON et le langage de programmation choisi.

Le site *json.org* fournit un parseur en C et donne une liste de parseurs pour d'autres langages.

L'échange de données

La récupération d'un fichier peut se faire à partir de JavaScript de plusieurs façons :

- inclusion directe du fichier dans la page HTML au même titre qu'un fichier .js de JavaScript ;
- chargement par une commande JavaScript ;
- emploi de `XMLHttpRequest`.

Le fichier JSON est parsé par la fonction JavaScript `eval()`.

Le transfert d'un fichier au serveur se fait par `XMLHttpRequest`. Le fichier au format texte est traité par le parseur du langage de programmation utilisant le fichier.

Copyright © 2015 Dunod.

Exemple

Le code `XMLHttpRequest` :

```
var req = new XMLHttpRequest();
req.open("GET", "fichier.json", true);
req.onreadystatechange = monCode;   // la fonction de prise en charge
req.send(null);
```

Le code JavaScript :

```
function monCode()
{
   if (req.readyState == 4)
   {
        var doc = eval('(' + req.responseText + ')');
   }
}
```

Utilisation des données :

```
var nomMenu = document.getElementById('jsmenu');   // trouver un champ
nomMenu.value = doc.menu.value;           // assigner une valeur au champ
```

Comment on accède aux données :

```
doc.commandes[0].titre      // lire la valeur de "titre" dans le tableau
doc.commandes[0].action     // lire la valeur de "action" dans le tableau
```

Il existe des implémentations permettant de définir des schémas JSON[1].

2.2.4 Mappings en Java

Il existe de nombreuses implémentations de JSON en Java, les plus populaires sont :

- JSON Java[2]
- GSON[3]
- *Java API for JSON Processing* (JSON-P)[4]
- Jackson[5]

Le projet Jersey[6] d'Oracle (*RESTful Web Services in Java*) fournit également un support pour JSON.

1. *http://json-schema.org*
2. *http://json.org/java/*
3. *https://code.google.com/p/google-gson/*
4. *https://jersey.java.net/documentation/latest/media.html# json.json-p*
5. *https://github.com/FasterXML/jackson*
6. *https://jersey.java.net/*

Copyright © 2015 Dunod.
© Dunod – Toute reproduction non autorisée est un délit.

Dans la suite nous utiliserons une implémentation simple, JSON Simple[1].

2.2.5 Mise en œuvre

Il suffit de mettre le fichier `json-simple-1.1.1.jar` dans le CLASSPATH avant de compiler le code.

On peut alors écrire un code Java pour écrire un fichier JSON :

```
String fichierJson = "json.txt";
JSONObject jsonObject = new JSONObject();
jsonObject.put("URL", "www.afceurope.com");
jsonObject.put("Nom du site", "AFC Europe");
jsonObject.put("Livres", 2);
JSONArray jsonArray = new JSONArray();
jsonArray.add("Java Reparti");
jsonArray.add("Contraintes");
jsonObject.put("Bibliotheque", jsonArray);
FileWriter jsonFileWriter = new FileWriter(fichierJson);
jsonFileWriter.write(jsonObject.toJSONString());
jsonFileWriter.flush();
jsonFileWriter.close();
System.out.print(jsonObject);
```

Dans cet exemple, les attributs sont de plusieurs natures : chaînes, nombres, tableau.

On obtient le fichier suivant :

```
{"Livres":2,"Bibliothèque":["Java Reparti","Contraintes"],"Nom du site":"AFC
Europe","URL":"www.afceurope.com"}
```

Pour le relire, on utilise cette fois un `Parser` pour l'analyse du fichier.

On notera qu'il est possible d'utiliser des accents français.

```
String fichierJson = "json.txt";
JSONParser jsonParser = new JSONParser();
FileReader fileReader = new FileReader(fichierJson);
JSONObject jsonObject = (JSONObject) jsonParser.parse(fileReader);
String nomSite = (String) jsonObject.get("Nom du site");
System.out.println("Nom du site: " + nomSite);
Long livres = (Long) jsonObject.get("Livres");
System.out.println("Livres : " + livres);
String url = (String) jsonObject.get("URL");
System.out.println("URL: " + url);
JSONArray bibliotheque = (JSONArray) jsonObject.get("Bibliothèque");
                for (Object ouvrage : bibliotheque) {
                     System.out.println(ouvrage);
                }
```

1. *https://code.google.com/p/json-simple/*

Copyright © 2015 Dunod.

2.2.6 Conclusion

JSON est un format de sérialisation facile à mettre en œuvre et très populaire sur le web, car le format est directement lisible par JavaScript. Il est plus compact que XML.

2.3 PROTOCOL BUFFERS : UNE TECHNIQUE DE SÉRIALISATION PERFORMANTE ET DÉCLINÉE SUR PLUSIEURS LANGAGES

2.3.1 Une sérialisation performante et versatile

Les **Protocol Buffers** sont une nouvelle sérialisation pour les données structurées définies par Google. Le projet est en open source et se veut indépendant du langage et extensible. Les *Protobuffs* sont plus simples et plus rapides que XML, et plus compacts que JSON (voir paragraphe précédent). Ils offrent plus de flexibilité que la simple sérialisation Java et également une interopérabilité avec d'autres langages tels que C++, Java et Python.

Le principe est simple : écrire une description de haut niveau dans un fichier `.proto`, puis utiliser des outils pour générer le *marshalling/demarshalling*. Ce fichier proto est analogue à un fichier IDL en Corba (voir chapitre 6).

Ce fichier est traité par un compilateur, le *Protocol Buffer Compiler*, avec la commande `protoc`. Cette compilation génère les méthodes pour lire et écrire des objets dans différents flux (*streams*).

Le format des messages est simplissime. Chaque type d'objet comporte un ou plusieurs champs numérotés. Les types primitifs peuvent être des nombres, des booléens, des chaînes de caractères, des octets, des collections ou des énumérations. Mais on peut imbriquer aussi les objets de manière hiérarchique comme en JSON. Les champs peuvent être optionnels, obligatoires, ou répétés.

Le format est une représentation à plat et compressée des objets.

Les protobuffs sont extensibles, et il est possible de modifier la structure des objets dans une phase ultérieure sans perturber les programmes déjà écrits avec l'ancien format. Ceci est un point important car Java lui-même peut évoluer, et par exemple la sérialisation native est incompatible entre versions de Java. Tant que l'on utilise des champs optionnels dans les objets, il sera possible de désérialiser de vieux enregistrements sans avoir à les mettre à jour.

2.3.2 Mise en œuvre

On commence par télécharger le code sur le site de Google :
https://developers.google.com/protocol-buffers/docs/downloads

Copyright © 2015 Dunod.
© Dunod – Toute reproduction non autorisée est un délit.

et on compile l'utilitaire `protoc` ou on charge une version précompilée pour Windows :
https://github.com/google/protobuf/releases/download/v2.6.1/protoc-2.6.1-win32.zip

Définition du fichier .proto

Le fichier `.proto` définit la structure des données. On peut trouver un exemple ici :

```
option java_package = "com.exemple.tutoriel";
option java_outer_classname = "AdresseProto";

message Personne {
  required string nom = 1;
  required int32 id = 2;
  optional string courriel = 3;

  enum TypeTelephone {
    MOBILE = 0;
    PERSONNEL = 1;
    PROFESSIONNEL = 2;
  }

  message NumeroTelephone {
    required string numero = 1;
    optional TypeTelephone type = 2 [default = PERSONNEL];
  }

  repeated NumeroTelephone telephone = 4;
}

message CarnetAdresse {
  repeated Personne personne = 1;
}
```

Il est possible de définir un package Java pour les fichiers `.proto` à l'aide du mot-clé `java_package`. Ceci permet d'éviter des collisions entre les codes générés à différentes dates ou par plusieurs personnes.

Le fichier définit la classe cible `AdresseProto`. Ensuite viennent les définitions des messages. Les messages sont des structures de données avec des champs numérotés et typés. Il existe de nombreux types de base tels que `bool`, `float`, `int32`. Il est possible d'imbriquer des messages pour structurer les données. Dans l'exemple ci-dessus le message `CarnetAdresse` contient des messages `Personne`, et le message `Personne` contient des messages `NumeroTelephone`.

Il est également possible de définir des types énumérés comme dans l'exemple des types de numéros de téléphone (`PERSONNEL, PROFESSIONNEL`).

Les numéros de champs identifient l'ordre des champs du message dans la représentation binaire.

Copyright © 2015 Dunod.

Les valeurs de numéros de 1 à 15 seront stockées sur un octet, alors que les valeurs de 16 à 2047 seront stockées sur 2 octets. Google recommande d'utiliser les numéros 1 à 15 pour les données fréquentes ou répétées et de réserver des numéros dans cette plage pour les versions ultérieures.

Chaque champ doit être suivi d'une des annotations suivantes :

- `required` (obligatoire) : il faut fournir une valeur d'initialisation au champ, sans quoi la lecture lèvera une erreur IOException.
- `optional` (facultatif) : le champ peut être ou non initialisé. S'il ne l'est pas, on utilise une valeur par défaut, soit définie par l'utilisateur, soit implicite pour le langage (false pour les booléens, chaîne vide pour les chaînes, zéro pour les nombres).
- `repeated` (répété) : le champ peut être répété un nombre quelconque de fois, y compris zéro. L'ordre des valeurs sera conservé. Les champs répétés sont analogues à des tableaux de taille dynamique.

D'autres possibilités sont décrites dans la documentation : *https://developers.google.com/protocol-buffers/docs/proto*.

Compilation du fichier .proto

Le fichier `.proto` va permettre de générer les classes d'accès aux données.

Pour cela on utilise l'utilitaire protoc pour compiler la définition, ici par exemple le fichier **adresses.proto**.

```
protoc –I=$SRC_DIR - -java_out=$DST_DIR adresses.proto
```

ou en particulier si tous les sources sont dans le sous-répertoire **src** :

```
protoc –I=src - -java_out=src addresses.proto
```

Le mot-clé `- -java_out` signifie que l'on génère les classes en Java.

On observe que la compilation a généré une classe `com.exemple.tutoriel.AdresseProto.java`.

Dans cette classe, on remarque en particulier la création d'interfaces pour `Personne` avec les méthodes suivantes :

```
//required string nom = 1
public boolean hasNom() ;
public String getNom() ;
...
//optional string courriel =3
public Boolean hasCourriel();
public String getCourriel();
...
//repeated NumeroTelephone telephone = 4
```

Copyright © 2015 Dunod.
© Dunod – Toute reproduction non autorisée est un délit.

```
java.util.List<com.exemple.tutoriel.AdresseProto.Personne.NumeroTelephone>
        getTelephoneList();
public int getTelephoneCount();
```

Il y a ainsi des accesseurs simples pour chaque champ. Les champs répétés ont des méthodes supplémentaires pour compter les occurrences, ici `getTelephoneCount()`. Les noms des méthodes sont dérivés des noms des attributs.

Pour les types énumérés, une énumération est générée ici :

```
/**
  * Protobuf enum {@code tutorial.Personne.TypeTelephone}
  */
 public enum TypeTelephone
     implements com.google.protobuf.ProtocolMessageEnum {
   /**
    * <code>MOBILE = 0;</code>
    */
   MOBILE(0, 0),
   /**
    * <code>PERSONNEL = 1;</code>
    */
   PERSONNEL(1, 1),
   /**
    * <code>PROFESSIONNEL = 2;</code>
    */
   PROFESSIONNEL(2, 2),
   ;
   /**
    * <code>MOBILE = 0;</code>
    */
   public static final int MOBILE_VALUE = 0;
   /**
    * <code>PERSONNEL = 1;</code>
    */
   public static final int PERSONNEL_VALUE = 1;
   /**
    * <code>PROFESSIONNEL = 2;</code>
    */
   public static final int PROFESSIONNEL_VALUE = 2;


   public final int getNumber() { return value; }

   public static TypeTelephone valueOf(int value) {
     switch (value) {
       case 0: return MOBILE;
       case 1: return PERSONNEL;
       case 2: return PROFESSIONNEL;
       default: return null;
     }
   }
```

Copyright © 2015 Dunod.

Utilisation des classes générées

Pour utiliser les classes générées, on fera appel en écriture à un `Builder`.

```
CarnetAdresse.Builder carnetAdresse =CarnetAdresse.newBuilder();
Personne.Builder personne = Personne.newBuilder();
    personne.setId(1);
    personne.setNom("Véronique");

    Personne.NumeroTelephone.Builder numeroPersonnel =
             Personne.NumeroTelephone.newBuilder().setNumero("123456");
    numeroPersonnel.setType(Personne.TypeTelephone.PERSONNEL);
    personne.addTelephone(numeroPersonnel);

    carnetAdresse.addPersonne(personne);
    carnetAdresse.build().writeTo(new FileOutputStream("carnet.txt"));
```

En lecture on peut retrouver les adresses dans le carnet en itérant sur la liste :

```
CarnetAdresse.Builder carnetAdresse = CarnetAdresse.newBuilder();
carnetAdresse.mergeFrom(new FileInputStream("carnet.txt"));
System.out.println("Nombre d'adresses "+carnetAdresse.getPersonneCount());
               for (Personne pers : carnetAdresse.getPersonneList())
                    System.out.println(pers.getNom());
       }
```

Protocol Buffers fournit des outils et un canevas pour générer une sérialisation simple d'objets structurés non seulement en Java mais dans d'autres langages. Il est extensible et compact, et permet de s'adapter à des évolutions de version des structures de données.

2.4 UTILISATION DES THREADS : ACTIVITÉS EN PARALLÈLE

2.4.1 Thread et processus

Processus

La plupart des systèmes d'exploitation traditionnels sont multitâches. On entend par là qu'ils sont capables de déclencher plusieurs applications en parallèle, et d'allouer une portion du temps d'utilisation du processeur à chaque application, de manière équitable (éventuellement pondérée par la notion de priorité). Ces applications ne s'exécutent donc pas réellement en parallèle, mais plutôt à tour de rôle, chacune étant fréquemment interrompue par l'ordonnanceur du système qui joue le rôle d'arbitre en donnant la main à toutes les applications à tour de rôle (c'est du moins le comportement des systèmes d'exploitation préemptifs, tels que Windows et les principaux Unix). On dit également que chaque application s'exécute dans son propre **processus**, c'est-à-dire que le système lui réserve un certain espace mémoire, quelques

Copyright © 2015 Dunod.
© Dunod – Toute reproduction non autorisée est un délit.

registres de travail, ainsi qu'un identifiant (PID) parmi la liste des processus déclenchés en parallèle.

Toutefois au-delà de ce pseudo-parallélisme les nouvelles architectures de processeurs dites *multi-cores* (plusieurs cœurs) sont de plus en plus répandues, et permettent un parallélisme réel. D'autres architectures encore permettent un parallélisme réel avec mémoire partagée. Dans le domaine de la gestion des activités avec parallélisme ou pseudo-parallélisme, Java possède un avantage depuis longtemps.

Thread

Dans son processus, chaque application s'exécute de manière linéaire (qu'elle soit développée avec un langage procédural, orienté objet, ou déclaratif) du début à la fin de son point d'entrée.

Pour qu'une application devienne elle-même multitâche, il faut avoir recours à un découpage des activités au sein même de l'application ; c'est cette notion de « sous-activité » que nous nommerons **thread** (même si cette acception peut être implémentée de manière très variable en fonction des systèmes d'exploitation et des machines virtuelles. En particulier, les architectures récentes permettent d'utiliser des threads matériels qui permettent un réel parallélisme et non plus une simple simulation).

Certains langages (Fortran, Ada, XSLT), grâce à la notion de pragma ou à un calcul des limites des effets de bords des fonctions, sont capables de procéder à un découpage automatique d'une application en plusieurs *threads* parallélisables. Mais avec l'avènement de la programmation orientée objet, le flux d'exécution des programmes s'est terriblement complexifié, rendant ce mécanisme difficilement automatisable ; c'est donc au développeur (ou au *framework* utilisé par le développeur) de paralléliser lui-même les tâches, qu'il doit déclarer explicitement auprès de son système d'exécution (système d'exploitation ou machine virtuelle).

2.4.2 Simulation du parallélisme

Ce petit paragraphe rappelle les notions fondamentales sur les threads et insiste uniquement sur les aspects utiles à la suite de ce livre. On parle en anglais de « programmation concurrente » pour l'utilisation d'activités en parallèle, et Java 5, puis Java 6 et 7 ont rajouté de nombreuses primitives dans le package `java.util.concurrent` pour permettre une riche sémantique dans ce domaine. On se reportera à l'un des ouvrages de référence [Lea2003], [NIO1].

Il est possible en Java de simuler ou de profiter de l'exécution parallèle de plusieurs programmes en utilisant des `Thread`. Un thread est un fil d'activité qui possède un cycle de vie (démarrage, attente, fin), et qui peut lui-même lancer des threads fils.

Les threads seront très utilisés dans les applications réparties qui doivent servir plusieurs requêtes à la fois. Un programme Java comporte toujours au moins un thread principal.

Copyright © 2015 Dunod.

2.4.3 Création d'un thread

Un thread est un objet de la classe `Thread`. Il existe deux méthodes pour créer un thread en Java. L'une consiste à instancier une classe dérivée de la classe `Thread`, l'autre repose sur l'instanciation d'une classe spéciale, appelée `Runnable`.

Voyons tout de suite à quoi ressemble la première méthode à l'aide d'un exemple basique (affichage des chiffres de 1 à 10) :

```
public class ThreadTest extends Thread{
     public void run()
    {
       for(int i = 1; i<= 10; i++)
       System.out.println(i + " " + getName());
    }
    public static void main (String[] args)
    {
       Thread t1 = new ThreadTest();
       t1.start();
    }
}
```

La méthode `run()` décrit le corps d'activité du thread, qui peut très bien se terminer après quelques instructions. Elle est appelée implicitement lors du `start` du thread. L'instance de `Thread` est créée comme un objet standard.

Si l'on utilise à présent deux threads, on observera une imbrication des exécutions sur la trace qui ressemble au code suivant. La trace peut dépendre de l'exécution car l'imbrication des activités des threads va dépendre des quantièmes alloués à chaque thread alternativement, du système d'exploitation et des performances de la machine :

```
1 Thread-0
2 Thread-0
3 Thread-0
4 Thread-0
5 Thread-0
6 Thread-0
1 Thread-1
7 Thread-0
2 Thread-1
8 Thread-0
```

Si l'on souhaite contrôler exactement l'exécution de la trace, on doit **synchroniser** les threads. Nous y reviendrons.

On peut éviter de dériver de la classe `Thread` en implémentant l'interface `Runnable` (figure 2.3). Dans ce cas la classe `Runnable` sera passée en paramètre à un `Thread` anonyme.

```
public class ThreadTest2 implements Runnable
{
    // Exécution en parallèle de deux threads non synchronisés
```

© Dunod – Toute reproduction non autorisée est un délit.
Copyright © 2015 Dunod.

```
    public static void main (String[] args)
    {
        ThreadTest2 t1 = new ThreadTest2();
        ThreadTest2 t2 = new ThreadTest2();
        new Thread(t1).start();
        new Thread(t2).start();
    }
    public void run()
    {
        for(int i = 1; i<= 10; i++)
            System.out.println(i);
    }
}
```

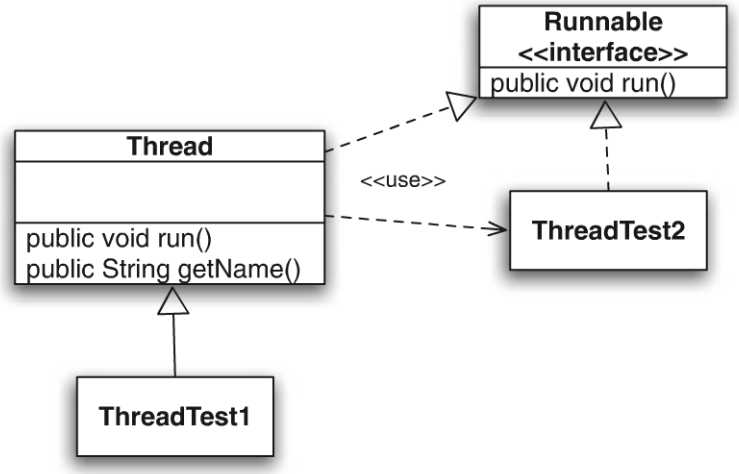


Figure 2.3 — Deux méthodes pour lancer un thread : par héritage ou par délégation

2.4.4 Partage de mémoire entre threads

Les threads partagent en Java la même zone mémoire, il est nécessaire alors de verrouiller l'accès simultané aux variables partagées.

On utilise pour cela les primitives de synchronisation et le mot-clé `synchronized`. Ce mot-clé va verrouiller la variable pour les autres threads, qui seront bloqués en attente.

```
synchronized (someObject) {
            // code protégé
    }
```

Par exemple, si l'on suppose plusieurs threads en parallèle effectuant des transferts de fonds entre différents comptes et travaillant sur la même liste de soldes, il faudra verrouiller la variable statique `soldes` pour toutes les opérations de transfert ou de consolidation (figure 2.4) :

Copyright © 2015 Dunod.

```
static void transfer(int compteDebite, int compteCredite, double montant) {
        synchronized (soldes) {
            soldes[comptedebite] -= montant;
            soldes[compteCredite] += montant;
        }
    }
    static void consolidation() {
        int total = 0;
        synchronized (soldes) {
            for (int n=0; n<nbComptes; n++) {
                 total += soldes[n];
        }
    }
    System.out.println("Le solde total est " + total);
}
```

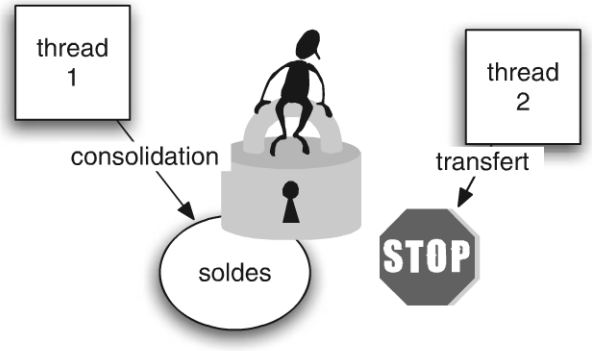


Figure 2.4 — On verrouille l'accès partagé à la variable `soldes`

On vérifiera à titre d'exercice que sans synchronisation le résultat peut varier.

2.4.5 Utilisation de ExecutorService et sémantique d'exécution

On veut pouvoir spécialiser la sémantique de lancement d'un `Runnable` dans un `Thread` : on veut par exemple utiliser un seul thread instantané en bloquant les autres requêtes, ou retarder l'exécution. Pour cela, on peut utiliser des objets de type `ExecutorService`. La classe `Executors` est une fabrique d'objets (*Factory*) qui permet de créer toutes sortes d'objets avec des sémantiques différentes :

- Exécution des différents appels à la queue leu leu dans un seul thread : `newSingleThreadExecutor()`.

```
Executors.newSingleThreadExecutor().execute(runnable) ;
```

- Exécution retardée : `newSingleThreadScheduledExecutor()`. L'exécution ne sera lancée qu'après un délai de 100 ms :

Copyright © 2015 Dunod.
© Dunod – Toute reproduction non autorisée est un délit.

```
Executors.newSingleThreadScheduledExecutor().schedule(runnable, 100,
  TimeUnit.MILLISECONDS)
```

Il existe bien d'autres sémantiques possibles, par exemple pour sécuriser l'exécution ou gérer le nombre de threads simultanés : le lecteur pourra approfondir ces thèmes au-delà de ce livre[1].

2.5 UTILISATION DES FUTURS

2.5.1 Programmation asynchrone

Java 5 introduit un mécanisme fondamental pour la programmation asynchrone : ce sont les **futurs**. Il existe l'équivalent en .Net sous le nom de *delegate*. Imaginons que pour des raisons de performance l'on veuille lancer un calcul important en tâche de fond et continuer à travailler en local tant que l'on n'a pas besoin du résultat.

Si l'on écrit dans le code l'instruction :

```
resultat = calcul() ;
```

On sera bloqué en attente du résultat. On peut imaginer lancer un thread, mais comment être averti de la fin du thread et renvoyer le résultat ? En effet, `run()` ne renvoie rien !

```
new Thread(){public void run(){calcul() ;}}
```

Au lieu de `Runnable`, on peut utiliser l'interface `Callable<T>` qui va elle renvoyer un résultat de type T. Ci-dessous la variable `calculateur` est de type `Callable<String>` et renvoie une `String` en resultat du `call()`. On peut imaginer que la méthode `call` fait appel à un calcul.

```
Callable<String> calculateur= new Callable<String>(){
        public String call(){ return calcul();}
        public String calcul(){ return "succes";}};
String resultat=calculateur.call();
```

2.5.2 Synchronisation sur attente du résultat

Par ailleurs pour synchroniser le retour du thread avec l'attente de résultat, on peut utiliser `join()`. Cette primitive permet d'attendre la fin de l'exécution d'un thread dans le contexte d'appel (figure 2.5).

1. *http://docs.oracle.com/javase/7/docs/api/java/util/concurrent/Executors.html*

Copyright © 2015 Dunod.

```
thread.start() ;
thread.join() ; // Le résultat est alors prêt
```

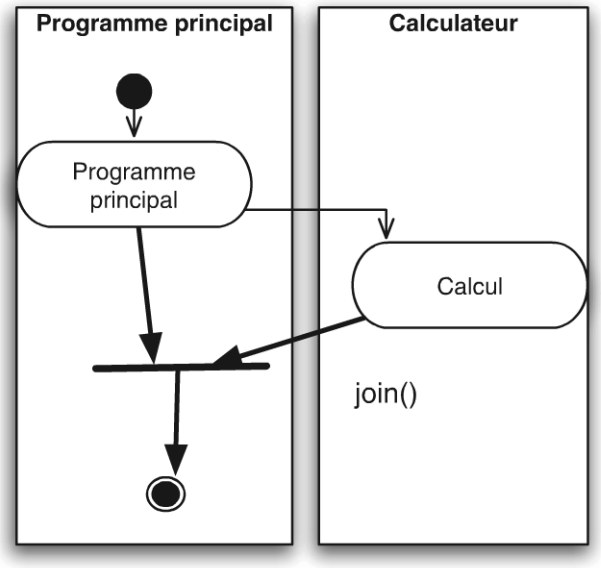


Figure 2.5 — Synchronisation sur le résultat avec `join()`

Mais ceci bout à bout ne nous permet pas simplement d'utiliser un `Callable` avec un `Thread` synchronisé sur `join()` !

Pour cela le « *futur* » est un mécanisme simple. Il va gérer la synchronisation du résultat différé d'un `Callable`, comme si `join()` était implicite.

Tout comme `Callable`, il est paramétré par le type d'argument attendu : `Future<String>`. Par contre, il faut utiliser un service d'exécution pour lancer la tâche.

Nous avons vu précédemment comment les services d'exécution permettaient de lancer un `Runnable` en différé. On choisira par exemple un exécuteur simple :

```
ExecutorService interprete= Executors.newSingleThreadExecutor();
```

Un service d'exécution permet de lancer un `Callable` en créant un futur pour la synchronisation des résultats. Au lieu de la méthode `execute` utilisée pour les `Runnable`, on utilise la méthode `submit()` :

```
Future<String> future = executor.submit(calculateur) ;
contructionFenetre(); // On fait d'autres choses en parallèle
afficheTexte(future.get());
```

Dans l'exemple précédent, l'appel `future.get()` sera bloqué en attente de retour du `call` sur le calculateur qui a été lancé bien avant dans le code. Ce code permet par exemple de construire de manière simple une fenêtre Swing en attendant le résultat

Copyright © 2015 Dunod.
© Dunod – Toute reproduction non autorisée est un délit.

de la requête à une base de données. Par contre l'affichage du résultat dans un champ de la fenêtre ne se fera qu'à disponibilité du résultat (figure 2.6).

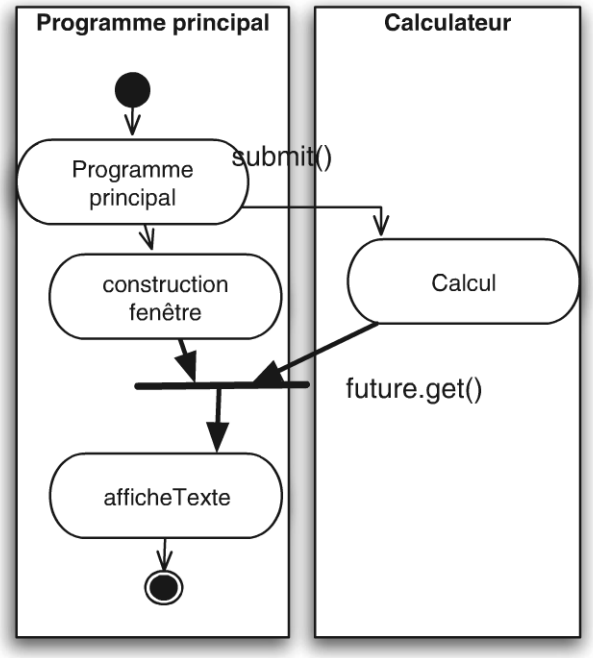


Figure 2.6 — Les futurs permettent une synchronisation implicite sur la disponibilité du résultat

Il est possible de combiner ce mécanisme à des appels distants tels que RMI (chapitre 5), CORBA (chapitre 6) ou même SOAP (chapitre 9) pour obtenir un comportement asynchrone sur des appels synchrones.

Il est également possible d'annuler un `Future` ou de tester s'il est terminé.

```
future.cancel(false)  // annulation si pas exécuté
future.cancel(true)  // interrompt la tâche en cours
future.isCancelled()         // teste l'annulation
future.isDone()               // teste la complétion de la tâche
```

En somme :
– un `Future` est un conteneur pour un résultat qui sera disponible à une date ultérieure.
– Il est sous-tendu par un *thread*.
– Il peut lancer une exception sur erreur de calcul

Un `Future` permet un type de programmation asynchrone, c'est-à-dire sans attente du résultat, mais nécessite l'utilisation d'un blocage. Combinés à des `Callables`, ils sont moins chers et plus faciles à maintenir que des *threads*.

Copyright © 2015 Dunod.

2.5.3 Utilisation de la puissance de calcul des multiprocesseurs avec les tâches Fork-Join en Java 7

Nous avons vu que l'`ExecutorService` a simplifié l'exécution en parallèle de tâches en s'affranchissant de la manipulation directe des threads. Cette classe propose un pool de threads qui seront utilisés pour exécuter une liste de tâches. Mais cette solution souffre de quelques limitations parmi lesquelles :

- Tout va bien si les tâches sont indépendantes : une tâche s'exécute et une fois terminée, le thread l'ayant exécutée est libre pour traiter la suivante... mais si une tâche doit attendre le résultat d'une autre, qui s'exécute en parallèle, le thread qu'elle monopolise n'est pas rendu au *pool* tant que l'autre tâche n'est pas terminée et que la tâche courante n'a pas exploité le retour de l'autre.
- Apprécier la taille optimale du pool est toujours une tâche délicate.
- Le nombre de tâches parallélisables souhaitées ne peut pas être déraisonnablement grand, car limité par le nombre de threads que cela induit. Trop de threads tue le thread ! La concurrence induite par un grand nombre de threads sur les ressources devient nuisible à la rapidité globale d'exécution.

Java 7 introduit une nouvelle implémentation de `ExecutorService` afin de tirer parti de la puissance de calcul des architectures multiprocesseurs.

Le principe de base est la subdivision des traitements en plus petites unités de travail. Les traitements à un niveau de subdivision seront lancés en parallèle et les résultats obtenus en retour, agrégés. Ce principe est appelé **fork/join**. Chaque tâche, si sa taille est au-dessus d'un certain seuil, sera subdivisée en sous-tâches (*fork*) qui seront exécutées individuellement et parallèlement (ou subdivisées encore) puis agrège (*join*) les résultats de chacune. Le comportement est nouveau ici car lors des périodes d'attente, le thread n'est plus monopolisé pour rien mais est affecté à l'exécution d'une autre unité de travail.

Tout d'abord il peut être utile de connaître le nombre de processeurs dont dispose votre ordinateur. Il suffit pour cela d'interroger Java :

```
System.out.println("Nombre de processeurs : "+
    Runtime.getRuntime().availableProcessors());
```

On souhaite par exemple effectuer des calculs intensifs et récursifs sur une machine comportant un grand nombre de processeurs. On prend comme exemple simple le calcul des nombres de Fibonacci :

```
Fibonacci(n)= Fibonacci(n-1) + Fibonacci(n-2).
```

On crée donc une classe qui va effectuer le calcul avec une variable d'instance indiquant le numéro de séquence de la série à calculer, et un constructeur.

```
public class FibonacciMonoProcesseur {
    private int nombre;
    private int resultat;
```

Copyright © 2015 Dunod.
© Dunod – Toute reproduction non autorisée est un délit.

```
    public FibonacciMonoProcesseur(int unEntier){
        nombre= unEntier;}
    private int calculElementaire(int n){
           if (n <= 1) {
             return n;
         } else {
             return calculElementaire(n - 1) + calculElementaire(n - 2);}
        }
public void compute() {
       resultat= calculElementaire(nombre);
}
public int resultat(){ return resultat;}
public static void main(String[] args) {
     long depart = System.currentTimeMillis();
     FibonacciMonoProcesseur f= new FibonacciMonoProcesseur(44);
     f.compute();
     int resultatDefinitif = f.resultat();
     System.out.println("Reponse : " + resultatDefinitif);
     long fin = System.currentTimeMillis();
     System.out.println("Temps de calcul :"+(fin-depart)); }}
```

Au passage, on mesure le temps de calcul. Si l'on veut au contraire tirer parti de l'architecture multiprocesseur, on va reprendre la même classe, mais la faire dériver de la classe `RecursiveAction` introduite en Java 7. Cette classe va soit faire le calcul elle-même, soit le déléguer à d'autres de ses semblables en fonction d'un seuil.

Une classe héritant de `RecursiveAction` doit nécessairement implémenter une méthode `compute()`. On marque en gras les différences avec le code précédent. On constate que le calcul élémentaire est le même, mais que l'arbitrage de division de tâche est effectué dans la méthode compute.

```
public class Fibonacci extends RecursiveAction {
   protected  static int SEUIL= 10;
   volatile private int nombre;
   volatile private int resultat;
   public Fibonacci(int unEntier){
      nombre= unEntier;
   }
   @Override
   protected void compute() {
       int n = nombre;
           if (nombre <= SEUIL) { // seuil de granularité
                                  // sous lequel on ne parallélise plus
               resultat = calculElementaire(n);
           } else {
               final Fibonacci f1 = new Fibonacci(n - 1); // subdivision
               final Fibonacci f2 = new Fibonacci(n - 2);
               invokeAll(f1, f2); // invocation en parallèle
                                  // des sous-tâches (fork & join)
               resultat = f1.resultat() + f2.resultat();
                                  // agrégation des résultats
           }
   }
private int calculElementaire(int n){
        if (n <= 1) {
```

Copyright © 2015 Dunod.

```
            return n;
        } else {
            return calculElementaire(n - 1) + calculElementaire(n - 2);}
        }
    public int resultat(){ return resultat;} }
```

Le mot-clé `volatile` est utilisé sur les variables qui peuvent être modifiées de manière asynchrone. C'est-à-dire que plusieurs threads peuvent y accéder simultanément. Ces accès peuvent être pour la lecture et/ou la modification du contenu.

En indiquant que la variable est `volatile`, on oblige la JVM à rafraîchir son contenu à chaque fois qu'elle est utilisée. On est ainsi certain que la valeur de la variable n'est pas une valeur mise en cache, mais bel et bien sa valeur exacte. Ainsi chaque thread a accès à la valeur la plus récente de la variable.

Ce mot-clé, `volatile`, est relativement peu utilisé et toutes les JVM ne le prennent pas en compte.

Il reste alors à écrire le code pour le lancement du programme. Si l'on a constaté que l'ordinateur possédait deux processeurs, on va par exemple dimensionner la taille du groupe à 2.

```
public static void main(String[] args) {
        int tailleGroupe = 2;
        ForkJoinPool pool = new ForkJoinPool(tailleGroupe);
        Fibonacci f = new Fibonacci(44);
        long depart = System.currentTimeMillis();
        pool.invoke(f);
        int resultatDefinitif = f.resultat();
        System.out.println("Reponse : " + resultatDefinitif);
        long fin = System.currentTimeMillis();
        System.out.println("Temps de calcul :"+(fin-depart));}
```

On constate un gain de performance par rapport au code précédent !

2.5.4 Nouvelle API asynchrone en Java 8

Introduction rapide aux lambdas

Java 8 introduit un nouveau concept qui permet une modification radicale du code : les **lambdas**.

Une lambda est une fonction anonyme décrite par la liste de ses arguments et le corps de la fonction. Ici :

```
() -> 100
```

est une lambda qui n'a pas d'arguments et qui renvoie la valeur 100.

Voici un petit exemple de l'utilisation d'une lambda à deux arguments pour faire un tri sur une liste de noms :

© Dunod – Toute reproduction non autorisée est un délit.
Copyright © 2015 Dunod.

```
List<String> noms = new ArrayList<String>();
noms.add("Betty");
noms.add("Anna");
noms.add("Chantal");
Collections.sort(noms, (a, b) -> a.compareTo(b));
for (String nom : noms)
     System.out.println(nom);
}
```

Dans cet exemple la lambda est la fonction à deux arguments, `a` et `b`, qui renvoie la comparaison :

```
(a, b) -> a.compareTo(b)
```

CompletionService introduit par Java 6

Java 6 a incorporé la JSR 166x, `CompletionService`. Avec un `ExecutorService`, après avoir soumis les requêtes de calcul, il faut écrire du code pour récupérer les résultats. `CompletionService` va automatiser cela.

Voici un petit exemple pour illustrer l'apport de `CompletionService` par rapport à `ExecutorService` et par rapport à une boucle classique. Cet exemple va imprimer des caractères toutes les 200 ms en utilisant trois threads.

Écrivons la liste des caractères à imprimer :

```
  private final List<String> printRequests =
         Arrays.asList("1", "2", "3", "4", "5", "6", "7", "8", "9", "10",
                       "1", "2", "3", "4", "5", "6", "7", "8", "9", "10",
                       "1", "2", "3", "4", "5", "6", "7", "8", "9", "10",
                       "1", "2", "3", "4", "5", "6", "7", "8", "9", "10",
                       "1", "2", "3", "4", "5", "6", "7", "8", "9", "10",
                       "1", "2", "3", "4", "5", "6", "7", "8", "9", "10",
                       "1", "2", "3", "4", "5", "6", "7", "8", "9", "10",
                       "1", "2", "3", "4", "5", "6", "7", "8", "9", "10",
                       "1", "2", "3", "4", "5", "6", "7", "8", "9", "10",
                       "1", "2", "3", "4", "5", "6", "7", "8", "9", "10",
                       "1", "2", "3", "4", "5", "6", "7", "8", "9", "10",
                       "1", "2", "3", "4", "5", "6", "7", "8", "9", "10",
                       "1", "2", "3", "4", "5", "6", "7", "8", "9", "10"
                       );
private static final int delai = 200; // en ms
private static final int nombreDeThreads = 3;
```

Voici la boucle naturelle :

```
void boucleNormale() {
             for (String requete : requetes) {
                     Thread.sleep(waittime);
                     System.out.print(requete);
             }
     }
```

Copyright © 2015 Dunod.

On introduit à présent un `Callable`, `Imprimante`, qui matérialise le délai d'attente.

```
private class Imprimante implements Callable<String> {

        private final String toPrint;

        public Imprimante(String toPrint) {
                this.toPrint = toPrint;
        }

        public String call() {
                try {
                        Thread.sleep(delai);
                } catch (InterruptedException e) {
                        e.printStackTrace();
                }
                return toPrint;
        }
}
```

Voici à présent la boucle utilisant un `ExecutorService` pour appeler `Imprimante`. On soumet la tâche au service d'exécution, puis on va chercher le résultat avec `get()` :

```
ExecutorService executor = Executors.newFixedThreadPool(nombreDeThreads);
Set<Future<String>> impressionsFutures = new HashSet<Future<String>>();
for (final String requete : requetes) {
         impressionsFutures.add(executor.submit(new Printer(printRequest)));
                        }
for (Future<String> future : impressionsFutures) {
                             System.out.print(future.get());
                        }
```

Et enfin la boucle utilisant le `CompletionService`. La méthode `take()` va renvoyer un `Future`.

```
ExecutorService executor = Executors.newFixedThreadPool(nombreDeThreads);
                CompletionService<String> completionService = new
ExecutorCompletionService<String>(executor);
                for (final String requete : requetes) {
                        completionService.submit(new Imprimante(requete));
                }
                for (int t = 0, n = requetes.size(); t < n; t++) {
                                Future<String> f = completionService.take();
                                System.out.print(f.get());
                        }
```

Si l'on compare les temps d'exécution on obtient la trace suivante :

```
Normal Executor Service
98161393102572510691057572571554675965826106827216103264101957321010139281 3
7345710449829815109681079894484244633410797178824269138310636143235879610418
8511
Execution time : 9428
```

Copyright © 2015 Dunod.
© Dunod – Toute reproduction non autorisée est un délit.

```
Completion Service
231465789101234567891012345678910123465798102135467891102435768109123456789 1
021345687911042357681091324657891021354687910124357689101234567891012345678 9
10
Execution time : 9409
Normal Loop
123456789101234567891012345678910123456789101234567891012345678910123456789
101234567891012345678910123456789101234567891012345678910123456789101234567
8910
Execution time : 28013
```

On observe que l'impression du `CompletionService` a lieu trois par trois, puisque l'on a trois threads. Mais que l'`ExecutorService` affiche tout d'un trait après une attente.

L'impression `CompletionService` prend également légèrement moins de temps que `l'ExecutorService` et en voici l'explication.

L'`ExecutorService` attend toujours que le thread suivant termine avant d'imprimer le nombre, et le client doit attendre que tous les thread soient finis. Au contraire le `CompletionService` suit une philosophie producteur/consommateur : dès qu'un thread a fini, il met le résultat dans une queue non bloquante afin que le consommateur puisse y accéder par `take()`.

ForkJoinTask et RecursieAction introduits par Java 7

Après le `CompletionService`, Java 7 a introduit les `ForkJoinTask`. Nous avons vu un exemple au paragraphe précédent. Mais nous allons en faire une critique ici. Il s'agit d'une extension des `Future` qui permet de tirer avantage des multiprocesseurs afin d'optimiser les applications. Comme les `ExecutorService`, les `ForkJoinTask` distribuent les tâches dans un pool de threads. Mais cette fois les threads qui sont à court de travail peuvent voler des tâches à ceux qui sont actifs.

`ForkJoinTask` possède trois sous-classes, `RecursiveTask` qui prend un parameter, `RecursiveAction` qui n'en prend pas et `CountedCompleter`.

Nous allons illustrer l'utilisation de `RecursiveAction` pour diviser un gros calcul en plusieurs sous-calculs et tirer parti des architectures multiprocesseurs, en implémentant une stratégie de division et conquête (*divide and conquer*).

On souhaite calculer la somme des carrés d'une liste. Pour ce faire, on va essayer de ne jamais ajouter plus de trois carrés dans le même thread.

Voici le code de la classe `Carre` qui hérite de `RecursiveAction` :

```
class Carre extends RecursiveAction {
          final int LIMITE = 3;
          //on garde le resultat comme variable statique
          private static int resultat;
          int debut, fin; // les bornes de l'intervalle à calculer
          int[] donnees;
          Carre(int[] data, int start, int end) {
                 this.debut = start;
```

Copyright © 2015 Dunod.

```
                this.fin = end;
                this.donnees = data;
        }
        @Override
        protected void compute() {
                if((fin - debut)< LIMITE){
// intervalle suffisamment petit: on fait le calcul
                        for(int i= debut;i<fin;i++){
                                resultat =resultat + donnees[i]*donnees[i];
                        }
                }else {
// sinon on divise en deux sous processus
                        int milieu = (debut + fin)/2;
                        System.out.println("Je divise en deux");
                        Carre gauche = new Carre(donnees, debut, milieu);
                        Carre droit = new Carre(donnees, milieu, fin);
                        gauche.fork();
                        droit.fork();
                        gauche.join();
                        droit.join();
                }
        }
        public static int getResultat() {
                return resultat;
        }
        public static void setResultat(int result) {
                Carre.resultat = result;
        }
}
```

L'algorithme est le suivant : la classe `Carre` est caractérisée par les bornes de l'intervalle sur lequel elle fait le calcul, ici `debut` et `fin`. Si cet intervalle est de taille inférieure au grain de calcul, c'est-à-dire 3, on fait le calcul, sinon on divise l'intervalle en deux et on crée deux sous-processus avec un diamètre d'intervalle divisé par deux.

Pour utiliser cette classe, on va soumettre les requêtes à un `ForkJoinPool` qui va gérer les threads comme prévu.

```
ForkJoinPool pool = new ForkJoinPool();
int [] data = {1,2,3,4,5,6,7,8,9,10};
Carre app = new Carre(data, 0, data.length);
pool.invoke(app);
System.out.println(app.getResultat());
```

Lorsque l'on fait l'exécution, on constate 4 divisions par deux, et voici une trace d'exécution qui est schématisée dans la figure 2.7 :

```
Je divise en deux
Je divise en deux
Je calcule de 0 a 2
Je divise en deux
Je calcule de 2 a 3
Je calcule de 3 a 5
Je divise en deux
```

Copyright © 2015 Dunod.
© Dunod – Toute reproduction non autorisée est un délit.

```
Je calcule de 5 a 7
Je divise en deux
Je calcule de 7 a 8
Je calcule de 8 a 10
```

Cette technique se limite donc à une décomposition des tâches arborescente avec un strict parcours d'appel et retour.

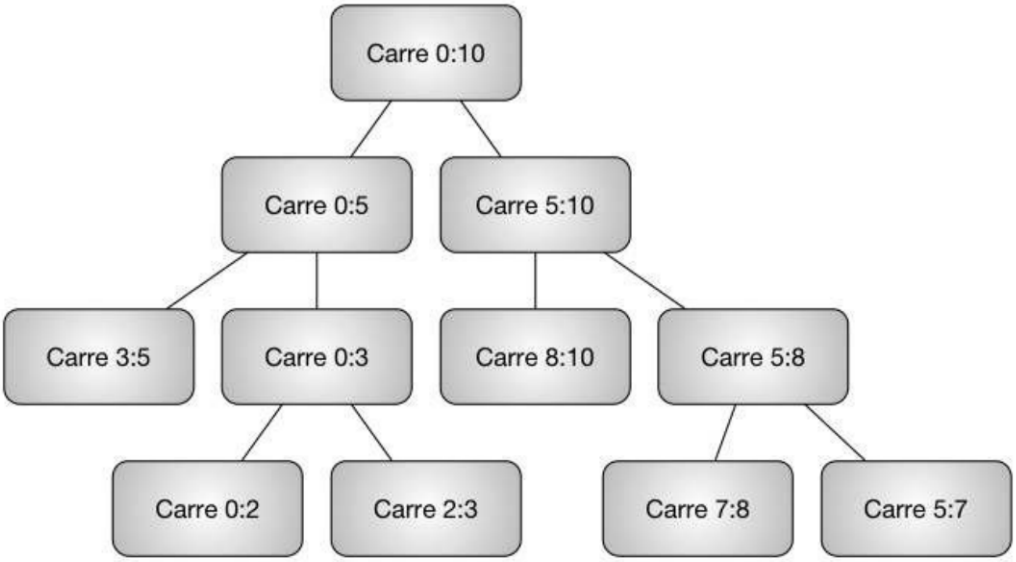


Figure 2.7 — Décomposition arborescente des tâches avec l'utilisation de `RecursiveAction`

De plus cette technique ne répond par correctement aux problèmes de complétion de la tâche, car la méthode `complete()` est ambigüe comme l'illustre l'exemple suivant :

```
ForkJoinTask<Integer> tache= ForkJoinPool.commonPool().submit(()->{
       Thread.sleep(1000);
       return 42;});

       Thread.sleep(10);
       tache.complete(43);
       System.out.println(tache.get());
       //renvoie 43
       Thread.sleep(1500);
       System.out.println(tache.get());
       //renvoie 42
```

Lors du premier appel à `get`, la tâche n'est pas terminée et renvoie la valeur par défaut. Le bon résultat n'est renvoyé que lors du deuxième appel.

Le problème est que lorsqu'une tâche est accomplie sa valeur est immuable. Comment y remédier ? En mettant la variable dans un conteneur, le `Future`.

Copyright © 2015 Dunod.

Les classes CompletableFuture et CompletableStage

Java 8 introduit une nouvelle API asynchrone, `CompletionStage`.

Pour optimiser l'enchaînement des tâches en Java, il existe deux approches différentes :

- Les *futurs*, avec l'interface `java.util.concurrent.Future`, qui définit des méthodes pour tester ou attendre la fin d'une tâche.
- Les *callbacks*, où l'on précise qui prendra la main lors de la fin d'une tâche. Pour cela on utilise `CompletionStage` ou `CompletionFuture`.

La classe `CompletionStage` permet de faire des calculs asynchrones, en définissant quelle phase suivante prendra la main : c'est donc une tâche qui se déclenche à la fin d'une autre et qui peut à son tour en déclencher de nouvelles. Il s'agit d'une étape dans un traitement global. Une tâche peut dépendre de plusieurs autres, ou en déclencher plusieurs. On peut ainsi décrire un *workflow* complexe avec des synchronisations et des ramifications.

On s'intéressera à la classe `CompletableFuture` qui implémente à la fois `Future` et `CompletionStage`. Les objets de cette classe peuvent être dans trois états :

- en train d'être calculés,
- calculés, ayant produit un résultat,
- calculés, ayant généré une exception.

Les méthodes suivantes sont disponibles :

```
boolean cancel(boolean peutInterrompreEnCours) ;
```

```
boolean isCanceled() ;
boolean isDone() ;
```

```
V get() ; // appel bloquant
V get(long timeout, TimeUnit timeUnit)
      Throws InterruptedEception, ExecutionException, TimeoutException
// appel avec armement d'un délai
```

Méthodes de type « Future » :

```
V join() ; // synchronisation sur terminaison
V getNow(V valeurParDefaut) ;
```

Il y a trois manières de représenter les tâches :

- `Function` : prend un argument, retourne une valeur.
- `Consumer` : prend un argument.
- `Runnable`

Copyright © 2015 Dunod.
© Dunod – Toute reproduction non autorisée est un délit.

En utilisant respectivement les méthodes `apply`, `accept` ou `run`.

Il est possible de créer un `CompletableFuture` par son constructeur :

```
CompletableFuture completableFuture = new CompletableFuture();
```

Ou en lui passant une tâche :

```
CompletableFuture completableFuture = CompletableFuture.supplyAsync( ( ) ->
//gros calcul
            100) ;
```

Cette syntaxe utilise les nouvelles lambdas expression introduites par Java 8.

Pour obtenir les résultats d'un `CompletableFuture` il y a plusieurs options :

- Attente sur la fin de la tâche ou son annulation :

```
completableFuture.get() ;
```

- Renvoie de la valeur si elle disponible, sinon de la valeur par défaut :

```
completableFuture.getNow("valeurParDefaut") ;
```

- Attente sur armement d'un délai, ici 3 secondes :

```
completableFuture.get( 3, TimeUnit.SECONDS )
```

Voici un petit exemple simple pour créer une `CompletionStage` qui renvoie la valeur 10.

```
CompletableFuture<String> grosCalcul = CompletableFuture.supplyAsync(() ->
// gros calcul
"10");
System.out.println(grosCalcul.get());
System.out.println(grosCalcul.getNow("valeurParDefaut"));
```

Le premier `get` renvoie la valeur attendue, mais le `getNow` renvoie également la même valeur car elle est disponible !

Chaînage

Il est possible de chaîner des tâches en utilisant respectivement `thenApply`, `thenAccept` ou `thenRun`.

Voici un petit exemple d'un calcul suivi d'un post-traitement.

On commence par créer un gros calcul... qui renvoie 10 :

Copyright © 2015 Dunod.

```
CompletableFuture<Integer> grosCalcul1 = CompletableFuture.supplyAsync(() ->
       // gros calcul
       10);
```

On crée ensuite le post-traitement sous forme de `Runnable`. Ici on se contente d'imprimer le mot « Fin » :

```
Runnable postTraitement = new Runnable() { public void run() {
System.out.println("Fin");}};
```

On peut alors créer une tâche enchaînée :

```
CompletableFuture<Void> enchainement = grosCalcul1.thenRun(postTraitement);
```

Et évaluer sa valeur :

```
System.out.println(enchainement.get());
```

Combinaison de tâches

On peut également combiner des tâches en précisant l'opérateur de combinaison. Dans l'exemple suivant, on va effectuer deux gros calculs, puis comparer numériquement les résultats obtenus.

Premier calcul qui renvoie la valeur 10 :

```
CompletableFuture<Integer> grosCalcul1 = CompletableFuture.supplyAsync(() ->
    // gros calcul 1
    10);
```

Deuxième calcul qui renvoie la valeur 100 :

```
CompletableFuture<Integer> grosCalcul2 = CompletableFuture.supplyAsync(() ->
    // gros calcul 2
    100);
```

Combinaison des deux par comparaison :

```
CompletableFuture<Integer> combine =
grosCalcul1.thenCombine(grosCalcul2,Integer::compareTo);
System.out.println(combine.get());
```

Le résultat obtenu est –1 puisque la valeur du premier calcul est inférieure à la valeur du deuxième.

Copyright © 2015 Dunod.
© Dunod – Toute reproduction non autorisée est un délit.

Synchronisation de plusieurs tâches

La synchronisation de plusieurs tâches va utiliser la primitive `join`. Cette primitive va attendre la terminaison de la tâche.

Voici un exemple pour lancer plusieurs calculs. On commence par créer une méthode qui renvoie le carré d'un nombre :

```
private static int carre(int a){
        return a*a;
    }
```

On crée ensuite une liste de nombres :

```
List<Integer> list = Arrays.asList(10,20,30,40);
```

On va parcourir cette liste avec une stream et associer à chaque élément un `CompletableFuture` qui appelle la fonction carre :

```
list.stream().map(data->CompletableFuture.supplyAsync(()->carre(data)))
```

On enchaîne chaque tâche obtenue avec une tâche définie par une lambda :

```
n->n*n
```

Cette lambda calcule elle aussi le carré ! On obtiendra donc le carré du carré du nombre.

On attend sur tous les résultats et on les imprime :

```
list.stream().map(data->CompletableFuture.supplyAsync(()->carre(data))).
                map(compFuture->compFuture.thenApply(n->n*n)).map(t->t.join())
                .forEach(s->System.out.println(s));
```

On remarquera la nouvelle syntaxe d'enchaînement des méthodes de Java 8, qui donne un code particulièrement compact.

Il est également possible de prendre en compte le calcul le plus rapide comme dans l'exemple suivant. Un premier calcul va prendre 2,3 secondes et renvoyer 2 :

```
CompletableFuture<Integer> calcul1= CompletableFuture.supplyAsync(()->{
       Thread.sleep(2300);
       return 2;});
```

Un deuxième calcul va prendre 2,2 secondes et renvoyer 3 :

```
CompletableFuture<Integer> calcul2= CompletableFuture.supplyAsync(()->{
       Thread.sleep(2200);
       return 3;});
```

Copyright © 2015 Dunod.

On peut alors constater que la mise au carré du plus rapide est bien 9, c'est-à-dire que l'on prend le résultat du calcul2.

```
CompletableFuture<Integer> lePlusRapide=calcul1.applyToEither(calcul2,
          (r)->r*r);
System.out.println(lePlusRapide.get());
```

Conclusion

Les nouvelles classes de synchronisation `CompletableFuture`, `CompletionStage` permettent ainsi de représenter un graphe quelconque de dépendance entre processus, pas seulement des arbres, et résolvent le problème de la complétion d'une tâche. Elles offrent bien d'autres possibilités pour faire des calculs asynchrones et synchroniser sur une tâche parmi *n*, renvoyer des exceptions, déclencher un *handler* lors de la fin de tâche, mais nous nous limiterons à ces quelques exemples rudimentaires. Ces techniques sont utilisées pour obtenir des graphes de traitements complexes dans les applications de traitement des données intensives (Big Data), et nous laissons le soin au lecteur d'approfondir le sujet par lui-même.

2.6 DESIGN PATTERNS

Les **design patterns** ou *modèles de conception* décrivent des organisations pratiques de classes objets [Gof4]. Ces organisations résultent souvent d'une conception empirique, le concepteur des objets tente de faciliter la réutilisation et la maintenance du code. On peut donc concevoir un modèle de conception comme une forme d'organisation transposable à plusieurs applications. Ces systèmes peuvent apparaître complexes aux débutants voire inutiles, il est pourtant très important d'en connaître plusieurs et de les appliquer systématiquement (dans les cas reconnus comme pouvant évoluer). L'architecte objet se construit petit à petit un « panier » de modèles.

Les design patterns ne sont pas réellement normalisés, mais on peut les découper en trois grandes catégories :

- **Les modèles de création** : ces modèles sont très courants pour déléguer à d'autres classes la construction des objets.
- **Les modèles de structure** : ces modèles tendent à concevoir des agglomérations de classes avec des composants de haut niveau.
- **Les modèles de comportement** : ces modèles tentent de répartir les responsabilités entre chaque classe (l'usage est plutôt dynamique).

Si on voulait faire un parallèle avec UML, les deux premiers modèles seraient liés à des diagrammes statiques (de classes) alors que le dernier modèle est davantage lié à un diagramme dynamique (de séquence).

Nous ne présentons ci-dessous que les patterns les plus utilisés dans ce livre, mais nous mentionnerons au cours de l'ouvrage les patterns d'Observateur et de Délégation que le lecteur pourra retrouver dans [Gof4].

Copyright © 2015 Dunod.
© Dunod – Toute reproduction non autorisée est un délit.

2.6.1 Pattern de la fabrique

Une fabrique de création (*factory*) est une classe qui n'a pour rôle que de construire des objets. Cette classe utilise des interfaces ou des classes abstraites pour masquer l'origine des objets et masquer la classe utilisée concrètement pour créer les objets. Dans le livre [Gof4] la fabrique est utilisée pour choisir dynamiquement le type d'objet à créer par exemple dans un graphique. De manière plus générale, on parlera de *fabrique* pour désigner une classe responsable de créer d'autres objets. Nous verrons ainsi des fabriques de connexions pour JMS (`ConnectionFactory`), mais aussi des fabriques d'objets distants.

La figure 2.8 montre le lien entre la fabrique et les objets produits. La méthode de création renvoie des objets du type Produit.

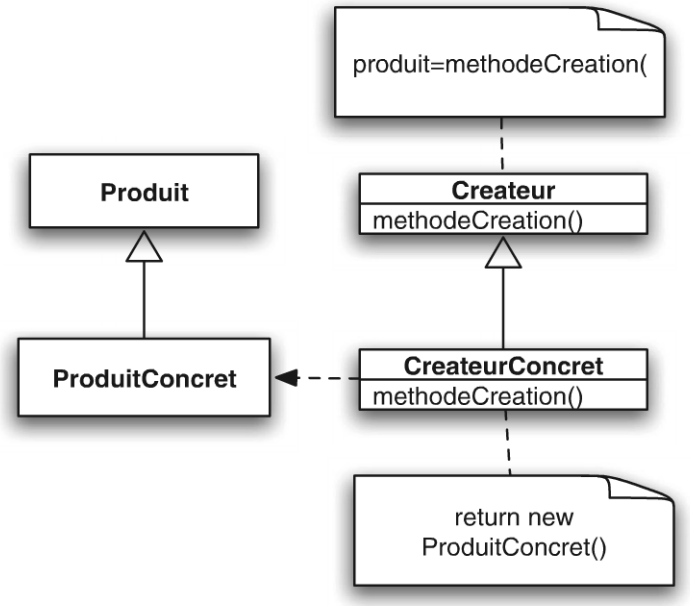


Figure 2.8 — Fabrique et produit concret

On distingue parfois deux formes de fabrique :

- **Les fabriques abstraites** comme celle que nous allons voir reposant sur l'exploitation de classes génériques (`Point` et `Line`).
- **Les fabriques concrètes** masquant toutes les méthodes nécessaires à la création et à l'initialisation de l'objet.

Exemple

- Interface de description d'un point :

```
public interface Point {
    /** Retourne l'abscisse du point */
    public int getX();
```

Copyright © 2015 Dunod.

```
    /** Retourne l'ordonnée du point */
    public int getY();
}
```

- Interface de description d'une ligne :

```
public interface Line {
    /** Retourne les coordonnées du premier point */
    public int getX1();
    public int getY1();
    /** Retourne les coordonnées du deuxième point */
    public int getX2();
    public int getY2();}
```

- Voici à présent le code d'une fabrique retournant des points ou des lignes :

```
public class CanvasFactory {
    /** Retourne un Point aux coordonnées x,y */
    public Point getPoint( int x, int y ) {
    return new PointImpl( x, y );
    }
    /** Retourne une Ligne aux coordonnées x1,y1,x2,y2 */
    public Line getLine( int x1, int y1, int x2, int y2 ) {
    return new LineImpl( x1, y1, x2, y2 );
    }
}
```

Dans cet exemple, nous définissons deux interfaces `Point` et `Line` représentant deux classes abstraites. Ces classes Point et Line désignent des objets retournés par la classe `CanvasFactory`. Cette classe masque la véritable nature des objets. Ici nous retournons par les méthodes d'accès des objets `PointImpl` et `LineImpl` qui implémentent respectivement les interfaces `Point` et `Line`. Ainsi, l'application utilisera la classe `CanvasFactory` pour obtenir des éléments graphiques, et lors d'une évolution l'utilisateur pourra changer facilement la nature des objets (avec d'autres classes implémentant les interfaces `Point` et `Line`...).

Dans le chapitre 8 sur JMS, nous verrons que ce pattern est appliqué afin de permettre plusieurs implémentations de la même API JMS et de permettre à une application de modifier facilement l'implémentation. Les points d'entrée correspondent à des fabriques (`ConnectionFactory`) interchangeables selon les implémentations. Nous verrons également comment utiliser des fabriques pour créer des objets distants : en effet, les classes en Java ne sont pas accessibles à distance, et il faut alors passer par des objets intermédiaires, les fabriques.

2.6.2 Pattern du mandataire

Un mandataire (*proxy*) intercepte les méthodes de l'objet qu'il contrôle et les effectue à la place du client, mais le plus souvent avec sa coopération. Les proxies sont en général transparents pour le client qui effectue les requêtes, et ils remplacent la référence de l'objet auprès du client.

Copyright © 2015 Dunod.
© Dunod – Toute reproduction non autorisée est un délit.

Il existe un certain nombre de proxies bien connus :

- **Proxy distant** (*remote proxy*) : il fournit une référence sur un objet situé dans un autre espace d'adresses ou une machine différente (objet distant).
- **Proxy virtuel** (*virtual* proxy) : il permet la création à la demande d'un objet consommateur de mémoire. Cet objet sera créé à son utilisation (évaluation paresseuse). Par exemple une image ne sera chargée que lors de son affichage.
- **Proxy de copie par nécessité** (*copy-on-write proxy*) : il retarde la copie d'un objet à l'instant de son utilisation réelle. C'est un cas particulier de proxy virtuel.
- **Proxy de protection** (*protection (access) proxy*) : il fournit différents niveaux d'accès à différents clients, notamment pour la sécurité.
- **Proxy de cache** (*cache proxy*) : il fournit un stockage temporaire des résultats d'une opération gourmande en ressources afin de partager ce résultat soit entre plusieurs clients soit entre plusieurs requêtes.
- **Proxy de pare-feu** (*firewall proxy*) : il protège l'accès à l'objet des indésirables.
- **Proxy de synchronisation** (*synchronization proxy*) : il contrôle les accès concurrents pour le compte de l'objet qu'il protège.
- **Proxy intelligent** (*smart reference proxy*) : il rajoute un comportement sur les actions déléguées à l'objet cible, par exemple la journalisation ou le comptage de références à l'objet.

La figure 2.9 montre la structure du pattern de proxy : le proxy implémente toute l'interface du sujet réel et fait appel aux méthodes de l'objet réel lors de l'appel de ses propres méthodes.

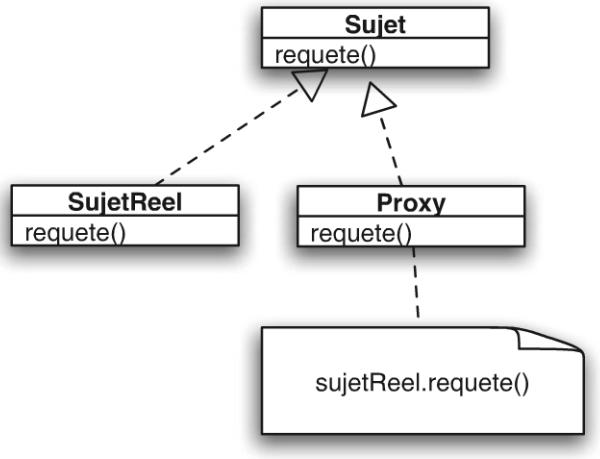


Figure 2.9 — Structure du pattern de proxy

L'utilisateur d'un sujet réel passe toujours par un proxy pour effectuer des opérations sur le sujet réel. Le proxy possède également un lien vers le sujet réel (figure 2.10).

Copyright © 2015 Dunod.

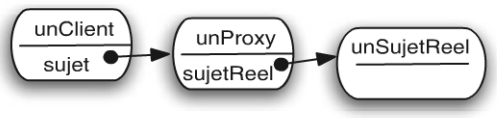


Figure 2.10 — Collaboration entre client et sujet réel au travers du proxy

2.6.3 Proxies dynamiques

L'inconvénient des proxies est qu'ils sont intimement liés à l'objet qu'ils représentent et doivent notamment implémenter la totalité des méthodes de l'objet.

Depuis Java 1.3, Sun a introduit les proxies dynamiques qui peuvent implémenter plusieurs interfaces, à condition qu'elles n'aient pas de conflits de noms de méthodes. Il existe d'autres restrictions :

- Les objets qu'ils représentent doivent être l'implémentation d'interfaces.
- Les interfaces doivent être dans le même package et visibles du même `ClassLoader`.

Dans l'exemple suivant, on souhaite journaliser toutes les actions d'une voiture. On va donc utiliser un proxy dynamique pour ajouter des écritures à chaque action de la voiture.

La voiture implémente l'interface simple suivante :

```
public interface InterfaceVoiture {
    public void demarre();
    public void arrete();}
```

Et voici la classe `Voiture` associée à cette interface :

```
public class Voiture implements InterfaceVoiture{
    public void demarre() {}
    public void arrete() {}
}
```

Il faut alors écrire une classe qui implémente `java.lang.reflect.InvocationHandler` et qui représente le corps du proxy avec délégation des appels vers le sujet. Cette classe possède un pointeur vers la voiture, le sujet du pattern de proxy. Elle se contente de rajouter une trace à l'appel de toutes les méthodes de la voiture.

```
public class JournalInvocationHandler implements InvocationHandler {
    Voiture sujet;
    public Object invoke(Object objet, Method methode, Object[] arguments)
    throws Throwable {
    System.out.println("Invocation de : "+methode.getName());
    return methode.invoke(sujet, arguments); }
public JournalInvocationHandler(Voiture v) {sujet = v; }}
```

© Dunod – Toute reproduction non autorisée est un délit.
Copyright © 2015 Dunod.

Nous pouvons à présent tester le programme.

On crée une voiture et un `InvocationHandler`. Il est nécessaire également d'associer le proxy à un `ClassLoader`, qu'on prendra ici identique à celui de la voiture. On doit également spécifier la liste des interfaces implémentées par le proxy dynamique, ici seulement l'interface de voiture.

```
Voiture voiture = new Voiture();
ClassLoader cl = voiture.getClass().getClassLoader();
Class[] listeClasses= new Class[]{InterfaceVoiture.class};
JournalInvocationHandler journal= new JournalInvocationHandler(voiture);
InterfaceVoiture proxy=(InterfaceVoiture) Proxy.newProxyInstance(cl,
    listeClasses, journal);
proxy.demarre();
proxy.arrete();
```

Lorsque l'on exécute le programme, on observe la trace de toutes les opérations effectuées sur la voiture.

2.7 PROPRIÉTÉS EN JAVA

On peut vouloir passer des propriétés en argument d'un programme Java lors de son lancement. Ceci s'effectue en utilisant le paramètre `-D` sur la ligne de commande comme ceci :

```
java -Dpropriete1=valeur1 -Dpropriete2=valeur2 package.Classe
```

Toutefois, il existe un équivalent en renseignant les propriétés dans le code.

```
import java.util.Properties ;
Properties props= System.getProperties() ;
props.put(propriete, valeur) ; // sous forme de chaînes
```

Il existe de nombreuses propriétés du système prédéfinies telles que `user.home`, `java.home`, etc. Les applications réparties font appel à de nombreuses propriétés pour renseigner la configuration du bus logiciel ou spécifier des éléments de sécurité. Par convention, ce livre utilisera les propriétés dans le code, afin de présenter un code uni, sans avoir besoin de rattacher le lancement d'une application à son code effectif.

2.8 RÉFLEXION

Depuis Java 1.2, Java fournit une interface d'introspection des classes : on parle de réflexion, pour indiquer la connaissance réflexive qu'une classe a d'elle-même.

Copyright © 2015 Dunod.

On peut ainsi connaître (figure 2.11) :

- la liste des attributs d'une classe,
- la liste de ses opérations,
- la liste des interfaces qu'elle implémente,
- la liste de ses constructeurs.



Figure 2.11 — Une classe connaît la liste de ses attributs et de ses opérations

Nous retrouverons ces éléments dans l'interface d'administration JMX au chapitre 10.

2.8.1 Classe d'un objet

La classe d'un objet est une entité de type `java.lang.Class`, que l'on peut obtenir soit à partir d'un objet, soit à partir d'un symbole.

Les types primitifs possèdent également une classe. Java 5 permet à présent de faire la conversion implicite entre types natifs et classes associées, par exemple entre `int` et `Integer` (*autoboxing*).

```
Class c = unObjet.getClass();
Class c = java.lang.String.class;
Class c = int.class; // aussi les types primitifs
```

On peut forcer le chargement d'une classe ainsi :

```
Class c = Class.forName (''package.nomClasse'')
```

Et tester l'appartenance d'un objet à une classe :

```
boolean b = "ma jolie chaine" instanceof String
```

En Java les classes ne sont pas des objets. On ne peut donc pas créer une classe ainsi : `new Class().setMethod...`

Copyright © 2015 Dunod.
© Dunod – Toute reproduction non autorisée est un délit.

Le pedigree de la classe

On peut connaître plusieurs choses d'une classe :

- Son package : `Package p=c.getPackage();`
- Si c'est une interface : `boolean b=c.isInterface ();`
- Sa superclasse : `Class c1 = c.getSuperclass ()`
- La liste des interfaces qu'elle implémente :
  `Class[] c2 = c.getInterfaces ()`

2.8.2 Méthodes d'une classe

La classe `Method` va permettre d'obtenir des informations sur une méthode. La liste des méthodes d'une classe est obtenue ainsi :

```
Methods[] m1 = uneClasse.getDeclaredMethods (); // non héritées
Methods[] m1 = uneClasse.getMethods (); // toutes les méthodes héritées ou non
```

On peut obtenir le type de retour d'une méthode :

```
m.getReturnType() == int.class
```

Ou accéder à la méthode d'une classe en connaissant sa signature (le tableau des types de ses arguments) :

```
Method m = uneClasse.getMethod (''methodName'', new Class[] {...}) ;
```

Exemple

```
Method method1 = String.class.getDeclaredMethod("indexOf",
    new Class[]{int.class});
Method method2= String.class.getDeclaredMethod("indexOf",
    new Class[]{String.class,int.class });
```

Nous ne parlerons par ici des attributs et des constructeurs d'une classe, car le but de l'exposé est de montrer l'invocation dynamique, et d'introduire ce concept pour les applications réparties.

Invocation d'une méthode

On peut invoquer une méthode dynamiquement en utilisant `invoke` sur la méthode et en lui passant l'objet qui recevra le message et la liste des arguments.

```
Method m..
m.invoke (objetInvoque, new Object[] {...})
```

Copyright © 2015 Dunod.

Le récepteur de la méthode (`objetInvoque`) est passé en argument. Les arguments sont passés en tant que tableau d'objets : `new Object[]`

On peut par exemple invoquer toutes les méthodes d'un objet commençant par « `get` » :

```
if (methods[i].getName().startsWith("get")) {
    Object value= methods[i].invoke(objet,null);}// liste d'arguments vide
```

Ainsi on peut comparer l'appel statique et l'appel dynamique de la recherche d'une sous-chaîne dans une chaîne :

- Appel classique :

```
"un bel avenir".indexOf("bel");// Renvoie 3
```

- Appel dynamique :

```
// la méthode indexOf d'une String
Method m=String.class.getMethod("indexOf", new Class[] {String.class};
// on appelle la méthode avec le paramètre "bel" sur la chaîne "un bel avenir"
Object result=m.invoke("un bel avenir",new Object[] {"bel"}); // renvoie 3
```

En résumé

Ce chapitre a présenté les technologies de base pour utiliser les applications réparties en Java. La sérialisation permet de comprendre comment transitent les structures d'objets sur le réseau. Les threads vont être utiles pour programmer les activités concurrentes sur les serveurs qui exécutent plusieurs requêtes en parallèle. Les futurs sont un ajout de Java 5 fondamental pour programmer des applications asynchrones. Ils peuvent être combinés à des appels RMI, SOAP ou CORBA. Java 6 et 7 ont enrichis également ces mécanismes.

Ce livre utilisera plusieurs design patterns, tel que le Proxy ou la Factory. Pour finir, la réflexion en Java permet de réaliser l'invocation dynamique d'une méthode. Nous verrons que les applications réparties en tirent profit pour un déploiement facilité.

Copyright © 2015 Dunod.
© Dunod – Toute reproduction non autorisée est un délit.

Copyright © 2015 Dunod.

3

Sockets et Java NIO

Objectif

Le but de ce chapitre est de rappeler des concepts de base pour la programmation d'applications en réseau, et de montrer également comment les nouvelles bibliothèques de Java NIO (nouvelle API Java pour les entrées-sorties) permettent d'ajouter des fonctionnalités pour une performance accrue.

3.1 COMMUNICATION EN RÉSEAU

3.1.1 Couches basses du système

La manière la plus ancienne de communiquer entre applications réparties a été l'utilisation des sockets. Les sockets sont des ports de communication ouverts au niveau des couches de réseau du système d'exploitation, et permettant de faire passer des flux (*streams*) d'octets.

Les sockets permettent de communiquer en point à point en mode client/serveur. La socket serveur va accepter des connexions sur un port de communication donné, puis va analyser le flux obtenu ou écrire dans le flux. L'analyse du flux doit être faite manuellement. Le client va pour sa part se connecter et faire de même.

En Java l'utilisation de sockets est portable quel que soit le système d'exploitation. Java permet d'accéder aux ressources réseau de bas niveau, par exemple pour retrouver l'adresse ou le nom de l'ordinateur sur lequel on travaille. Par convention `localhost` désignera toujours l'ordinateur local, de même que l'adresse IP au format IPv4 `127.0.0.1`. En IPv6 l'adresse sera `::1`.

Copyright © 2015 Dunod.
© Dunod – Toute reproduction non autorisée est un délit.

```
InetAddress.getByName("localhost")
InetAddress.getLocalHost()
// retourne le nom et l'adresse IP de l'ordinateur local
```

On notera également la possibilité d'utiliser des adresses IPv4 ou IPv6.

IPv6 est un protocole défini au sein de l'organisme de normalisation IETF destiné à succéder à IPv4 et qui permet d'utiliser des adresses réseau sur 128 bits au lieu de 32 bits, permettant ainsi de multiplier les adresses disponibles. Ceci est devenu nécessaire par le succès de l'internet et l'apparition du concept d'internet des objets, alors que l'on est proche d'avoir attribué toutes les adresses disponibles en IPv4. Avec 128 bits, on dispose de beaucoup plus d'adresses disponibles.

3.1.2 Adresses en IPv4, IPv6 et interfaces réseau

On peut en Java obtenir un objet adresse soit en donnant le nom de la machine si le réseau dispose d'un service de nommage (DNS, *Domain Name Server*), soit en donnant l'adresse IP.

Par exemple si l'on souhaite connaître l'adresse IP sur serveur de mail d'Orange, on peut utiliser :

```
InetAddress.getByName("smtp.orange.fr") ;
```

Et l'on obtiendra le résultat :

```
smtp.orange.fr/193.2252.22.64
```

Si l'on souhaite obtenir un objet Java pointant sur l'adresse IPv4 192.168.1.6 on pourra utiliser :

```
InetAddress.getByAddress(new byte[]{(byte)192,(byte)168,(byte)1,(byte)6});
```

On remarquera l'utilisation de `getByAddress()` au lieu de `getByName()`.

`InetAddress` comporte deux sous-classes pour les deux protocoles IPv4 et IPv6, et si l'on précise que l'on souhaite une adresse IPv6 on pourra utiliser :

```
Inet6Address adresse=(Inet6Address)
Inet6Address.getByName("fe80:0:0:0:8810:534d:fd8e:fc65%11");
Inet6Address.getByName("::1"); // nom conventionnel de l'ordinateur local
```

Que se passe-t-il à présent si l'ordinateur est connecté au réseau par plusieurs interfaces, par exemple une prise Ethernet et une carte wifi ? Chaque interface réseau aura sa propre adresse, de même si l'on utilise une virtualisation telle que VMWare, l'adresse sera différente sur la machine virtuelle. Java a introduit pour cela la classe

Copyright © 2015 Dunod.

`NetworkInterface`, qui permet de faire la différence entre ces différentes connexions au réseau. Voici un programme simple pour afficher la liste de toutes les interfaces :

```
   Enumeration<NetworkInterface> nets = NetworkInterface
            .getNetworkInterfaces();
      for (NetworkInterface netint : Collections.list(nets))
         displayInterfaceInformation(netint);
   }
static void displayInterfaceInformation(NetworkInterface netint)
         throws SocketException {
      out.printf("Description : %s\n", netint.getDisplayName());
      out.printf("Nom: %s\n", netint.getName());
      Enumeration<InetAddress> inetAddresses = netint.getInetAddresses();
for (InetAddress inetAddress : Collections.list(inetAddresses)) {
         out.printf("Adresse Internet : %s\n", inetAddress);
      }
      out.printf("En marche ? %s\n", netint.isUp());
      out.printf("Loopback? %s\n", netint.isLoopback());
      out.printf("Point a Point? %s\n", netint.isPointToPoint());
      out.printf("Supporte multicast? %s\n", netint.supportsMulticast());
      out.printf("Virtuelle ? %s\n", netint.isVirtual());
      byte[] adresseMaterielle = netint.getHardwareAddress();
      out.printf("Adressse materielle : %s\n",
      Arrays.toString(adresseMaterielle));
out.printf("Unite Maximum de Transmission (MTU): %s\n", netint.getMTU());
      out.printf("\n");
```

On pourra constater que certaines interfaces ont plusieurs adresses, typiquement une pour IPv4 et une pour IPv6.

3.1.3 TCP et UDP

Il existe essentiellement deux protocoles de communication par socket, l'un par TCP (*Transmission Control Protocol*), l'autre par UDP (*User Datagram protocol*). TCP est dit fiable car il nécessite un point de rendez-vous entre le client et le serveur au moment de la connexion du client : on est assuré que les deux parties sont présentes au moment de la communication. En UDP par contre, il n'existe pas de point de rendez-vous, et donc le serveur n'est pas assuré que l'information a été reçue réellement.

UDP va par contre permettre une communication en diffusion à plusieurs clients (*multicast*).

3.1.4 Port de communication

Un serveur est défini par un port de communication sur une machine donnée. Comme le nombre de ports de communication est limité dans un système, et que beaucoup sont utilisés pour toutes sortes d'utilités systèmes (port 21 pour FTP, port 25 pour le mail,

Copyright © 2015 Dunod.
© Dunod – Toute reproduction non autorisée est un délit.

port 80 pour le trafic HTTP, etc.[1]) le nombre de ports disponibles est limité. Les pare-feu mettent souvent en place des stratégies de filtrage sur les ports de communication, et les systèmes Linux/Unix interdisent toute connexion à un utilisateur non root sur les ports inférieurs à 1024.

Un exemple de création de socket de communication sur la machine locale sur le port 5555 est présenté ici :

```
InetSocketAddress adresse = new InetSocketAddress("localhost", 5555);
System.out.println(adresse.getPort());
```

Nous verrons par la suite comment l'utilisation d'objets répartis va permettre de mutualiser un port de communication entre plusieurs objets.

3.1.5 Télécharger un fichier web à partir d'une URL

Il est très facile de télécharger un fichier web depuis une URL en Java, puisque Java considère qu'une URL est juste une extension d'un chemin de fichier. On peut donc facilement utiliser les outils fournis pour les fichiers pour télécharger la page. Nous verrons au chapitre 5 RMI qu'il est également possible de télécharger du code depuis une URL.

Voici donc le code pour obtenir la page http://www.afceurope.com/index.htm

```
public class Telecharge
{
   public static void getFile(String host)
   {
      InputStream input = null;
      FileOutputStream writeFile = null;
      try
      {
         URL url = new URL(host);
         URLConnection connection = url.openConnection();
         int fileLength = connection.getContentLength();
if (fileLength == -1)
         {
            System.out.println("URL erronée.");
            return;
         }
         input = connection.getInputStream();
         String fileName =
url.getFile().substring(url.getFile().lastIndexOf('/') + 1);
         writeFile = new FileOutputStream(fileName);
         byte[] buffer = new byte[1024];
         int read;
while ((read = input.read(buffer)) > 0)
            writeFile.write(buffer, 0, read);
         writeFile.flush();
```

1. http://www.iana.org/assignments/port-numbers

Copyright © 2015 Dunod.

```
        }
        catch (IOException e)
        {
           System.out.println("Erreur en chargeant le fichier.");
           e.printStackTrace();
        }
        finally
        {
           try
           {
              writeFile.close();
              input.close();
           }
           catch (IOException e)
           {
              e.printStackTrace();
           }}}
public static void main(String[] args)
   {getFile("http://www.afceurope.com/index.htm");}}
```

3.2 COMMUNICATION AVEC TCP

3.2.1 Mode client serveur

La communication par socket s'effectue en mode client serveur. Le serveur va écouter les requêtes des clients sur un port défini par l'application. Dans l'exemple qui suit, le client a l'initiative de la conversation (figure 3.1), mais le client et le serveur peuvent envoyer des informations sur la socket à tout moment, car les flux dans chaque sens sont indépendants. Les applications doivent par contre gérer l'état du dialogue.

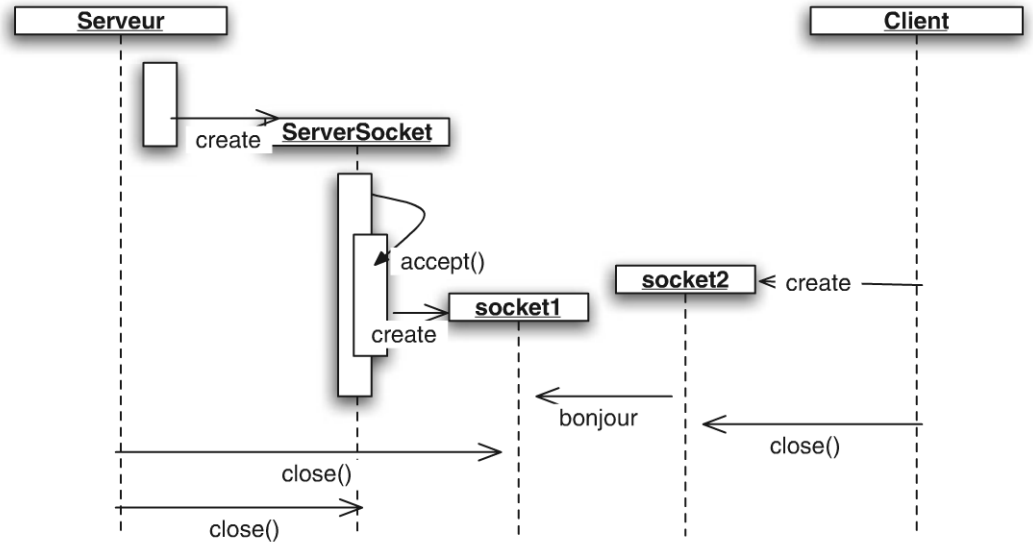


Figure 3.1 — Communication par sockets

Copyright © 2015 Dunod.
© Dunod – Toute reproduction non autorisée est un délit.

3.2.2 Code d'une socket serveur et d'une socket cliente

Ici on a choisi arbitrairement le port 6666 pour la connexion des clients. La socket est ensuite bloquée en attente de connexion de la part des clients : si l'affichage « Serveur prêt » s'effectue après le `accept`, vous ne le verrez s'afficher qu'au moment de la connexion.

```
ServerSocket serverSocket = new ServerSocket(6666);
System.out.println("Serveur prêt");
Socket socket1 = serverSocket.accept(); // attente bloquée
```

On notera l'utilisation de deux sortes de socket : la socket serveur et la socket provenant de la connexion d'un client. La socket serveur ne permet que la connexion, alors que la socket normale permet la lecture et l'écriture dans le flux de communication à l'aide de `InputStream` et `OutputStream` – les flux d'entrée-sortie associés à la socket.

Dans l'exemple suivant, la socket cliente va se connecter au serveur, puis écrire dans le flux de sortie. On a nommé volontairement cette socket `socket2`, pour montrer la symétrie avec la `socket1` du serveur. Le point d'entrée va être l'adresse de l'hôte du serveur, ici `localhost` sur le port `6666` :

```
Socket socket2 = new Socket("localhost", 6666);
PrintWriter writer = new PrintWriter(socket.getOutputStream());
writer.print("bonjour");
writer.flush();
socket2.close();
```

| Ne pas oublier le `flush` pour délimiter la fin de l'écriture.

On récupère le flux de sortie de la socket à l'aide de `getOuputStream()`.

3.2.3 Réception de l'information par le serveur

Dès que le client va se connecter, le serveur sera débloqué et pourra exécuter le code après le `accept` pour récupérer l'information du client :

```
InputStream stream = socket1.getInputStream();
BufferedReader reader = new BufferedReader(new InputStreamReader(stream));
String line = "";
while((line=reader.readLine()) != null ){
    System.out.println(line);
    }
socket1.close();
serverSocket.close();
```

L'objet `socket1` provient de la connexion du client dans le code précédent. Cette fois-ci on utilise le flux d'entrée pour lire les informations.

Copyright © 2015 Dunod.

`socket1` sur le serveur pourrait écrire dans le flux tout comme `socket2` dans le client, et leurs rôles sont une fois la connexion établie absolument symétriques. On peut ainsi nouer un dialogue.

On remarquera que la fermeture de la socket cliente ne lève pas d'exception, et n'est détectée que par le fait que le flux devient nul.

C'est une bonne pratique également de fermer les sockets, qui sont des ressources systèmes en nombre limité. Il est en particulier absolument nécessaire de fermer la socket serveur, sinon il est impossible de relancer le serveur sans obtenir une exception du type : `Address already in use`.

3.2.4 Utilisation de TCP

Il est déconseillé d'utiliser TCP pour communiquer au niveau applicatif pour des raisons de sécurité (intrusion directe sur la socket), d'agilité du code (le dialogue et l'échange des messages doivent être gravés dans la pierre). En particulier, le dialogue doit prendre en compte les déroutements sur gestion d'exceptions. En pratique, un dialogue complexe peut avoir des ramifications multiples, et le gestionnaire du dialogue du client et du serveur doit implémenter un analyseur syntaxique (*parser*) élaboré pour implémenter le protocole convenu entre le client et le serveur :

- Si je reçois « OK » et que l'état du système est « en marche », alors je renvoie « 200 ».
- Si je reçois « OK » et que l'état du système est « démarré », alors je renvoie « 300 »...

Mais TCP sous-tend toutes les implémentations des autres couches abstraites de RPC que nous verrons par la suite, et il est également utilisé pour la connexion entre serveurs, par exemple entre un serveur Apache et un moteur de servlet Tomcat, pour la connexion aux annuaires d'applications réparties, pour le recueil de journaux (logs) répartis, etc.

3.3 UTILISATION DE THREADS DANS UN SERVEUR

3.3.1 Connexion de plusieurs clients

Dans le modèle précédent, le serveur sert un seul client et puis s'arrête. Il a donc peu d'intérêt en pratique !

Une première idée est de boucler en séquence sur le `accept` (figure 3.2), afin de permettre des connexions successives de plusieurs clients.

```
boolean estActif ;
while(estActif) socket1 = serverSocket.accept() ;
//... traitement de la requête
```

© Dunod – Toute reproduction non autorisée est un délit.
Copyright © 2015 Dunod.

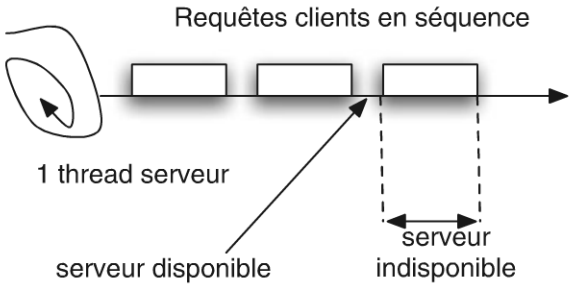


Figure 3.2 — Traitement des requêtes en séquence

Le problème est que si le traitement de la requête est trop long, les clients suivants pourront se connecter, mais leur requête ne sera pas traitée : on risque de perdre des clients.

On utilise donc en général des threads pour permettre le traitement simultané de plusieurs clients.

3.3.2 Utilisation de threads pour le traitement de requêtes en parallèle

Dans ce modèle, un serveur va lancer plusieurs threads en parallèle afin de traiter les requêtes de manière simultanée (figure 3.3).

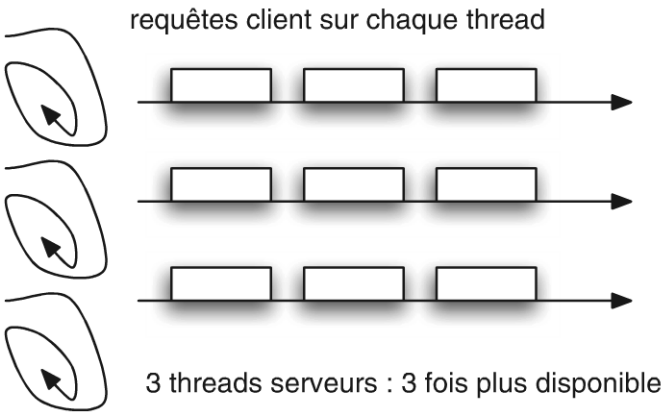


Figure 3.3 — Traitement en parallèle à l'aide de threads

Afin de rendre le code plus modulaire, on crée une classe auxiliaire, le `Handler`, qui hérite de `Thread` et définit la gestion de la requête. Ce `Handler` va prendre en paramètre la socket et gérer la requête :

```
class Handler extends Thread {
    private final Socket socket;
    // constructeur
    public Handler(Socket socket) {
        this.socket=socket;
}}
```

Copyright © 2015 Dunod.

Il définit dans la méthode d'exécution `run()` la gestion de la requête (les exceptions d'entrées-sorties ne sont pas mentionnées ici) :

```
public void run() {
    BufferedReader reader = new BufferedReader(
        new InputStreamReader(socket.getInputStream()));
    String line = "";
    while((line=reader.readLine()) != null ){
        System.out.println(line); // gère l'information reçue
    }
    socket.close();}
```

De cette manière, le serveur va se contenter de gérer les connexions, avant de déléguer le traitement aux instances de `Handler` :

```
while(estActif){new Handler(serverSocket.accept()).start();}
```

3.3.3 Limitations du nombre de threads : pool de threads

Un pool est un ensemble de ressources limité

Dans le code précédent, le nombre de connexions est virtuellement infini. Hélas, dans un système d'exploitation, le nombre de threads est très grand (32 000 sous Windows), mais limité. De plus, historiquement l'allocation des threads était coûteuse. Ce n'est plus vrai actuellement, mais on peut vouloir souhaiter maîtriser l'activité d'un serveur, soit pour des raisons de maîtrise de la gestion mémoire, soit pour d'autres raisons, par exemple l'équilibrage de charge entre applications.

Depuis Java 5, on dispose d'un package `java.util.concurrent` issu des travaux de Doug Lea [Lea2003] sur la programmation concurrente, et qui offre une boîte à outils très riche pour gérer les threads. On va donc créer à l'avance un nombre fixe de threads `nbThreads` qui seront recyclés au fur et à mesure de l'utilisation. Toute requête au-delà du compte sera alors rejetée ou traitée en exception.

```
ExecutorService pool = Executors.newFixedThreadPool(nbThreads);
while(estActif) {
    handler=new Handler(serverSocket.accept());
    pool.execute(handler);}
```

Il existe des variantes de `pool` :

- **newCachedThreadPool** : création de nouveaux threads ad libitum, mais réutilisation des existants.
- **newScheduledThreadPool** : exécution retardée ou périodique...

Le lecteur est invité à consulter les API de `Executors` pour approfondir ce sujet.

Copyright © 2015 Dunod.
© Dunod – Toute reproduction non autorisée est un délit.

Traitement des clients en excédent

Le pool de thread ne peut servir qu'un nombre fini de clients simultanément. Par exemple pour un pool de trois threads, il faut définir ce que devient le quatrième client : on peut définir une politique de rejet pur et simple, ou une politique d'attente.

Pour la **mise en attente**, on va utiliser des queues synchronisées telles que `PriorityBlockingQueue` ou `ArrayBlockingQueue`. Le client sera bloqué en attente de libération de ressources de threads. On peut limiter la durée d'attente avec la méthode `setKeepAliveTime`.

Pour la **politique de rejet**, il existe des politiques prédéfinies en Java comme l'annulation ou le rejet du plus vieux client en attente par exemple :

- `ThreadPoolExecutor.AbortPolicy`
- `ThreadPoolExecutor.DiscardOldestPolicy`

Mais on peut définir une politique personnalisée en définissant une nouvelle classe héritant de `RejectedExecutionHandler` et implémentant la méthode `rejectedExecution` :

```
pool.setRejectedExecutionHandler(new RejectedExecutionHandler() {
    public void rejectedExecution(Runnable r ,ThreadPoolExecutor executor)
    {System.out.println("tache rejetee");}});
```

En résumé

Les pools de threads sont utilisés pour des tâches courtes et massivement parallèles. Ils permettent de limiter le nombre de ressources simultanées utilisées et il existe une large palette de possibilités pour la personnalisation du comportement.

De manière générale, de nombreux serveurs d'application utilisent des pools de threads pour toutes sortes de services, en particulier pour les connexions JDBC (*Java Database Connectivity*) à des bases de données ou les connexions à des serveurs JMS (voir chapitre 8).

3.4 AUTRES TYPES DE SOCKETS

3.4.1 Sockets avec UDP

Le mode UDP est un mode non connecté, qui va permettre de gagner en performance. Par contre ce mode est moins fiable, car on ne teste pas la présence des émetteurs et récepteurs comme lors de la connexion en TCP. Ce protocole nécessite de segmenter l'information en paquets (figure 3.4), et il est très utilisé dans le monde multimédia, car il correspond bien à un mode de transmission rapide de données de même nature et en paquets.

Copyright © 2015 Dunod.

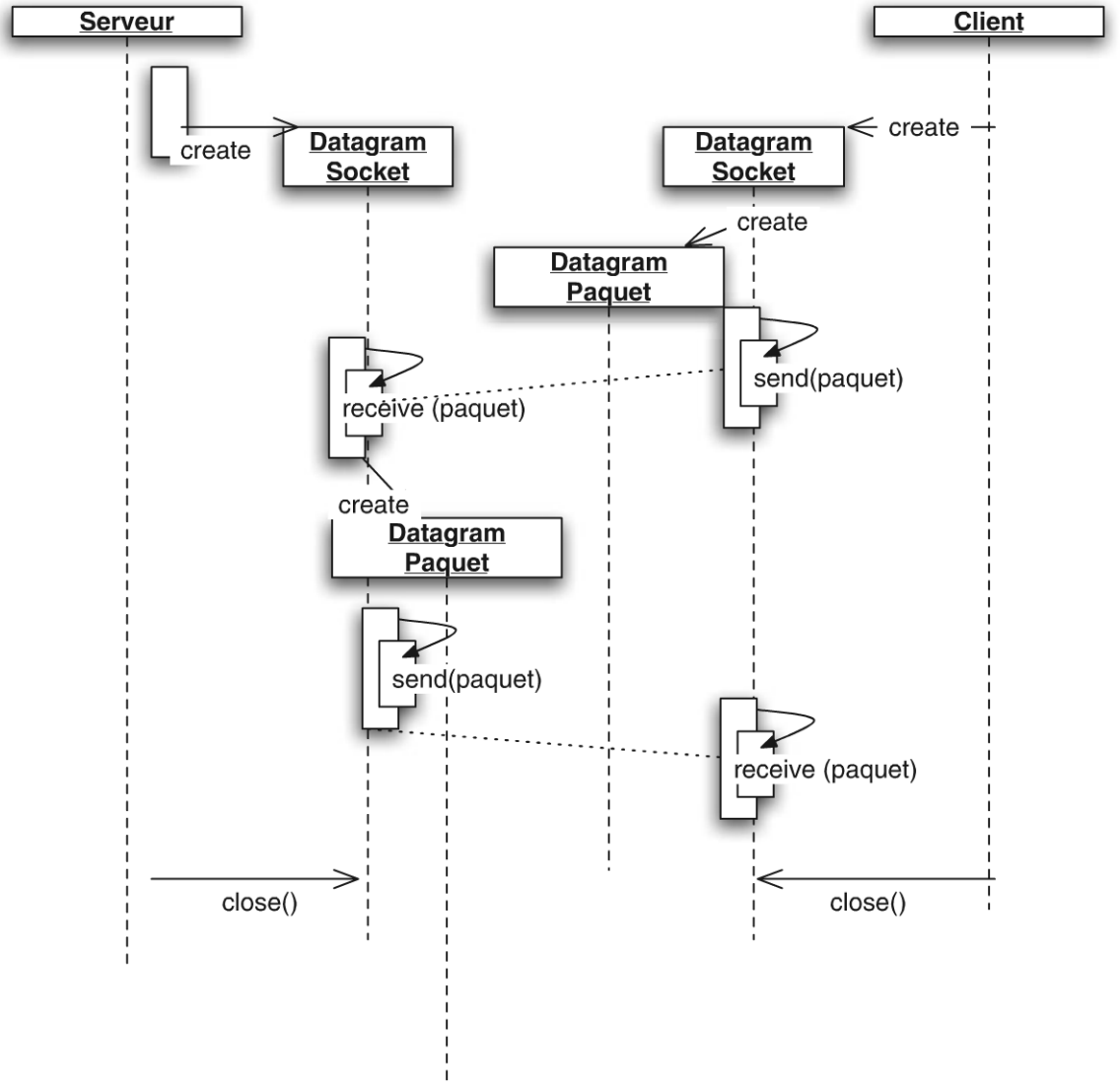


Figure 3.4 — UDP communique par échange de paquets

Serveur UDP

Un serveur UDP va commencer par ouvrir une socket UDP sur un port, ici le 4666 :

```
DatagramSocket socket = new DatagramSocket(4666);
```

On utilise ensuite deux tableaux d'octets servant de tampons, l'un pour l'émission, l'autre pour la réception, mais on est obligé de dimensionner le buffer, et l'on crée deux paquets pour l'émission et la réception.

```
byte[] sendbuffer = null;
byte[] receivebuf =new byte[256];
```

Copyright © 2015 Dunod.
© Dunod – Toute reproduction non autorisée est un délit.

```
DatagramPacket receivepacket = new DatagramPacket(receivebuf,
receivebuf.length);
DatagramPacket sendPacket = null;
System.out.println("Serveur prêt");
```

Le serveur est alors prêt à recevoir :

```
while(true){
    socket.receive(receivepacket);
```

On affiche le contenu du paquet reçu (on reçoit des octets que l'on convertit en chaîne de caractère) :

```
System.out.println(new String(receivepacket.getData()));
```

On envoie en réponse un message indiquant d'où vient l'appel en transformant la chaîne de caractères en octets (`getBytes`) :

```
InetAddress addr=receivepacket.getAddress();
int port = receivepacket.getPort();
sendbuf=("Vous êtes connecté depuis la machine " +addr+ "et sur le port "
    +port).getBytes();
sendPacket= new DatagramPacket(sendbuf, sendbuf.length, addr, port);
socket.send(sendPacket);
```

On remarquera que le serveur doit utiliser une adresse spécifique pour envoyer son datagramme. Il la récupère dans le message qu'il reçoit. Le client se voit allouer en effet un numéro de port de manière dynamique. On pourrait également imaginer qu'il écoute sur un port fixe. Client et serveur ont alors des rôles symétriques.

Client UDP

Le client UDP va ouvrir une socket de type `Datagram` et envoyer un texte en binaire : « Bonjour ». On remarquera que seul le paquet comporte l'adresse du serveur, et non la socket, et que la taille du paquet d'envoi est ajustée à la taille de l'information envoyée.

Par contre, la taille du tampon de réception est arbitraire, et le client et le serveur doivent s'accorder sur cette taille. La méthode `getData` va éliminer les octets non remplis par le serveur.

```
InetAddress addr = InetAddress.getByName("localhost");
DatagramSocket socket = new DatagramSocket();
byte[] buf ="Bonjour".getBytes();
DatagramPacket sendPacket = new DatagramPacket(buf, buf.length,addr,4666);
buf = new byte[256];
DatagramPacket receivePacket=new DatagramPacket(buf, buf.length,addr,4666);
socket.send(sendPacket);
socket.receive(receivePacket);
System.out.println(new String(receivePacket.getData()));
```

Copyright © 2015 Dunod.

Utilisation de UDP

UDP est un protocole non connecté qui va être utilisé essentiellement pour des transferts de données numériques. On peut typiquement découper une image ou un film en paquets de taille égale, et effectuer une boucle sur le client pour recevoir ces paquets. Toutefois il s'agit d'un protocole point à point. Dans le paragraphe suivant, nous allons voir comment permettre la diffusion vers plusieurs récepteurs.

3.4.2 Multicast

Il est possible d'utiliser des adresses multicast permettant la diffusion simultanée d'une information à plusieurs destinataires. Il existe des plages d'adresses réservées pour les adresses multicast. Ici l'adresse multicast utilisée est le 230.0.0.2. On utilise également les datagrammes. Tous les clients actifs au moment de l'émission et qui écoutent sur cette adresse recevront simultanément la même information. La diffusion est sélective d'où le nom de multicast qui s'oppose à broadcast (diffusion).

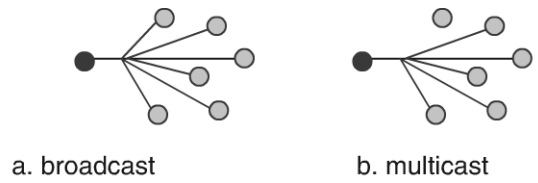


Figure 3.5 — Tous les clients actifs reçoivent la même information

Code du serveur

```
DatagramSocket socket = new DatagramSocket();
byte[] buf = "Hello".getBytes();
InetAddress group = InetAddress.getByName("230.0.0.2");
DatagramPacket packet = new DatagramPacket(buf, buf.length,group, 4666);
socket.send(packet);
```

On remarquera que le code est identique au code UDP, mais l'adresse est cette fois une adresse multicast : on ignore quels destinataires recevront ce message.

Code du client

Sur le client, on notera le code caractéristique utilisant `join`, puisque le client va se joindre à un groupe en écoute sur l'adresse multicast. On notera également l'utilisation de la classe `MulticastSocket`.

```
InetAddress addr= InetAddress.getByName("230.0.0.2");
InetSocketAddress address = new InetSocketAddress(addr,4666);
MulticastSocket socket= new MulticastSocket(4666);
socket.joinGroup(addr);
byte [] buf = new byte[256];
```

Copyright © 2015 Dunod.
© Dunod – Toute reproduction non autorisée est un délit.

```
DatagramPacket packet =new DatagramPacket(buf, buf.length);
socket.receive(packet);
System.out.println(new String(packet.getData()));
```

Utilisation du multicast

En dehors de la diffusion d'informations numériques, le multicast est également très utilisé pour le clustering de serveurs ou la réplication de bases de données.

3.5 UTILISATION DE JAVA NIO

3.5.1 Une nouvelle bibliothèque

Java NIO est une nouvelle bibliothèque introduite dans le JDK 1.4 permettant d'optimiser les performances des entrées-sorties en Java. Ainsi une copie de fichier pourra être jusqu'à dix fois plus rapide, au prix d'une gestion plus fine des ressources, et donc d'un code un peu plus complexe. Ceci s'effectue avec l'utilisation de traitement de données par blocs dans des buffers, et non plus au fil de l'eau avec des streams.

Avec les streams, les entrées-sorties sont traitées octet par octet, produisant ou consommant un seul octet à la fois. Avec les blocs, on peut optimiser le traitement en lisant un bloc d'un coup, mais au prix d'une augmentation de la complexité.

Les flux d'entrées-sorties vont être couplés à des canaux de communication (*Channel*) afin d'accélérer le transfert. Les *channels* sont bidirectionnels (figure 3.6).

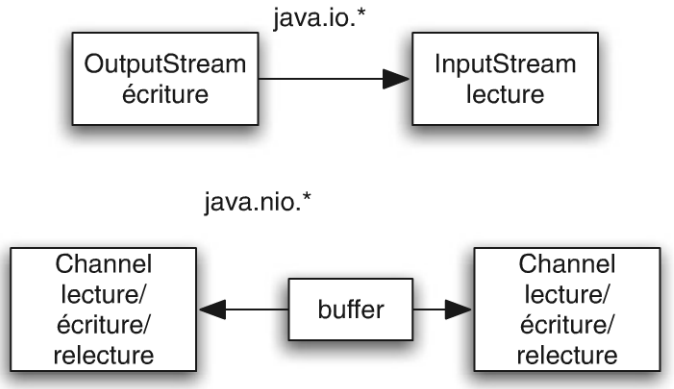


Figure 3.6 — Les *channels* sont bidirectionnels

Les deux bibliothèques `java.io.*` et `java.nio.*` continuent à coexister dans le JDK, mais de nombreuses primitives de `java.io.*` ont été réécrites.

- Exemple de code utilisant `java.io.*` :

```
System.out.println ("Hello World");
```

Copyright © 2015 Dunod.

- Le même code en utilisant `java.nio` :

```
String hello = "Hello World" + System.getProperty ("line.separator");
ByteBuffer bb = ByteBuffer.wrap (hello.getBytes ("UTF-8"));
WritableByteChannel wbc = Channels.newChannel (System.out);
wbc.write (bb);
wbc.close();
```

On voit donc un accroissement de la complexité. Au passage, on notera l'encodage des caractères en UTF-8. Les caractères peuvent en effet avoir un encodage différent selon la langue, en particulier pour les langues non occidentales. Ceci ne sera pas traité dans ce chapitre.

3.5.2 Canaux et tampons

Les canaux (*channels*) et les tampons (*buffers*) sont au centre de l'architecture `java.nio.*`.

Les buffers

Les données sont d'abord regroupées dans des *buffers*, avant de transiter par des *channels*. On ne peut plus lire une donnée directement dans un *stream*, on doit tout d'abord la lire ou l'écrire dans un *buffer*. Un *buffer* est juste un conteneur similaire à un tableau qui accumule les données. Ces *buffers* sont naturellement typés selon la nature des données : `ByteBuffer`, `IntBuffer`, `DoubleBuffer`... implémentent tous l'interface `Buffer` ; `ByteBuffer` (ou tampon d'octets) est le pivot de tous les autres.

Un *buffer* est accessible en lecture et écriture :

- soit en séquentiel : `get()`, `put(value)`
- soit en absolu : `get(i)`, `put(i, value)`

Son état est défini par quatre attributs :

- `capacity` : nombre d'éléments – invariable – fixé à la création.
- `limit` : index du premier élément non accessible.
- `position` : index du premier élément accessible.
- `mark` : la position de réinitialisation lors du *reset*.

> La relation suivante est toujours vraie :
> `0 <= mark <= position <= limit <= capacity`

On peut créer un buffer de trois manières :

```
ByteBuffer buffer1 = ByteBuffer.allocate(1024);
ByteBuffer buffer2 = ByteBuffer.allocateDirect(1024);
// lié à l'OS : mapping direct
ByteBuffer buffer3 = ByteBuffer.wrap(new String("hello").getBytes());
```

© Dunod – Toute reproduction non autorisée est un délit.
Copyright © 2015 Dunod.

Les Channels

Les `Channel` sont comparables aux `Stream` de `java.io.*`, et sont des canaux de communication pour transférer les données. Toutefois ils sont bidirectionnels, à l'inverse des `Stream` qui sont typés par leur sens (`InputStream`, `OutputStream`), et ainsi ils reflètent mieux la réalité du système d'exploitation. On ne peut lire et écrire dans un `Channel` qu'au travers d'un `Buffer`.

De manière non intuitive, aux opérations de lecture du `Channel` (`read`) correspond un `put` du `Buffer`, et réciproquement aux opérations d'écriture (`write`) correspond un `get` du `Buffer`. Des opérations clés du *buffer* permettent de basculer entre les modes écriture et lecture :

- `clear()` : prépare le put de *buffer* (`read` du channel associé) – voir aussi `compact()`.
- `flip()` : prépare le get du *buffer* (`write` du channel).
- `rewind()` : permet de relire à nouveau le *buffer*.

Exemples de lecture et d'écriture dans un fichier avec des Channel

L'exemple suivant permet de copier un fichier de manière performante. On commence par ouvrir un `Channel` sur un fichier.

```
FileInputStream fin = new FileInputStream( "readandshow.txt" );
FileChannel fc = fin.getChannel();
```

On alloue ensuite un buffer, indépendamment de la taille du contenu lu.

```
ByteBuffer buffer = ByteBuffer.allocateDirect( 2048);
```

Et on boucle sur la recopie des données par lecture et écriture.

```
int r ;
while (true) {
buffer.clear();
int r = fcin.read( buffer );
  if (r==-1) {
break; }
buffer.flip();
fcout.write( buffer );
  }
```

On remarquera l'utilisation du buffer à double sens : les méthodes `clear` et `flip` permettent de repositionner le curseur de lecture ou écriture au bon endroit.

Copyright © 2015 Dunod.

3.5.3 Sockets avec NIO

Communication par sockets asynchrone : les sélecteurs

Les entrées-sorties asynchrones sont une méthode pour lire et écrire des données sans bloquer l'application en attente. D'habitude, lorsque le code effectue une lecture avec `read()`, l'exécution est bloquée jusqu'à ce que la donnée soit disponible. De même pour l'écriture avec `write()`.

Les appels d'entrée-sortie asynchrones au contraire ne sont pas bloquants. C'est tout comme lorsque l'on vous donne un ticket à la poissonnerie pour attendre votre tour (figure 3.7), vous devez vous enregistrer pour votre intérêt pour un événement d'entrée-sortie particulier (arrivée de données, connexion de socket, etc.) et le système vous avertit lorsque cela survient.

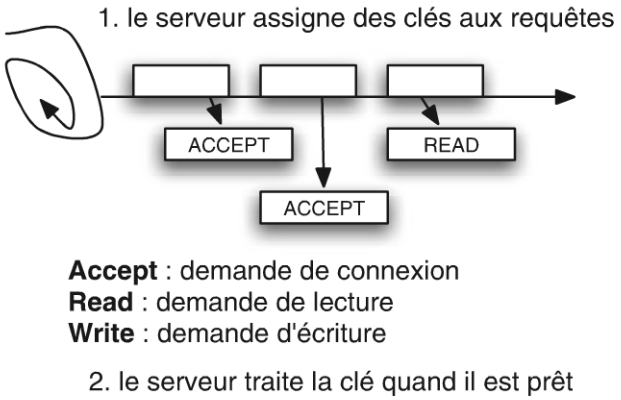


Figure 3.7 — Attribution de clés de sélection aux requêtes

Un des grands avantages est que cela vous permet de gérer un grand nombre de requêtes simultanées. À l'inverse, les programmes synchrones doivent souvent utiliser les techniques de boucle d'attente (*polling*) pour les requêtes, ainsi que la création de threads en grand nombre pour chaque connexion.

Les sélecteurs

L'objet central d'une entrée-sortie est appelé le **sélecteur** (`Selector`). C'est lui qui vous indique votre place dans la file d'attente, il assigne la clé à la requête du client :

```
Selector selector= Selector.open() ;
```

Dans la section suivante, on indiquera quel est le sujet d'intérêt en envoyant la méthode `register()` au sélecteur. On passe toujours le sélecteur en argument de `register()`.

Ouverture d'un canal de socket pour le serveur (ServerSocketChannel)

Afin de recevoir des connexions, il faut un mécanisme serveur appelé `ServerSocketChannel`. En fait, chaque port d'écoute doit être géré par un canal serveur :

Copyright © 2015 Dunod.
© Dunod – Toute reproduction non autorisée est un délit.

```
ServerSocketChannel ssc = ServerSocketChannel.open() ;
ssc.configureBlocking(false) ; // on précise le mode asynchrone
```

On crée ensuite la socket associée en l'attachant à un port :

```
int port = 6789 ;
ServerSocket ss=ssc.socket() ;
InetSocketAddress address = new InetSocketAddress(port) ;
ss.bind(address);
```

Les clés de sélection

On enregistre alors l'intérêt de la socket serveur pour l'attente de connexion (`ACCEPT`) :

```
selector = Selector.open();
SelectionKey serverKey=serverSocketChannel.register(selector,
SelectionKey.OP_ACCEPT);
```

`OP_ACCEPT` est le seul événement possible pour les sockets de type serveur, alors que les sockets classiques utiliseront des événements `OP_READ` et `OP_WRITE`.

On notera aussi le ticket d'enregistrement donné par le sélecteur : la clé de sélection. Celle-ci permettra plus tard de se désenregistrer du sélecteur.

La boucle d'attente de connexion

Le serveur va à présent boucler en attente de connexions.

```
while(serveurActif) {
  int count = selector.select();
  // Pas de clients
  if (count == 0) {continue; }
  Set keySet = selector.selectedKeys();
  Iterator itor = keySet.iterator();
  // Itération sur les clients
  while (itor.hasNext()) {
     SelectionKey selectionKey = (SelectionKey) itor.next();
  // Traitement de la demande...
}
```

Filtrage des clés

Lorsque l'on parcourt les clés du sélecteur, il faut vérifier que la clé est bien de type `OP_ACCEPT` :

```
if (selectionKey.isAcceptable()) {
// Traitement de la connexion
}
```

Copyright © 2015 Dunod.

Traitement de la connexion

Comme on est sûr de l'événement de la clé, on peut accepter la connexion associée :

```
ServerSocketChannel ssc = (ServerSocketChannel)key.channel();
SocketChannel sc = ssc.accept();
```

Il existe une symétrie entre les `ServerSocket/Socket` et les `ServerSocketChannel/SocketChannel`.

L'étape suivante est de configurer cette nouvelle socket comme non bloquante, et de l'enregistrer pour les événements de lecture (`OP_READ`).

```
sc.configureBlocking( false );
SelectionKey newKey = sc.register( selector, SelectionKey.OP_READ );
```

Suppression de la clé traitée

Lorsque la clé de sélection a été traitée, elle doit être supprimée en fin de corps de boucle :

```
itor.remove() ;
```

Traitement d'une requête

À l'intérieur de la boucle principale si la clé n'est pas de type `OP_ACCEPT`, elle peut être de type `OP_READ` :

```
else if (selectionKey.isReadable()) {
// Lecture de la donnée
  SocketChannel sc = (SocketChannel)key.channel();
  ByteBuffer sharedBuffer = ByteBuffer.allocateDirect(BUFFER_SIZE);
  sharedBuffer.clear();
  int bytes = -1;
  while ( (bytes = sc.read(sharedBuffer)) > 0) {
    System.out.println("Lecture...");
    sharedBuffer.flip();
// écho du code lu dans la socket en écriture
    while (sharedBuffer.hasRemaining()) {
      System.out.println("Writing...");
      socketChannel.write(sharedBuffer);
    }
    sharedBuffer.clear();}
selectionKey.cancel();
socketChannel.close();
}
```

Copyright © 2015 Dunod.
© Dunod – Toute reproduction non autorisée est un délit.

Code du client

Le client de son côté peut soit utiliser une socket classique, soit utiliser une socket NIO. Cela n'a pas d'intérêt d'utiliser un sélecteur du côté du client. Voici toutefois en rappel un petit programme de test du serveur :

```
Socket socket = new Socket("localhost", 6789);
InputStream is = socket.getInputStream();
OutputStream os = socket.getOutputStream();
Writer writer = new OutputStreamWriter(os, "US-ASCII");
PrintWriter out = new PrintWriter(writer, true);
out.println("Hello, World");
BufferedReader in = new BufferedReader
  (new InputStreamReader(is, "US-ASCII"));
String line;
while ((line = in.readLine()) != null) {
  System.out.println(i + ": " + line);
  }
socket.close();
```

On remarquera toutefois la spécification du jeu d'encodage des caractères « US-ASCII ». Le lecteur intéressé par les nouveaux jeux de caractères de `java.nio.*` pourra approfondir par lui-même.

3.6 NOUVELLE SOCKET ASYNCHRONE EN JAVA 7 : POUR PLUS DE PERFORMANCE

3.6.1 Intérêt des sockets asynchrones

Java 7 a introduit une nouvelle socket asynchrone. L'intérêt des sockets asynchrones est de gagner de la performance et de la montée en charge dans les serveurs en utilisant des primitives non bloquantes.

C'est un canal asynchrone pour les sockets réseau connectées.

Ce type de canal permet l'utilisation de threads simultanés multiples.

Dans le cas des sockets synchrones classiques, on a plusieurs points d'appels bloquants :

- `ServerSocketChannel.accept()` : attente de connexion.
- `SocketChannel.read(ByteBuffer)` : attente de lecture.
- `SocketChannel.write(ByteBuffer)` : attente de fin de l'écriture.

Toutes ces attentes minent la performance. Les sockets asynchrones permettent de multiplexer les activités pendant ces attentes.

La mise en œuvre est très proche de la mise en œuvre classique de sockets, à la différence que la connexion sur le `accept` renvoie un `Future` et non directement une socket. Même chose pour la lecture et l'écriture.

Copyright © 2015 Dunod.

Dans l'exemple qui suit le client utilise un `AsynchronousSocketChannel` et le serveur un `AsynchronousServerSocketChannel`.

`AsynchronousSocketChannel` est une classe abstraite qui implémente `NetworkChannel` et `AsynchronousByteChannel`.

3.6.2 Programme du serveur

Le programme commence par l'ouverture du canal, comme en synchrone.

```
AsynchronousServerSocketChannel serverChannel =
AsynchronousServerSocketChannel.open();
```

Puis, il lie le canal à une adresse internet, ici sur le port 3883 de l'ordinateur local :

```
InetSocketAddress adresseServeur = new InetSocketAddress("localhost", 3883);
                 serverChannel.bind(adresseServeur);
```

Le serveur est alors en écoute de connexions :

```
Future<AsynchronousSocketChannel> acceptResult = serverChannel.accept();
AsynchronousSocketChannel clientChannel =  acceptResult.get();
```

On notera que le programme attend ici la connexion du client. Le `accept` renvoie un `Future`.

Il peut alors recevoir des messages :

```
ByteBuffer buffer = ByteBuffer.allocate(128);
Future result = clientChannel.read(buffer);
```

Et afficher le résultat obtenu :

```
String message = new String(buffer.array()).trim();
System.out.println(message);
```

On peut finalement fermer la socket :

```
                 clientChannel.close();
                 serverChannel.close();
```

Voici le code complet du programme du serveur :

```
public class Serveur {
       private boolean estActif =true;
       public static void main(String[] args) throws IOException,
InterruptedException, ExecutionException {
              new Serveur().go();
```

Copyright © 2015 Dunod.
© Dunod – Toute reproduction non autorisée est un délit.

```
        }

        private void go() throws IOException, InterruptedException,
ExecutionException {
                AsynchronousServerSocketChannel serverChannel =
AsynchronousServerSocketChannel.open();
                InetSocketAddress adresseServeur = new
InetSocketAddress("localhost", 3883);
                serverChannel.bind(adresseServeur);
                System.out.println("Serveur en écoute sur le port: " +
adresseServeur.getPort());
                System.out.println("Attente de connexion de clients... ");

                Future<AsynchronousSocketChannel> acceptResult =
serverChannel.accept();
                AsynchronousSocketChannel clientChannel =  acceptResult.get();
                System.out.println("Messages du client : ");

                if ((clientChannel != null) && (clientChannel.isOpen())) {
                        while (estActif) {
                                ByteBuffer buffer = ByteBuffer.allocate(128);
                                Future<Integer> result =
clientChannel.read(buffer);
                                while (! result.isDone()) {
                                        //ne fait rien
                                }
                                buffer.flip();
                                String message = new
String(buffer.array()).trim();
                                System.out.println(message);
                                if( message.equals("Au revoir")) estActif= false;
                               buffer.clear();
                        }
                        }
                clientChannel.close();
                serverChannel.close();
                }
        }
```

3.6.3 Programme du client

Le programme du client utilise lui aussi les primitives asynchrones.

Ouverture du canal :

```
AsynchronousSocketChannel client = AsynchronousSocketChannel.open();
```

Connexion au serveur en utilisant son adresse et son port, ici 3883, avec obtention là encore d'un `Future` :

```
InetSocketAddress adresseServeur = new InetSocketAddress("localhost",3883);
Future<Void> future= client.connect(adresseServeur);
future.get();
```

Copyright © 2015 Dunod.

Noter que la méthode `get` renvoie `null` lors d'une connexion réussie.

Le client peut alors envoyer des messages au serveur :

```
for (String message: messages) {
       ByteBuffer buffer = ByteBuffer.wrap(message.getBytes());
       Future<Integer> result = client.write(buffer);
```

Puis ferme la socket :

```
client.close();
```

Voici le programme complet du client :

```
public class Client {
       public static void main(String[] args) throws IOException,
InterruptedException, ExecutionException {
       //ouverture du canal

AsynchronousSocketChannel client = AsynchronousSocketChannel.open();
       //connexion au serveur
InetSocketAddress adresseServeur = new InetSocketAddress("localhost",3883);
Future<Void> future= client.connect(adresseServeur);
future.get();
System.out.println("Démarrage du client : " + client.isOpen());
//envoi d'un message au serveur
String[] messages = new String[] {"Le temps passe.", "Et à présent ?", "Au
revoir"};
for (String message: messages) {
       ByteBuffer buffer = ByteBuffer.wrap(message.getBytes());
       Future<Integer> result = client.write(buffer);
        while (! result.isDone()) {
               System.out.println("... ");
        }
        System.out.println(message);
        buffer.clear();
        Thread.sleep(2000);
        }
//ferme le canal
client.close();
       }
}
```

Les sockets asynchrones permettent de gagner de la performance avec une mise en œuvre très compacte et proche des sockets synchrones. Si l'on recherche de la performance, il existe d'autres implémentations de communications asynchrones en Java, on pensera en particulier à Netty (http://netty.io/) et Grizzly (https://grizzly.java.net/).

Copyright © 2015 Dunod.
© Dunod – Toute reproduction non autorisée est un délit.

En résumé

Java permet une grande portabilité entre différentes plates-formes de la programmation des communications en réseaux par sockets. Il supporte la communication par TCP, UDP et multicast. L'utilisation de threads permet d'optimiser la montée en charge des serveurs. La nouvelle API `java.nio.*` augmente les performances des entrées-sorties en Java non seulement sur les fichiers mais aussi sur les serveurs de connexions avec les mécanismes de sélecteurs.

Copyright © 2015 Dunod.

4

Interfaces et pliage des arguments

Objectif

Le but de ce chapitre est de montrer comment les interfaces jouent un rôle important pour communiquer entre applications réparties. Les interfaces vont établir un contrat entre un client et un serveur, et vont également permettre de vérifier la syntaxe et de gérer un premier niveau de sécurité. Certaines notions abordées rapidement dans ce chapitre seront développées plus en détail dans les chapitres ultérieurs plus spécifiques, mais elles donnent dès maintenant une vue d'ensemble des différentes technologies.

4.1 L'INTERFACE, UN CONTRAT ENTRE CLIENT ET SERVEUR

4.1.1 Insuffisance des sockets

Le premier mode de communication entre les applications utilisait les sockets. Les sockets utilisent des flots de caractères continus (*streams*) sans structuration. Elles nécessitent donc d'établir un protocole syntaxique entre le client et le serveur, en énumérant les commandes possibles. Prenons l'exemple suivant :

Copyright © 2015 Dunod.
© Dunod – Toute reproduction non autorisée est un délit.

```
A envoie prêt à B
B analyse le flux et extrait la commande prêt de A
    et envoie données 1 2 à A
A analyse le flux et extrait la commande et ses arguments
Si la commande est données , alors A stocke les données
Si la commande est fin , le processus se termine...
```

Chaque interlocuteur doit donc spécifier une syntaxe de commandes, analyser le flux en permanence, et implémenter un gigantesque aiguillage prenant en compte toutes les commandes possibles. L'ajout d'une commande implique de réécrire tout celui-ci. Ceci est vrai également de certaines implémentations de XML-RPC.

Les interfaces permettent aux applications de formaliser ce dialogue, et rejettent sur le compilateur l'écriture de l'aiguillage. Avec une interface, le compilateur peut également vérifier le type des données. En effet, le codage des arguments est toujours un souci pour des structures de données complexes, en particulier lorsque l'on communique entre machines hétérogènes. Voir la section 4.2 sur le pliage des arguments (*marshalling*).

Un autre défi à prendre en compte est de pouvoir traiter les exceptions dans un dialogue aussi complexe : non seulement dérouter le fil de conversation, mais aussi transmettre la nature de l'exception. Les interfaces vont permettre également de transmettre et de partager l'information sur les exceptions du système (voir paragraphe suivant).

Les interfaces ne sont pas des contrats

Les interfaces vont décrire l'aspect des appels distants, et non la manière dont l'appel est envoyé : elles décrivent le nombre d'arguments et leur type, la liste des méthodes, mais elles ne disent rien du temps d'attente de la réponse, ou de l'envoi synchrone ou asynchrone, des threads ou de l'état d'activité du serveur. Ceci génère une forte ambiguïté dans les applications réparties.

Bertrand Meyer [Meyer1991] a introduit un niveau supplémentaire de spécification des interfaces entre appelant et appelé : un **contrat**. Un contrat est une liste d'assertions ou d'expressions booléennes qui décrivent l'état du système, ordinairement divisées en pré-conditions et post-conditions. Les pré-conditions doivent être satisfaites avant appel, les post-conditions sont des promesses à remplir par l'appelé.

Les interfaces ne sont pas des contrats. Elles spécifient un couplage faible entre le client et le serveur, limité seulement à la vérification de types et de signature de méthode. Client et serveur doivent partager les mêmes types de données, et cela aura un impact important sur le déploiement.

4.1.2 Spécification des interfaces

On spécifie les interfaces d'un système ordinairement en les regroupant par modules (packages ou espaces de nommage), et en détaillant la signature des méthodes.

Copyright © 2015 Dunod.

Voici un exemple simple pour une application CORBA (voir chapitre 6) définissant un additionneur à deux arguments dans un module `Calculateur` :

```
module Calculateur {
    interface Additionneur {
        long ajout(in long x, in long y) ;
    }
}
```

Dans cet exemple, l'interface n'expose qu'une seule méthode, `ajout`, comportant deux arguments, `x` et `y`, et renvoyant un résultat de type `long`. On remarquera le style inspiré du langage Ada spécifiant si l'argument est en entrée ou en sortie (`in, out`).

La syntaxe des interfaces varie selon le mode de communication choisi :

- En **RMI** (voir chapitre 5), les interfaces sont des interfaces Java standard. Ceci facilite le processus de développement car on utilise le même environnement de développement et le même compilateur pour les interfaces réparties et le code. Les interfaces RMI utilisent de plus les exceptions Java standard enrichies de quelques exceptions propres à RMI (`RemoteException`).
- **CORBA** (voir chapitre 6) utilise un langage spécifique pour décrire les interfaces : IDL (*Interface Definition Language*). Ceci est nécessaire dans la mesure où CORBA fédère des langages hétérogènes : C, Lisp, Python, Java, etc. Ceci signifie que les types primitifs (`long`, `int`, `char`) doivent être traduits du langage IDL vers le langage cible. CORBA rajoute un peu de sémantique dans l'interface :
  - Les exceptions, tout comme en RMI. Mais certains langages cibles n'utilisent pas d'exceptions, comme en C, à moins d'utiliser une bibliothèque spécifique.
  - Le sens de passage des arguments : en entrée (`in`), en sortie (`out`) ou les deux (`inout`). Les arguments `inout` sont modifiés par le programme.
  - Un peu de sémantique sur l'appel avec l'instruction `oneway`. L'idée est de ne pas attendre de réponse de l'appelé. La spécification précise que `oneway` ne garantit pas l'acheminement de l'appel. Mais ceci dépend fortement de l'implémentation CORBA choisie.
- En **SOAP** (voir chapitre 9 sur les services web), la description de l'interface est beaucoup plus compliquée. Elle est donc le plus souvent générée par un outil et non écrite manuellement. La modification pour l'adaptation d'un outil à l'autre peut être plus compliquée, et souvent on peut rajouter une couche d'incompatibilité entre outils de développement (typiquement entre Java et .Net) même si les interfaces elles-mêmes sont compatibles. L'interface SOAP utilise un langage de description à la XML : WSDL. Les interfaces SOAP spécifient non seulement la signature des opérations, mais également le protocole de communication sous-jacent et également des éléments de déploiement tels que l'URL du service web et le numéro de port.

Copyright © 2015 Dunod.
© Dunod – Toute reproduction non autorisée est un délit.

- En **JMS** (voir chapitre 8), on communique par messages : on n'utilise pas d'interfaces, mais on retrouve le même souci de codage et d'extraction de l'information qu'avec les sockets.

Exceptions

Une partie souvent négligée de la spécification des interfaces est la définition des exceptions. Dans un outil UML ancien, IBM rajoutait un compartiment supplémentaire à la classe permettant de préciser les exceptions levées par la classe (figure 4.1).

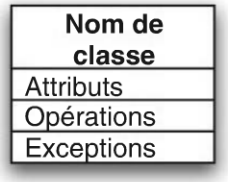


Figure 4.1 — Description des exceptions dans une classe

C'est en général un bon choix de conception de définir minutieusement les exceptions dans un système réparti. En effet surtout lorsque l'on utilise des langages et des environnements de développement différents, il est rare de pouvoir utiliser des déverminneurs distants, et les exceptions vont être un pivot central du déverminage et de la compréhension du déroulement de la machine distante, notamment pour les applications enfouies.

4.1.3 Partager des types de données entre les applications

En Java, on fait en général l'approximation de confondre les classes et les types de données, aux types primitifs près (`long`, `char`, `int`...). Un type de données servira au compilateur. Dans les applications réparties, il faut prendre davantage de précautions : un type est en effet une version de classe donnée dans une machine virtuelle donnée. Ceci a les conséquences suivantes :

- Une classe dans deux versions différentes ne sera pas considérée comme la même classe, même si elle satisfait les mêmes interfaces. Ceci impose le plus souvent de pouvoir harmoniser les versions de tous les programmes dans toutes les applications réparties.
- Une référence à un objet distant n'a pas de type connu dans la machine locale : l'utilisation d'une référence à un objet distant nécessite d'effectuer un « cast » dans une interface connue ou d'utiliser comme en CORBA un objet `Helper` pour extraire le type correct de la référence.

Copyright © 2015 Dunod.

C'est ici que l'on voit une des limitations des langages à typage fort tels que Java, puisque la compilation se limite à une machine virtuelle locale et ne peut prétendre à faire une vérification de types entre deux espaces de mémoire différents. La programmation répartie est de ce fait plus facile avec des langages dynamiques tels que Lisp, Python ou Smalltalk.

Exemples

- En RMI :

```
Hello objetDistant=(Hello) contexte.lookup("rmi://localhost:1099/hello");
```

- En CORBA :

```
Hello objetDistant = HelloHelper.narrow(namingContext.resolve_str(''hello''));
```

Les interfaces permettent de compiler une classe cliente sans disposer du code de la classe serveur : elles permettent d'assurer le typage statique à la compilation. Lors du déploiement, les interfaces doivent être présentes à la fois sur le client et sur le serveur.

4.2 PLIAGE DES ARGUMENTS (MARSHALLING)

4.2.1 Représentation des données

Le transfert des données et la modification des représentations internes aux machines impliquent des mécanismes insoupçonnés.

Les données transitent toujours par le réseau qui ne connaît pas de structure de données, et travaille avec des flux de caractères ou d'octets uniformes. La structure des objets doit être « aplatie » dans un flux d'octets. C'est ce qu'on appelle le pliage des arguments (*marshalling*).

De plus des machines différentes peuvent utiliser des représentations de données internes différentes : selon les processeurs (`big endian, small endian`), la représentation binaire d'un entier peut avoir un bit de poids fort à droite ou à gauche (figure 4.2). L'intergiciel ou le bus logiciel doit assurer ce transcodage.

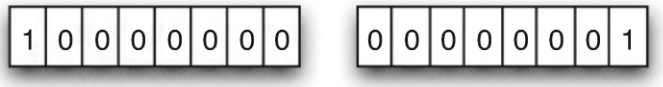


Figure 4.2 — Cet entier est-il le nombre 1 ou 128 ?

Ceci doit être pris en compte également pour les architectures 64 bits et 32 bits, et il faut veiller à ce que le bus de communication soit réellement compatible avec du 64 bits le cas échéant.

Copyright © 2015 Dunod.
© Dunod – Toute reproduction non autorisée est un délit.

L'internationalisation ou le système d'exploitation peuvent affecter la représentation des chaînes de caractères : les lettres accentuées dans les langues d'Europe de l'ouest telles que « é » ou « ö » peuvent avoir un codage ASCII différent dans une configuration française ou allemande. Pour le même langage, les systèmes d'exploitations Windows, Linux et Mac OS ont des codages différents. Un texte accentué au format `.txt` sera généralement mal recodé d'une machine Linux à une machine Windows.

Les autres alphabets ou idéogrammes utilisent des jeux de caractères codés sur 16 bits au lieu de 8 afin de couvrir la palette de leurs alphabets. L'organisme de normalisation ISO a défini des jeux de caractères pour différents groupes de langues, par exemple le code ISO-8859-1 codera les langues de l'Europe occidentale. Java 5 incorpore à présent le codage de caractères (voir les classes `Charset`, `CharsetEncoder` et `CharsetDecoder`) ainsi que des outils d'internationalisation. CORBA spécifie par exemple le type `wstring` (*wide string*) pour ce type de chaînes. Voir également la méthode de lecture de caractères UTF `DataInput.readUTF()`.

La spécification de l'encodage est une des innovations importantes de XML, qui sert de trame à SOAP et aux services web (voir chapitre 9). Cet encodage assure une compatibilité de lecture des chaînes entre machines et systèmes d'exploitation.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
```

En résumé, un bon intergiciel doit pouvoir prendre en compte les conversions de types natifs entre les machines, ainsi que les conversions de codage de caractères. Il doit également prendre en compte la structuration des données.

Structures d'objets

Dans le chapitre 2 Rappels, nous avons vu que les objets venaient le plus souvent par grappes, et que la sérialisation prenait en compte les graphes d'objets. Dans une certaine mesure, le codage des objets en XML permet de conserver une structuration des objets, mais on perd complètement la notion de pointeur qui est préservée par la sérialisation. Ceci aura un impact sur la sérialisation des objets dans les services web au chapitre 9.

4.2.2 Passage des arguments par valeur ou par référence

Lorsque deux objets distants communiquent au travers d'appels de méthodes distants, ils peuvent passer les arguments par valeur ou par référence :

- Lors d'un passage **par valeur**, on va envoyer une copie de l'objet par le réseau.
- Lors d'un passage **par référence**, on va seulement envoyer une référence distante, propre à l'intergiciel.

Copyright © 2015 Dunod.

Passage des arguments par référence – service de nommage

La gestion des références distantes est une des tâches essentielles des différents intergiciels (RMI, CORBA...) : le bus doit préserver l'unicité des références sur le réseau, et parfois aussi gérer le cycle de vie ou d'activation des références. Dans DCOM, l'ancien intergiciel de Microsoft, les références étaient stockées dans le registre du système.

Les références en Java sont stockées dans un silo appelé service de nommage (*naming service*), et parfois dans ce livre « service d'annuaire » (voir chapitre 7). Le service de nommage que vous utilisez quotidiennement est le service des URL (*Uniform Resource Locator*) d'Internet : chaque adresse correspond à un point physique unique sur le globe, par exemple http://www.dunod.com. De plus en plus de services de nommage utilisent les techniques d'URL.

C'est une erreur fréquente de considérer que les références distantes peuvent être traitées comme des pointeurs ordinaires.

On peut l'illustrer par l'histoire traditionnelle chinoise suivante : deux seigneurs de la guerre étaient dans une embarcation lorsque l'épée de l'un deux tomba à l'eau. Il se précipita pour marquer à la craie une croix sur le bateau à l'endroit où l'épée était tombée. De retour au port, il plongea dans l'eau à l'endroit de la marque sur le bateau.



Figure 4.3 — Une référence distante est un élément mouvant

Il en est de même pour les références distantes. Si l'on conserve la référence en cache dans l'application cliente, la référence peut avoir disparu, ou bien certaines informations liées à l'objet peuvent ne pas transiter par le réseau. Le plus souvent, l'objet distant peut avoir été modifié par un autre programme, sans qu'il soit possible de s'en apercevoir.

Passage des arguments par valeur

Les paramètres des méthodes d'une interface distante doivent pouvoir être passés par référence ou par valeur. Les techniques de passage par valeur utilisent en général la sérialisation ou encore le mapping XML (voir chapitre 2) : une copie de l'objet est envoyée par le réseau.

Le passage des arguments sous forme binaire sera généralement plus compact et moins gourmande en bande passante qu'en XML – jusqu'à un facteur 5 –, mais

Copyright © 2015 Dunod.
© Dunod – Toute reproduction non autorisée est un délit.

nécessite une unité de compilation entre les interlocuteurs. En effet, XML s'appuie généralement sur un codage ISO UTF8, qui représente les caractères non plus sur 8 bits, mais sur 16 ou 8 bits. Il existe d'autres techniques de représentation binaire, par exemple celles utilisées par ASN1 en télécommunications.

Le passage des arguments en mode texte peut se révéler parfois utile pour vérifier la compatibilité de deux applications, ou gérer des versions différentes de classes (voir paragraphe 2.1 Rappels sur la gestion de version de la sérialisation).

Sémantique de copie lors du passage par valeur

Lorsque l'on utilise le passage par valeur, une copie de l'objet distant transite par le réseau. Cette copie reflète naturellement l'état instantané de l'objet copié, et reste figée dans cet état. Elle peut donc être désynchronisée de l'objet distant si celui-ci est modifié par la suite.

Prenons l'exemple d'une agence bancaire qui souhaite se connecter à un serveur central pour saisir des mouvements sur un compte. Si l'on copie l'état du compte en agence, celui-ci fera apparaître le solde au moment de la copie. Si l'agence effectue une saisie sur la copie, celle-ci ne sera pas prise en compte par le serveur.

Si le serveur par contre prend en compte des modifications en provenance d'autres agences, celles-ci ne seront pas connues de la copie dans l'agence locale (figure 4.4).

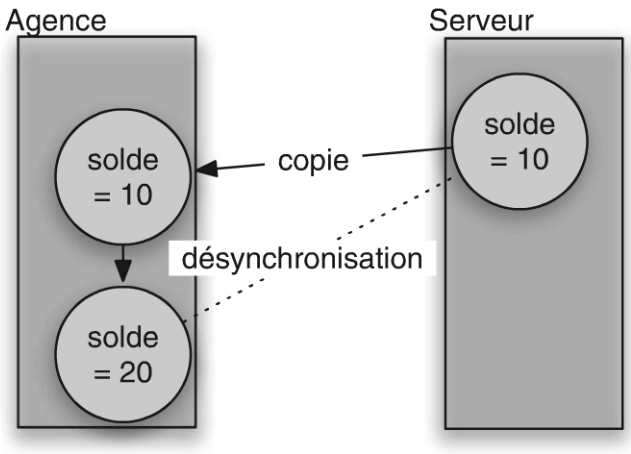


Figure 4.4 — Désynchronisation des objets lors d'une copie

Ceci est en particulier vrai pour les services web qui utilisent un passage par valeur (voir chapitre 9).

Comparaison du passage par valeur et du passage par référence

En dehors des problèmes de copie d'objet, on peut également comparer le passage par valeur et par référence selon d'autres critères :

- **Bande passante** : lors de la copie d'un objet, on peut également copier le graphe des objets liés à l'objet et il faut surveiller la taille totale. On peut

Copyright © 2015 Dunod.

penser à élaguer le graphe, comme vu au paragraphe 2.1. Naturellement lors du passage par référence, seule la référence est passée. Par ailleurs si l'on utilise la sérialisation ou tout format binaire, le transfert sera plus compact que par chaînes de caractères.

- **Appels du serveur** : si l'on travaille avec une copie locale, on évite de nombreux allers et retours avec le serveur, et mettre la copie en cache peut permettre de gagner en performance. Typiquement, si l'on vient de faire une simulation de calcul à distance on peut obtenir le nom de la simulation, le résultat, la durée totale. Si l'on utilise une référence à la simulation, on fera un appel distant pour obtenir chacune des informations (figure 4.5) :

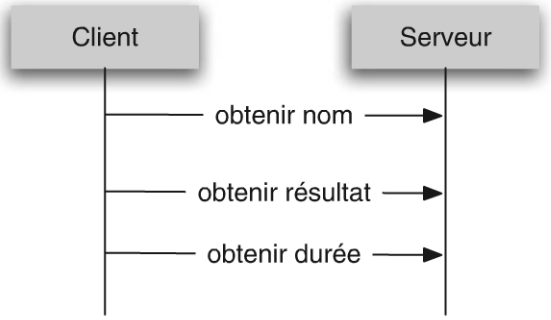


Figure 4.5 — Appels vers le serveur pour chaque attribut

Une idée simple est alors de grouper les informations en paquet comme les techniques utilisées dans les *views* des bases de données. Selon les cas d'utilisation, on peut utiliser plusieurs conteneurs, selon le niveau de détail souhaité sur l'objet : par exemple résumé de la simulation, détails de la simulation, etc.

Étude de cas : bibliothèque de CD-Roms

On souhaite parcourir une bibliothèque de CD à distance. La bibliothèque sera typiquement un objet volumineux, auquel on va accéder par une référence distante unique. Par contre il faut envisager les cas d'utilisation pour choisir le mode de passage des CD :

- Si l'on souhaite seulement faire l'état des stocks de CD, et que l'information est en seulement en consultation, on pourra utiliser un passage par valeur. On devra toutefois s'assurer que la classe CD est présente sur le client. La classe CD ne comporte aucun comportement de modification en local (figure 4.6).
- Si à présent on doit saisir de nouveaux prix sur la bibliothèque, on doit s'assurer qu'aucune application cliente n'a gardé en cache de vieux prix, et l'on utilise alors des références distantes. De même si l'information du CD est volumineuse et contient des extraits de musique, on évitera le passage par valeur.

Copyright © 2015 Dunod.
© Dunod – Toute reproduction non autorisée est un délit.

Lorsque l'on conçoit une application répartie, on doit vérifier les points suivants :
– Tous les arguments des méthodes doivent être des références distantes ou des objets sérialisables.
– Préférer les références pour de gros objets.
– Éviter les graphes d'objets sérialisables avec des dépendances incontrôlées.
– Penser à utiliser un cache pour éviter les allers-retours, mais prendre garde à la désynchronisation possible.
– Utiliser des références lorsque plusieurs personnes partagent simultanément la même information.

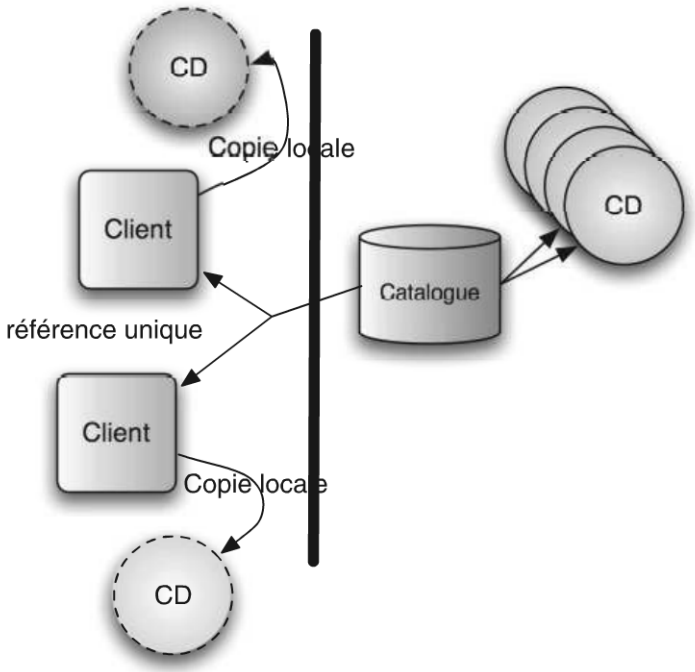


Figure 4.6 — Consultation d'une bibliothèque de CD-Roms

4.3 INTERFACE, IMPLÉMENTATION ET DÉPLOIEMENT

4.3.1 Une implémentation pour plusieurs interfaces

Il est souvent utile de pouvoir utiliser un objet sous plusieurs interfaces : on prend l'analogie d'un diamant qui rayonne sous plusieurs facettes. Le terme de facette (*facet*) est utilisé par les composants CORBA (CCM, *CORBA Component Model*) pour décrire les interfaces d'un même objet.

Une interface est seulement une projection des méthodes de l'objet, et permet de présenter plusieurs masques selon le client distant. Ceci peut être utile pour une architecture sécurisée ou pour partitionner le développement.

Copyright © 2015 Dunod.

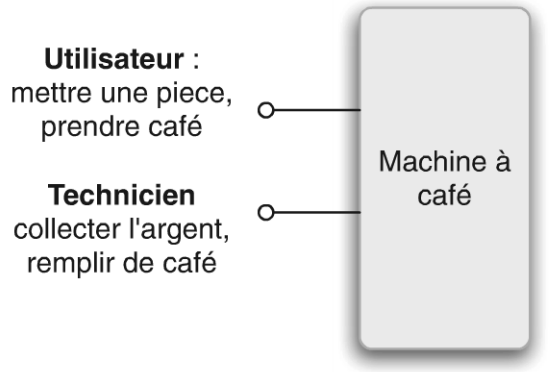


Figure 4.7 — Plusieurs interfaces pour la machine à café

La figure 4.7 illustre l'utilisation de plusieurs interfaces pour la même implémentation : une machine à café. L'utilisateur peut seulement effectuer deux types d'opérations, mettre une pièce et prendre un café. Le technicien aura lui le droit de collecter l'argent du distributeur et de remettre du café. Il est utile par sécurité de protéger l'accès au collecteur de pièces.

Pour partitionner un système en différentes interfaces, on prendra également l'exemple d'un équipement de télécommunications, qui offrira différentes interfaces pour la taxation des communications, pour la configuration du réseau, ou pour l'utilisation opérationnelle en mode de communication (*billing*, *operations*, *provisioning*) (figure 4.8).

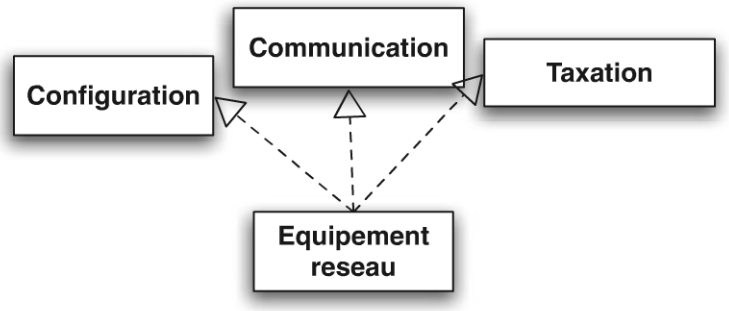


Figure 4.8 — Plusieurs interfaces distantes pour le même équipement.

4.3.2 Plusieurs implémentations de la même interface

Il est aussi possible de fournir plusieurs implémentations de la même interface. Ceci peut être utile pour changer de version à chaud si l'application conserve la même interface, ou pour harmoniser par une façade des systèmes différents. Nous verrons qu'avec RMI la souche utilisée par le client pour l'acheminement des messages au serveur dépend de l'implémentation, et donc qu'il n'est pas possible de modifier

Copyright © 2015 Dunod.
© Dunod – Toute reproduction non autorisée est un délit.

l'implémentation sans modifier le client. En CORBA ou en SOAP par contre ce n'est pas le cas.

En résumé

Les interfaces permettent de structurer les applications réparties et d'augmenter la qualité du logiciel en assurant une compatibilité syntaxique entre les clients et les serveurs. Elles servent aussi à la compilation. Les interfaces toutefois ne définissent aucune sémantique en dehors de la syntaxe.

Le pliage des arguments (*marshalling*) est un problème important et spécifique à chaque technologie utilisée.

Le passage des arguments par valeur ou par référence a un impact fort sur l'architecture du système.

Copyright © 2015 Dunod.

5

RMI

Objectif

Le but de ce chapitre set de présenter les techniques standard de programmation par appel de procédures distantes en Java, RMI (*Remote Method Invocation*), et de montrer également les modifications apportées par Java 5, 6 et 7.

5.1 COMMENT UTILISER RMI

5.1.1 Positionnement de RMI

RMI (*Remote Method Invocation*) est une API de communication entre objets distants de Java présente depuis longtemps dans la version standard de Java du JRE (*Java Runtime Environment*). Contrairement à CORBA (voir chapitre 6), RMI ne permet que la communication entre programmes Java à l'exclusion de tout autre langage ; par contre, il assure des échanges entre systèmes d'exploitation différents possédant une machine virtuelle Java. RMI s'appuie sur un protocole de type RPC (*Remote Procedure Call*). Il peut être utilisé également pour la communication entre deux machines virtuelles Java sur la même machine physique.

RMI est souvent vu comme une version légère de CORBA, car il présente le même modèle de programmation, mais le fait d'utiliser uniquement Java simplifie le processus de développement, et de plus RMI ne possède pas les couches de services complexes qui ont été spécifiées avec CORBA : découverte des interfaces, persistance, etc. RMI a toutefois rajouté dans les versions récentes des mécanismes d'activation de serveurs et de persistance. Il possède enfin ses propres mécanismes de sécurité.

Copyright © 2015 Dunod.
© Dunod – Toute reproduction non autorisée est un délit.

Les appels de méthodes distantes en RMI nécessitent toutefois une réécriture de certaines interfaces, et il est donc nécessaire d'avoir la maîtrise du code pour déployer une application de manière répartie en RMI.

Vue d'ensemble de la mise en œuvre

L'écriture d'un programme en RMI présente peu de différence par rapport à l'écriture d'un programme normal. Il nécessite toutefois une conception préliminaire pour déterminer où se situent les frontières entre le programme distant et le programme local, et quels sont les objets serveurs de part et d'autre.

RMI définit un mode client serveur, où les serveurs seront des objets et les clients d'autres objets, mais rien n'impose de mettre tous les objets serveurs dans la même machine virtuelle (JVM) ou la même machine physique (figure 5.1).

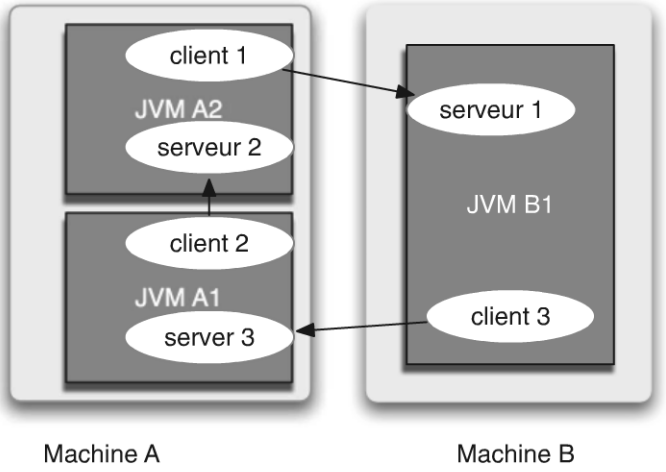


Figure 5.1 — Les objets serveurs et clients peuvent résider sur la même machine
Les machines virtuelles communiquent par RMI

Par contre tous les échanges entre objets clients et serveurs doivent être analysés pour déterminer les méthodes qui sont échangées, et les arguments qui transitent entre les deux. Un diagramme de séquence UML peut être un bon support pour analyser le flux entre les objets.

5.1.2 Définition des objets distants

À titre d'exemple, considérons un petit diagramme de séquence entre une fenêtre graphique et un serveur (figure 5.2).

Le diagramme de séquence de la figure 5.2 se lit ainsi : lors de la mise à jour d'une interface graphique, le modèle se connecte à un entrepôt pour assurer le stockage, et l'entrepôt rappelle le modèle pour obtenir une estampille de stockage.

Dans ce diagramme de séquence, il faut déterminer les objets résidant sur chacune des machines. Il est parfois nécessaire de refaire une analyse de couplage des objets pour minimiser les échanges, de rajouter des objets intermédiaires.

Copyright © 2015 Dunod.

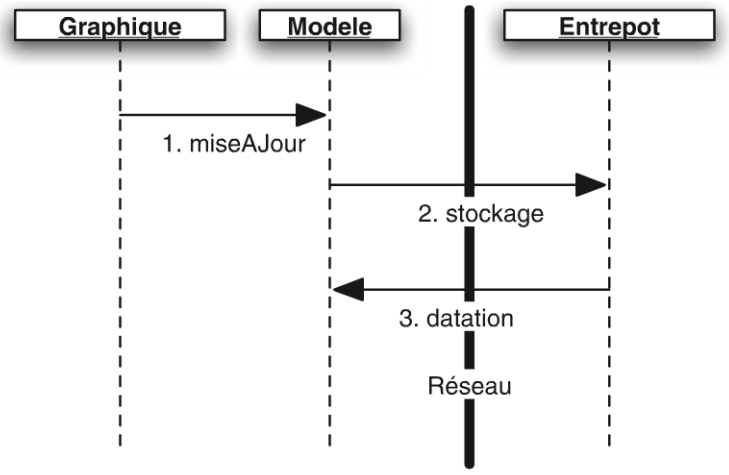


Figure 5.2 — Diagramme de séquence d'une application en réseau

Dans notre diagramme, les objets `Graphique` et `Modele` seront sur la même machine, alors que `Serveur` sera sur une autre machine. La méthode `miseAJour` sera une méthode Java usuelle, alors que les méthodes `stockage` et `datation` seront des méthodes distantes qui vont transiter par le réseau.

Les objets distants seront alors les objets implémentant des méthodes distantes : ici `Serveur` et `Modele`. Si `Serveur` n'envoyait pas de méthode en retour (*callback*), `Modele` pourrait rester un objet Java normal.

5.2 MISE EN ŒUVRE DE RMI

La mise en œuvre de RMI procède par étapes : certaines sont des étapes d'écriture de code, d'autres des étapes de lancement d'annuaire et de génération de code. On va ainsi procéder aux opérations suivantes :

- Définition des interfaces
- Implémentation de l'objet distant
- Génération du code des souches
- Écriture du code de mise à disposition (serveur)
- Écriture du code d'exploitation (client)
- Lancement de l'annuaire
- Lancement du serveur puis du client

5.2.1 Définition des interfaces distantes

La première étape pour la conception d'une application est de définir les interfaces. Ce sera aussi vrai de la démarche CORBA (voir chapitre 6). Par contre avec SOAP (chapitre 9), les interfaces seront dérivées du code. Le même schéma permet

Copyright © 2015 Dunod.
© Dunod – Toute reproduction non autorisée est un délit.

d'identifier les interfaces distantes. L'objet `Modele` implémente les méthodes `miseAJour` et `datation`. Toutefois on ne mettra dans l'interface distante que les méthodes qui sont accédées à travers réseau à distance, soit `datation`.

L'interface qui est exposée est une projection de toutes les méthodes de l'objet, qui peut masquer des méthodes locales.

En RMI, l'interface est une interface Java simple, mais héritant de l'interface `java.rmi.Remote` et lançant obligatoirement l'exception `java.rmi.RemoteException` :

```
public interface ModeleInterface extends Remote {
    public java.util.Date datation() throws RemoteException;
}
```

Dans la définition d'une interface, on doit s'assurer que tous les arguments sont bien sérialisables. C'est ici le cas de `Date` (ou de tout autre type natif comme les nombres, etc.).

Toutes les méthodes accessibles doivent être publiques. C'est une question de bon sens, mais qui doit être prise en compte pour déployer une application existante.

On remarquera qu'il n'est pas possible d'utiliser une interface préexistante sans lui rajouter les exceptions `RemoteException`. Ceci est un inconvénient majeur pour la réécriture de programmes existants de manière aveugle. Il serait possible d' utiliser des *proxies* (voir paragraphe 2.4) pour encapsuler une application existante sans modification.

5.2.2 Implémentation de l'interface

Implémentation par héritage

Depuis Java 1.4, il est possible d'implémenter un objet distant soit par héritage du mécanisme Java préexistant, soit par mise à disposition de l'objet par un mécanisme de servant : un servant est un objet qui va gérer le cycle de service d'un autre objet.

Par héritage, l'objet doit hériter de `java.rmi.server.UnicastRemoteObject` :

```
public class Modele extends UnicastRemoteObject implements ModeleInterface {
    protected Modele() throws RemoteException {super();}
    public Date datation() throws RemoteException
        {return new GregorianCalendar().getTime();}
    public void miseAJour(){};}
```

Attention, cet héritage implique de définir un constructeur qui lance une exception `RemoteException` comme ceci :

```
protected Modele() throws RemoteException {
    super();}
```

La figure 5.3 montre l'héritage de l'objet distant de `UnicastRemoteObject`. `Modele` implémente l'interface `ModeleInterface` tout comme sa souche `Modele_Stub` qui va être générée.

Copyright © 2015 Dunod.

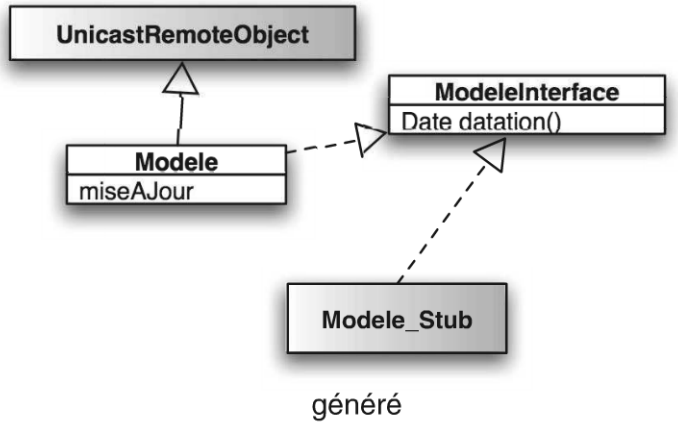


Figure 5.3 — L'objet distant hérite de `UnicastRemoteObject`

Implémentation par délégation et non par héritage

On peut également utiliser le mécanisme de mise en service pour éviter l'héritage. Dans l'exemple suivant la classe `Modele` n'hérite plus de `UnicastRemoteObject`.

```
public class Modele implements ModeleInterface {..}
```

Dans ce cas, inutile de déclarer un constructeur spécifique. Voici alors le code pour la mise à disposition de l'objet distant :

```
Modele modele = new Modele() ;
ModeleInterface stub = UnicastRemoteObject.exportObject(modele,0) ;
```

Le paramètre `0` permet de définir une allocation dynamique du port pour le serveur, mais il peut être également spécifié explicitement, notamment pour l'utilisation de pare-feu.

La figure 5.4 montre la modification d'implémentation de `Modele`.

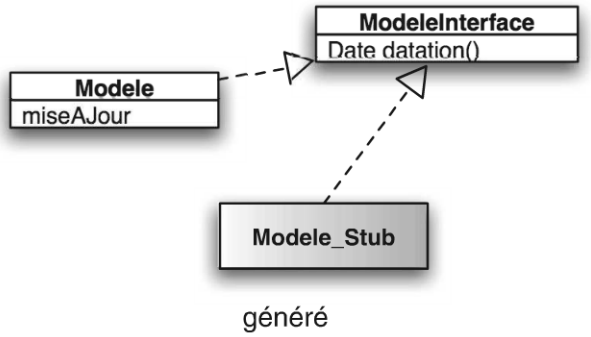


Figure 5.4 — `Modele` n'hérite plus de `UnicastRemoteObject`

Copyright © 2015 Dunod.
© Dunod – Toute reproduction non autorisée est un délit.

5.2.3 La génération de la souche

Le client ne dispose pas de la classe de l'objet serveur. Il doit donc obtenir un relais vers le serveur contenant la référence distante à l'objet serveur : c'est le *stub* (souche ou moignon en français). Le stub implémente un pattern de mandataire (*proxy*) comme décrit dans le paragraphe 2.4 sur les Rappels. Il implémente donc l'interface du serveur, et remplace celui-ci côté client. Il va masquer au client plusieurs aspects complexes :

- la localisation de la référence sur le serveur (transparence de localisation) ;
- l'appel des méthodes à distance ;
- le pliage et le dépliage des arguments (voir chapitre 4 sur les interfaces) ;
- l'unicité de l'objet serveur et la cohérence d'état entre les différents appels sur serveur ;
- le ramasse-miettes des références.

Le stub était jusqu'alors généré par un outil Java appelé `rmic`. Le stub est généré à partir du code compilé de la classe, et non à partir de l'interface comme en CORBA ou avec SOAP. Ceci a un impact important sur le déploiement, car cela interdit au client de générer un stub de connexion sans l'avoir reçu du serveur lors du déploiement. Il existe des mécanismes de téléchargement de *stub* qui ne seront pas présentés ici. Les arguments ont la même convention de nommage que la commande `java` :

```
rmic package.NomdeClasse.
```

`rmic` doit voir la classe sur son *classpath*, et le plus souvent se lance à la racine du projet Java. La souche est une classe Java de nom `package.classe_Stub`. Ainsi la commande :

```
rmic simple.Modele
```

génère la classe `simple.Modele_Stub`.

Il est possible de visualiser le code Java généré avec l'option –`keep` :

```
rmic –keep simple.Modele
```

Il existe d'autres options de `rmic` qui ne seront pas discutées ici.

La génération des souches peut être fastidieuse, car il faut parcourir chacune des classes distantes. Il existe des plugins pour Eclipse, NetBeans ou d'autres outils intégrés qui génèrent ceux-ci automatiquement, et il est possible aussi dans Eclipse de câbler la génération de souches[1].

1. Voir le site http://www.afceurope.com/RMI.html pour l'intégration de rmic dans Eclipse.

Copyright © 2015 Dunod.

5.2.4 Enregistrement de l'objet dans l'annuaire et code serveur

Utilisation de rmiregistry

Mais à présent, comment l'objet client va-t-il trouver la référence au serveur ? Il va s'adresser à un service intermédiaire d'annuaire (voir chapitre 7), accessible lui par le nom de machine et le numéro de port, comme une socket ordinaire. L'annuaire va stocker de manière persistante le lien entre un nom logique et une référence à un objet distant. L'annuaire est un outil Java disponible sous la version standard :

```
rmiregistry
```

Par défaut, l'annuaire de RMI appelé `rmiregistry` est lancé sur le port 1099. Il doit disposer du code de la souche sur son *classpath*, et il est donc généralement lancé à la racine du code du serveur.

On peut ensuite enregistrer l'objet dans l'annuaire avec les commandes `bind` ou `rebind`. `rebind` permet d'écraser le nom d'une référence existante en utilisant la classe `java.rmi.Naming` :

```
Naming.bind("topmodel", modele) ;
```

Enfin, si le stub est généré à l'aide de `UnicastRemoteObject` avec la méthode `exportObject` comme précédemment, on va mettre dans l'annuaire non pas la classe mais le *stub* :

```
registry.bind("topmodel",stub)
```

Dans ce cas, il faut prendre garde à générer la souche avant de lancer le `registry`.

Lancement de l'annuaire par code

Il est possible de lancer l'annuaire non par la commande `rmiregistry` mais par code sur un numéro de port donné, par exemple 1099, en utilisant la classe `java.rmi.registry.Registry`:

```
Registry registry = LocateRegistry.createRegistry(1099);
Modele modele= new Modele() ;
registry.bind("topmodel", modele) ;
```

On notera qu'il n'est pas possible de créer un annuaire sur une machine distante. Une des limitations du service d'annuaire de RMI est de nécessiter un annuaire sur la même machine que l'objet serveur.

Copyright © 2015 Dunod.
© Dunod – Toute reproduction non autorisée est un délit.

Parcours de tous les éléments du rmiregistry

Il peut être commode de parcourir tous les éléments afin de vérifier que tous les objets ont été liés correctement, ou de regarder l'utilisation d'autres applications.

Pour cela il suffit de reprendre la classe `LocateRegistry` et de lui envoyer la méthode `list()`. On obtient un tableau de `String` comportant la liste des noms présents dans `rmiregistry`.

```
String[] annuaire=LocateRegistry.getRegistry().list();
```

5.2.5 Appel distant depuis le client

Le client va d'abord obtenir une référence au serveur au travers de l'annuaire en connaissant le nom logique : ici `topmodel`. Il va ensuite pouvoir appeler des méthodes sur cette référence tout comme si elle était locale. Ceci illustre la notion de transparence de RMI. Nous verrons que celle-ci disparaît avec les services web.

On notera que le client n'importe jamais le package `java.rmi.server` mais seulement le package `java.rmi`.

```
Registry registry = LocateRegistry.getRegistry("localhost");
// port 1099 par défaut ici sur la machine locale
ModeleInterface modele = (ModeleInterface)registry.lookup("topmodel");
Date uneDate = modele.datation() ;
```

On utilise ici `getRegistry` et non plus `createRegistry`, puisque l'annuaire doit être déjà lancé pour obtenir une référence.

L'annuaire est local pour le serveur, mais il peut être distant pour un client : le client doit alors connaître tout comme pour les sockets le numéro IP de la machine ou son nom dans un DNS (*Domain Name Server*). Au chapitre 7, nous verrons qu'il est possible également d'utiliser des pseudo-URL pour l'accès à l'annuaire.

Les différentes étapes sont résumées dans la figure 5.5 :

1. Le serveur enregistre l'objet dans un annuaire.
2. Le client va chercher la référence dans l'annuaire.
3. Le client appelle directement les méthodes comme si la référence était locale.

5.2.6 Arrêt d'un objet serveur

Nous avons vu qu'il était possible d'enregistrer un objet dans le `rmiregistry` ainsi :

```
MyRemoteObject obj = MyRemoteObject.new();
LocateRegistry.createRegistry(1099).bind(''MonObjet'', obj);
```

L'enregistrement dans l'annuaire est bloquant et le code reste en attente de requêtes, sans terminer le programme.

Copyright © 2015 Dunod.

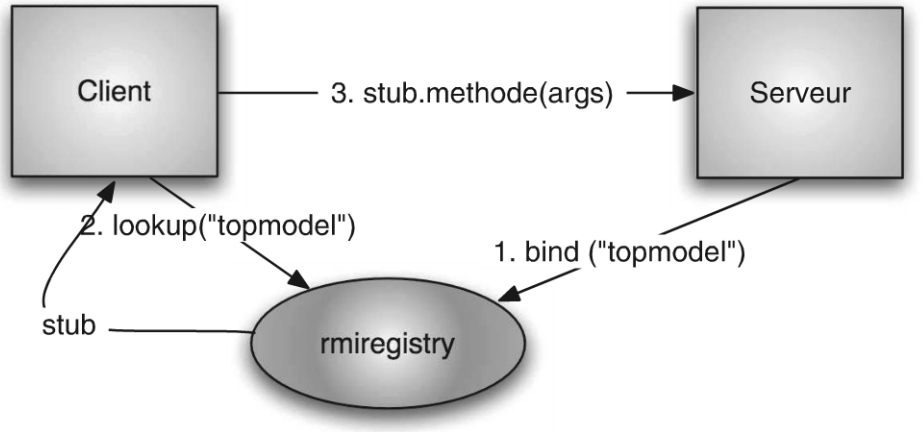


Figure 5.5 — Étapes de connexion entre un client et un serveur

Pour désactiver un objet serveur, il suffit de le désenregistrer du service de nommage.

```
LocateRegistry.getRegistry(1099).unbind(''MonObjet'');
```

Performances : discussion

Il est important de garder ces étapes en tête : lorsque l'on va mesurer des performances de RMI, on va mesurer séparément le temps pour obtenir une référence et le temps d'appel de la méthode à proprement parler en point à point. De même pour CORBA. En général, le temps d'établissement de la connexion est bien plus long que l'appel lui-même, et très souvent le client gardera une référence sur le serveur afin d'optimiser le temps de mise en relation.

Il ne s'agit pas d'une utilisation de ressources, comme sur une base de données qui va monopoliser une connexion. Toutefois, la conservation longue d'une référence peut générer des erreurs si le serveur n'est plus actif. Elle a aussi un impact sur la récupération de la mémoire que nous verrons plus loin.

Pour l'optimisation des performances, on peut également penser à regrouper dans un même paquet de données plusieurs informations, au lieu d'aller faire plusieurs appels successifs à distance pour chercher les éléments un à un.

On peut enfin stocker une référence distante dans un cache local, afin d'éviter de faire appel à l'annuaire. Toutefois, on risque d'avoir à un moment donné une référence qui pointe dans le vide.

Copyright © 2015 Dunod.
© Dunod – Toute reproduction non autorisée est un délit.

5.3 PASSAGE DES ARGUMENTS

5.3.1 Passage des arguments par référence

Dans le chapitre 4 *Interfaces*, nous avons vu la différence entre passage par valeur et par référence.En RMI, il est possible de passer les arguments des deux manières : tous les arguments de fonctions distantes doivent être soit sérialisés, soit des objets distants. Dans l'exemple précédent, la date est renvoyée par valeur, et l'objet `Date` est sérialisé sur le modèle et désérialisée sur l'entrepôt[1].

On peut également passer un objet distant en argument d'une méthode RMI pour un passage par référence.

Intéressons-nous à présent à l'entrepôt. Comme il est accessible du modèle et implémente des méthodes distantes, il doit lui aussi être un objet distant. Voici son interface :

```
public interface EntrepotInterface extends Remote {
    public void stockage(ModeleInterface modele) throws RemoteException; }
```

La méthode `stockage` prend en argument un `Modele` qui est un objet distant.

On peut alors implémenter l'entrepôt par héritage de `UnicastRemoteObject` sans oublier le constructeur :

```
public class Entrepot extends UnicastRemoteObject implements
EntrepotInterface
{protected Entrepot() throws RemoteException {super();}
public void stockage(ModeleInterface modele) throws RemoteException {
Date estampille= modele.datation();
System.out.println("Objet stocké le "+estampille); }}
```

La méthode `stockage` va appeler les méthodes sur le modèle tout simplement. Ici, inutile de passer par l'annuaire car la référence est passée automatiquement en argument.

Dans un logiciel réel, on mettrait seulement l'entrepôt dans l'annuaire, mais pas les modèles. Le modèle obtient donc d'abord une référence sur l'entrepôt dans l'annuaire, puis l'entrepôt rappelle le modèle de manière simple.

Égalité des références

Modifions légèrement l'implémentation pour que l'estampille du modèle soit propre à celui-ci.

```
public Date datation() throws RemoteException {
    return estampille; // variable d'instance estampille
}
```

1. Reportez-vous au pararaphe 2.1 pour une discussion approfondie sur la sérialisation.

Copyright © 2015 Dunod.

On modifie le scénario en rajoutant une mise à jour de l'estampille du modèle, et un autre appel de l'entrepôt à celui-ci après modification de l'estampille (figure 5.6).

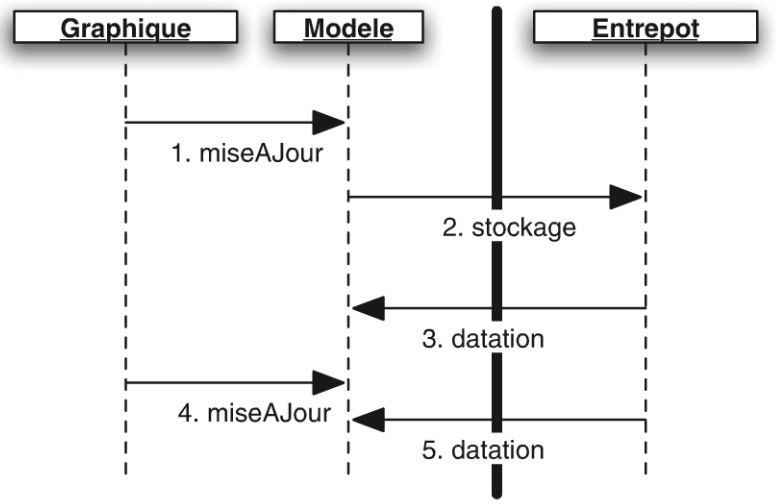


Figure 5.6 — Deux appels successifs à la même référence

On observera alors que si le modèle change d'estampille, la datation obtenue par l'entrepôt reflète cette modification.

Côté modèle	Côté entrepôt
`setEstampille(valeur1)`	`modele.datation()` renvoie `valeur1`
`setEstampille(valeur2)`	`modele.datation()` renvoie `valeur2`

On vérifiera à titre d'exercice que ceci est également vrai des variables statiques.

5.3.2 Passage des arguments par valeur

Imaginons à présent de passer le modèle par valeur. `Modele` devient un objet normal qui n'hérite plus de `UnicastRemoteObject`, mais qui implémente `Serializable`.

```
public class Modele implements Serializable, ModeleInterface {
    private Date estampille;
    public void setEstampille(Date uneDate){estampille = uneDate; }
    public Date datation() {return estampille; }}
```

L'entrepôt peut toujours utiliser un objet de type `ModeleInterface` avec la méthode stockage :

```
public void stockage(ModeleInterface modele) throws RemoteException {
    Date estampille= modele.datation();
    System.out.println("Objet stocke le "+estampille);}
```

© Dunod – Toute reproduction non autorisée est un délit.
Copyright © 2015 Dunod.

Toutefois, la méthode `datation()` sera appelée cette fois-ci en local sur une copie du modèle. L'entrepôt doit disposer de l'implémentation de la classe `Modele` sur son *classpath*, et non plus seulement de l'interface et du stub. Et cette fois, si le modèle modifie son estampille, la modification ne sera pas propagée à la copie locale à l'entrepôt.

Côté modèle	Côté entrepôt
`setEstampille(valeur1)`	`modele.datation()` renvoie `valeur1`
`setEstampille(valeur2)`	`modele.datation()` renvoie toujours `valeur1`

5.3.3 Exceptions

Les exceptions sont sérialisables en RMI : elles seront donc passées comme retour d'appel de méthodes le cas échéant.

De plus, si un client appelle un serveur disparu ou inactif, il obtiendra immédiatement une exception : ceci permet d'obtenir un minimum de qualité de service, et d'être sûr que le message a été reçu ou non.

RMI définit des exceptions spécifiques, liées d'une part à l'annuaire (références non existantes (`NotBoundException`), référence déjà liée(`AlreadyBoundException`), etc.) d'autre part au fonctionnement du bus logiciel (`RemoteException`). Une exception distante sera levée en local lorsqu'elle revient : ceci est très important pour déverminer une application distante.

Créons par exemple une exception de datation :

```
public class ExceptionDeluge extends Exception {
    public String getMessage() {return "Avant le deluge";}}
```

Modifions l'interface `ModeleInterface` pour prendre en compte cette exception :

```
public interface ModeleInterface extends Remote {
    public java.util.Date datation() throws RemoteException, ExceptionDeluge;}
```

Et modifions également l'implémentation :

```
public class Modele extends UnicastRemoteObject implements ModeleInterface{
    protected Modele() throws RemoteException {super();}
    public Date datation() throws ExceptionDeluge{
        throw new ExceptionDeluge();}}
```

Copyright © 2015 Dunod.

Lorsque le client va appeler la méthode `datation()` il va bien relever l'exception :

```
try {ModeleInterface modele = (ModeleInterface)
     LocateRegistry.getRegistry().lookup("modele");
  modele.datation());}
catch (ExceptionDeluge e) {System.out.println(e.getMessage());}
```

5.3.4 Création d'objets à distance

En Java les classes ne sont pas des objets, et ne peuvent être stockées dans un annuaire ou passées en référence Il est donc **impossible** d'écrire quelque chose comme :

```
Naming.bind("fabrique",maClasse).
```

On utilise donc le *design pattern* de la *Factory* présenté au chapitre 2 et on délègue la génération des objets à une fabrique distante qui pourra appeler en local la classe voulue et renvoyer soit une référence, soit une copie d'un objet initialisé localement, selon que `Modele` est sérialisable ou hérite de `UnicastRemoteObject`.

```
public ModeleInterface creeModele(Date estampille) {
return new Modele(estampille);}
```

5.4 RMI EN MULTITHREAD

Les objets serveurs en RMI peuvent naturellement répondre à plusieurs clients simultanément. Par implémentation, les méthodes RMI exposées comme `Remote` seront en mode multithread, plusieurs clients pourront exécuter des appels de méthodes simultanément. Comment doivent-ils alors gérer les accès simultanés ? Il va falloir prendre garde aux accès concurrents aux variables d'instances.

On identifie d'abord les variables d'instances qui sont utilisées dans les méthodes accessibles à distance.

Il peut exister plusieurs besoins sur le serveur pour gérer des threads multiples :

- on peut avoir besoin de coordonner des threads ;
- on peut avoir besoin de mettre des priorités sur des threads.

5.4.1 Exclusion mutuelle pour éviter des accès concurrents à une variable

Nous avons vu au chapitre 2 Rappels qu'il était possible de verrouiller l'accès concurrent à des variables par plusieurs threads. Il est tout à fait possible d'utiliser cette politique pour éviter les accès concurrents par des méthodes distantes.

Par exemple pour un compte en banque qui implémente les méthodes `retrait` et `depot`, selon l'implémentation de l'objet, on pourra envisager plusieurs solutions.

Copyright © 2015 Dunod.
© Dunod – Toute reproduction non autorisée est un délit.

Première solution

Le compte est représenté par une seule variable d'instance `solde` : on envisage de verrouiller tout l'objet, et un seul client pourra accéder au compte en simultané. On rappelle que l'utilisation de `synchronized nomMethode` est équivalent à `synchronized(this) nomMethode`, soit un verrou sur l'objet total.

L'inconvénient de cette solution est d'interdire le retrait en même temps que le dépôt. L'autre inconvénient est celui de la performance dégradée en cas d'appels nombreux à une méthode donnée.

```
public synchronized boolean depot(int montant) throws RemoteException {
    solde = solde + montant;
    return true;
}
public synchronized boolean retrait(int montant) throws RemoteException {
    solde = solde - montant;
    return true;
    }
public synchronized int getSolde() throws RemoteException {
    return solde;}
```

Deuxième solution

On peut alors penser à une deuxième solution avec des verrous plus fins sur les variables d'instance, et non sur l'objet entier. On garde un historique des retraits, un historique des dépôts, et on effectue les additions uniquement sur le calcul du solde. De cette manière, il sera possible d'effectuer un retrait et un dépôt en simultané, mais les retraits devront se faire l'un après l'autre. De même pour les dépôts. Les historiques peuvent être implémentés comme `ArrayList`.

```
public boolean depot(int montant) throws RemoteException {
    synchronized(depots) {depots.add(montant);
        return true;}
    }
public boolean retrait(int montant) throws RemoteException {
    synchronized(retraits) {retraits.add(montant);
        return true;
    }
public int getSolde() throws RemoteException {
    synchronized(depots) {
    synchronized(retraits) {
        total= somme(depots)-somme(retraits);
}
```

5.4.2 Synchronisation sur le client

Si le client souhaite synchroniser l'appel à l'objet distant de son côté, il peut utiliser également des méthodes `synchronized`. Mais attention, cette synchronisation va être locale à son contexte, et non partagée par les autres clients ! En effet, le verrou va être posé sur le *stub* et non sur l'objet distant lui-même.

Copyright © 2015 Dunod.

Afin de le vérifier, rallongeons le temps de réponse du modèle de 20 secondes :

```
public Date datation() throws RemoteException {
    Thread.sleep(20000);//attente de 20 000 ms
    return new GregorianCalendar().getTime();
```

À présent on lance deux clients simultanément, en synchronisant l'appel ainsi :

```
ModeleInterface modele = (ModeleInterface)
    LocateRegistry.getRegistry().lookup("modele");
Date date = null;
synchronized (modele){date=modele.datation();};
System.out.println(date);
```

Qu'observe-t-on ? Si le verrou était posé correctement à distance, la date des clients devrait être espacée au moins de 20 secondes. Or ce n'est pas le cas. On constate donc que le verrou est bien local sur la souche et qu'il n'a aucun effet. Par contre, il permet de synchroniser les différents appels distants d'un client particulier.

5.5 ACTIVATION AVEC RMI

5.5.1 Modifications de l'implémentation du serveur pour l'activation

L'activation en RMI évite de maintenir des serveurs actifs en mémoire en attente de clients : un démon Java va créer l'objet distant seulement lorsqu'un client se connecte.

L'activation en RMI est délicate à mettre en œuvre car elle implique plusieurs démons Java :

- `rmiregistry` pour permettre au client de trouver la référence de l'objet distant
- `rmid`, le démon d'activation qui va lancer le serveur à la demande

Le code du client n'est absolument pas modifié, mais par contre l'objet distant doit hériter de `Activatable` plutôt que de `UnicastRemoteObject`, et naturellement implémenter une ou plusieurs interfaces.

On peut toutefois éviter d'hériter de `Activatable` en plaçant le code suivant dans le constructeur :

```
Activatable.exportObject(this, id, 0) ;
```

Voyons tout d'abord l'interface distante :

```
public interface Bourse extends Remote {
 public double obtenirPrix( String symbole ) throws RemoteException;}
```

Copyright © 2015 Dunod.
© Dunod – Toute reproduction non autorisée est un délit.

Puis l'objet activable qui l'implémente :

```
public class ActivatableBourse extends Activatable implements Bourse {
   private Map<String, Double> prix;
   protected ActivatableBourse(ActivationID arg0,
         MarshalledObject<Map<String, Double>> data) throws IOException,
         ClassNotFoundException {
      super(arg0, 0);
      prix = data.get();
      System.out.println("Bourse en marche");
   }
   public double obtenirPrix(String symbole) throws RemoteException {
      return prix.get(symbole);}}
```

5.5.2 Activation de l'objet distant

Pour mettre l'objet à disposition il faut fournir :

- Un ID d'activation
- Un objet permettant l'initialisation de l'objet distant. Ce paramètre doit être de type `MarshalledObject` et doit contenir tous les paramètres nécessaires et sérialisables, par exemple danse une liste.

Dans le constructeur de l'objet d'implémentation on utilise la méthode `get` pour récupérer les paramètres d'initialisation, dans notre exemple la liste de prix.

Ensuite on construit un programme pour mettre l'objet à disposition. On commence par créer un groupe d'activation avec des sécurités Java permettant la connexion en réseau :

```
Properties props = new Properties();
props.put("java.security.policy", "/chemin/vers/server.policy");
ActivationGroupDesc group = new ActivationGroupDesc(props, null);
```

On crée ensuite un ID pour l'appel :

```
ActivationGroupID id = ActivationGroup.getSystem().registerGroup(group);
```

Puis les descripteurs d'activation qui nécessite une URL de téléchargement de code :

```
String codebase = "http://localhost:8080/";
      ActivationDesc desc = new ActivationDesc(id, "ActivatableBourse",
            codebase, param);
```

Dans ce cas on a choisi un téléchargement par http, et nous verrons comment télécharger un serveur de code simple.

Copyright © 2015 Dunod.

Voici donc le code d'activation complet :

```
public class MySetup {
   public static void main(String[] args) throws Exception {
      System.out.println("Mise en route de l'activation...");
      Properties props = new Properties();
            props.put("java.security.policy",
            new File("server.policy").getCanonicalPath());
            ActivationGroupDesc group = new ActivationGroupDesc(props, null);
      ActivationGroupID id = ActivationGroup.getSystem().registerGroup(group);
      Map<String, Double> prix = new HashMap<String, Double>();
      prix.put("LVMH", 10.8);
      prix.put("EDF", 8.0);
      MarshalledObject<Map<String, Double>> param = new
MarshalledObject<Map<String, Double>>(
            prix);
      String codebase = "http://localhost:8080/";
      ActivationDesc desc = new ActivationDesc(id, "ActivatableBourse",
            codebase, param);
      Bourse bourse = (Bourse) Activatable
            .register(desc);
      System.out.println("Liaison de la bourse a rmiregistry...");
      Context namingContext = new InitialContext();
      namingContext.bind("rmi:bourse", bourse);
      System.out.println("Sortie...");
   }
}
```

5.5.3 Code du client

Le code du client est peu changé si ce n'est qu'il doit pouvoir télécharger le code du stub.

Voici le code du client :

```
public class Client {
public static void main(String[] args) {
      Bourse stub;
      SecurityManager manager = new RMISecurityManager();
      System.setProperty("java.security.policy", "client.policy");
      System.setSecurityManager(manager);
      try {
         stub = (Bourse) LocateRegistry.getRegistry().lookup("bourse");
         System.out.println(stub.obtenirPrix("EDF"));
      } catch (Exception e) {
         e.printStackTrace();
      }
   }}
```

On remarquera l'utilisation du `RMISecurityManager` nécessaire pour pouvoir télécharger le code du stub.

Copyright © 2015 Dunod.
© Dunod – Toute reproduction non autorisée est un délit.

5.5.4 Mise en œuvre

Nous commençons donc tout d'abord par télécharger le mini-serveur http en Java NanoHTTP à l'adresse suivante : https://github.com/NanoHttpd/nanohttpd.

Lors du téléchargement de code, il est essentiel que `rmiregistry` ne soit pas lancé dans le répertoire où se trouve le code, car cela génère des conflits de sécurité pour les `ClassLoader`, qui permettent de charger du code dans la machine virtuelle Java.

On va donc séparer les éléments du code afin d'éviter les conflits de sécurité dans quatre répertoires avec le contenu suivant :

- **Répertoire Client**
  - `bourse.class`
  - `client.class`
  - `client.policy`
  - `lancerClient.bat`

  Avec le fichier `client.policy` :

```
grant
{
   permission java.net.SocketPermission  "*:1024-65535", "connect,resolve";
};
```

  Et le fichier `lancerClient.bat`

```
java Client -Djava.security.policy=client.policy
```

- **Répertoire Serveur**
  - `ActivatableBourse.class`
  - `Bourse.class`
  - `server.policy` (pour la sécurité Java)
  - `MySetup.class`
  - `lancerServeur.bat`

  avec le fichier `server.policy` tel que celui-ci :

```
grant
{
    permission java.net.SocketPermission "*:1024-65535",
"connect,accept,resolve";
};
```

  et le fichier `lancerServeur.bat` comme suit :

```
java  -Djava.rmi.server.codebase=http://localhost:8080/  MySetup
```

Copyright © 2015 Dunod.

- **Répertoire Download –** Ce répertoire va permettre au client de télécharger dynamiquement le stub avec le mini serveur web `NanoHTTPD`.
  On va donc y mettre le code compilé de NanoHTTPD ainsi que les classes suivantes :
  - `ActivatableBourse.class`
  - `Bourse.class`

  Voici la commande `lancerNano.bat` où l'on spécifie que Nano utilise le port `8080`

```
java NanoHTTPD -p 8080
```

On remarquera au passage que le stub va être généré automatiquement lorsque le client va le demander.

- **Répertoire Rmi –** Ce répertoire va servir à mettre les fichiers de sécurité pour `rmid` et à lancer les démons. Voici le fichier de sécurité `rmid.policy` :

```
grant {
permission com.sun.rmi.rmid.ExecPermission "${java.home}${/}bin${/}java";
permission com.sun.rmi.rmid.ExecOptionPermission
"-Djava.security.policy=*";};
```

Voici la commande pour lancer `rmid lancerRmid.bat` :

```
rmid -J-Djava.security.policy=rmid.policy
```

On va ensuite lancer dans l'ordre :

- `rmiregistry` dans le répertoire rmi (sans aucun code !)
- `LancerRmid` dans le répertoire `rmi`
- `LancerNano` dans le répertoire `download`
- `LancerServeur` dans le répertoire `serveur`
- `LancerClient` dans le répertoire `client`

On observera que `LancerServeur` termine immédiatement, contrairement aux programmes RMI classiques. On observera également la trace des téléchargements dans la console de lancement de Nano.

5.6 AUTRES ASPECTS DE RMI

5.6.1 Gestion du ramasse-miettes

RMI tire parti de l'utilisation du seul langage Java pour gérer la mémoire répartie de manière automatique : on parlera de DGC (*Distributed Garbage Collector*). Ceci n'est

© Dunod – Toute reproduction non autorisée est un délit.
Copyright © 2015 Dunod.

pas possible en CORBA par exemple (voir chapitre 6), car CORBA peut connecter des langages sans ramasse-miettes comme C ou C++, et il n'est pas alors possible de faire des hypothèses sur les références. La gestion de la mémoire répartie en RMI s'effectue par comptage de référence, mécanisme qui ne détecte pas les cycles d'utilisation entre plusieurs objets.

La clé de l'utilisation est de maîtriser la durée du bail d'un objet distant à l'aide de la propriété : `java.rmi.dgc.leaseValue`.

On modifie le code du serveur en diminuant considérablement la valeur du bail, ici 100 ms.

```
Modele modele= new Modele();
System.getProperties().put("java.rmi.dgc.leaseValue", new Integer(100));
LocateRegistry.createRegistry(1099).rebind("modele", modele);
```

On constate alors que le serveur termine très rapidement, et qu'aucun client ne peut plus se connecter.

Un objet distant pourra être alerté de la fin de son bail en implémentant l'interface `Unreferenced`.

Toutefois ici le programme termine avant de pouvoir lancer la méthode `unreference`.

Il est également fréquent de modifier l'intervalle d'exécution du ramasse-miettes même si ceci se fait de manière spécifique à une JVM :
`sun.rmi.dgc.server.gcInterval` ou `sun.rmi.dgc.client.gcInterval`

5.6.2 Modification du protocole de RMI

Le protocole JRMP natif de RMI ou même le protocole IIOP ne permettent pas de faire communiquer des objets au travers de pare-feu. Il faut donc utiliser du *tunneling* au-dessus d'un autre protocole, typiquement HTTP.

Le protocole RMI va donc s'exécuter non plus sur TCP mais sur une couche HTTP qui va émuler les envois de part de d'autre. On utilise pour cela des fabriques de sockets (`SocketFactory`) permettant de modifier le protocole.

Ces fabriques peuvent avoir de nombreux autres usages, notamment pour chiffrer RMI au-dessus de SSL ou compresser les données. Ces aspects avancés ne seront pas présentés dans cette édition.

5.7 RMI IIOP

RMI peut utiliser deux protocoles : le protocole natif appelé JRMP (*Java Remote Method Protocol*) ou IIOP (*Interoperable Internet Object Protocol*). IIOP est le protocole de CORBA et le protocole recommandé pour les EJB (*Entreprise JavaBeans*). Il est en particulier utilisé dans les serveurs d'applications Java EE.

Copyright © 2015 Dunod.

Il est possible de mettre un objet RMI à disposition d'une application cliente CORBA écrite en Java ou un autre langage à l'aide de RMI sur IIOP.

5.7.1 Premières étapes de mise en œuvre

La démarche est très similaire à la mise en œuvre de RMI avec quelques petites modifications qui sont présentées ici.

On part toujours d'une interface RMI :

```
public interface ModeleInterface extends Remote {
    public java.util.Date datation() throws RemoteException; }
```

Mais cette fois l'implémentation doit hériter de `PortableRemoteObject`.

```
public class Modele extends PortableRemoteObject implements ModeleInterface
{
protected Modele() throws RemoteException {super();}
public Date datation() throws RemoteException {
  return new GregorianCalendar().getTime();}
public void miseAJour(){};}
```

Et la génération des souches utilise `rmic` avec l'option `-iiop` :

```
rmic -iiop Modele
```

Les fichiers générés cette fois sont `_Modele_Tie.class` et `_ModeleInterface_Stub.class`. Le `Tie` correspond au squelette CORBA qui permet le pliage et dépliage des arguments côté serveur.

5.7.2 Nouvel annuaire CORBA

Java possède un annuaire CORBA (*naming service*), en plus de l'annuaire RMI. Il se trouve avec tous les outils Java sous `$JAVA-HOME/bin`. Nous verrons au chapitre 7 sur les annuaires comment obtenir un accès universel à ces annuaires avec JNDI. Nous verrons également au chapitre 6 sur CORBA l'annuaire CORBA en détail.

Il suffit toutefois pour RMI-IIOP de lancer le démon `orbd` sur le port 1050 par exemple :

```
orbd -ORBInitialPort 1050
```

L'accès à cet annuaire par code utilise l'interface JNDI (*Java Naming Directory Interface*) qui sera présentée en détail au chapitre 7 des annuaires. Elle nécessite le package `javax.naming`.

L'objet qui va permettre d'enregistrer un objet distant dans l'annuaire est de classe `Context`.

Copyright © 2015 Dunod.
© Dunod – Toute reproduction non autorisée est un délit.

```
Hashtable env = new Hashtable();
env.put(Context.INITIAL_CONTEXT_FACTORY,
    "com.sun.jndi.cosnaming.CNCtxFactory");
env.put(Context.PROVIDER_URL,"iiop://localhost:1050");
Context ctx=new InitialContext(env);
```

Le code du serveur va utiliser ce contexte avec peu de différences avec RMI normal.

```
Modele modele= new Modele();
ctx.rebind("modele", modele);
```

Le code du client va par contre utiliser une méthode particulière pour convertir la référence obtenue du serveur : la méthode `narrow`, qui est à peu près l'équivalent d'un `cast`.

```
Object objref = (ModeleInterface) ctx.lookup("modele");
modele =(ModeleInterface) PortableRemoteObject.narrow(objref,
    ModeleInterface.class);
System.out.println(modele.datation());
```

En résumé

RMI est une API ancienne de Java qui permet de faire communiquer des applications Java en mode RPC de manière transparente mais intervient dans de nombreux outils serveurs. Java fournit un annuaire appelé `rmiregistry` pour trouver les références distantes et les objets distants publient leurs interfaces sous forme d'interfaces Java standard.

RMI permet de passer des arguments par valeur ou par référence et gère la durée de vie des références. L'interface RMI-IIOP permet de rendre des objets Java accessibles à d'autres langages par CORBA.

Copyright © 2015 Dunod.

6

CORBA

Objectif

Le but de ce chapitre est de présenter les techniques relatives à CORBA, et de montrer en quoi celles-ci sont encore actuelles. CORBA est en effet une technologie déjà ancienne née en 1989, mais elle a bénéficié de la maturité de nombreuses expériences dans de nombreux domaines (télécommunications, finance, industrie de la défense, etc.) et propose des solutions sophistiquées par exemple pour l'activation des objets et l'appel dynamique de méthodes.

6.1 ÉVOLUTION DE CORBA

6.1.1 L'OMG

L'architecture CORBA[1] (*Common Object Request Broker Architecture*) est née en 1991 au sein du consortium de l'OMG (*Object Management Group*) afin de définir l'interopérabilité entre différents langages de programmation et différents systèmes d'exploitation. Ce consortium qui regroupe plus de 800 sociétés et organisations dans le monde définit des spécifications pour fournir un modèle de coopération entre objets répartis. CORBA n'est qu'une des nombreuses spécifications de l'OMG qui travaille également sur UML (*Universal Modeling Language*) dans le domaine de la conception d'application et sur l'approche MDA (*Model Driven Architecture*).

La première version de CORBA date de 1991, la deuxième version majeure de 1995, et une autre version importante de 2002. Une grande étape a été l'élaboration

1. http://www.corba.org/

© Dunod – Toute reproduction non autorisée est un délit.
Copyright © 2015 Dunod.

d'un modèle d'interopérabilité entre ORB (*Object Request Broker*) appelés encore bus logiciels. Ainsi l'ORB inclut dans Java 5 est capable de communiquer avec une autre implémentation Java, avec un autre langage ou sur un autre système d'exploitation.

La caractéristique majeure de CORBA est de permettre une communication en mode RPC (*Remote Procedure Call*) avec de nombreux autres langages autres que Java, à l'inverse de RMI. Comme en RMI un serveur CORBA peut héberger plusieurs objets CORBA accessibles à distance, et une même application peut abriter à la fois des objets clients et des objets serveurs.

1991 peut sembler une date préhistorique, mais ce serait un tort de négliger la maturité acquise par CORBA, ainsi que la richesse des concepts développés autour de l'architecture. RMI en ce sens peut être considéré comme une simplification de CORBA, qui rajoute à chaque étape une nouvelle complexité (par exemple pour l'activation de serveurs). SOAP peine à mettre en œuvre toutes les propriétés non fonctionnelles (sécurité, transactions, services d'événements, etc.) et la normalisation des interfaces est loin d'avoir atteint la stabilité et la maturité de CORBA. De plus, les aspects non fonctionnels associés à SOAP (sécurité, transaction) commencent à peine à voir le jour, alors que CORBA les a définis depuis longtemps sous la forme de services CORBA (voir paragraphe 6.2.6).

En plus de son interopérabilité avec d'autres langages et de sa montée en force dans l'embarqué et le temps réel, CORBA reste une architecture à étudier et une alternative à prendre en compte.

CORBA a évolué au cours du temps, et les dernières versions (notamment la spécification 2.6) ont beaucoup assoupli le déploiement des applications qui a longtemps été le point d'achoppement de l'architecture.

Voici un bref historique :

- 1989 : création de l'OMG.
- 1991 : première version de CORBA.
- 1992 : version 1.1 ; 200 membres à l'OMG.
- 1995 : CORBA 2.0 ; interopérabilité entre solutions de vendeurs différents.
- 1996-1998 : définition des services (nommage, notification, transaction, etc.)
- 1998 : version 2.2 ; création du *Portable Object Adapter*.
- 1999 : version 2.3 ; passage de paramètres par valeur (*Value Types*), intégration du mapping Java vers IDL.
- 2000 : version 2.4 ; service de nommage interopérable (*Interoperable Name Service*), messagerie asynchrone, CORBA temps réel.
- 2002 : version 3.0 ; modèle de composants, révision du modèle de traversée des pare-feu.

Copyright © 2015 Dunod.

Il est possible de voir quelle version Java implémente sur le site d'Oracle[1], mais il s'agit plus ou moins de la version 2.3 avec un ajout de service interopérable.

6.1.2 CORBA et Java

Même si CORBA est né avant Java au moment d'une prédominance du langage C, la communauté Java a toujours été très active au sein de l'OMG, et des concepts tels que le passage par valeur viennent de la sérialisation connue en Java.

Java fait partie des nombreuses projections de CORBA vers les différents langages de programmation[2]. Et dès 1998, Sun a introduit dans le JDK 1.2 le Java IDL (*Interface Definition Language*) permettant l'interopérabilité de Java avec CORBA. Java IDL incluait à la fois un ORB supportant le protocole de communication propre à CORBA, IIOP. Il comportait également un compilateur permettant de générer le code CORBA en Java à partir de l'IDL tant pour les adaptateurs pour la partie cliente (*stubs* ou encore proxies) que les adaptateurs côté serveur (*skeletons*).

Actuellement, Java 5 supporte les *Portable Object Adapters* (voir paragraphe 6.2.3), les intercepteurs de protocole, un service de nommage interopérable ainsi que les types anonymes *any* dynamiques.

Quelques implémentations commerciales ou Open Source

Il y a de moins en moins de valeur ajoutée dans les ORB CORBA, et il reste peu d'acteurs qui en font encore des produits à part entière : la plupart du temps CORBA va être enfoui au sein d'un serveur d'application, ou d'un progiciel plus spécifique. CORBA est devenu une commodité : vous le trouverez sous Linux avec l'implémentation Orbit, dans les connecteurs Oracle, dans les drivers d'imprimantes... Les services CORBA (notification, équilibrage de charge, etc. voir paragraphe 6.2.6) restent toutefois des produits à forte valeur ajoutée.

Citons tout de même les implémentations suivantes en Java en dehors du JDK :
– Microfocus Orbix[3] Orbacus[4] Visibroker[5] et OpenFusion[6] (commerciale)
– Jacorb[7] (Open Source)
– OpenORB[8] (Open Source)

1. http://docs.oracle.com/javase/7/docs/technotes/guides/idl/compliance.html
2. *Java to IDL Mapping*, http://www.omg.org/cgi-bin/doc?formal/01-06-07.
3. http://www.microfocus.com/products/corba/orbix/
4. http://microfocus.com/products/corba/orbacus/index.aspx
5. https://www.microfocus.fr/produits/visibroker/index.aspx
6. http://www.microfocus.com/products/corba/openfusion/index.aspx
7. http://www.jacorb.org
8. http://openorb.sourceforge.net

© Dunod – Toute reproduction non autorisée est un délit.
Copyright © 2015 Dunod.

Et quelques implémentations Open Source pour C++ : Mico, Omniorb, TAO, Orbit. Orbit permet par exemple de programmer les interfaces GTK+ sous Unix, TAO a toujours été spécialisé sur le temps réel, Omniorb est facile d'accès.

Le choix d'un ORB CORBA va dépendre des caractéristiques offertes par l'ORB, de la version de la norme implémentée, de la compatibilité de versions entre les différentes machines et langages. Mais aussi de la liste des bibliothèques implémentées (les services CORBA, voir paragraphe 6.2.6).

Le JDK implémente peu de services, et il est possible de rajouter des bibliothèques complémentaires pour l'utilisation de ces services.

6.1.3 Pourquoi utiliser CORBA ?

CORBA est une alternative intéressante pour communiquer avec de nombreux langages, aussi divers que Python, Lisp, C++, Smalltalk. Il permet surtout d'obtenir un code très compact et efficace, économe en bande passante, et il existe de nombreuses implémentations pour l'embarqué prenant en compte des exigences pour le temps réel. La bande passante pourra être un argument important contre l'utilisation de services web.

Il permet également de faire une vérification de typage fort par l'utilisation de description d'interfaces, et il a surtout un mécanisme d'activation de service très sophistiqué.

Il nécessite peu de bibliothèques et possède des profils pour le temps réel et l'embarqué. Par contre, il ne traverse pas les pare-feu.

6.2 ARCHITECTURE DE CORBA

6.2.1 Appels distants

Tout comme RMI, CORBA permet d'effectuer des appels de méthodes sur des objets à distance (*Remote Procedure Call*) de manière bloquante et synchrone. Il nécessite généralement l'utilisation de souches. Il utilise un bus logiciel appelé ORB (*Object Request Broker*) et s'appuie sur un protocole d'échange d'objets appelé IIOP (*Internet Inter-Orb Protocol*).

Chaque objet va exposer son interface avec la signature de ses méthodes en utilisant l'IDL (*Interface Description Language*). Comme CORBA peut avoir différents langages pour cibles, cet IDL ne peut être une interface Java comme en RMI. En particulier, il permet de décrire des types de données qui doivent être compris de plusieurs langages. IIOP décrit une liste de règles de formatage des données sur le réseau appelée CDR (*Common Data Representation*) adaptée aux types de données décrits dans l'IDL.

Copyright © 2015 Dunod.

6.2.2 L'ORB

L'architecture globale de CORBA inclut un bus logiciel appelé ORB (*Object Request Broker*) avec un rôle très analogue au bus RMI :

- Maintien des références sur les objets distants et gestion de leur cohérence : l'ORB doit renvoyer une exception si l'objet serveur a disparu.
- Pliage et dépliage des arguments (*marshalling*).
- Acheminement des requêtes vers le bon objet à partir de sa référence au travers du réseau et gestion des références d'objets. Il n'y a pas de récupération automatique de la mémoire en CORBA, contrairement à RMI.

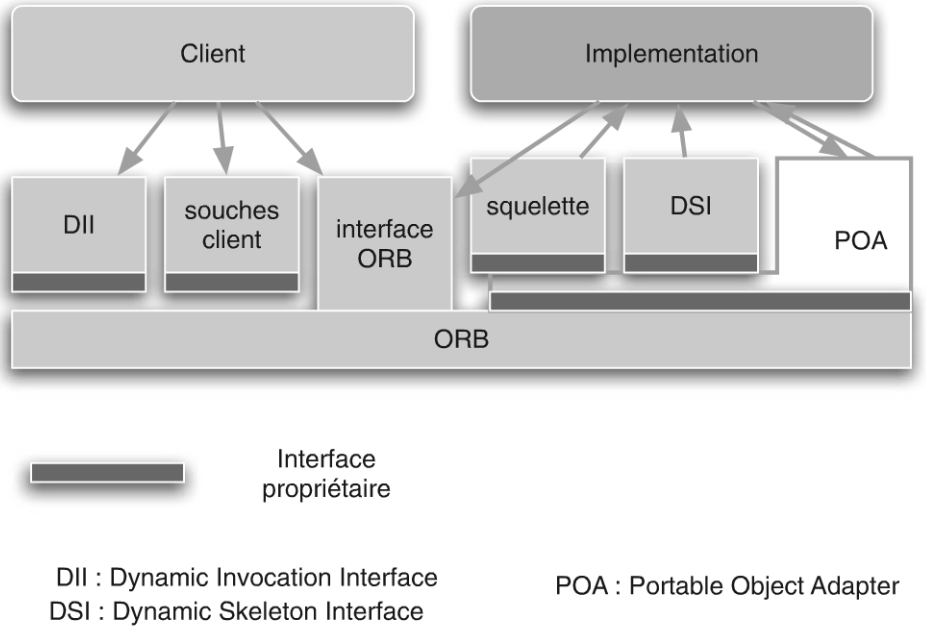


Figure 6.1 — Schéma d'ensemble de la communication en CORBA

L'architecture d'une application CORBA est présentée à la figure 6.1. L'ORB est accessible au travers d'un ensemble de primitives normalisées par l'OMG mais l'accès du code est toujours fait en local. Si l'on dispose de deux langages différents dans deux applications, on cherchera donc une implémentation des bibliothèques de l'ORB pour chaque langage, par exemple Omniorb pour C++ et Jacorb pour CORBA. Java depuis la version 5 fournit également en standard un ORB. Cet ORB est lancé par une commande du JDK appelée `orbd`.

En C++, l'édition de liens s'effectuera avec les bibliothèques de l'ORB. Pour une autre implémentation en Java que le JDK standard, il faudra également importer des fichiers `JAR` spécifiques.

L'ORB est un objet local, il n'a pas de référence distante et ne peut être accédé à distance. Il fournit toutes les fabriques vers les interfaces pivots de CORBA, notamment le service de nommage.

Copyright © 2015 Dunod.
© Dunod – Toute reproduction non autorisée est un délit.

Java permet d'utiliser une autre implémentation ainsi :

```
Properties p = new Properties();
p.put("org.omg.CORBA.ORBClass", "UneAutreImplementation");
ORB orb = ORB.init(args, p);
```

6.2.3 Souches et POA

Côté client : la souche

Le pattern du mandataire (*Proxy*) est utilisé du côté du client : c'est la **souche** (*stub*)qui assure l'intermédiation avec le bus logiciel :

- Elle prépare les paramètres d'entrée de l'invocation.
- Elle décode les paramètres de sortie et le résultat.
- Elle assure la transparence de référence du serveur : l'utilisateur de la souche ne connaît pas le numéro IP de la machine distante.

Il existe deux modes essentiels de communication en CORBA :

- **L'appel statique** qui est très semblable à RMI, avec génération d'une souche statique à partir de l'IDL. Comme en RMI, le client doit disposer d'autant de souches que de classes d'objets serveurs.
- **L'invocation dynamique** qui permet une découverte dynamique de l'interface du serveur en cours d'exécution, sans connaître l'interface à la compilation. En C++ on parle alors de souche dynamique, qui est commune à toutes les classes d'objets serveurs, en Java elle n'est pas nécessaire. L'invocation dynamique peut être utile lorsque CORBA sert seulement de relais vers un autre serveur. Elle est en quelque sorte le pendant de la réflexion en Java (voir chapitre 2). Avec l'invocation dynamique, le client n'a besoin d'aucun code, même pas l'interface du serveur.

Côté serveur : le concept de servant

En CORBA, on distingue l'objet distant de l'objet qui le met à disposition dans l'ORB : on appelle celui-ci le **servant**. Ceci permet de gérer l'activation de l'objet distant à la demande. C'est exactement comme si vous vous adressiez au bureau d'accueil d'une entreprise plutôt que d'appeler directement l'un de ses employés. L'hôtesse vous répondra s'il est disponible, ira le chercher à l'étage... ou récréera une instance de votre interlocuteur ! Qui avec la persistance ne sera même pas amnésique et se rappellera l'état précédent. Le servant est matérialisé par le *Portable Object Adapter* qui est décrit ci-après.

Côté serveur : le POA

L'objet serveur est mis à disposition par le POA (*Portable Object Adapter*) qui sert d'interface entre le serveur et l'ORB notamment pour le pliage et dépliage des arguments pour l'appel et le retour. Le POA est depuis la version CORBA 2.2 le successeur du

Copyright © 2015 Dunod.

BOA (*Basic Object Adaptor*). Jusqu'à la version CORBA 2.2, l'adaptateur était défini de manière vague, et appelé *Basic Object Adapter* (BOA). Son implémentation variait énormément d'un vendeur à l'autre, et n'assurait pas la portabilité du code entre ORB : le POA assure la portabilité entre les vendeurs. Il permet par ailleurs l'invocation statique ou dynamique. La classe `DynamicImplementation` qui permettait de choisir une invocation dynamique est obsolète, de même que la notion de squelettes statiques et dynamiques.

Un *Object Adapter* :

- génère et interprète les références sur les objets distants ;
- gère les autorisations d'accès à l'objet ;
- gère l'activation des implémentations des objets ;
- gère les références des objets (*Object Id*) ;
- permet l'invocation des méthodes sur l'objet.

Un *Object Adapter* peut servir plusieurs objets, et assure dans ce cas la répartition des requêtes entre les objets. Tous ces objets partagent alors les mêmes propriétés non fonctionnelles définies par le POA : la persistance ou non de l'état de l'objet entre deux appels, l'utilisation de threads ou de sécurité, etc.

Le POA comporte réellement une sémantique de **conteneur**, qui est complètement absente des EJB (*Entreprise JavaBeans*). Les EJB proposent une sémantique pour les composants, mais l'implémentation du conteneur et la manière dont il va gérer les aspects non fonctionnels et l'activation des objets reste spécifique au vendeur et inaccessible par une interface quelconque.

Un POA est aussi un espace de **nommage** pour les objets qui sont repérés par des identificateurs. Un POA est créé par filiation d'un autre POA. Bien sûr le POA racine appelé `RootPOA` est accessible par l'ORB.

```
orb.resolve_initial_references("RootPOA") ;
```

On peut alors créer des POA fils. Nous verrons que chaque POA peut implémenter une politique particulière pour les propriétés non fonctionnelles.

```
POA fils = rootPOA.create_POA("Fils",null,null);
```

Hiérarchie de POA

Les hiérarchies de POA assurent la manière dont le serveur va être activé et ensuite éteint.

Prenons l'exemple à la figure 6.2, cette hiérarchie assure que la destruction respectera l'ordre suivant :

- Le POA `Trefle` sera détruit avant le POA `FabriqueTrefle`, de même pour `Pique`.

Copyright © 2015 Dunod.
© Dunod – Toute reproduction non autorisée est un délit.

- Les POA `FabriqueTrefle`, `FabriquePique` et `Administration` seront détruits avant le `RootPOA`.

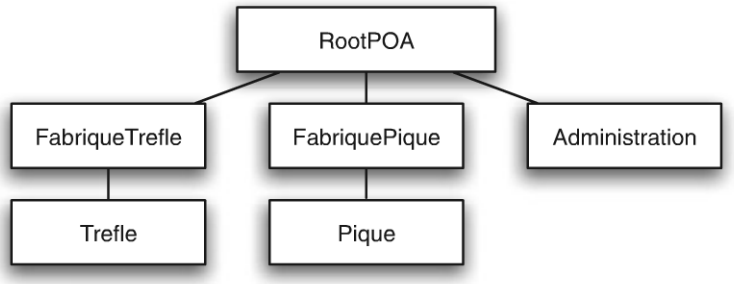


Figure 6.2 — La hiérarchie de POA est liée à la dépendance entre objets serveurs

En général, on pourra créer un POA pour chaque interface IDL, mais il est souhaitable de garder peu de profondeur dans l'arborescence des POA[1].

Il est possible d'implémenter de nombreuses fonctionnalités dans les POA, les plus connues sont les suivantes :

- Gestion de cache.
- Activation paresseuse (à la demande). Ce type de POA permet de gérer la montée en charge de nombreux objets distants en soulageant la mémoire, et en n'incarnant l'objet que s'il est absent de la table active.

POAManager

Les gestionnaires de POA (*POAManager*) sont tout à fait semblables à des robinets qui vont commander l'afflux ou non de requêtes vers les POA (figure 6.3).

Un POAManager peut prendre les états suivants :

- **HOLDING** : le gestionnaire est inactif. Toutes les requêtes sont mises en attente.
- **DISCARDING** : le gestionnaire est inactif. Toutes les requêtes renvoient une exception au client.
- **ACTIVE** : c'est le mode normal de fonctionnement.
- **INACTIVE** : le gestionnaire est inactif. C'est typiquement l'état en cours d'extinction du serveur. Les requêtes sont rejetées de manière propre à l'implémentation.

Un `POAManager` gère typiquement plusieurs POA. C'est une bonne pratique d'utiliser deux `POAManager` par exemple pour gérer les interfaces opérationnelles et les interfaces d'administration.

1. http://www.ciaranmchale.com/corba-explained-simply/concepts-for-server-side-programming.html# toc32

Copyright © 2015 Dunod.

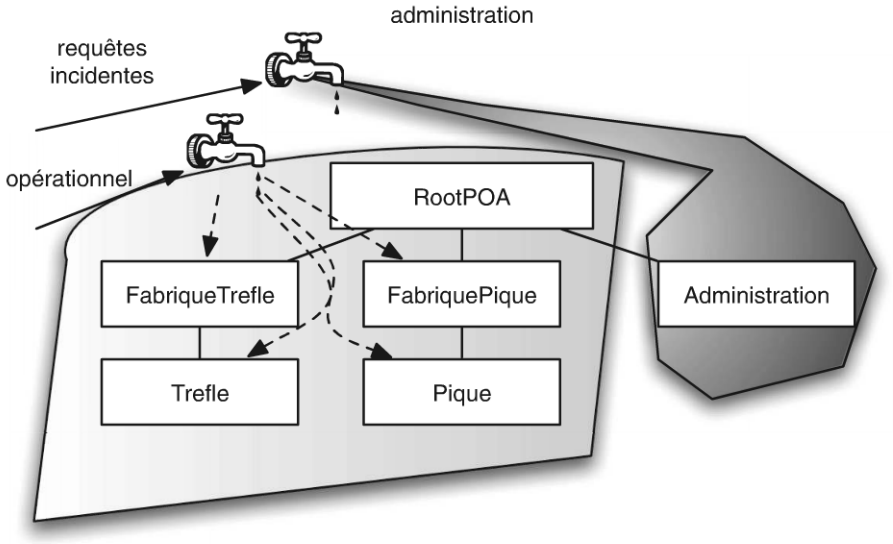


Figure 6.3 — Mode opérationnel et mode administration commandés par les POAManager

Il est ainsi possible de gérer dynamiquement la disponibilité de l'interface d'administration, par exemple en mode de test ou en mode de production (figure 6.3).

Panorama de l'appel distant

Lors d'un appel d'un objet client à un objet serveur, l'appel de méthode va transiter par le *stub*, puis par l'ORB local qui communique avec l'ORB distant, puis enfin attaque le serveur au travers du POA (figure 6.4).

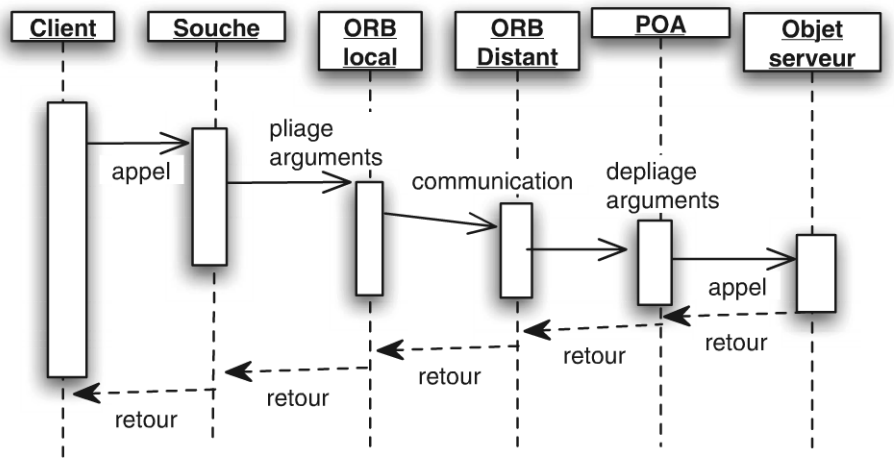


Figure 6.4 — Cheminement d'un appel CORBA

© Dunod – Toute reproduction non autorisée est un délit.
Copyright © 2015 Dunod.

6.2.4 L'IDL

L'IDL assure l'interopérabilité des langages

Comme en RMI, un objet serveur va exposer son interface. Comme cette interface doit être commune à tous les langages il n'est pas possible cette fois d'utiliser les interfaces Java. On utilise donc un langage spécifique, l'IDL (*Interface Definition Language*). C'est le langage qui permet de définir les interfaces des objets du bus CORBA de façon indépendante de l'implantation. Les interfaces de l'ORB, des objets clés de CORBA et des services sont, entre autres, spécifiées dans ce langage par l'OMG.

Pour assurer l'interopérabilité entre langages, il faut trouver un plus petit commun dénominateur entre tous les types de données. Par exemple Java n'implémente pas les entiers non signés, alors que C les prend en compte. À l'origine, le modèle de données a été donné par C. En dehors des types primitifs CORBA présente un modèle de composition de types tout à fait spécifique. Pour les services web, ce modèle de composition de données utilise des schémas XML.

Il est aussi possible de personnaliser les types CORBA en définissant des **typecodes** qui permettent de définir des correspondances arbitraires entre types de données d'une machine à l'autre.

Les types primitifs

Voici une bijection entre les types primitifs de CORBA et les types primitifs Java :

Java	CORBA
`void`	`void`
`boolean`	`boolean`
`char`	`char/wchar`
`byte`	`octet`
`short`	`short/ unsigned short`
`int`	`long/ unsigned long`
`long`	`long long /unsigned long long`
`float`	`float`
`double`	`double`
`String`	`string, wstring`

L'IDL permet de définir des constantes :

```
const nom = valeur ;
```

Copyright © 2015 Dunod.

Les types complexes

On peut définir des types complexes pour passer les arguments. On peut définir un nouveau type abstrait de données :

```
typedef long duree;
```

Mais aussi des types complexes. En voici quelques-uns :

- Les énumérations ne se traduisent pas par des énumérations Java mais se décrivent ainsi :

```
enum mois {janvier,fevrier};
```

- Les structures ressemblent au C et permettent de passer des informations par valeur :

```
struct erreur {
    long date;
    string message;
};
```

- Les exceptions sont très proches des structures :

```
exception creditDepasse { long montant ; };
```

- Les tableaux posent souvent des problèmes d'interopérabilité entre langages, qui transposent parfois lignes et colonnes. C'est l'ORB qui va vérifier la taille effective :

```
typedef float vecteur[10] ;
```

- Les séquences sont des tableaux sans limites. L'ORB effectue alors la réservation mémoire voulue :

```
typedef sequence <long> historique ;
```

- Il existe en fin un type anonyme appelé `any` qui peut contenir n'importe quel type de donnée.

Les modules et les interfaces

Les **modules** sont l'équivalent des packages Java et permettent de grouper des interfaces. Les interfaces décrivent naturellement la signature des opérations exposées par un objet distant :

Copyright © 2015 Dunod.
© Dunod – Toute reproduction non autorisée est un délit.

```
module finance {
    exception creditDepasse {};
    interface banque {
        attribute double montantMaximumAutorise ;
        void depot(in double montant) ;
        void retrait(in double montant) raises (creditDepasse);
        double solde() ;
    };
};
```

Les attributs se traduisent par des fonctions d'accès en lecture et écriture. Les arguments des méthodes peuvent être en entrée (*in*), en sortie (*out*) ou en entrée-sortie (*inout*). On aura ainsi après génération de code à partir de l'IDL l'interface suivante dans le package `finance` :

```
public interface banqueOperations
    {double montantMaximumAutorise ();
    void montantMaximumAutorise (double newMontantMaximumAutorise);
    void depot (double montant);
    void retrait (double montant) throws finance.creditDepasse;
    double solde ();
} // interface banqueOperations
```

Helper et Holder

Comme les arguments peuvent être passés en IDL soit en entrée (*in*) soit en sortie (*out*), soit les deux, il faut un conteneur typé pour les arguments `out` et `inout`. Ce conteneur s'appelle `Holder`. Pour chaque objet distant, et pour chaque type complexe définit dans l'IDL on aura une classe de type `Holder`, générée automatiquement lors de la compilation de l'IDL.

Le `Helper` va permettre de décoder la donnée dans le flux de la requête et effectuer le dépliage des arguments et des références. L'appel de la méthode `narrow` sur le `Helper` se comporte exactement comme un `cast` en RMI :

```
banque mabanque = banqueHelper.narrow(reference);
```

6.2.5 Les IOR

Les références d'objets sont appelées IOR (*Interoperable Object Reference*) en CORBA. Ce sont des identifiants uniques sur le réseau permettant de trouver un objet de manière univoque, indépendamment de l'implémentation de l'ORB. C'est un problème récurrent en applications réparties de générer des identifiants uniques, et l'on trouve cela également dans les GUID des ActiveX de plate-forme Windows.

En CORBA, les IOR comportent de manière codée l'adresse de la machine et le port d'écoute, ainsi que des clés spécifiques à l'objet distant. Les IOR peuvent

Copyright © 2015 Dunod.

également contenir des profils pour une personnalisation plus poussée par exemple du protocole sous-jacent. Il existe un utilitaire en ligne pour disséquer un IOR[1].

Exemple

```
IOR:000000000000001b49444c3a506c6163654d61726368652f426f757273653a312e30000
0000000100000000000000860001020000000000c3139322e3136382e312e3300c02b000000
000031afabcb0000000020391f165b0000000100000000000000010000000 8526f6f74504f4
1000000000800000001000000001400000000000000200000001000000200000000000010001
0000000205010001000100200001010900000001000101000000002600000000020002
```

Si on le compare à une URL d'un service web, il est clair que cette référence ne peut être transmise que par un fichier ou un service d'annuaire qui va stocker la référence sous un nom plus facile. En CORBA, on parlera de service de noms (*Naming Service*). Toutefois les dernières versions de CORBA permettent d'utiliser des pseudo-URL telles que `corbaloc` ou `corbaname` qui seront illustrées au paragraphe 6.5.4.

6.2.6 Les services CORBA

Les services CORBA[2] sont des bibliothèques qui enrichissent le comportement du bus logiciel avec des fonctions supplémentaires : transactions, sécurité, notification d'événements...

Il est à noter que chaque implémentation d'ORB peut ou non implémenter ces suppléments, et les ORB vont se distinguer selon les services qu'ils implémentent. Attention le concept de service ici n'a rien à voir avec le concept de service au sens de services web(voir chapitre 9). Il s'agit plutôt d'un ajout de *qualité de service*. Voici une liste des services les plus courants :

- **Naming Service** : service d'annuaire ou service de nom, similaire à `rmiregistry`, le service le plus utilisé.
- **Event Service, Notification Service :** le service d'événement est assez utilisé, notamment en télécommunications, et permet d'envoyer des événements asynchrones entre applications. Le service de notification le complète avec une notion de qualité de service.
- **Trader Service** : le service de courtier permet d'implémenter un mode de recherche par propriétés et non par nom comme un service de pages jaunes.
- **Transaction Service** : le service de transaction permet de fiabiliser les appels distants.
- **Persistent State Service** : ce service permet de rendre l'état d'un serveur persistant entre deux activations. Ceci est similaire à l'utilisation de RMI avec des `Marshalled Object`.

1. http://www2.parc.com/istl/projects/ILU/parseIOR/
2. http://www.omg.org/spec/# CS

Copyright © 2015 Dunod.
© Dunod – Toute reproduction non autorisée est un délit.

- **Concurrency Service** : ce service permet de gérer les accès concurrents à un serveur, notamment en posant des verrous.
- **Security Service** : permet de gérer les accès sécurisés aux serveurs, l'identification, l'intégrité et la confidentialité des messages.

Il existe bien d'autres services spécifiés. [Daniel] cite les suivants : *Collection service*, *Property Service*, *Enhanced View of Time*, *Query Service*, *Relationship Service*, *Externalization Service*, *Time Service*, *Licensing Service*, *Life Cycle Service*.

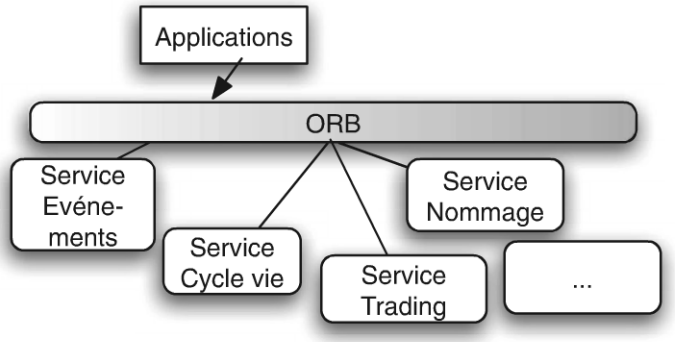


Figure 6.5 — Les services CORBA rajoutent des fonctionnalités à l'ORB

Les services (figure 6.5) sont issus de travaux de normalisation et d'un consensus entre vendeurs : certains sont très utilisés et répandus, comme le service de Notification mais ils ont des fortunes diverses, et certains sont trop complexes pour avoir une large diffusion, comme le service de Transaction ou le service de Trader. On notera toutefois que l'API JTA (*Java Transaction API*) d'Oracle s'inspire du service de Transaction CORBA, en le simplifiant.

Alors que le modèle CORBA est l'un des plus évolués en termes de services, on comprend encore mal comment combiner des services au sein d'un même composant. Ce problème se pose aussi pour SOAP (voir chapitre 9).

6.2.7 Utilisation de CORBA en Java

CORBA est à présent inclus comme une bibliothèque standard dans J2SE. Java fournit également un outil pour démarrer le service d'annuaire, `orbd`, qui remplace l'ancien `tnameserv`. L'outil `idlj` permet enfin de générer les souches CORBA à partir des fichiers d'IDL.

Copyright © 2015 Dunod.

RMI/IIOP et CORBA IDL

Il y a deux modes d'utilisation de CORBA vraiment différents :

- soit l'on dispose d'un IDL existant, provenant par exemple d'un programme avec lequel on doit s'interfacer, et dans ce cas on utilise CORBA IDL, avec la démarche classique de génération de souches à partir de l'IDL[1].
- soit au contraire on part d'un programme Java que l'on souhaite rendre disponible à d'autres applications CORBA en mode serveur. Dans ce cas, on pourra utiliser la démarche liée à RMI/IIOP. RMI IIOP est décrit à la fin du chapitre 5.

6.3 UN EXEMPLE STATIQUE SIMPLE : CLIENT ET SERVEUR EN JAVA

Dans l'exemple qui suit, on va créer un client et un serveur en Java. Les étapes seront les suivantes :

- Écriture de l'IDL
- Génération des classes côté client
- Génération des classes côté serveur
- Implémentation du code serveur
- Implémentation du code client

6.3.1 IDL et classes générées

IDL

Comme dans toutes les communications RPC, on commence par définir l'interface exposée aux clients distants. Ici il s'agit d'une bourse qui renvoie une cotation sous forme de réel à partir du nom de la société :

```
module PlaceMarche {
    interface Bourse {
        double obtenirPrix( in string symbol );
    };
};
```

On remarquera la propriété `in` attachée au paramètre de l'opération, à la mode du langage Ada.

1. https://docs.oracle.com/cd/E28280_01/web.1111/e13717/corba_idl.htm# SACLT202

© Dunod – Toute reproduction non autorisée est un délit.
Copyright © 2015 Dunod.

Génération des classes à partir de l'IDL côté client

L'interface va à présent permettre de générer les classes nécessaires au client : la souche, qui représentera l'objet distant sur le client, l'interface Java, ainsi que toutes les classes `Holder` et `Helper` des types complexes. On utilise pour cela l'outil `idlj` avec l'option `-fclient`. En CORBA on ne peut plus se reposer sur la sérialisation pour empaqueter les types d'objets que l'on aurait la paresse de définir comme des objets distants : il faut décrire absolument tous les types de données qui transitent par le réseau. L'utilisation de `idlj` peut ainsi générer un grand nombre de classes.

```
idlj -fclient banque.idl
```

Cette commande engendre les fichiers suivants :

- `BourseOperations`
- `BourseHolder`
- `BourseHelper.`
- `Bourse.java`
- `_BourseStub.java`

L'interface Java à proprement parler s'appelle `Bourse`. Elle hérite de `BourseOperations`. On remarquera que contrairement à RMI, l'interface `BourseOperations` est tout à fait standard et ne lance pas de `RemoteException`. La hiérarchie est décrite à la figure 6.6.

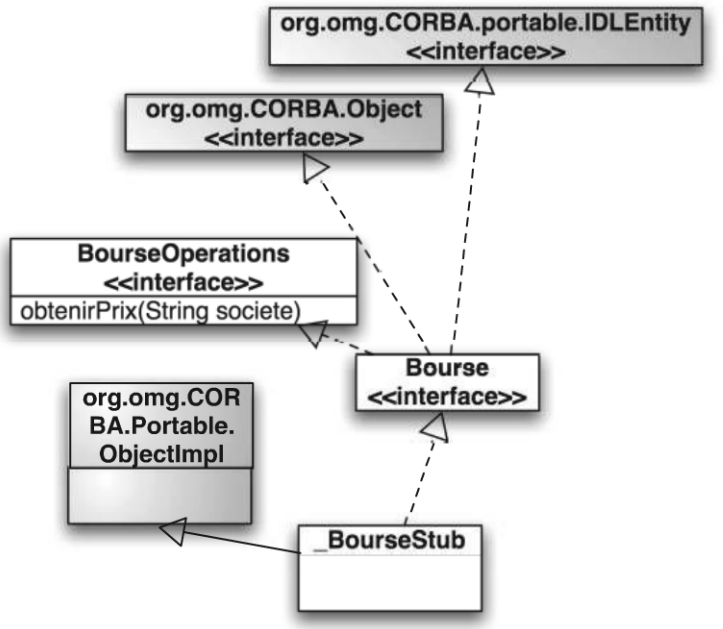


Figure 6.6 — Hiérarchie des interfaces CORBA pour le client

Copyright © 2015 Dunod.

Génération des classes côté serveur

On utilise de même la commande `idlj` avec l'option `-fserver` ou `-fall` pour toutes les classes (*all*) :

```
idlj -fall Bourse.idl
```

En plus des classes générées pour le client on trouvera la classe `BoursePOA`, qui va servir l'objet à implémenter. `BoursePOA` hérite de `org.omg.CORBA.portable.InvokeHandler` qui implémente la méthode :

```
_invoke(String method, InputStream input, ResponseHandler handler)
```

Cette méthode fait furieusement penser aux proxies dynamiques (voir chapitre 1) utilisés à présent en RMI et à l'interface `java.lang.reflect.InvocationHandler` :

```
invoke(Object proxy, Method method, Object[] args)
```

Ceci permet d'expliquer pourquoi un POA peut servir plusieurs objets : en effet l'interface est intrinsèquement dynamique !

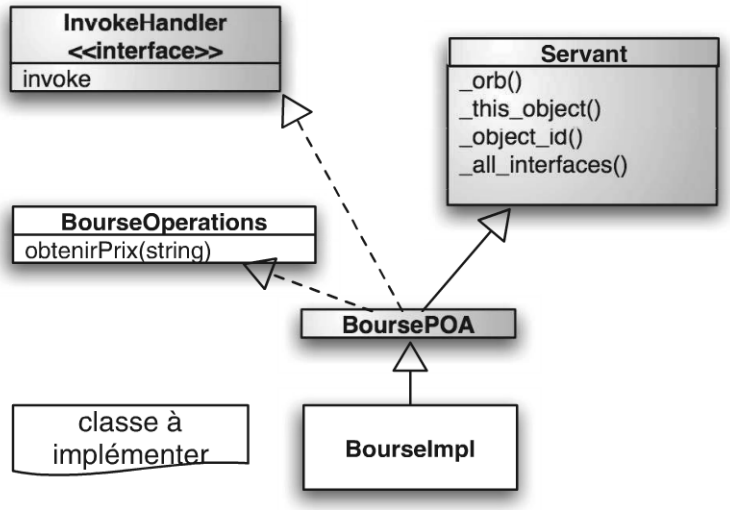


Figure 6.7 — Implémentation du serveur par héritage

Le diagramme UML qui représente ces différents objets est décrit dans la figure 6.7. On notera que `BoursePOA` implémente non pas l'interface `Bourse` mais l'interface `BourseOperations`.

On s'assure également en regardant le code que le servant `BoursePOA` est bien responsable du pliage des arguments. Ce code est généré et les variables ne sont pas toutes présentées ici. Il suffit de comprendre qu'à un moment donné la souche va lire les arguments et écrire la réponse dans le flux de sortie :

Copyright © 2015 Dunod.
© Dunod – Toute reproduction non autorisée est un délit.

```
String symbol = in.read_string ();
double $result = (double)0;
$result = this.obtenirPrix (symbol);
out = $rh.createReply();
out.write_double ($result); ...
```

6.3.2 Implémentation des classes

Implémentation de l'objet distant

L'implémentation de l'objet distant est minimaliste :

```
package PlaceMarche;
public class BourseImpl extends BoursePOA {
public double obtenirPrix(String symbol) {
return 45.0;}}
```

On remarquera encore qu'aucun constructeur spécial n'est requis, contrairement à RMI.

Écriture du code du serveur

Dans cet exemple, le serveur va mettre à disposition l'objet distant simplement en publiant son IOR. Nous verrons par la suite comment utiliser le service d'annuaire, qui est plus souvent utilisé. Mais l'exportation d'IOR est une solution rustique qui permet souvent de tester l'interopérabilité de deux ORB dans les premières phases de maquettage.

```
// Initialisation de l'ORB
ORB orb = ORB.init(args, null); // args provient du main, et peut être
                                // un tableau de String vide new String[]
// Création de l'objet Bourse
BourseImpl brongniart = new BourseImpl();
```

On doit à présent faire appel au POA pour obtenir une référence et activer le `POAManager` :

```
// Activation du POAManager
POA rootpoa =POAHelper.narrow(orb.resolve_initial_references("RootPOA"));
rootpoa.the_POAManager().activate();
// On obtient une référence sur l'objet du servant
org.omg.CORBA.Object ref = rootpoa.servant_to_reference(brongniart);
```

On doit typer cette référence avec l'interface, et là intervient l'objet `Helper` :

```
Bourse href = BourseHelper.narrow(ref);
```

Copyright © 2015 Dunod.

On récupère l'IOR de l'objet à partir de sa référence et on stocke l'IOR dans un fichier :

```
OutputStream f = new FileOutputStream(System.getProperty("user.home")+
    System.getProperty("file.separator")+"Bourse.ior");
DataOutputStream out = new DataOutputStream(f);
String ior = orb.object_to_string( (Object) href);
out.writeBytes(ior);
out.close();
}
```

On doit finir par une commande bloquante, pour éviter que le serveur ne termine :

```
orb.run() ;
```

En RMI pour rappel, le code de mise à disposition de l'objet distant avec enregistrement dans `rmiregistry` est bloquant. On peut aussi écrire un code bloquant si l'`orb` a déjà été lancé avec attente d'une touche ou avec le code suivant :

```
java.lang.Object sync = new java.lang.Object();
    synchronized (sync) {
        sync.wait();}
```

Écriture du code client

Le code client va consister à initialiser l'ORB, à récupérer la référence dans l'IOR et à effectuer un appel distant :

- Création et initialisation de l'ORB :

```
ORB orb = ORB.init(args, null);
```

- Récupération de la référence IOR dans le fichier :

```
InputStream f = new FileInputStream(System.getProperty("user.home")+
    System.getProperty("file.separator")+"Bourse.ior");
BufferedReader reader= new BufferedReader(new InputStreamReader(f));
    String ior reader.readLine();
```

- On peut alors tout simplement récupérer l'objet et effectuer l'appel distant :

```
Bourse brongniart=(Bourse) orb.string_to_object(ior);
System.out.println(brongniart.obtenirPrix("Orange"));
```

Copyright © 2015 Dunod.
© Dunod – Toute reproduction non autorisée est un délit.

6.4 VARIANTES DE CODE

6.4.1 Implémentation par délégation et non par héritage

Héritage et délégation

Dans certaines applications, il peut être impossible d'implémenter un serveur par héritage d'une classe POA. On va alors utiliser un mécanisme par délégation[1]. On parlera de `Tie` pour désigner le conteneur de l'objet « lié » à l'ORB.

Le code du client est le même que précédemment.

Génération de code sur le serveur

On commence par générer les éléments nécessaires au serveur, en partant toujours du même IDL :

```
idlj -fallTIE bourse.idl
```

On observe alors une nouvelle classe, `BoursePOATie`, qui hérite de `BoursePOA` et comporte une variable d'instance `_delegate`, ainsi qu'un pointeur vers le POA. Le diagramme de classes UML est modifié comme le montre la figure 6.8.

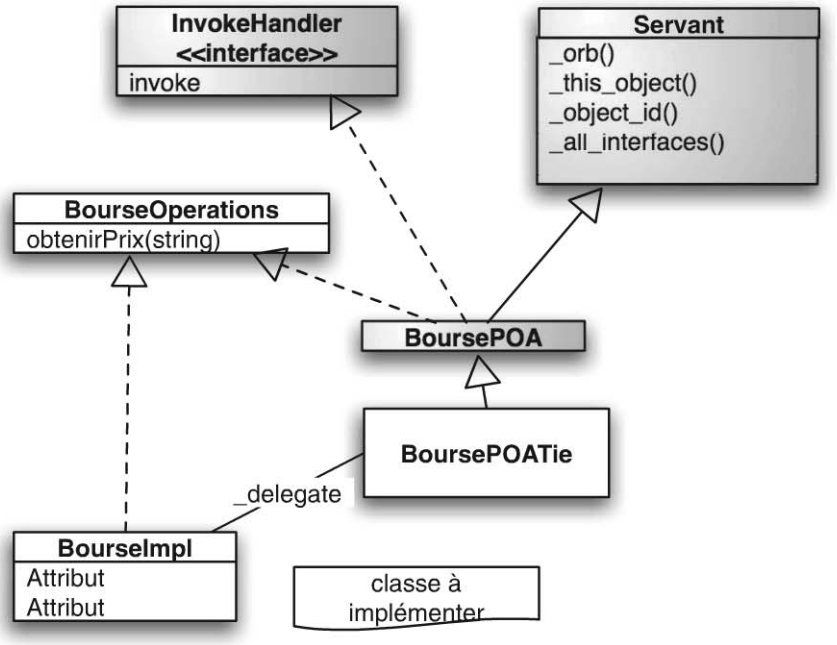


Figure 6.8 — Modification de la hiérarchie d'héritage

1. http://docs.oracle.com/javase/7/docs/technotes/guides/idl/jidlTieServer.html

Copyright © 2015 Dunod.

La classe à implémenter, `BourseImpl`, n'a plus de nécessité d'hériter de `BoursePOA`. Elle doit toutefois continuer à implémenter les opérations de l'interface :

```
public class BourseImpl implements BourseOperations {
    public double obtenirPrix(String symbol) {
    return 45.0;
}}
```

Mise à disposition dans le serveur

La manière dont on obtient la référence est modifiée. On crée d'abord le `Tie` en lui passant l'implémentation en argument, ainsi que le POA.

La référence est obtenue à partir de l'ORB :

```
BourseImpl brongniart = new BourseImpl();
BoursePOATie tie = new BoursePOATie(brongniart, rootpoa);
//On obtient la référence et on active l'objet
Bourse href = tie._this(orb);
String ior = orb.object_to_string( (Object) href); ....
```

6.4.2 Appel dynamique d'un client

Motivation pour l'appel dynamique

Dans l'exemple qui suit le code du serveur ne change pas.

Nous avons vu que le POA permet de recevoir les appels dynamiques. Le client peut également faire l'économie de la souche et appeler la méthode à distance par construction de requête, et non appel direct de la méthode. Ceci peut être utile dans de nombreux cas :

- pour éviter de télécharger de nombreuses souches sur un grand nombre de clients et simplifier les problèmes de déploiement ;
- lorsqu'un client fait appel à de nombreux serveurs, et que l'on souhaiterait « factoriser » les souches ;
- lorsque l'interlocuteur est un langage à typage dynamique tel que Smalltalk, mais peut-être aussi Python et Lisp ;
- lorsque l'interface n'est pas encore figée et en mode de développement. Il est alors inutile de recompiler le client à chaque modification.

La dégradation des performances en mode dynamique n'est pas significative, et l'OMG a même produit des benchmarks en faveur du mode dynamique. Rappelons que la réflexion a été optimisée depuis Java 5[1].

1. Voir chapitre 2 pour des rappels sur la réflexion en Java.

© Dunod – Toute reproduction non autorisée est un délit.
Copyright © 2015 Dunod.

L'implémentation de l'appel

Le pivot d'un appel dynamique est une liste de la classe `NVList` contenant un tuple pour chaque argument. Ces tuples de classe `NamedValue` comportent trois informations :

- le nom de l'argument,
- sa valeur de type `any` (joker qui remplace tous les types),
- son sens de passage, `IN`, `OUT` ou `INOUT`.

On commence tout d'abord par initialiser l'ORB :

```
ORB orb = ORB.init (args, null);
```

On récupère l'objet comme précédemment :

```
org.omg.CORBA.Object brongniart= orb.string_to_object(ior);
```

On va à présent créer la liste des arguments de taille 1 (un seul argument) :

```
NVList argList = orb.create_list (1);
Any argument = orb.create_any ();
argument.insert_string("Orange"); // la valeur de l'argument
    // ajout du tuple dans la liste d'arguments
argList.add_value ("symbol", argument, org.omg.CORBA.ARG_IN.value);
```

On fait de même pour les résultats. Cette fois par contre le mode est `OUT` et non `IN` :

```
Any result = orb.create_any ();
result.insert_double (0); // cette valeur sera remplacée par le résultat
NamedValue resultVal = orb.create_named_value ("resultat", result,
    org.omg.CORBA.ARG_OUT.value);
```

Il suffit alors d'invoquer la requête et de récupérer les résultats. La création de la requête utilise comme arguments le nom de la méthode `obtenirPrix`, la liste des arguments `argList`, le nom de la variable de retour `resultVal`, ainsi qu'un contexte d'appel contenant des propriétés ici à `null`.

```
Request requete = brongniart._create_request (null, "obtenirPrix", argList,
    resultVal);
requete.invoke ();
resultat = requete.result().value ();
System.out.println ("Prix : " + resultat.extract_double ());
```

6.5 UTILISATION D'UN ANNUAIRE

Généralement, il n'est pas souhaitable d'échanger des fichiers d'IOR sur le réseau. On utilise comme en RMI un service d'annuaire ou service de noms (*Naming Service*).

Copyright © 2015 Dunod.

L'IDL du service d'annuaire est normalisé par CORBA, et depuis CORBA 2.4 on peut utiliser des services d'**annuaires interopérables** : plusieurs ORB de vendeurs différents partageront alors le même annuaire.

6.5.1 Utilisation de l'annuaire côté serveur

Dans l'exemple qui suit on reprend le code du serveur de bourse. Pour utiliser l'annuaire il faut tout d'abord lancer le service d'annuaire de Java : `orbd`

```
orbd -ORBInitialPort 1050
```

Par défaut, le port est le 900, mais sous Unix on ne peut le lancer qu'en tant qu'administrateur. On prend donc le port 1050 pour cet exemple. On doit spécifier lors du lancement de l'ORB quel est le port et la machine utilisés par l'annuaire. On remarquera que contrairement à RMI où l'annuaire doit être local au serveur, l'annuaire CORBA peut être distant.

```
Properties properties = System.getProperties();
properties.put( "org.omg.CORBA.ORBInitialHost","localhost" );
properties.put( "org.omg.CORBA.ORBInitialPort","1050" );
ORB orb = ORB.init(args, properties);
```

On obtient ensuite une référence sur l'annuaire, exactement de la même manière que l'on obtient une référence sur le POA :

```
NamingContextExt annuaireDistant =
NamingContextExtHelper.narrow(
    orb.resolve_initial_references("NameService"));[1]
```

Pour mémoire, la manière dont on obtient une référence sur le POA est la suivante :

```
rootpoa = POAHelper.narrow(orb.resolve_initial_references("RootPOA"));
```

On crée une référence sur l'objet comme précédemment :

```
BourseImpl brongniart = new BourseImpl();
Bourse referenceBrongniart =
    BourseHelper.narrow(rootpoa.servant_to_reference(brongniart););
```

Il suffit comme en RMI de lier le nom à la référence de l'objet.

```
String nom = "Palais Brongniart";
NameComponent[] chemin = annuaireDistant.to_name(nom);
annuaireDistant.rebind(chemin, referenceBrongniart);
```

1. `NamingContextExt` est la version interopérable de `NamingContext`.

Copyright © 2015 Dunod.
© Dunod – Toute reproduction non autorisée est un délit.

6.5.2 Hiérarchie de contextes

La structure des noms en CORBA est hiérarchique. Un nom est donc un tableau de `NameComponent`, de la racine des contextes à la feuille désignant l'objet. Un `NameComponent` est une structure de données simple comportant un nom et un type. On peut ainsi créer les espaces de nommage par exemple pour séparer les références des différentes applications (figure 6.9).

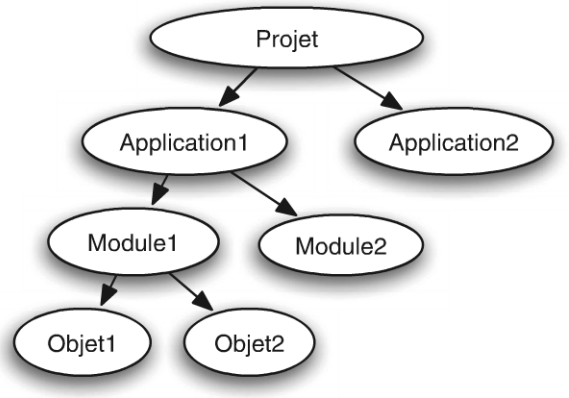


Figure 6.9 — Hiérarchie de contextes de nommage

```
String nom = "Palais Brongniart";
NameComponent[] chemin=new NameComponent[2];
chemin[0]=new NameComponent("application1","");
chemin[1]=new NameComponent(nom,"");
annuaireDistant.rebind(chemin, referenceBrongniart);
```

On peut alors constater la création d'un répertoire local `orbd.db` avec des fichiers représentant chacune des références liées.

6.5.3 Code du client

Le code du client est très similaire. On initialise les propriétés de l'ORB comme précédemment. On récupère l'annuaire comme précédemment. On utilise seulement la méthode `resolve` au lieu de `bind` (l'équivalent du `lookup` RMI).

```
refAnnuaire = orb.resolve_initial_references("NameService");
NamingContextExt annuaireDistant =
        NamingContextExtHelper.narrow(refAnnuaire);
// récupération de la référence de l'objet
Bourse brongniart=BourseHelper.narrow(annuaireDistant.resolve_str("Palais
Brongniart"));
System.out.println(brongniart.obtenirPrix("Orange"));
```

Copyright © 2015 Dunod.

6.5.4 Service d'annuaire interopérable

Il est possible d'adresser un annuaire d'un autre vendeur sur une autre machine en utilisant les URL `corbaloc`, par exemple :

```
corbaloc:iiop:1.2@MaBanque.com:2050/NamingService.
```

Il est également possible pour un client d'aller rechercher une URL directement sans faire appel à un `NameService` comme ceci :

```
// initialisation de l'ORB ...
Bourse bourse=BourseHelper.narrow(orb.string_to_object(
    "corbaname:iiop:1.2@localhost:1050#PalaisBrongniart"));
```

Cette URL est préfixée par `corbaname`. Vient ensuite le protocole dont on précise la version, `iiop:1.2`, puis le nom de la machine et le port d'écoute du service d'annuaire, recherché implicitement par le service de noms.

6.6 PASSAGE D'ARGUMENTS

6.6.1 Passage d'arguments par référence

Il est possible en CORBA de passer des objets par référence. Si les objets sont créés sur le client, il faudra les activer sur place avant de les passer en argument. Pour cela il faut naturellement que ces objets dérivent d'une classe POA. Le petit exemple suivant montre comment obtenir un objet `Compte` d'un objet `Banque`.

Création de l'IDL

Dans l'IDL suivant, l'interface `Banque` utilise le type de l'interface `Compte`. Attention la définition de `Compte` doit précéder la définition de `Banque` qui lui fait référence, ou alors être précédé de la déclaration `forward` :

```
interface Banque ;
```

Voici le fichier `banque.idl` :

```
module finance {
interface Compte {
double obtenirSolde( in string symbol );
long obtenirNumero();
};
interface Banque {
Compte creerCompte( in long numero );
void enregistrerCompte(in Compte compte);
};
};
```

Copyright © 2015 Dunod.
© Dunod – Toute reproduction non autorisée est un délit.

Génération du code côté serveur

On utilise naturellement :

```
idlj -fall banque.idl
```

Et cette fois on obtient les classes pour `Banque` et `Compte`.

On implémente `BanqueImpl` et `CompteImpl` par exemple comme ceci :

```
public class CompteImpl extends ComptePOA {
    public int obtenirNumero() {
        return 1234;}
    public double obtenirSolde(String symbol) { return 1000.0; }
}
```

Et voici l'implémentation de la `Banque` :

```
public class BanqueImpl extends BanquePOA {
    private Vector comptes= new Vector(); // Vector est synchronisé
    public Compte creerCompte(int numero) {
        try{
// Création de l'objet Compte
    CompteImpl compte = new CompteImpl();
    org.omg.CORBA.Object ref =
    Compte referenceCompte =
        CompteHelper.narrow(_poa().servant_to_reference(compte));
    return referenceCompte ;
        }catch(Exception e) {...return null ;}
}
public void enregistrerCompte(Compte compte) {
    comptes.add(compte);
    System.out.println(compte.obtenirNumero());}} // appel distant
                                                  // sur le client
```

On remarquera l'utilisation de `_poa()` puisque la banque a déjà un POA, inutile de revenir à la racine des POA (`rootPOA`).

Code du client

Le code du client doit cette fois faire appel au POA. On initialise l'ORB comme précédemment :

```
Properties properties = System.getProperties();
properties.put( "org.omg.CORBA.ORBInitialHost","localhost" );
properties.put( "org.omg.CORBA.ORBInitialPort","1050" );
ORB orb = ORB.init(args, properties);
```

Mais cette fois on doit rechercher le POA et activer le POAManager en local sur le client :

Copyright © 2015 Dunod.

```
POA rootpoa =
    POAHelper.narrow(orb.resolve_initial_references("RootPOA"));
rootpoa.the_POAManager().activate();
```

On obtient une référence sur la banque :

```
Banque banque=BanqueHelper.narrow(
orb.string_to_object(
    "corbaname:iiop:1.2@localhost:1050#BanqueHasard"));
```

Et on peut alors créer le compte et le passer en argument :

```
CompteImpl compte= new CompteImpl();
rootpoa.activate_object(compte);
Compte ref = CompteHelper.narrow(rootpoa.servant_to_reference(compte));
    banque.enregistrerCompte(ref);
```

On remarquera que le compte réside sur le client. Le seul moyen pour une autre application d'obtenir une référence est de passer par la banque : en effet le compte n'est pas dans le service d'annuaire.[1]

Lorsque le client appelle la méthode banque.`enregistrerCompte`, la banque affiche sur le serveur le numéro du compte comme le montre la figure 6.10.

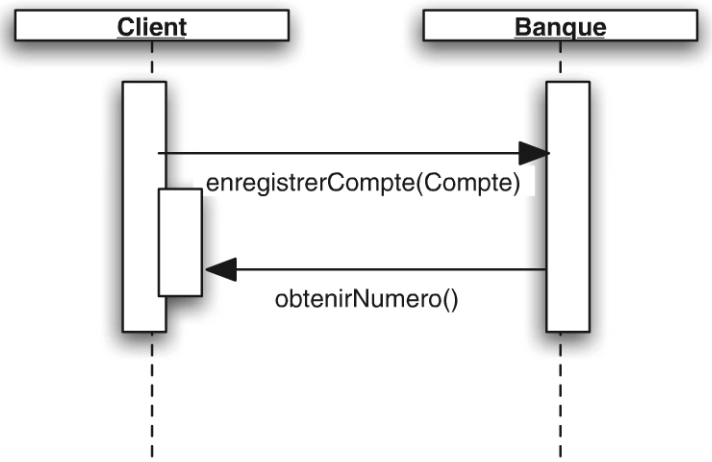


Figure 6.10 — Appel du serveur en retour sur le client

1. C'est un cas similaire aux objets Modele et Entrepot du chapitre 5 sur RMI.

Copyright © 2015 Dunod.
© Dunod – Toute reproduction non autorisée est un délit.

6.6.2 Passage des arguments par valeur

Depuis la version 2.3, il est possible de passer des arguments par valeur, un peu comme la sérialisation en RMI. Pour cela on utilise un type de données particulier appelé `ValueType`.

Définition de l'IDL

On reprend à peu près l'IDL précédent, mais cette fois `Compte` n'est plus une interface mais un `ValueType`.

```
module finance {
valuetype Compte {
    long obtenirNumero();
    factory init(in long numero, in double solde);
};
interface Banque {
    Compte creerCompte( in long numero );
};};
```

On remarquera également la définition d'une `Factory` pour créer des objets de type `Compte` (voir *Design pattern de la Factory* au chapitre 2).

La génération du code par `idlj` crée une classe `Compte` qui implémente `org.omg.CORBA.portable.StreamableValue`.

Elle crée également une classe `CompteDefaultFactory` qui implémente l'interface `CompteValueFactory` :

```
public interface CompteValueFactory extends
    org.omg.CORBA.portable.ValueFactory
    {Compte init (int numero, double solde); }
```

Cette fabrique est référencée par la classe `CompteHelper`. On trouvera d'autres informations dans l'aperçu du package CORBA[1].

Classes d'implémentation

Il faut cette fois implémenter `BanqueImpl` par héritage de `BanquePOA`, `CompteImpl` par héritage de `Compte`.

`CompteImpl` a une implémentation triviale :

```
public class CompteImpl extends Compte {
private int numero;
private double solde;
public CompteImpl(int unNumero, double unSolde) {
    numero=unNumero;
    solde=unSolde;
```

1. http://docs.oracle.com/javase/7/docs/api/org/omg/CORBA/package-summary.html

Copyright © 2015 Dunod.

```
}
public CompteImpl() {}
public int obtenirNumero() {return numero; }
public double obtenirSolde(String nom) {return solde;}
```

Par contre `BanqueImpl` va utiliser la `Factory` :

```
public class BanqueImpl extends BanquePOA {
public Compte creerCompte(int numero) {
    CompteDefaultFactory factory= new CompteDefaultFactory();
    return factory.init(numero, 0.0);
} }
```

Code du client

Le code du serveur est tout à fait le même que précédemment : on doit enregistrer un objet `Banque` dans l'annuaire.

Pour le client par contre, le code est identique à ce qu'il serait avec un passage par référence :

```
// Initialisation de l'ORB
// Obtention d'une référence sur la banque
Compte A = banque.creerCompte(1234);
System.out.println(A.obtenirNumero());
```

Toutefois, le résultat attendu n'est pas le bon : le numéro de compte affiché n'est pas 1234 mais 0 !

En effet, le compte est passé par valeur et non par référence. C'est donc une copie qui est envoyée, et non le compte lui-même. La sérialisation de Java n'est pas utilisée, puisque CORBA assure lui-même le pliage des arguments. Pour s'assurer de la sérialisation correcte pour CORBA, il faut implémenter les méthodes suivantes dans la classe `CompteImpl` :

```
public void _read (org.omg.CORBA.portable.InputStream istream) {
    numero=istream.read_long();} // sérialisation du numéro seul
public void _write (org.omg.CORBA.portable.OutputStream ostream) {
    ostream.write_long(numero);}
```

6.7 UTILISATION DES POLITIQUES DU POA

Le POA permet de définir des politiques particulières de mise en service des objets distants. Contrairement aux EJB avec lesquels la politique est liée à l'objet lui-même, en CORBA c'est le conteneur qui définit la politique indépendamment de l'objet : aucune implémentation particulière n'est nécessaire pour l'objet, avec une réelle orthogonalité tout comme en AOP (*Aspect Oriented Programming*).

Copyright © 2015 Dunod.
© Dunod – Toute reproduction non autorisée est un délit.

Parmi les politiques possibles, on en trouvera pour gérer :

- l'activation des threads,
- la persistance de l'état du serveur entre deux activations,
- l'unicité des identificateurs d'objets, etc.

La persistance va faire appel à un nouvel outil Java appelé `servertool`.

Écriture de l'IDL et génération des classes

On rajoute une méthode `shutdown` pour la désactivation de l'objet. Le qualificatif `oneway` permet de ne pas être bloqué en attente de retour de `shutdown` (ce qui paraît logique).

```
module PlaceMarche {
    interface Bourse {
        double obtenirPrix( in string symbol );
        oneway void shutdown();};};
```

On génère le code comme précédemment avec `idlj`.

Classes d'implémentation

Peu de modifications à `BourseImpl`, si ce n'est le rajout d'un lien vers l'ORB pour permettre le `shutdown`.

```
public class BourseImpl extends BoursePOA {
    private ORB orb;
    public BourseImpl(ORB unOrb){ orb=unOrb; }
    public double obtenirPrix(String symbol) {return 45.00; }
    public void shutdown() {orb.shutdown( false );
    // false indique qu'on n'attend pas la terminaison
}}
```

Implémentation du serveur

La grande modification ici sera la création d'un POA fils qui puisse appliquer la politique de persistance sur le servant `BourseImpl`.

```
// Initialisation de l'ORB...
POA rootPOA =
    POAHelper.narrow(orb.resolve_initial_references("RootPOA"));
// Création de la politique de persistance
Policy[] persistentPolicy = new Policy[1];
persistentPolicy[0] = rootPOA.create_lifespan_policy(
    LifespanPolicyValue.PERSISTENT);
// Association de la politique de persistance au POA fils
POA persistentPOA = rootPOA.create_POA("POApersistant", null,
    persistentPolicy );
// Activation du POAManager
persistentPOA.the_POAManager().activate( );
```

Copyright © 2015 Dunod.

On va ensuite continuer comme précédemment avec la création de l'objet `BourseImpl`, son activation dans le POA persistant et son enregistrement dans le service d'annuaire :

```
BourseImpl brongniart = new BourseImpl(orb);
persistentPOA.activate_object( brongniart );
// enregistrement dans le service de nommage
orb.run() ;
```

Code du client

Le code du client est sensiblement le même, sauf qu'il répète ses appels distants toutes les minutes en terminant à chaque fois par `shutdown`.

```
ORB orb = ORB.init(args, null);
// Utilisation des URL
org.omg.CORBA.Object obj = orb.string_to_object
    "corbaname::localhost:1050#PalaisBrongniart");
Bourse brongniart = BourseHelper.narrow( obj );
// Appel des cours toutes les minutes et extinction du serveur
while( true ) {
    double cours = brongniart.obtenirPrix("Orange");
    System.out.println("Cours de bourse: " +cours );
    System.out.println( "Shutting down Persistent Server.." );
    brongniart.shutdown( );
    Thread.sleep( 60000 );}...
```

Lancement du service d'annuaire

Cette fois-ci on rajoute un argument au service d'annuaire.

```
orbd -ORBInitialPort 1050 -serverPollingTime 200
```

L'argument `serverPollingTime` indique que l'annuaire va vérifier ici toutes les 200 ms la présence de l'objet distant persistant. S'il est en erreur, il sera relancé.

Par défaut, le paramètre est de 1 s.

Lancement du registre de persistance (servertool)

Le registre de persistance est un outil Java un peu analogue au démon `rmid` d'activation des serveurs RMI (voir chapitre 5).

Il doit être lancé sur le même port que le service d'annuaire, et après celui-ci :

```
servertool -ORBInitialPort 1050 (Unix)
start servertool -ORBInitialPort 1050 (Windows)
```

Copyright © 2015 Dunod.
© Dunod – Toute reproduction non autorisée est un délit.

Une console apparaît alors pour enregistrer l'application (pas le servant mais le code serveur lancé avec `main()`) :

```
Servertool> register -server Serveur -applicationName bourse -classpath
    cheminDuCodeServeur
```

Déroulement de l'application

À présent nous avons lancé :

- L'annuaire `orbd`
- Le service d'activation `servertool`

Il suffit alors de lancer le client. Le service d'activation va lancer automatiquement le serveur. Une minute plus tard le client réémet une requête vers le serveur. Or celui-ci a été désactivé (`shutdown`), et la première requête est mise en erreur. Le service d'activation réactive alors le serveur et la requête est réémise correctement.

6.8 INTRODUCTION À DDS

DDS (*Data Distribution Service*) a été récemment spécifié par l'OMG pour l'échange de données en réseau avec des contraintes de fiabilité et de performance. Il s'agit d'un modèle de communication de données par publication/abonnement avec des topics un peu comme JMS (voir chapitre 8) mais avec des exigences de performance et de qualité de service. Il est basé sur un modèle de communication par événements, par exemple la mise à disposition de données, asynchrone, et avec gestion de priorités.

On peut trouver une introduction ici :
http://www.slideshare.net/Angelo.Corsaro/intro-javacxx
http://www.slideshare.net/kerush/Slidesennohiddendata

et télécharger le code ici :
http://code.google.com/p/datadistrib4j/

La dernière version datait de novembre 2011.

Copyright © 2015 Dunod.

En résumé

Il resterait encore bien d'autres sujets à aborder, par exemple comment effectuer la journalisation avec des intercepteurs, comment mettre en place un service d'événement un peu analogue au publier/s'abonner de JMS (voir chapitre 8)...

Retenons que CORBA reste une technologie éprouvée qui a expérimenté de nombreux concepts de mise en œuvre (activation à la demande, politiques de service du POA, etc.) qui ont bénéficié de nombreuses contributions des membres de l'OMG. Elle demeure une technologie mal documentée mais elle reste à l'abri de la floraison de concepts conflictuels qui survient actuellement dans les services web. C'est une technologie performante et sûre, adaptée aux applications embarquées et temps réel, même si elle paraît parfois obsolète dans la presse. Elle est également adaptée à la gestion de graphes de données complexes, puisqu'elle travaille avec les références d'objets et non les valeurs.

Les services CORBA restent normalisés et interopérables. L'interopérabilité des services web (sécurité, transactions, etc.) est plus complexe.

Copyright © 2015 Dunod.

Copyright © 2015 Dunod.

7

Annuaires-LDAP-JNDI

Objectif

Le but de ce chapitre est de montrer comment une architecture répartie va se déployer autour du partage de références : l'annuaire. Tout comme dans un réseau l'attribution d'adresses est vitale pour communiquer entre machines, la gestion des références et l'organisation de celles-ci est vitale pour le déploiement et la supervision des applications réparties.

Ce chapitre abordera les sujets suivants :

– Plusieurs types d'annuaires : RMI, CORBA.

– Comment aborder les annuaires en phase de conception.

– Des techniques pour les annuaires pages jaunes.

– L'API Java JNDI[1] et l'utilisation d'annuaires d'entreprise, LDAP en particulier (*Light Directory Application Protocol*).

7.1 RÉFÉRENCES D'APPLICATIONS ET ANNUAIRES

7.1.1 Gestion des références en réseau

Un des problèmes fondamentaux pour assurer l'appel d'objets distants sur le réseau est de découvrir où se trouvent ces objets, comment y accéder et comment gérer des références distantes. Avant de pouvoir chercher le numéro de téléphone de quelqu'un dans un annuaire pour le joindre il faut naturellement connaître son nom !

1. http://www.oracle.com/technetwork/java/faq-141356.html
http://docs.oracle.com/javase/jndi/tutorial/getStarted/overview/

© Dunod – Toute reproduction non autorisée est un délit.
Copyright © 2015 Dunod.

Dans un monde d'objets, on n'appelle pas les objets par leur nom mais par une référence, un pointeur en quelque sorte sur l'emplacement mémoire. Mais cet emplacement mémoire est propre à une machine, et même à une JVM, et on ne peut plus accéder à l'objet de cette façon dans une application répartie.

Pour résoudre ce problème, il est nécessaire de nommer des objets accessibles sur le réseau afin de pouvoir les localiser. Les services d'annuaire (*Naming Service*) vont permettre d'associer et de maintenir des références distantes sur ces objets.

Les services d'annuaire fonctionnent tout comme un annuaire téléphonique : il faut enregistrer l'objet, et ces services assurent ensuite la persistance de la référence, et la recherche de celle-ci en fonction du nom de l'objet. On parlera d'un fonctionnement en mode **pages blanches**, par rapport à un mode de fonctionnement en mode **pages jaunes** où l'on va rechercher un objet non plus en fonction de son nom mais selon des critères de sélection. Tout comme dans un annuaire téléphonique, on peut faire une recherche pour trouver tous les numéros de la rubrique « restaurant » dans une ville donnée, il existera des extensions de service d'annuaires permettant de rechercher des objets non plus en fonction de leur nom mais de leurs propriétés. En anglais on distingue le concept de *Naming Service* correspondant aux pages blanches et le concept de *Directory Service* correspondant aux pages jaunes. Par anglicisme, ce livre parlera souvent de service de nommage au lieu de service d'annuaire, pour bien préciser qu'il s'agit de *Naming Service*.

7.1.2 Références distantes

Les **références distantes** sont ordinairement gérées par les bus logiciels, avec quelques différences subtiles sur le cycle de vie et le ramasse-miettes (*garbage collector*). Le bus logiciel doit assurer que la même référence pointe toujours sur le même objet physique à différents instants. Pour un serveur réel, le cycle de vie d'un objet peut être très long, comment s'assurer qu'un ramasse-miettes n'est pas passé par là ou que le serveur distant n'est pas éteint ?

La gestion du cycle de vie d'une référence est un problème central. En RMI, il est possible de gérer le ramasse-miettes de références distantes avec le DGC (*Distributed Garbage Collection*) (voir chapitre 5 sur RMI). Avec CORBA, cela est impossible, on ne peut faire aucune hypothèse sur les deux interlocuteurscar ils peuvent être de langages différents et gérer ou non la mémoire. En C et C++ par exemple, la mémoire est gérée par le programmeur contrairement à Java, ce qui donne lieu à beaucoup d'erreurs. Ce genre d'erreur peut se produire également sans utiliser d'applications réparties, tout simplement en essayant d'appeler un programme C++ depuis un programme Java avec JNI (*Java Native Interface*). .Net de Microsoft définit par exemple environ six modes de coopération entre des langages C++ avec ou sans gestion de la mémoire et des programmes C# qui gèrent automatiquement la mémoire.

Exemple des références en RMI

Dans le package `java.rmi` la classe `RemoteRef` permet de représenter une référence distante. On remarquera immédiatement qu'elle est `Serializable`, et qu'elle implémente

Copyright © 2015 Dunod.

une méthode appelée `remoteEquals`. En effet le test d'identité de références dans une application répartie est fondamental, afin de pouvoir assurer toujours le même objet derrière la même référence (voir paragraphe 4.2).

7.1.3 Points d'entrée dans le système

Un des problèmes fondamentaux pour utiliser des références distantes est de tirer le premier bout de laine de l'écheveau qui va entraîner tous les autres de proche en proche. Pour cela il faut un **point d'entrée**.

Lorsque l'on utilise des sockets, le point d'entrée est composé du nom de la machine ou de son adresse IP ainsi que du port de communication, par exemple :

```
192.168.1.0:1058
```

Nous avons vu dans le chapitre 3 sur les sockets que ce point d'entrée ne pouvait révéler qu'un seul serveur. Les services de noms vont permettre d'abriter de nombreux objets derrière le port de communication de l'annuaire qu'ils exposent comme le montre la figure 7.1. On communiquera toujours en premier avec le service de noms au travers d'une socket, avant de communiquer avec les objets en utilisant un protocole de plus haut niveau.

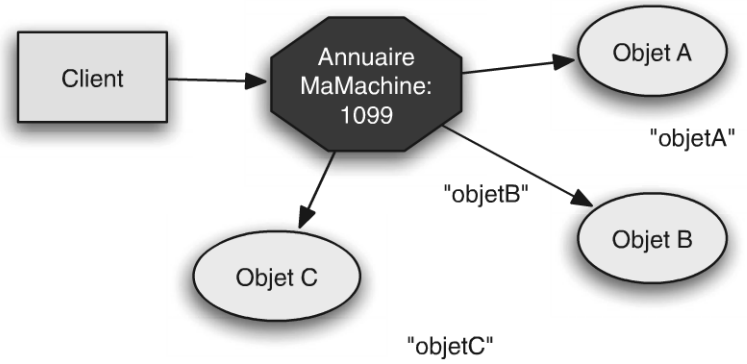


Figure 7.1 — Un service de noms contient les références de plusieurs objets distants

7.2 ENREGISTREMENT D'UN OBJET DANS UN ANNUAIRE

Afin d'être accessibles au travers d'un service de nommage, il faut enregistrer les objets dans celui-ci. On appellera souvent **serveur ou servant** le programme qui effectue cette opération et met l'objet à disposition.

© Dunod – Toute reproduction non autorisée est un délit.
Copyright © 2015 Dunod.

7.2.1 Unicité du nom

Tout comme dans un annuaire ou un numéro représente une ligne et une seule, un service de noms doit assurer qu'un nom est unique, et qu'il ne pointe que vers une seule référence.

Pour des applications simples, il est facile de répertorier tous les objets mis à disposition, mais pour une application à grande échelle, il faut envisager un plan de nommage, tout comme on assure un plan de nommage des numéros IP de machines sur le réseau ou un plan d'annuaire téléphonique.

En CORBA (voir chapitre 6), il est possible de hiérarchiser les noms dans des contextes, tout comme on peut hiérarchiser des fichiers dans des répertoires. Par exemple on peut imaginer partitionner entre applications ou entre unités de travail :

```
/MonApplication1/MonObjet1
/MonApplication2/MonObjet2
```

Cette notation hiérarchique est aussi utilisée pour les URL (*Universal Resource Locator*) sur Internet :

```
http://www.java.com
```

L'URL précédente reflète une hiérarchie implicite du sous-domaine `www` dans le domaine `java` défini dans la zone `com`. Le protocole (`http` en l'occurrence) figure en premier. Ces URL sont naturellement utilisées pour les services web, mais récemment CORBA a également introduit la notion d'URI (*Universal Resource Identifier*). Nous avons vu des exemples d'URI avec CORBA au paragraphe 6.5. Ces URI reflètent la hiérarchie des contextes de noms qui évitent les homonymes dans des applications différentes.

Nous avons vu que service de nommage de RMI se nommait `rmiregistry`, tout comme le `registry` de Windows permet de répertorier les composants ActiveX présents dans le système. Contrairement à l'annuaire CORBA, `rmiregistry` n'est pas hiérarchique, ce qui peut mener à des collisions de noms. De plus, il doit être localisé sur la même machine que le serveur RMI, contrainte qui n'existe pas en CORBA.

7.2.2 Mise à disposition d'un objet

Le servant va enregistrer l'objet sous un nom unique dans un service de nommage. En RMI, l'application servante va être bloquée en attente de requêtes. Il s'agit de la méthode classique qui permet de conserver l'objet dans un état actif. Depuis le JDK 1.4, on peut toutefois activer l'objet à la demande à l'aide d'un démon qui va le réveiller, sans laisser de programme bloqué en attente. Ce démon peut ou non assurer la persistance de l'objet distant entre deux activations (voir chapitre 5).

En CORBA, le processus d'activation est plus élaboré (voir le chapitre 6) et s'appuie sur le POA (*Portable Object Adapter*).

Copyright © 2015 Dunod.

7.2.3 Impact sur les performances

Il est fondamental lorsque l'on évalue les performances d'une application répartie de scinder l'analyse en deux étapes :

1. Le client obtient une référence sur le service de nommage.
2. Le client utilise cette référence pour communiquer avec l'objet distant.

Il n'a plus besoin du service de nommage pour communiquer. C'est en ce sens que l'on parle de « courtier » (*broker*) pour CORBA, le bus logiciel intervient dans la phase de négociation entre les objets, mais pas dans la transaction qui s'ensuit comme le montre la figure 7.2.

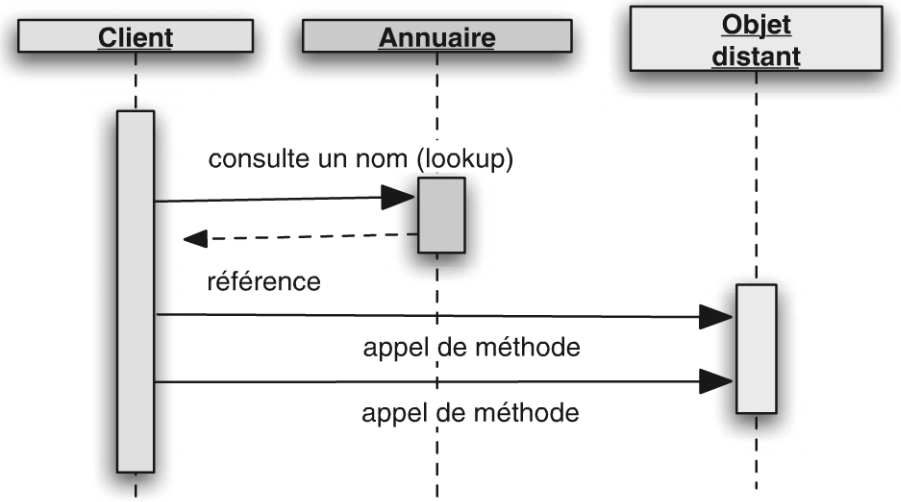


Figure 7.2 — La mise en relation entre client et objet distant utilise l'intermédiaire d'un annuaire.

Le surcoût d'un appel de méthode sur un objet distant par rapport à l'appel en local va donc comporter deux facteurs :

- l'accès au service d'annuaire,
- la communication à proprement parler.

Ordinairement, l'étape la plus longue est l'accès au service d'annuaire. Pour optimiser les performances, on pourra donc penser à mettre cette référence distante en cache, au risque d'avoir une référence qui pointera dans le vide si la référence ne vérifie pas régulièrement l'état actif du serveur. Les appels ultérieurs à cet objet éviteront la recherche coûteuse de la référence distante.

La performance va ensuite dépendre du transfert des arguments sur le réseau, en particulier de l'étape de pliage (*marshalling*). On conseillait ordinairement pour un passage d'arguments par valeur d'écrire une méthode spécifique pour la sérialisation parce que la sérialisation utilise la réflexion (voir chapitre 2) qui est lente. Les performances de la réflexion ont toutefois beaucoup progressé depuis Java 5.

Copyright © 2015 Dunod.
© Dunod – Toute reproduction non autorisée est un délit.

7.2.4 Modification de référence d'un objet

Le but du service de noms est de présenter toujours le même objet sous le même nom au client. Il est important de comprendre que cet objet va changer si sa localisation change, par exemple pour une migration sur un autre serveur, et surtout si la définition de sa classe a changé. On doit donc enregistrer à nouveau un objet dont l'interface ou la localisation a changé. Pour RMI, on doit également regénérer la souche lorsque l'implémentation change, ce qui n'est vrai ni de CORBA ni des services web. En effet la souche en RMI est générée à partir de l'implémentation et non de l'interface.

La mise à disposition d'un objet par un servant est un contrat entre le client et le serveur : dans les coulisses, le servant peut faire ce qu'il veut, en particulier jouer sur l'activation de l'objet. Dans les EJB (*Entreprise JavaBeans*), ce mécanisme permet de décharger les objets non utilisés de la mémoire sur le disque, et de recharger leurs informations sur demande.

C'est le bus logiciel qui est responsable de réactiver l'objet à la demande, et de s'assurer que la référence est proprement restaurée : tout cela doit être transparent pour le client (voir chapitres 5 et 6 sur l'activation d'objets).

Si plusieurs objets sont interchangeables et que la conversation avec un client ne modifie pas leur état, il est possible de les grouper afin de répartir la charge. Il est possible de faire ceci simplement avec des pointeurs intelligents (*smarts pointers*), qui peuvent être implémentés avec des proxies dynamiques (voir chapitre 2).

7.3 CONCEPTION D'UN SERVICE D'ANNUAIRE

7.3.1 Quels objets stocker dans le service de nommage ?

Ordinairement le service d'annuaire assure une bonne montée en charge, mais il n'est pas toujours conseillé de mettre tous les objets dans le service d'annuaire par exemple dans les cas suivants :

- Il y a une **large collection d'objets** de même nature et il est fastidieux de les nommer un par un. Par exemple, on peut imaginer des plots d'avion sur un radar de contrôle aérien qui doivent être échangés entre deux sites.
- Certains objets sont fortement dynamiques, avec un **cycle de vie court**, et il est difficile de tracer exactement leur apparition et leur disparition afin d'éviter les références mortes.
- Les objets n'exposent une interface distante que pour les *callbacks* : ils peuvent alors être passés en arguments (voir exemple du Modele au chapitre 5 ou l'exemple de la banque au chapitre 6).
- Les objets doivent être **passés en valeur** et il est plus performant de les faire transiter par le réseau que d'implémenter `n` méthodes d'accès à tous leurs champs.

Copyright © 2015 Dunod.

Accès à un grand nombre d'objets dans une collection

Prenons l'exemple d'une bibliothèque de CD : au lieu d'enregistrer chaque CD dans le service de nommage, on préférera donner accès à la collection de tous les CD et implémenter une méthode sur celle-ci permettant d'obtenir chacun des CD par son titre soit par valeur, soit par référence (figure 7.3).

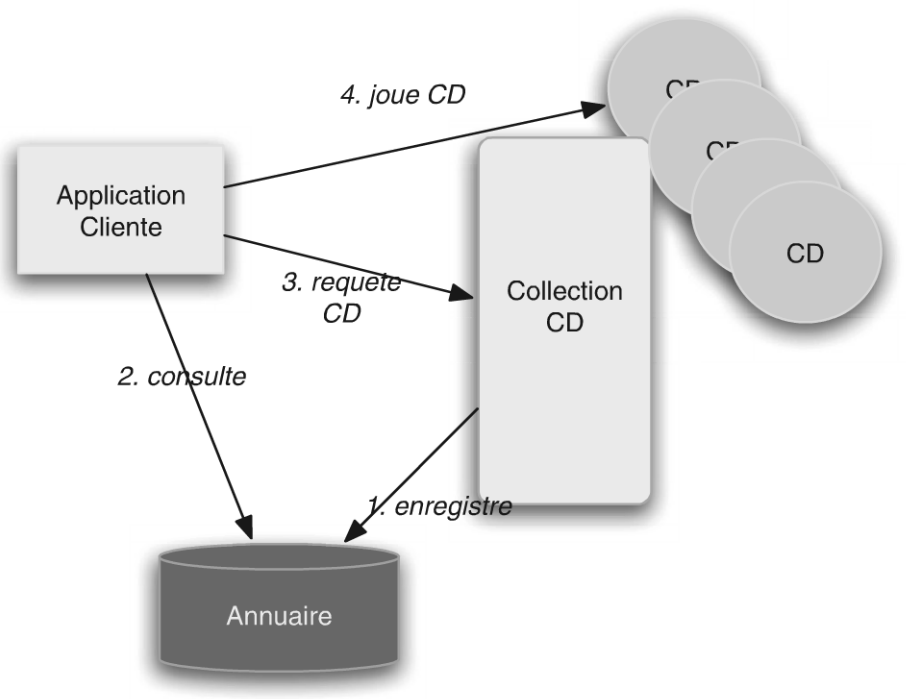


Figure 7.3 — Seule la collection figure dans l'annuaire

Fabriques à objets

Lorsque l'on veut pouvoir créer et détruire des objets à distance, il est nécessaire de passer par un intermédiaire. En effet, les classes en Java ne sont pas des objets, et il n'est donc pas possible d'enregistrer une classe dans un service d'annuaire et d'appeler le constructeur `new` sur une classe à distance. Ce serait possible si l'on avait des métaclasses (des classes de classes) !

On utilisera donc typiquement un autre objet chargé de créer les objets locaux en appelant la classe en local. On utilise le pattern de la `Factory` (voir chapitre 2), et la fabrique permettra de créer des objets. Typiquement les interfaces `EJBHome` des EJB (*Entreprise Java Beans*) servent à créer des EJB à distance.

7.3.2 Utilisation de plusieurs annuaires en parallèle

`rmiregistry` est lancé par défaut sur le port 1099, mais il est possible de modifier ce port par exemple pour faire tourner deux registres en parallèle. Pour quelle utilité ?

Copyright © 2015 Dunod.
© Dunod – Toute reproduction non autorisée est un délit.

Tout simplement pour le partitionnement d'applications à des fins de sécurité par exemple : certains objets ne seront accessibles qu'à certaines applications. Mentionnons brièvement qu'il est possible de sécuriser l'accès à ce registre à l'aide de sockets SSL (*Secure Socket Layer*), mais sans détailler le processus de création de clés de chiffrement. Voici un exemple du code à utiliser dans ce cas :

```
Registry registry = LocateRegistry.createRegistry(PORT,
new SslRMIClientSocketFactory(),
new SslRMIServerSocketFactory());
```

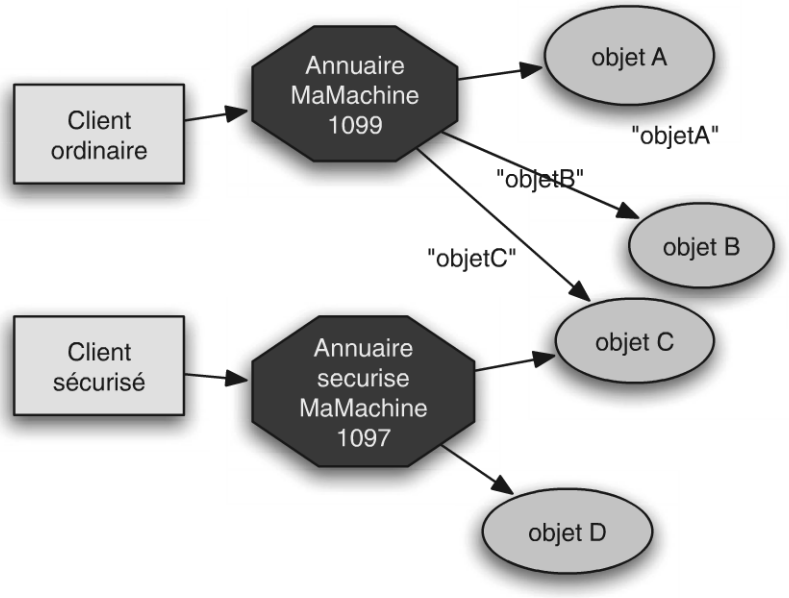


Figure 7.4 — Plusieurs annuaires peuvent avoir des droits d'accès différents

Nous voyons sur la figure 7.4 qu'il est nécessaire dans ce cas de faire une bonne partition entre les objets à sécuriser et les objets normaux. Ceci semble trivial sur le schéma, mais êtes-vous sûr de pouvoir faire une analyse de code montrant qu'un objet sécurisé ne sera jamais passé en paramètre de retour d'une méthode d'un objet non sécurisé ?

7.3.3 Utilisation de pointeurs intelligents pour référencer un ensemble d'objets

Il est très facile de représenter un ensemble d'objets de même nature avec un objet en façade. Par exemple, on souhaite représenter un centre d'appels où tous les agents ont le même rôle comme le montre la figure 7.5. On peut alors utiliser un pattern de Proxy et des proxies dynamiques (voir chapitre 2).

Copyright © 2015 Dunod.

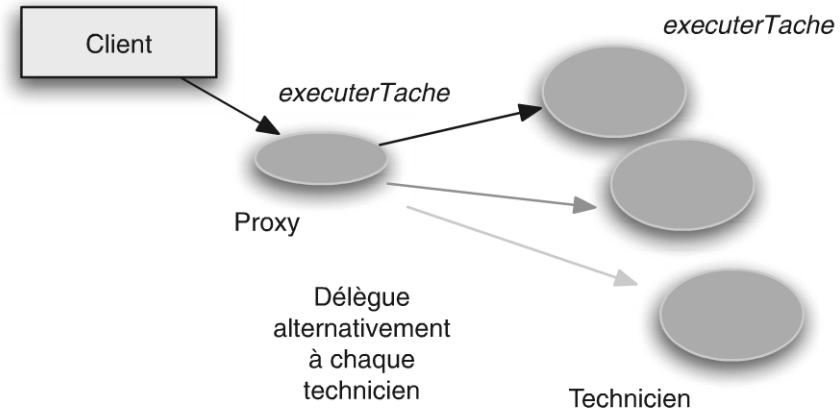


Figure 7.5 — Utilisation du pattern de proxy pour répartir les tâches

Le code pour le centre d'appels ressemblerait à ceci :

```
public String executerTache(String action) {
    ITechnicien candidat = selectionCandidat(action);
        return (String) candidat.executerTache(action); }
```

Ceci est simple tant que nous n'avons qu'une seule méthode `executerTache`, mais peut devenir fastidieux s'il faut écrire le même code pour toutes les méthodes. On pense alors naturellement à un proxy dynamique (voir paragraphe 2.4), véritable faussaire de références !

L'implémentation du `InvocationHandler` associé au proxy peut utiliser le code suivant :

```
public class GroupInvocationHandler implements InvocationHandler {
    private int numero ;
    private TechnicienInterface[] techniciens ;
    //constructeur
    public GroupInvocationHandler(TechnicienInterface[] aGroup){
        super();
        techniciens = aGroup; }
    public Object invoke(Object proxy, Method method, Object[] args)
        throws Throwable {// on appelle les techniciens à tour de rôle
        TechnicienInterface technicien=techniciens[numero];
        numero++ ;
        return method.invoke(technicien, args); }}
```

Copyright © 2015 Dunod.
© Dunod – Toute reproduction non autorisée est un délit.

7.4 JNDI

7.4.1 Service de noms et annuaire

Nous avons vu qu'un **service de noms** permet de faire correspondre un nom à une référence sur un objet. Ce service de noms va le plus souvent être hiérarchique, afin de permettre une recherche plus rapide et une isolation des différents contextes. Les annuaires sont typiquement optimisés pour la lecture des données, contrairement aux bases de données qui optimisent à la fois la lecture et l'écriture.

Un **service d'annuaire** (*Directory Service*) va enrichir un service de noms en rajoutant des attributs sur les références, afin de pouvoir effectuer des recherches non plus en mode pages blanches mais en mode pages jaunes.

L'interface **JNDI** (*Java Naming and Directory Interface*) va refléter cette double fonctionnalité en proposant au développeur essentiellement deux packages centraux :

- `javax.naming.*` pour le service de nommage ;
- `javax.naming.directory.*` pour le service d'annuaire.

Les deux interfaces pivots seront `Context` et `DirContext`, l'une pour le service de noms, l'autre pour le service d'annuaire comme le montre la figure 7.6.

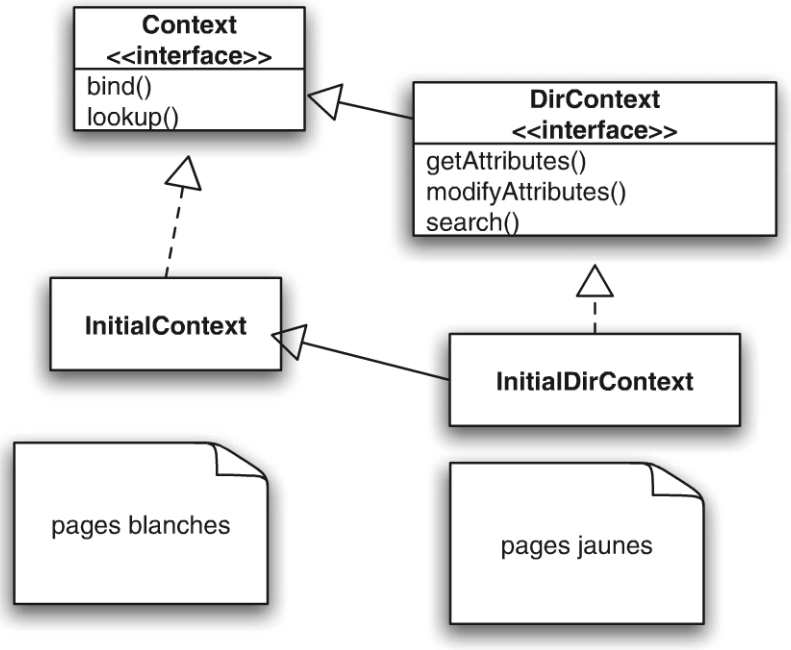


Figure 7.6 — Classes pivots de JNDI

Copyright © 2015 Dunod.

7.4.2 Les multiples implémentations de JNDI

JNDI fournit une API uniforme pour accéder à des ressources diverses dans un système comme le montre la figure 7.7 : service de nommage CORBA, `rmiregistry`, mais aussi bien d'autres annuaires tels que les annuaires LDAP (*Light Directory Access Protocol* voir paragraphe 7.6) ou Active Directory de Microsoft, le serveur DNS (*Domain Name Server*) des noms de machines en réseau, etc.

JNDI existe en standard dans J2SE, mais les implémentations font parfois appel à des tierces parties. Il existe par exemple des implémentations pour s'interfacer avec Active Directory, l'annuaire de Microsoft[1].

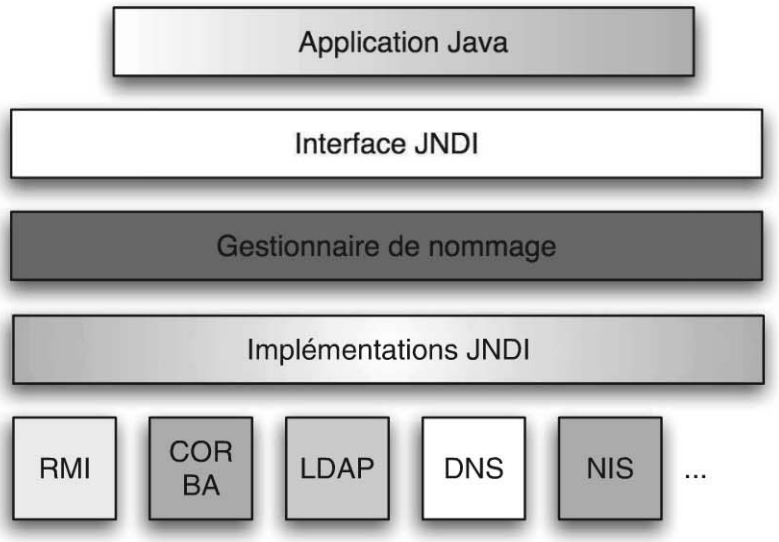


Figure 7.7 — Une seule API pour de multiples implémentations

7.4.3 Paramètres d'implémentation

Une implémentation de JNDI peut être identifiée de manière unique par deux paramètres :

- une **URL** pour le point d'entrée dans la hiérarchie de l'annuaire (`Context.PROVIDER_URL`) ;
- le **nom de la fabrique de contextes** (`Context.INITIAL_CONTEXT_FACTORY`), qui indique l'implémentation de JNDI utilisée (voir *Design pattern de la Factory* au chapitre 2).

Le point d'entrée dans l'annuaire peut être à un endroit quelconque de la hiérarchie comme le montre la figure 7.8.

1. http://www.programming-free.com/2012/09/query-active-directory-from-java-using.html

Copyright © 2015 Dunod.
© Dunod – Toute reproduction non autorisée est un délit.

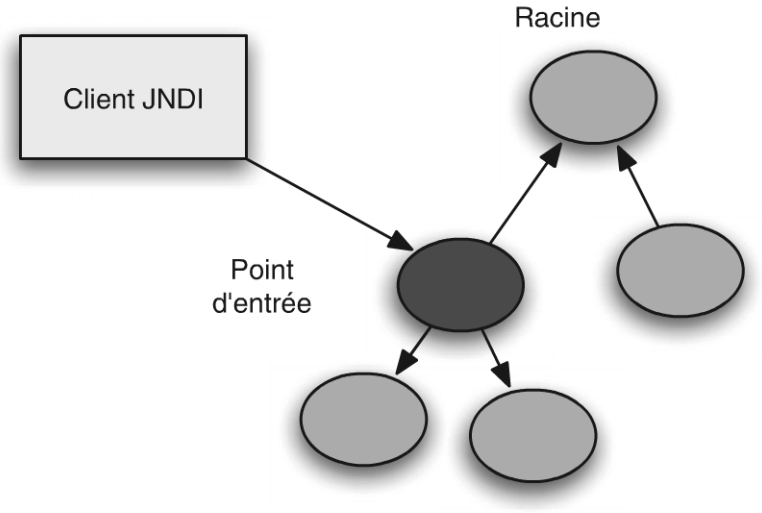


Figure 7.8 — Racine et point d'entrée peuvent être différents

Pour ce qui concerne les différents types de fabriques, le tableau suivant liste un certain nombre d'implémentations usuelles de JNDI avec le nom des fabriques de contextes associées :

DNS	`com.sun.jndi.dns.DnsContextFactory`
RMI	`com.sun.jndi.rmi.registry.RegistryContextFactory`
LDAP	`com.sun.jndi.ldap.LdapCtxFactory`
CORBA	`com.sun.jndi.cosnaming.CNCtxFactory`
File System	`com.sun.jndi.fscontext.RefFSContextFactory`

Très souvent les serveurs d'applications implémentent également leur propre annuaire sous forme de fichier ou de mini-base de données. Pour travailler avec de tels annuaires, il faut connaître le contexte initial ainsi que la classe d'implémentation.

Il est également possible de paramétrer ces éléments dans un fichier de propriétés systèmes (voir paragraphe 2.5) en utilisant les libellés suivants :

- `java.naming.factory.initial`
- `java.naming.factory.object`
- `java.naming.factory.state`
- `java.naming.factory.control`
- `java.naming.factory.url.pkgs`
- `java.naming.provider.url`
- `java.naming.dns.url`

Copyright © 2015 Dunod.

7.5 UTILISATION DE JNDI

7.5.1 Opérations de nommage

Lier et consulter les objets

Les opérations de nommage utilisent des classes `Context` qui disposent des opérations :

- `bind`, `rebind` pour lier les objets. `rebind` va écraser la référence si elle existe déjà, `bind` lèvera au contraire une exception dans ce cas. `unbind` permet de défaire une association.
- `lookup`, `list` pour consulter des objets et rechercher la référence associée.

Il est également possible de créer des sous-contextes ainsi :

```
Context sousContexte = ctx.createSubcontext("departement");
```

Exemple d'implémentation RMI

On peut ainsi utiliser JNDI pour accéder à un annuaire RMI[1]. Le contexte va être créé avec les paramètres décrits précédemment.

On suppose que `rmiregistry` est lancé :

```
LocateRegistry.createRegistry(1099);
```

On peut alors y accéder au travers de l'interface JNDI en utilisant la fabrique `RegistryContextFactory` :

```
Hashtable<String, String> env = new Hashtable(11);
    env.put(Context.INITIAL_CONTEXT_FACTORY,
    "com.sun.jndi.rmi.registry.RegistryContextFactory");
    env.put(Context.PROVIDER_URL, "rmi://localhost:1099");
Context ctx= new InitialContext(env);
Modele modele = new Modele();
ctx.bind("Modele", modele);
```

On peut également utiliser des **pseudos-URL**. Ces pseudos-URL vont résumer les deux paramètres essentiels : le protocole indique le type de fabrique, le reste de l'URL le contexte initial.

Un client RMI pourra ainsi rechercher l'objet modèle :

```
Context contexte = new InitialContext();
ModeleInterface modele = (ModeleInterface)
contexte.lookup("rmi://localhost:1099/Modele");
```

1. http://docs.oracle.com/javase/7/docs/technotes/guides/jndi/jndi-rmi.html

© Dunod – Toute reproduction non autorisée est un délit.
Copyright © 2015 Dunod.

Exemple d'implémentation CORBA

Nous avons déjà vu dans le chapitre 5 comment stocker un objet RMI IIOP dans un service de nommage CORBA. On lance démon `orbd` sur le port 1050 par exemple :

```
orbd -ORBInitialPort 1050
```

Puis, on écrit le code du serveur :

```
Hashtable env = new Hashtable();
env.put(Context.INITIAL_CONTEXT_FACTORY,
    "com.sun.jndi.cosnaming.CNCtxFactory");
env.put(Context.PROVIDER_URL,"iiop://localhost:1050");
Context ctx=new InitialContext(env);
ctx.rebind("modele", modele);
```

Ainsi pour le client il est possible également d'utiliser une pseudo-URL qui utilise le protocole `iiop` au lieu de `rmi`. Ceci serait aussi possible pour le serveur.

```
Context ctx= new InitialContext();
Object objref = (ModeleInterface) ctx.lookup("modele");
Object objref = ctx.lookup("iiop://localhost:1050/modele");
modele =(ModeleInterface) PortableRemoteObject.narrow(objref,
    ModeleInterface.class);
```

Il est ainsi possible dans un fichier de configuration de faire figurer les différentes pseudos-URL, quel que soit le protocole utilisé pour l'accès à l'objet.

Cette technique est utilisable pour de nombreuses implémentations de JNDI, comme par exemple les services de nommage RMI et CORBA :

- `rmi://host:port/nomObjet`
- `iiop://host:port/nomObjet`

Lister les éléments d'un annuaire

On peut lister les éléments de `rmiregistry` tout comme ce serait possible pour le service de noms CORBA :

```
NamingEnumeration liste=ctx.list("rmi://localhost:1099");
    while(liste.hasMore()) System.out.println(liste.next());
```

7.5.2 Opérations d'annuaire sur les attributs

L'interface DirContext

Les opérations d'annuaire utilisent l'interface `DirContext` qui hérite de l'interface `Context`.

Les opérations d'annuaires permettent de travailler sur les attributs. Les attributs sont des objets de classe `Attribute` ayant un nom et une ou plusieurs valeurs.

Copyright © 2015 Dunod.

Un contexte d'annuaire va donc exposer la liste de ses attributs et va permettre d'effectuer des recherches sur une valeur d'attribut :

- `getAttributes()`
- `search(...)`

Exemple d'accès au DNS en utilisant JNDI

Le DNS (*Domain Name Server*) est présent dans la grande majorité des réseaux d'entreprise ou chez le fournisseur d'accès internet de votre ADSL.

L'exemple suivant va permettre de donner la liste des machines présentes sur le réseau, ainsi que le nom de leur serveur de mail, un attribut de nom « MX » :

```
DirContext contexte = new InitialDirContext();
Attributes attrs = contexte.getAttributes(
    "dns://80.10.246.1/wanadoo.fr",
    new String[] { "MX" });
```

On remarquera l'utilisation d'une pseudo-URL, ainsi que le filtrage de la liste des attributs renvoyés.

Le résultat est une liste d'attributs de type `Attributes` qu'il faut parcourir avec une `NamingEnumeration`.

```
for (NamingEnumeration enum1=attrs.getAll(); enum1.hasMore();)
    {System.out.println(enum1.next());}
```

7.6 INTRODUCTION À LDAP

7.6.1 Annuaire d'entreprise

LDAP (*Light Directory Access Protocol*) est une implémentation d'annuaire d'entreprise très répandue. Elle s'appuie sur des schémas de données qui définissent la liste des attributs attendus pour un enregistrement de l'annuaire. LDAP est un protocole initialement développé par Novell pour permettre la centralisation d'annuaires d'entreprise pour simplifier l'utilisation des annuaires X500 : il existe notamment des schémas permettant de représenter les informations sur des utilisateurs tels que leur nom, leur login et mot de passe, leur mail et éventuellement leur photo voire leur certificat. Il est possible pour un même serveur de faire cohabiter plusieurs schémas pour décrire plusieurs types de ressources.

Le schéma d'annuaire le plus connu s'appelle `cosine.schema`[1] et définit la classe `inetorgperson` (pour une personne d'une organisation sur Internet) qui permet de décrire un utilisateur avec son login et mot de passe, son numéro de téléphone, son mail voire même sa photo et son certificat.

1. http://www.faqs.org/rfcs/rfc2798.html

© Dunod – Toute reproduction non autorisée est un délit.
Copyright © 2015 Dunod.

Il est possible d'ajouter des schémas de données tels que `java.schema` ou `corba.schema` pour stocker des objets Java ou des références à des objets Java sous forme sérialisée dans l'annuaire. Il est également possible de rajouter un schéma DNS pour coopérer avec un serveur d'adresses IP.

On peut trouver comment installer un serveur LDAP tel que OpenLDAP[1] et l'équiper de Java[2] dans le tutorial d'Oracle. Il existe également une implémentation Open Source de LDAP V3 en Java appelée *Apache Directory Server*[3].

Un annuaire peut fédérer également des références de natures très multiples permettant la gestion de nombreux aspects de l'entreprise (figure 7.9) :

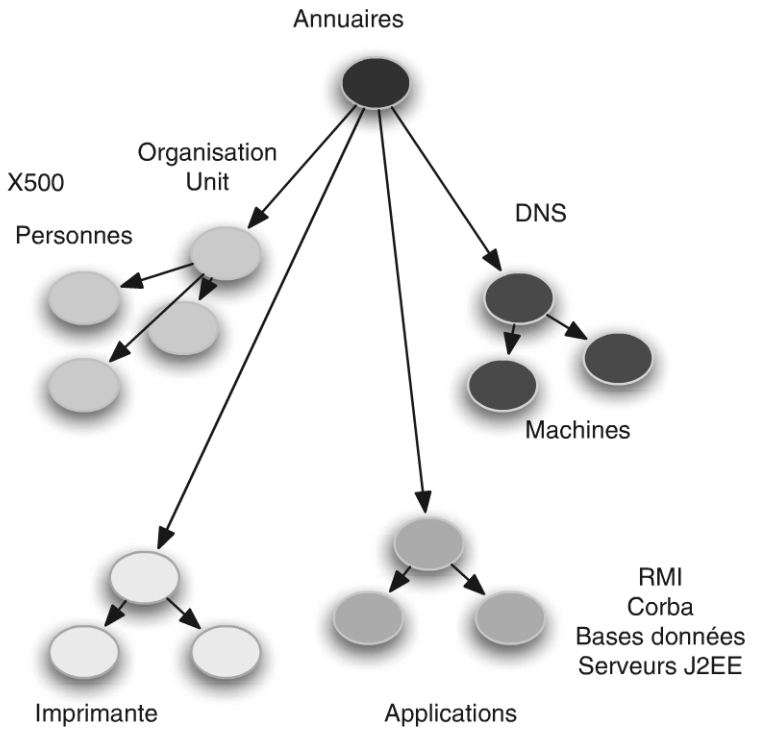


Figure 7.9 — Vue globale de l'entreprise dans un annuaire

- **Des annuaires de personnes** avec les mails et les numéros de téléphone, mais aussi les certificats d'identification, les mots de passe et les répertoires utilisateurs.
- **Des annuaires de matériel**, comportant les noms des machines comme le comporte typiquement un DNS (*Domain Name Service*) ou bien des imprimantes.
- **Des annuaires d'applications**, avec le stockage des références vers des serveurs distants (RMI, CORBA, serveurs web, etc.) ou encore des fabriques JMS (`ConnectionFactory`) ou des `DataSource` pour les accès aux bases de données.

1. http://www.openldap.org
2. http://docs.oracle.com/javase/jndi/tutorial/basics/prepare/content.html
3. http://directory.apache.org/

Copyright © 2015 Dunod.

7.6.2 La hiérarchie de noms en LDAP

LDAP utilise une convention hiérarchique de noms propres à l'organisation d'entreprise (figure 7.10) :

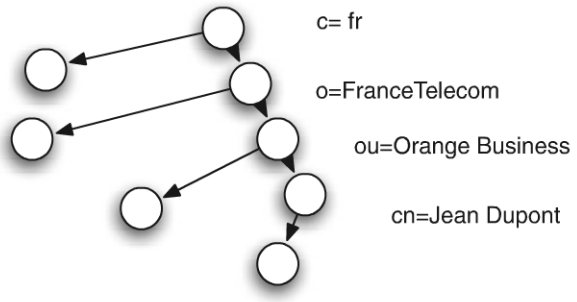


Figure 7.10 — Convention des noms LDAP
c (*country*) désigne le pays ; o (*organization*) désigne l'organisme
ou (*organization unit*) désigne un service ;
cn (*common name*) désigne une personne ou une feuille de l'arbre.

On appelle DN (*Distinguished Name*) la suite de noms désignant un contexte de manière unique.

Ainsi le `dn` de FranceTelecom sera :

```
o=FranceTelecom,c=fr
```

Celui de Jean Dupont :

```
cn=Jean Dupont, ou=Orange Business, o=FranceTelecom, c=fr
```

On désigne par RDN (*Relative Distinguished Name*) le nom unique désignant une feuille dans un contexte particulier.

Par exemple dans le contexte Orange Business (`ou=Orange Business, o = FranceTelecom, c=fr`) le RDN de Jean Dupont sera :

```
cn=Jean Dupont
```

Pour insérer des objets dans l'annuaire, il faut utiliser le RDN par rapport au contexte d'origine.

7.6.3 LDAP comme fédérateur d'annuaires

LDAP va pouvoir servir de proxy pour plusieurs annuaires plus spécialisés, ou répartis sur le réseau. LDAP peut ainsi servir de point d'entrée à des services de noms RMI ou CORBA répartis sur le réseau. En effet, `rmiregistry` doit être local aux objets serveurs

Copyright © 2015 Dunod.
© Dunod – Toute reproduction non autorisée est un délit.

qu'il référence, et il est impossible de partager le même `rmiregistry` entre plusieurs objets serveurs sur des machines différentes. LDAP permet un point d'accès unique aux informations de configuration, ce qui va faciliter le déploiement d'une application répartie. Il est également possible de crypter l'accès à LDAP en utilisant SSL (*Secure Socket Layer*), notamment pour rechercher les informations d'authentification des utilisateurs.

LDAP permet enfin de relier plusieurs instances LDAP entre elles par des liens logiques (figure 7.11), ce qui permet de fédérer des annuaires distants (*referral*) à l'aide de liens logiques (*alias*) comme sur les fichiers Linux, ce qui ne serait pas possible avec un modèle relationnel de bases de données.

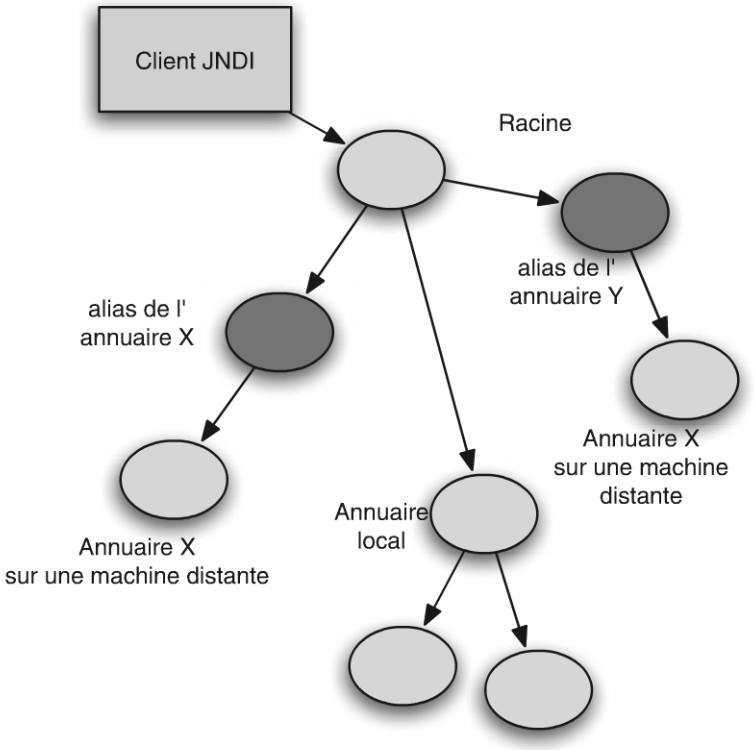


Figure 7.11 — Fédération d'annuaires

7.7 UTILISATION DE JNDI POUR LES ACCÈS LDAP

Pour stocker un objet Java dans un annuaire LDAP, il suffit que celui-ci implémente soit l'interface `Referenceable`, soit l'interface `Serializable`.

On va donc pouvoir stocker des souches d'objets distants CORBA ou RMI dans l'annuaire. Toutefois pour RMI, cela ne dispensera pas de mettre l'objet à disposition dans `rmiregistry`. Seule une référence sera stockée dans l'annuaire.

Copyright © 2015 Dunod.

On remarquera également que les objets de type `DataSource` permettant l'accès à une base de données avec le numéro de port, l'URL d'accès et le login implémentent `Referenceable` et peuvent être également stockés.

Ce livre ne parlera toutefois pas du stockage des accès `DataSource` aux bases de données avec JDBC (*Java Data Base Connection*).

7.7.1 Stockage d'un objet sérialisé dans un annuaire LDAP

Il faut en général utiliser un mot de passe pour accéder à l'annuaire LDAP. Une fois que l'on a obtenu le contexte avec son login, il suffit d'enregistrer l'objet sérialisé tout simplement.

On suppose que l'on a une classe `Fleur` sérialisable comme au chapitre 2 :

```
public class Fleur implements Serializable {...}
```

On doit connaître les informations d'accès à l'annuaire, et pour LDAP il est nécessaire de fournir un login et un mot de passe. Ici le mot de passe est « secret » et le RDN de l'utilisateur `"cn=TutorialAdmin,o=JNDITutorial"` :

```
Hashtable env = new Hashtable(11);
env.put(Context.INITIAL_CONTEXT_FACTORY,
    "com.sun.jndi.ldap.LdapCtxFactory");
env.put(Context.PROVIDER_URL, "ldap://localhost:389/o=JNDITutorial");
// Identification de l'utilisateur avec mot de passe
env.put(Context.SECURITY_AUTHENTICATION, "simple");
env.put(Context.SECURITY_PRINCIPAL, "cn=TutorialAdmin,o=JNDITutorial");
env.put(Context.SECURITY_CREDENTIALS, "secret");
DirContext ctx = new InitialDirContext(env);
```

On peut alors utiliser l'annuaire, tout comme on l'a vu avec RMI ou CORBA et appeler les méthodes `bind` et `lookup` sur un objet sérialisé.

```
Fleur hibiscus = new Fleur();
hibiscus.setCouleur(''rouge'');
ctx.bind(''cn=maFleur'', hibiscus).
```

Plus tard, on récupère l'objet avec la même création de contexte :

```
Fleur fleurRetrouvee=ctx.lookup("cn=maFleur");
```

Ou en utilisant une pseudo-URL, car en lecture il n'est pas nécessaire de préciser le login :

```
Context ctx = new InitialContext() ;
Fleur fleurRetrouvee=
ctx.lookup("ldap://localhost:389/cn=maFleur,o=JNDITutorial") ;
```

Copyright © 2015 Dunod.
© Dunod – Toute reproduction non autorisée est un délit.

On pourra de la même manière stocker dans le profil d'un utilisateur en plus de son login l'accès à son disque de stockage, mais aussi une interface graphique complète avec tous ses boutons pourvu qu'elle soit sérialisable !

7.7.2 Stockage d'un objet CORBA dans LDAP

Il est possible d'utiliser un annuaire LDAP pour le service de noms CORBA. Pour cela on crée un objet CORBA comme au chapitre 6 :

```
ORB orb = ORB.init(args, null);
// Création de l'objet Bourse
BourseImpl brongniart = new BourseImpl();
POA rootpoa =POAHelper.narrow(orb.resolve_initial_references("RootPOA"));
rootpoa.the_POAManager().activate();
org.omg.CORBA.Object ref = rootpoa.servant_to_reference(brongniart);
Bourse href = BourseHelper.narrow(ref);
Hashtable env = new Hashtable();
env.put(Context.INITIAL_CONTEXT_FACTORY,
    "com.sun.jndi.ldap.LdapCtxFactory");
env.put(Context.PROVIDER_URL, "ldap://localhost:389/o=JNDITutorial");
env.put("java.naming.corba.orb", orb);
// Création du contexte initial
DirContext ctx = new InitialDirContext(env);
// Stockage de l'objet dans l'annuaire
ctx.bind("cn=CorbaHello", ref);
```

7.7.3 Stockage d'une référence à un objet RMI dans un annuaire LDAP

Il est possible de stocker une référence à un objet RMI dans LDAP. LDAP ne sert alors que de proxy à un `rmiregistry` local. Mais comme `rmiregistry` ne peut enregistrer des objets sur une autre machine, c'est un moyen commode de fédérer plusieurs annuaires (figure 7.12).

Comme les EJB (*Entreprise Java Beans*) sont la plupart du temps des objets RMI, il est possible de stocker également des EJB.

On obtient le contexte d'écriture LDAP comme précédemment :

```
Context ctx=...
```

On crée ensuite une référence sur le nom (la référence !) de l'objet RMI :

```
String rmiurl= "rmi://localhost/modele" ;
Reference ref = new Reference("Modele",new StringRefAddr("URL", rmiurl));
```

On met la référence dans l'annuaire :

```
ctx.bind("cn=RefModele", ref);
```

Copyright © 2015 Dunod.

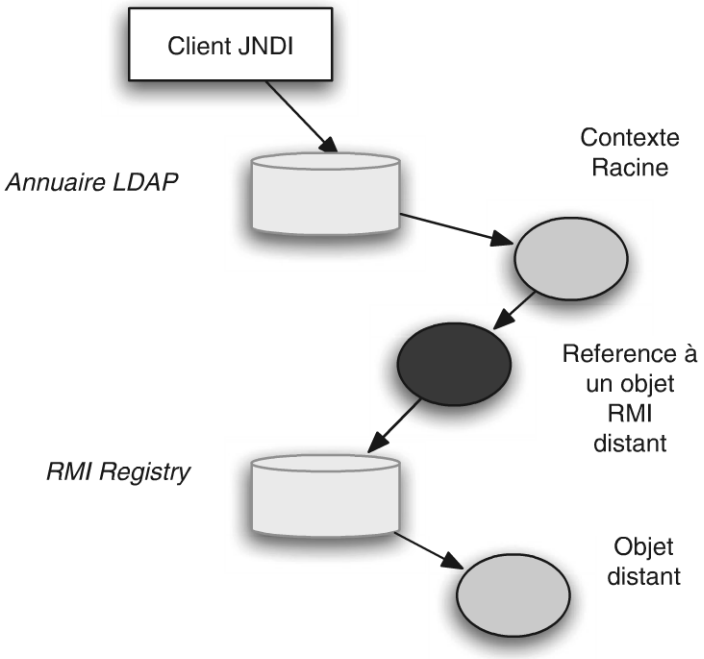


Figure 7.12 — Redirection d'un annuaire RMI dans LDAP

On crée l'objet distant et on l'enregistre dans `rmiregistry` avec le nom défini dans l'annuaire :

```
Modele modele= new Modele() ;
Naming.rebind(rmiurl, modele);
```

On peut alors tester la redirection :

```
ModeleInterface modele2 = (ModeleInterface)ctx.lookup("cn=RefModele");
System.out.println(modele2.datation()) ;
```

Sans oublier de fermer la connexion à la fin avec `ctx.close()`.

7.7.4 Stockage d'une ConnectionFactory JMS dans LDAP

Nous verrons dans le chapitre 8 JMS qu'une `ConnectionFactory` va regrouper tous les paramètres de connexion à un serveur JMS. Il est possible de stocker une `ConnectionFactory` dans LDAP afin de fournir un point d'accès unique et préconfiguré lorsque plusieurs clients partagent le même serveur. Ces clients n'auront alors plus besoin de fichier de configuration, ce qui est précieux lors du déploiement.

```
Context ctx = ...
ConnectionFactory factory= ...
ctx.bind("cn=jmsFactory", factory);
```

Copyright © 2015 Dunod.
© Dunod – Toute reproduction non autorisée est un délit.

Ce mécanisme permet également de changer l'implémentation JMS ou de déplacer le serveur sans impact sur le client qui utilise la même API.

7.8 RECHERCHE D'ÉLÉMENTS DANS LDAP

LDAP permet d'utiliser des attributs (mode pages jaunes) pour effecteur des recherches à l'aide de la méthode `search` de `DirContext`.

La recherche peut s'effectuer récursivement sur toute la hiérarchie ou à un seul niveau : c'est ce que l'on appelle les contrôles. Ce livre ne développera pas ces aspects présents dans JNDI. Voici simplement un exemple de recherche d'enregistrements liés à un numéro de téléphone qui utilise une pseudo-URL :

```
ctx = new InitialDirContext();
Attributes attrs= new BasicAttributes();
    attrs.put(new BasicAttribute("telephoneNumber","+1 408 555 5252"));
NamingEnumeration enum =
    ctx.search("ldap://localhost:389/ou=People,o=JNDITutorial", attrs);
while(enum.hasMore()) System.out.println((SearchResult)enum.next());
```

En résumé

Les services de noms et d'annuaires sont un des pivots du déploiement et de la supervision des applications réparties. Ils sont largement utilisés dans tous les serveurs d'applications tels que WebLogic ou WebSphere pour répertorier les ressources disponibles, que ce soient les bases de données ou les points d'accès des serveurs. Ils peuvent également être utilisés par JMS (voir chapitre 8) et RMI (voir chapitre 5). JNDI est une API Java qui permet d'uniformiser l'accès à de nombreuses ressources tant matérielles que logicielles, notamment avec l'utilisation de pseudo-URL : l'accès à un annuaire RMI ou CORBA se fait de la même façon.

LDAP permet une vision d'entreprise des annuaires et peut fédérer plusieurs implémentations ou plusieurs sites.

Copyright © 2015 Dunod.

8

JMS

Objectif

Le but de ce chapitre est de rappeler les concepts clés de la communication par message au travers de l'API normalisée de JMS (*Java Messaging Service*), de montrer les problèmes liés à la diffusion d'information non seulement en point en point mais aussi à un groupe de destinataires. Ce chapitre abordera également les notions de programmation asynchrone et de qualité de service.

8.1 JAVA MESSAGING SERVICE

8.1.1 Une API standard

Java Messaging Service[1] (JMS) est une API standard de Java édition entreprise (J2EE)[2]. Elle permet de fournir à Java une implémentation de MOM (*Message Oriented Middleware*) ou intergiciel de communication par messages, en quelque sorte un service de mél pour applications.

Toutefois, pour pouvoir utiliser JMS, il faut disposer d'une implémentation externe en Java. Il existe de nombreuses implémentations, tant commerciales que Open Source. Citons par exemple : WebSphere MQ d'IBM, Sonic MQ de Progress Software mais aussi Fiorano et Swift MQ pour les versions commerciales ; ActiveMQ, Joram, Open MQ, Open JMS pour les versions Open Source. Vous trouverez une liste non

1. http://www.oracle.com/technetwork/java/jms/index.html
2. On peut toutefois l'utiliser avec Java SE en utilisant une bibliothèque `jms.jar`.

© Dunod – Toute reproduction non autorisée est un délit.
Copyright © 2015 Dunod.

exhaustive mais souvent mise à jour à la page suivante : http://www.afceurope.com/JMS.html

Il existe également des utilitaires pour tester la conformité de l'implémentation à l'API[1].

> Notez que l'API JMS est définie dans la version serveur J2EE de Java et non la version standard J2SE. Il y a eu une remise à plat entre JMS 1.0 et JMS 1.1 afin d'unifier les noms de méthodes et les classes entre les deux modes de communication : par exemple `publish` et `send` sont unifiés en `send`. Ce livre s'appuie sur la version JMS 1.1. La version 2.0 sera documentée à la fin du chapitre.

8.1.2 Communication à plusieurs

Alors que dans les technologies RPC (*Remote Procedure Call*) telles que RMI ou CORBA il y a une communication synchrone directe entre un objet client et un objet serveur, JMS (*Java Messaging Service*) introduit une couche de médiation entre un ou plusieurs producteurs et un ou plusieurs consommateurs.

Un **producteur** de message va créer un message et utiliser une implémentation JMS pour l'envoyer. Un **consommateur** va en contrepartie lire le message (le consommer).

Typiquement, une communication RPC nécessitera de nombreux liens point à point entre clients et serveurs (figure 8.1).

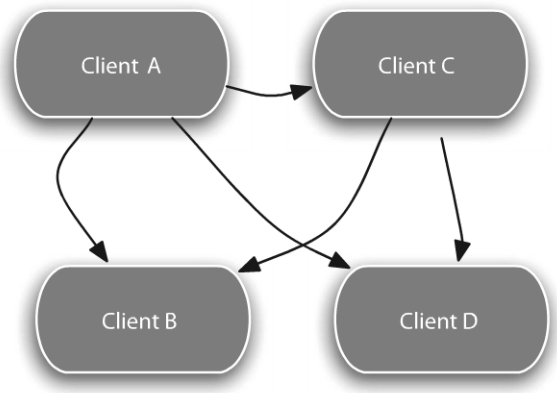


Figure 8.1 — Interconnexion entre plusieurs interlocuteurs en mode RPC

La couche de médiation JMS permet par contre une communication de un ou plusieurs producteurs vers zéro ou plusieurs consommateurs. Attention à ne plus parler de clients et de serveurs pour les consommateurs et les producteurs, car en quelque sorte tous les producteurs et consommateurs sont vus comme des clients par le courtier (*broker*) JMS (figure 8.2).

1. http://jmscts.sourceforge.net/

Copyright © 2015 Dunod.

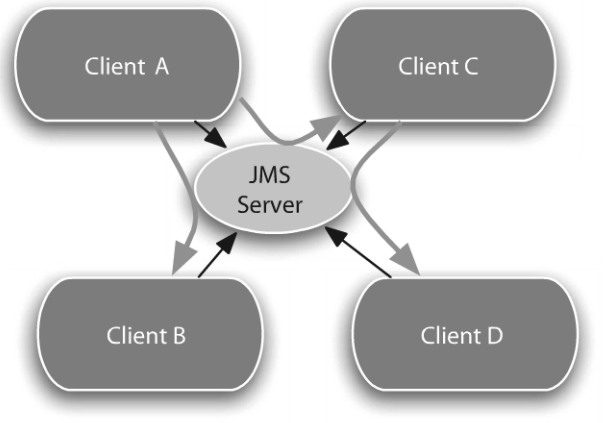


Figure 8.2 — Interconnexion entre plusieurs interlocuteurs au travers d'un courtier JMS

La communication s'établira au travers de deux segments : par exemple entre A et B, A établira une communication avec le courtier JMS, ensuite B fera de même, à un instant pouvant être différé. Le courtier peut en effet stocker des messages de façon persistante, afin d'assurer une communication asynchrone, tout comme lorsque vous envoyez du courrier à un correspondant, celui-ci pourra le lire à une date ultérieure depuis sa boîte aux lettres. Sa boîte aux lettres sera repérée par une `Destination`, de type `Queue` ou `Topic`, selon que l'on est en mode point à point ou en diffusion. Ceci sera présenté ci-après.

8.1.3 Le modèle de producteur/consommateur - les destinations

Le modèle JMS provient de l'unification de deux modes de communication :

- Un mode de communication en point à point avec un stockage temporaire dans une boîte à lettre appelée **Queue**, où un producteur va déposer un message lu ultérieurement par un consommateur.
- Un mode de communication par événements, ou en mode **publication/abonnement** (*publish/subscribe*), où zéro ou plusieurs consommateurs écoutent sur un même canal de diffusion appelé **Topic**, alimenté par un ou plusieurs producteurs.

Les **destinations** de type `Queue` ou `Topic` (figure 8.3) évitent au producteur et au consommateur de connaître leurs adresses respectives, et elles assurent la transparence de localisation, contrairement aux sockets (voir chapitre 3) où l'on doit connaître le port et le nom de machine pour communiquer.

Les destinations peuvent être ou non administrées : certaines implémentations nécessitent la création de `Queue` dans un outil d'administration et non dynamiquement dans un programme : en effet, les `Queue` sont souvent liées à une capacité de stockage physique. L'outil d'administration permet alors de consulter l'état de la `Queue`, le nombre de messages en attente. Les `Topic` peuvent également parfois être administrés.

© Dunod – Toute reproduction non autorisée est un délit.
Copyright © 2015 Dunod.

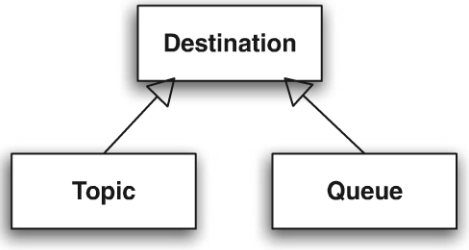


Figure 8.3 — `Topic` et `Queue` héritent de `Destination`

Le mode de communication en **point à point** est bien adapté à un *workflow* d'entreprise où les documents circulent avec un temps de traitement spécifique (figure 8.4).

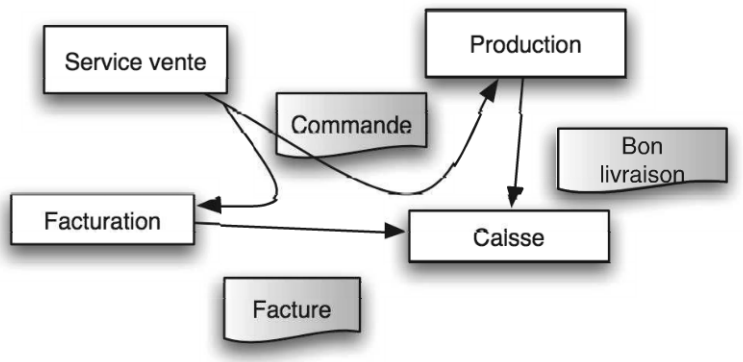


Figure 8.4 — Mode point à point de circulation de documents

Le mode de communication **par événement** convient au contraire pour diffuser des alertes à plusieurs destinataires (figure 8.5), par exemple des informations de bourse à des courtiers, ou bien des anomalies réseau en télécommunications, des points de synchronisation en vidéo, etc.

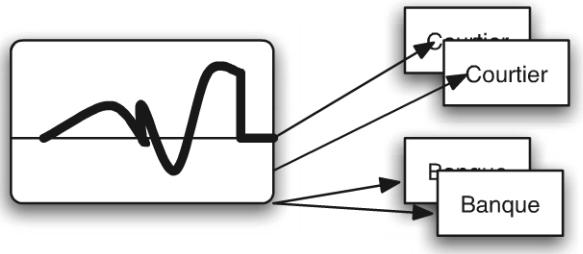


Figure 8.5 — Diffusion d'alertes de cours de change à la Bourse

Copyright © 2015 Dunod.

La plupart des implémentations de JMS permettent la communication sous les deux modes, toutefois certaines n'implémentent parfois que le mode `Topic` : elles ne sont alors pas conformes à la spécification JMS.

Chaque message individuel n'est produit que par un seul producteur, même si globalement une destination est alimentée par plusieurs producteurs.

En mode **point à point**, un message individuel n'est lu que par un seul consommateur (figure 8.6).

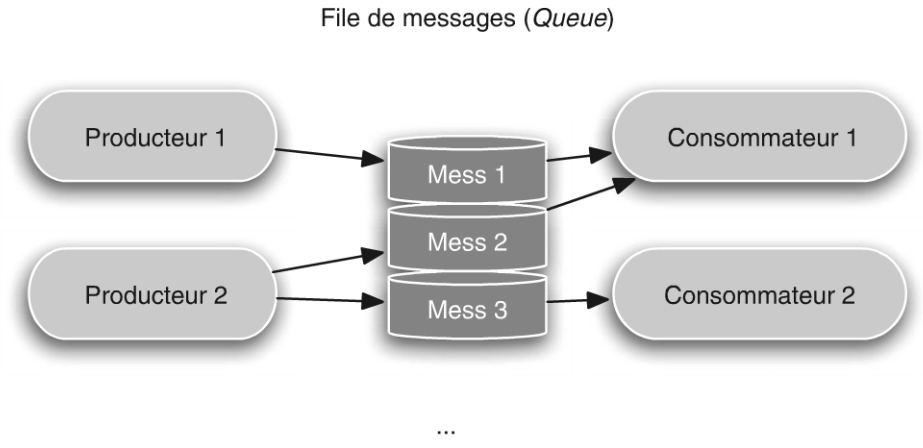


Figure 8.6 — Un message en point à point n'est lu que par un seul destinataire

Le message peut être stocké pour un certain temps par le courtier, et la lecture du message peut être différée par rapport à sa production. C'est en ce sens que l'on dira que JMS est un mode de communication asynchrone. Après lecture, le message est détruit de son lieu de stockage après accusé de réception du consommateur.

Dans un **mode publication/abonnement**, plusieurs consommateurs écoutent sur le même canal (`Topic`) et reçoivent collectivement le même message individuel (figure 8.7).

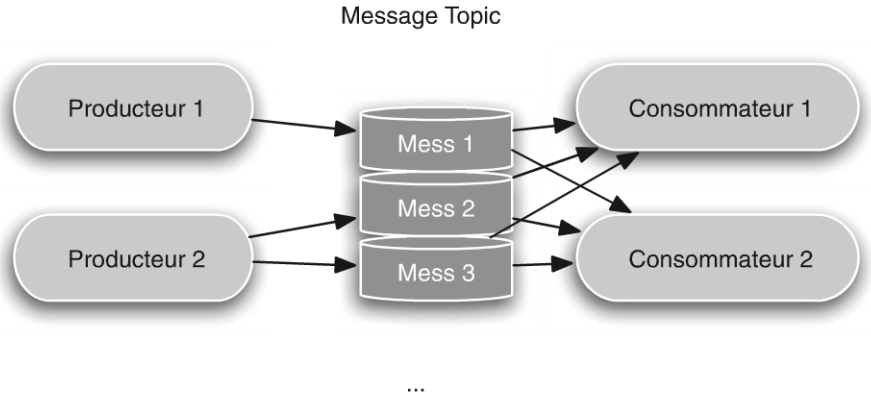


Figure 8.7 — Un même message peut être lu par plusieurs destinataires en mode publier/s'abonner

Copyright © 2015 Dunod.
© Dunod – Toute reproduction non autorisée est un délit.

Comme les consommateurs peuvent s'abonner et se désabonner dynamiquement, il n'est pas possible de connaître le nombre d'accusés de réception à attendre. Le message est donc détruit immédiatement, sauf dans le cas particulier des abonnés durables que nous verrons par la suite.

En mode publication/abonnement, un consommateur absent au moment de l'envoi du message ne le recevra donc pas. C'est pour cela qu'il est considéré moins fiable que le mode point à point : c'est la même analogie qu'entre le mode TCP et le mode UDP (voir chapitre 3).

À quel moment exact est détruit le message ? C'est ce que nous allons voir avec les concepts de session et de connexion.

8.1.4 Structure des messages

Contrairement aux modèles RPC (CORBA, RMI, SOAP), les consommateurs et les producteurs ne partagent pas d'interfaces (voir chapitre 4) et s'échangent uniquement des informations sous forme de messages. Un message est par définition la composition d'un en-tête, de propriétés et d'un contenu, ce contenu pouvant être de plusieurs natures, texte, binaire ou autre comme nous le verrons dans la suite. C'est pour cette raison que JMS est souvent utilisé pour l'intégration d'applications puisqu'il est moins contraignant sur la définition des interfaces, et s'adapte à l'existant sans modification.

Le message a un rôle ambivalent puisqu'il sert :

- de transport **d'information**,
- de point de **synchronisation** entre producteur(s) et consommateur(s) lorsque le consommateur utilise une réception synchrone avec une méthode `receive()` (voir paragraphe 8.2.2).

8.1.5 Connexion et session

Il existe deux abstractions en JMS qui n'ont pas d'équivalent en mode RPC : la **connexion** et la **session**. La connexion est connue des utilisateurs de bases de données et de l'interface JDBC (*Java Database Communication*), elle permet notamment de gérer la sécurité des accès et le déroulement des transactions vers le gestionnaire de bases de données. La session on le verra joue un rôle dans la qualité de service propre à JMS.

La connexion assure la fiabilité

Lorsque l'on utilise des sockets, un mode connecté précise que le client et le serveur doivent échanger une poignée de main (*handshake*) avant de rentrer en communication. Ce protocole assure que les deux parties sont disponibles au moment de la communication. Si l'un des deux raccroche, la communication est interrompue. C'est ainsi que le protocole TCP qui est en mode connecté est considéré comme

Copyright © 2015 Dunod.

plus fiable que le protocole UDP, où le message peut se perdre si le correspondant a raccroché. La connexion JMS joue exactement ce rôle : elle assure la fiabilité entre un client JMS et son courtier, et l'on peut récupérer les exceptions si la connexion est défaillante.

Pour une communication très fiable, on peut imaginer un battement de cœur (*heartbeat*) permettant de contrôler régulièrement que le serveur est disponible, en lui envoyant régulièrement des messsages.

En mode RPC, on peut détecter que le correspondant est défaillant sur réception d'une exception en retour d'un appel distant. En mode asynchrone, ce n'est évidemment pas possible.

On notera enfin que la connexion est également utilisée par l'interface JCA (*Java Connector Architecture*) pour communiquer avec des serveurs d'applications, des mainframes (CICS) et des bases de données.

La session assure la qualité de service

Une connexion peut ouvrir plusieurs sessions (figure 8.8). Une session est définie à l'intérieur d'une connexion comme une délimitation temporelle permettant de grouper des envois ou des réceptions de messages avec des propriétés spécifiques : mode transactionnel ou non, priorités des messages, séquence des messages, gestion des accusés de réception. C'est ce qu'on appellera la **qualité de service**.

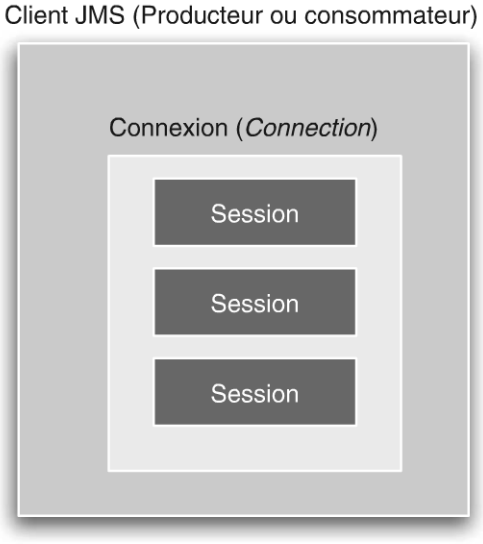


Figure 8.8 — Une connexion peut ouvrir plusieurs sessions

Naturellement deux sessions consécutives peuvent définir des modes différents. On notera que dans l'interface JCA définie en 2005 la notion de session est remplacée par plusieurs contrats de service séparés pour la gestion des connexions, la gestion des

Copyright © 2015 Dunod.
© Dunod – Toute reproduction non autorisée est un délit.

transactions et la gestion de la sécurité. En JMS la sécurité est gérée par la connexion, les transactions par la session.

8.2 EXEMPLE DE CODE POUR ENVOYER ET RECEVOIR UN MESSAGE EN JMS

8.2.1 Envoi d'un message

Un message est tout simplement un objet de classe `Message`. Nous prendrons ici le type de message le plus simple, `TextMessage`, dont le contenu est une chaîne de caractères. Les autres types de messages seront détaillés au paragraphe 8.3. On peut accéder au contenu d'un `TextMessage` en utilisant les méthodes `getText()` et `setText(String)`.

Pour envoyer un message, il faut disposer d'une `ConnectionFactory`, et connaître d'autre part le nom d'une destination, `Queue` ou `Topic`, (ici `MyQueue`). La `ConnectionFactory` utilise le *design pattern* de la fabrique (voir chapitre 2) et permet plusieurs connexions concurrentes. Elle encapsule tous les paramètres d'initialisation de la connexion. La `ConnectionFactory` peut provenir d'un annuaire JNDI (voir chapitre 7) ou être créée à partir de paramètres de configuration. La destination peut également être stockée dans un annuaire JNDI. Ceci est en particulier utilisé pour les destinations administrées et créées par l'outil d'administration propre à l'implémentation JMS.

La création d'une `ConnectionFactory` est spécifique à une implémentation, ici il s'agit de Sonic MQ. Le paramètre nécessaire est l'adresse du serveur et son numéro de port :

```
tcp://localhost:2506
```

On n'a besoin que d'un seul package pour JMS en plus des bibliothèques liées à l'implémentation :

```
import javax.jms.* ;
```

On crée alors la `ConnectionFactory` :

```
String broker= "tcp://localhost:2506";
ConnectionFactory qcf= (ConnectionFactory) new
progress.message.jclient.ConnectionFactory (broker);
```

La connexion nécessite un nom d'utilisateur et un mot de passe, mais il est possible d'utiliser le login par défaut :

```
Connection connexion= qcf.createConnection("Utilisateur", "MotDePasse");
```

Copyright © 2015 Dunod.

Après la création de la connexion, il faut une session. Ici, la session ne comporte pas de transaction (paramètre `false`), et l'acquittement des messages est automatique (`AUTO_ACKNOWLEDGE`) :

```
Session session = connexion.createSession(false, Session.AUTO_ACKNOWLEDGE);
```

La `Queue` peut avoir été créée dans le courtier avant utilisation, cela dépend des implémentations. On peut également avoir des destinations temporaires liées à la session. L'instruction suivante donne seulement une référence sur cet espace de stockage créé à l'avance par un outil d'administration propre à l'implémentation. Ici la destination a pour nom « MyQueue ».

```
Queue queue = session.createQueue("MyQueue");
```

On crée ensuite le producteur et le message. Noter que le producteur et le message n'existent que dans la session qui les crée.

```
MessageProducer sender = session.createProducer(queue);
TextMessage mess= session.createTextMessage("Hello");
sender.send( mess);
```

Ne pas oublier de fermer la connexion. Ceci ferme également la session et déclenche tous les messages de gestion d'accusés de réception.

```
session.close();
connexion.close();
```

Le code est exactement le même en mode publication/abonnement. La seule différence est le type de destination : ici un `Topic`. Les `Topic` sont le plus souvent créés dynamiquement contrairement aux `Queue`, car dans le mode usuel il n'y a pas d'acquittement des `Topic`, et donc pas de persistance des messages et de réservation de stockage.

```
Topic topic = session.createTopic(''MonTopic'');
```

La symétrie ente `Topic` est `Queue` est identique dans le code de réception du paragraphe suivant.

8.2.2 Réception d'un message synchrone

Pour la réception d'un message synchrone, la création d'une connexion et d'une session sont identiques, de même que la création de la référence à la `Destination`. Toutefois la session va cette fois créer un consommateur (`Consumer`) au lieu d'un

Copyright © 2015 Dunod.
© Dunod – Toute reproduction non autorisée est un délit.

producteur (`Producer`). On peut créer dans la même session des producteurs et des consommateurs :

```
Queue queue = session.createQueue("MyQueue");
// pour un Topic, on aurait de même createTopic...
MessageConsumer receiver = session.createConsumer(queue);
// Ne pas oublier de démarrer la connexion pour l'attente
// de réception de message
connexion.start();
// réception d'un message de type texte
TextMessage mess= (TextMessage) receiver.receive();
System.out.println("Message reçu : "+ mess.getText());}
// fermeture de la connexion et par là-même de la session
connexion.close();
```

Ne pas oublier de démarrer la connexion (`start`) pour se mettre en attente de réception des messages.

Ne pas oublier de fermer la connexion, afin de libérer des ressources. Le programme restera en attente tant que la connexion n'est pas fermée. Certaines implémentations nécessitent de fermer la session avant la connexion.

La commande de réception `receive` est bloquante pour les `Queue` : s'il n'y a pas de message dans la file, le consommateur sera bloqué en attente indéfiniment. Si l'on souhaite éviter ce blocage, on peut utiliser :

- `ReceiveNoWait()`: renvoie le premier message dans la `Queue` s'il existe, sinon renvoie `null`.
- `Receive(long timeout)`: attend un message pendant l'intervalle de temps `timeout`, sinon renvoie `null`.

8.2.3 Réception d'un message asynchrone

Pour une réception de message utilisant `receive`, le consommateur est bloqué en attente de réception (figure 8.9). En ce sens on parle de réception **synchrone.** Le terme est ambivalent, car on qualifiera JMS de mode de communication **asynchrone** entre producteurs et consommateurs, car le moment d'émission du message peut être totalement découplé du moment de réception du message, tout comme pour une lettre traditionnelle. La confusion provient du fait qu'une communication par JMS comporte deux segments distincts, l'un entre le courtier et l'émetteur et l'autre entre le courtier et le récepteur. Globalement la communication est asynchrone, mais le segment du récepteur peut être synchrone ou asynchrone.

L'attente bloquante peut détériorer les performances, ou générer des interblocages. Un **interblocage** survient lorsque deux interlocuteurs sont chacun en attente d'un message de l'autre. Par exemple si A attend un message de B pour lui envoyer une réponse, et B attend également un message de A pour lui envoyer une réponse.

Copyright © 2015 Dunod.

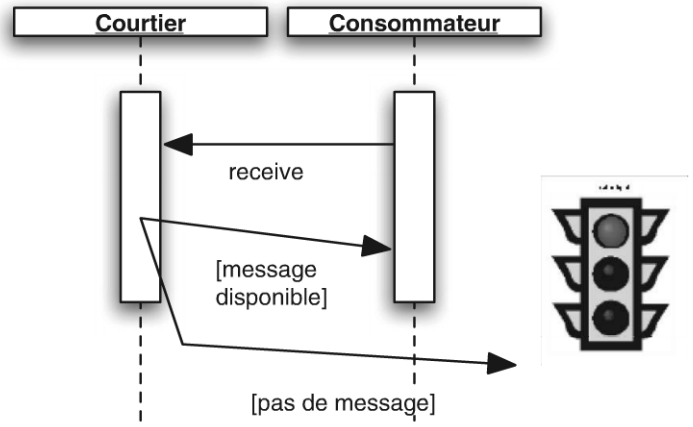


Figure 8.9 — Le consommateur est bloqué en attente sur un `receive` tant qu'il n'y a pas de message

JMS permet de faire de la réception asynchrone en utilisant le mécanisme classique en Java des `Listeners` (qui suit le *design pattern* de l'observateur). Un `MessageListener` créera un thread permettant de gérer la réception du message sur disponibilité de celui-ci (figure 8.10). Le `MessageListener` sera attaché au consommateur, mais peut ou non être une classe différente de celle qui abrite le consommateur. Il doit implémenter la méthode `onMessage(Message)`.

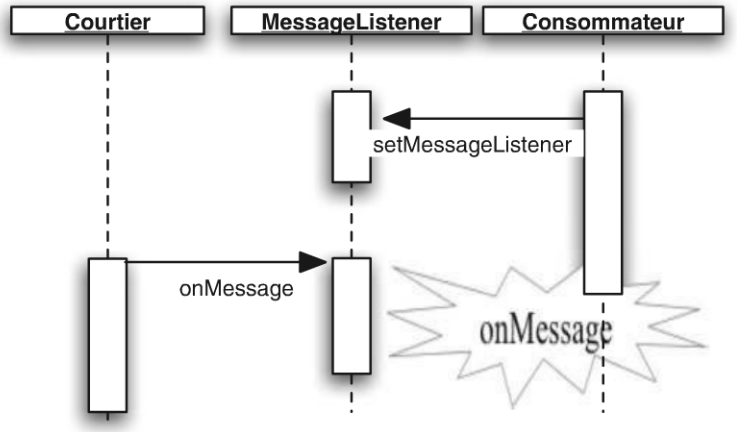


Figure 8.10 — L'arrivée d'un message en mode de réception asynchrone déclenche l'appel de la méthode `onMessage`

Cette technique évite le *polling* et améliore les performances. Le *polling* signifie que l'on va régulièrement scruter l'arrivée d'une donnée. C'est un mécanisme classique sur les drivers par exemple, mais très gourmand en temps : soit l'intervalle de scrutation est faible et l'on perd du temps à vérifier une donnée absente, soit l'intervalle de scrutation est long et l'on peut perdre la donnée. Par contre cette technique asynchrone génère un thread de réception pour la méthode `onMessage` et nécessite de faire attention

Copyright © 2015 Dunod.
© Dunod – Toute reproduction non autorisée est un délit.

à l'accès des variables partagées au thread de réception et au thread principal (voir chapitre 3 pour les accès concurrents aux variables).

Dans le code suivant, on suppose que la classe qui lance la réception implémente également la méthode `onMessage` définie dans l'interface `MessageListener` :

Code pour la mise en place du `Listener` :

```
Queue queue = session.createQueue("MyQueue");
MessageConsumer receiver = session.createConsumer(queue);
// On doit attacher le MessageListener avant de démarrer la connexion
receiver.setMessageListener(this);
connexion.start();
// Le code suivant permet de vérifier quel thread est utilisé
System.out.println("Thread lancement" +
        Thread.currentThread().getId());
...() ; // attente bloquante permettant de ne pas fermer la connexion
```

Le code suivant sera déclenché dans le `MessageListener` sur réception du message :

```
public void onMessage(Message message) {
// Affiche le contenu du message à la console
    System.out.println(((TextMessage) message).getText());
// Vérifie que le thread utilisé est différent
    System.out.println("Thread id onMessage "+
Thread.currentThread().getId());
...}
```

Une session ne peut comporter que des consommateurs **synchrones** ou que des consommateurs **asynchrones** dotés de `MessageListener`. Il n'est pas possible de mélanger les deux modes de réception dans la même session : on créera alors deux sessions.

8.2.4 Retour sur la création des connexions et des sessions

Le mécanisme de création de connexion et de session est parfaitement identique pour la création et la réception de messages (figure 8.11). Il suppose deux points d'entrée : la fabrique de connexions (`ConnectionFactory`) et le nom de la destination (`Queue ou Topic`).

La `ConnectionFactory` crée une connexion qui crée à son tour une ou plusieurs sessions.

Un `MessageProducer` ou un `MessageConsumer` n'a d'existence qu'à l'intérieur d'une session. Il est spécifique à une destination. On voit simplement que pour plusieurs `Topic` d'intérêt (par exemple football, rugby, etc.) on doit créer plusieurs consommateurs (resp. producteurs), car les consommateurs sont liés à une seule destination.

Un consommateur peut naturellement recevoir plusieurs messages, de même qu'un producteur peut en produire plusieurs dans la même session.

Une session peut créer plusieurs consommateurs et plusieurs producteurs.

Copyright © 2015 Dunod.

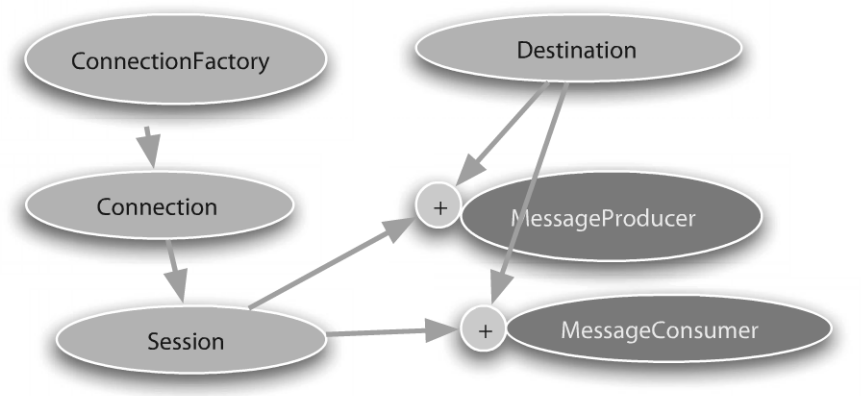


Figure 8.11 — Deux points d'entrée dans le système JMS : les destinations et les fabriques de connexions (`ConnectionFactory`)

Pourquoi utiliser plusieurs connexions ?

Une connexion est liée à une machine et un port sur lequel écoute le courtier. L'utilisation de plusieurs connexions permet de se connecter à plusieurs courtiers, par exemple pour faire de la redondance.

Une connexion est également attachée à une identification, typiquement utilisateur et mot de passe. Les implémentations de JMS les plus complètes permettent de restreindre l'accès à certaines ressources telles que des `Topic` ou des `Queue` pour certains utilisateurs. On utilisera éventuellement plusieurs connexions pour différents utilisateurs.

Pourquoi utiliser une ConnectionFactory ?

Le concept de fabrique (*Factory*) est présenté au chapitre 2. Il permet dans notre cas de créer des connexions personnalisées, avec des paramètres prédéfinis tels que l'adresse du courtier ou l'utilisateur à utiliser. De plus, on le stocke d'ordinaire dans un annuaire d'entreprise ou dans un annuaire applicatif au travers de l'interface JNDI qui a été présentée au chapitre 7. Cette fabrique permet aux clients JMS de disposer d'une configuration préétablie puisque la fabrique contient les paramètres d'initialisation. Voici un exemple de code en Joram[1] pour obtenir une connexion préconfigurée en accédant à l'annuaire propriétaire de Joram sur le port 16400 en local :

```
Properties properties = System.getProperties();
properties.put("java.naming.factory.initial",
    "fr.dyade.aaa.jndi2.client.NamingContextFactory");
properties.put("java.naming.factory.host", "localhost");
properties.put("java.naming.factory.port", "16400");
InitialContext jndi = new InitialContext();
```

1. http://joram.ow2.org/

© Dunod – Toute reproduction non autorisée est un délit.
Copyright © 2015 Dunod.

```
ConnectionFactory fabriqueDeConnexion =
    (ConnectionFactory)jndi.lookup("fabrique");
Connection connection = fabriqueDeConnexion.createConnection();
```

L'API JMS précise que les `Connectionfactory` doivent être `Referenceable` et `Serializable` afin de pouvoir les stocker dans un annuaire JNDI. Ce ne sont toutefois pas les seuls objets à figurer dans l'annuaire : les `Queue` et `Topic` peuvent également être découverts ou configurés dans un annuaire. Ainsi lors du déploiement d'une application, un client pourra obtenir dans un annuaire la liste des destinations qu'il doit adresser, sans configuration locale et statique.

L'utilisation de JNDI est également utile pour permettre d'interchanger les implémentations de JMS : en effet, comme la `ConnectionFactory` comporte tous les paramètres d'initialisation, l'obtention d'un tel objet permet un code identique quelle que soit l'implémentation :

```
ConnectionFactory factory= contexte.lookup("reference") ;...
```

Pourquoi utiliser plusieurs sessions ?

La session gère les paramètres de qualité de service des messages, ainsi que les Queues temporaires, les transactions (voir plus loin), l'ordre chronologique des messages, la réception synchrone ou asynchrone. Pour les transactions, la session servira de délimitation, elle permettra d'assurer l'ordre des messages à l'intérieur d'une même session. Elle assurera également la gestion des accusés de réception des messages. Des sessions trop longues peuvent typiquement générer des attentes d'accusés de réception des messages et détériorer les performances. Par contre, il est important de comprendre qu'une session peut comporter à la fois des producteurs et des consommateurs de messages, qui peuvent envoyer un nombre quelconque de messages.

8.2.5 Utilisation des hiérarchies de Topic

La plupart des implémentations JMS offrent des hiérarchies de `Topic` comme le montre la figure 8.12, toutefois la syntaxe de filtrage des `Topic` ou la création de l'arborescence diffère.

En supposant que le filtrage se fait par expressions régulières, il est ainsi possible de s'abonner à `/Sports/*` pour recevoir les `Topic` fils : `Sports`, `Football`, `Tennis`, `PSG` et `OL`.

Ceci peut être utile pour faire des diffusions plus ou moins ciblées, ou des communications de groupes : si chaque application écoute sur un `Topic` particulier, on peut les regrouper pour faire une diffusion générale.

Copyright © 2015 Dunod.

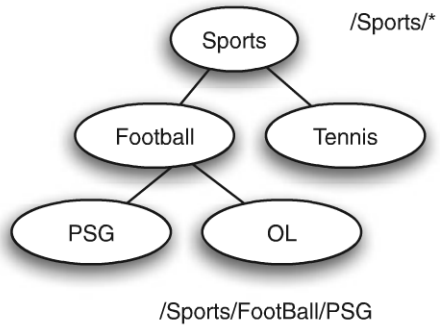


Figure 8.12 — Hiérarchie de `Topic`

8.3 NATURE DES MESSAGES ET CODAGE DE L'INFORMATION

Un message JMS est constitué de plusieurs parties :

- le **corps** du message qui véhicule les informations,
- les **propriétés** du message (une liste de paires attribut/valeur),
- **l'en-tête**, utilisé pour les informations de qualité de service et le routage, ainsi que le transport de propriétés associées au message.

8.3.1 Le corps du message

JMS fournit plusieurs natures de messages, permettant différents codages des informations (*marshalling*) : `TextMessage`, `BytesMessage`, `MapMessage`, `StreamMessage`, `ObjectMessage`.

TextMessage

Un `TextMessage` transporte du **texte** et peut utiliser les nouvelles classes d'encodage de caractères de `java.nio` dans toutes les langues. Un message de texte peut également transporter un contenu au format XML. Ceci permet par exemple à JMS de servir de protocole sous-jacent à SOAP (voir chapitre 9). Ceci permet également d'accorder la syntaxe du message entre les deux correspondants en utilisant par exemple un schéma XML pour vérifier la conformité de la syntaxe du texte.

Création d'un message par la session – la chaîne de caractère est passée en argument :

```
TextMessage messageTexte = session.createTextMessage("hello");
```

Copyright © 2015 Dunod.
© Dunod – Toute reproduction non autorisée est un délit.

Réception d'un message :

```
TextMessage messageTexte =(TextMessage) receiver.receive();
System.out.println(messageTexte.getText()) ;
```

On remarquera qu'on ignore la nature d'un message à réception, et que l'on ne peut filtrer sur la nature du message. Il faut donc concevoir l'application afin d'utiliser toujours le même type de message pour une destination donnée ou prévoir un aiguillage à l'aide de `instanceof`. Mais l'utilisation de `instanceof` est déconseillée en Java.

BytesMessage

Un `BytesMessage` transporte des données **binaires**. C'est utile pour les images, les fichiers binaires comme les feuilles de calcul Excel, et de manière générale tous les attachements que l'on connaît pour les courriels. JMS peut être considéré comme un système de courriels entre applications et non entre personnes.

On peut également transporter des messages encodés dans tout système de format binaire, tels que `ASN1` très utilisé en télécommunications et en cryptographie et plus compact que XML.

Exemple d'utilisation de `BytesMessage` :

```
BytesMessage outMessage = session.createBytesMessage();
byte[] buffer=readFile("photo.jpg");
outMessage.writeBytes(buffer);
```

Le problème ici est que l'on ne connaît pas forcément la taille du buffer envoyée !

On peut y remédier en mettant une propriété dans l'en-tête du message. Les propriétés seront détaillées au paragraphe suivant.

```
outMessage.setIntProperty("size",buffer.length);
```

Le récepteur du message pourra alors allouer le tampon de la taille voulue :

```
Message message = consumer.receive();
// On teste le type de message
if (message instanceof BytesMessage) {
// Récupère la taille dans une propriété
int size = inMessage.getIntProperty("size");
// on alloue un tampon adapté
byte[] replyBytes = new byte[size];
((BytesMessage) message).readBytes(replyBytes);
```

MapMessage

Un `MapMessage` est utilisé pour envoyer un **dictionnaire de clés et valeurs**. Il se marie facilement avec un `ResultSet` provenant d'une base de données, et fournit les mêmes types natifs (`int`, `float`, etc.).

Copyright © 2015 Dunod.

```
MapMessage message = session.createMapMessage();
message.setInt("Quantity", 4);
message.setStringProperty("itemType", "Screen");
```

Autres types de messages

Un `StreamMessage` peut être utilisé pour envoyer des objets sérialisés (voir chapitre 2) en Java ou tout autre flux. Ses méthodes sont très analogues à celle des classes `java.io.DataInputStream` et `java.io.DataOutputStream` utilisées pour la sérialisation.

Un `ObjectMessage` encodera un objet Java `Serializable`. L'utilisation est très proche des paramètres RMI `Serializable` (voir chapitre 5).

Certaines implémentations offrent d'autres formats de message, par exemple pour XML ou ASN1.

En conclusion, JMS permet une grande variété de formats d'échange, et ne préjuge en rien sur la manière dont ces données seront exploitées. Chaque client JMS peut modifier aisément le contenu sans recompilation, et sans adhérer à une syntaxe stricte d'interface. C'est un des succès de JMS puisqu'il permet à deux applications de communiquer avec un minimum de modifications. En contrepartie, il reste à la charge du développeur de s'assurer que les types de messages échangés sont cohérents, ainsi que la syntaxe de leur contenu : aucune erreur ne pourra être détectée à la compilation. On peut imaginer d'automatiser ce processus en définissant des schémas XML par exemple pour valider le contenu texte d'un message. On notera également que l'analyse syntaxique (*parsing*) d'un message trop long peut être très pénalisante en performances. Enfin, l'encodage du texte en chaînes de caractères sur 16 bits peut être pénalisant en bande passante, tout comme avec SOAP (voir chapitre 9 sur les services web).

8.3.2 Propriétés des messages : filtrage et routage

L'en-tête d'un message peut contenir des propriétés qui peuvent être utilisées pour filtrer les messages ou les re-router vers d'autres destinataires. Les propriétés sont des couples (nom, valeur) utilisables par le système ou l'application pour sélectionner les messages ou les destinataires. Ceci convient naturellement à la fois aux `Queues` et aux `Topics`.

Les propriétés peuvent être des types natifs comme `boolean`, `byte`, `short`, `int`, `long`, `float`, `double` et `String`.

Le filtrage des messages par propriétés (figure 8.13) permet de ne pas avoir à ouvrir le contenu du message avant de le router, et ainsi de ne pas faire jouer les accusés de réception.

Le filtrage s'appuie sur une syntaxe proche de SQL 92 afin de définir des expressions booléennes sur les propriétés.

```
(Country = ' UK') OR (Country = ' US') OR (Country = ' France')
age >= 15 AND age <= 19
```

Copyright © 2015 Dunod.
© Dunod – Toute reproduction non autorisée est un délit.

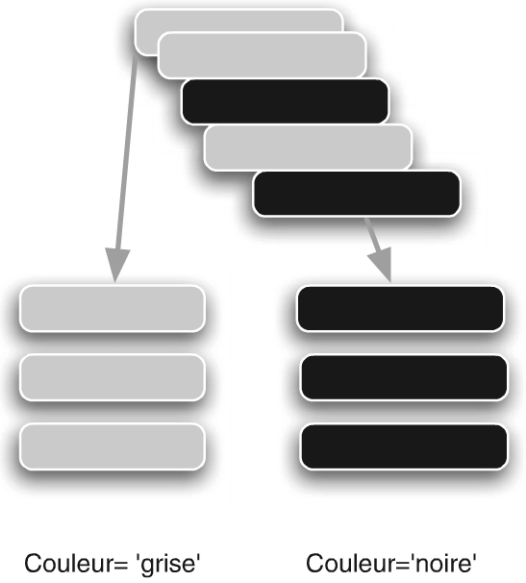


Figure 8.13 — Filtrage des messages par propriétés

Attention les chaînes sont notées comme en SQL avec de simples quotes : `'chaine'`

Envoi du message :

```
TextMessage message=session.createTextMessage("bonjour");
message.setStringProperty("couleur", "grise");
publisher.send(message);
```

À l'arrivée, le filtrage est attaché au consommateur, et non sur la méthode de réception :

```
MessageConsumer receiver = session.createConsumer(topic, "couleur='grise'");
connexion.start() ;
TextMessage mess= (TextMessage) receiver.receive();
```

On peut également utiliser la réception asynchrone en attachant un `MessageListener` au consommateur filtré. Par contre dans tous les cas, le filtre n'est plus modifiable après création du consommateur.

De nombreux usages découlent du filtrage : le routage dans un centre d'appels en fonction du motif d'appel, le routage d'information de bourses en fonction de l'importance du message, etc.

Copyright © 2015 Dunod.

8.3.3 En-tête des messages et qualité de service

L'en-tête du message véhicule des informations de qualité de service. Toutefois rien n'oblige une implémentation à mettre en œuvre tous ces aspects. De plus selon les implémentations les différents niveaux de définition seront implémentés ou non :

- lors de l'envoi du message :

```
producer.send(Message message,
    int deliveryMode, // persistant ou non
    int priority, // priorité
    long timeToLive) // durée de vie
```

- lors de la création du message :

```
message.setDeliveryMode(DeliveryMode.PERSISTENT)
```

- ou attachés au producteur pour tous ses messages :

```
producer.setTimeToLive(10000);
```

On trouve plusieurs attributs dans l'en-tête du message (méthode `getXXX` du message) :

Le mode de livraison (*deliveryMode*) peut être persistant ou non : `DeliveryMode.PERSISTENT` ou `NON_PERSISTENT`. En pratique, les messages seront quasiment toujours persistants, par contre on pourra jouer sur leur durée de vie (*TimeToLive*) exprimée en milliseconde (ms). Une durée de vie de 0 indique par convention que le message n'expirera jamais.

Si l'envoi d'un message persistant échoue, celui-ci sera renvoyé dans le mode en point à point ou le mode publication/abonnement avec abonnement durable. De même un message persistant ne sera supprimé que sur accusé de réception du consommateur.

Le renvoi du message peut-être détecté en inspectant le champ `JMSRedelivered`. En effet, JMS assure la livraison du message en réitérant celle-ci sur erreur.

Copyright © 2015 Dunod.

Champ d'en-tête	Fonction	Défini
`JMSDestination`	La `Queue` ou le `Topic`.	Automatiquement lors du `send`
`JMSDeliveryMode`	Définit si le message est persistant ou non.	Renseigné par code
`JMSExpiration`	Définit le temps de rétention d'un message persistant.	Renseigné par code lors du `send`
`JMSMessageID`	Identifiant unique du message.	Généré par le courtier
`JMSPriority`	Priorité du message.	Permet de traiter certains messages en priorité. N'a qu'une portée sur la session
`JMSTimestamp`	Estampille.	Générée par le courtier
`JMSCorrelationID`	Corrélation entre deux messages. Utilisé pour les requêtes-réponses.	Par code
`JMSReplyTo`	Champ optionnel permettant de donner une adresse de retour.	Par code
`JMSType`	Nature du contenu du message (Text, Bytes, etc.)	Courtier
`JMSRedelivered`	Indique s'il s'agit d'une nouvelle tentative d'envoi sur échec d'acquittement.	Courtier

8.4 FIABILITÉ DE LA COMMUNICATION

Une des motivations de l'utilisation de JMS va être la fiabilité du transport des informations.

8.4.1 Abonnés durables

Lors de la réception d'un message en mode publication/abonnement, le client doit être présent, il ne sera pas livré à nouveau. En effet, comme le nombre de destinataires est dynamique et non connu à l'avance, il est impossible de savoir si tout le monde a reçu le message et de renvoyer celui-ci si cela n'est pas le cas.

Pour fiabiliser la réception des messages et être sûr que tous les clients qui le souhaitent ont reçu le message, ceux-ci doivent souscrire un abonnement durable (*durable subscribers*). Cet abonnement permettra au courtier de s'assurer qu'ils ont reçu le message par acquittement, et de leur renvoyer si ce n'est pas le cas.

Pour cela, il suffit d'une part de donner un identificateur à la connexion, d'autre part de créer le consommateur en spécifiant un nom d'abonnement.

Copyright © 2015 Dunod.

```
ConnectionFactory qcf= .../// selon implémentation
Connection qc= qcf.createConnection("Utilisateur","MotDepasse");
qc.setClientID("MonAppliJMS");
Session s = qc.createSession(false, Session.AUTO_ACKNOWLEDGE);
Topic topic = s.createTopic("Developpement.Durable");
MessageConsumer receiver = s.createDurableSubscriber(topic,"Abonnement1");
receiver.setMessageListener(new Receiver());
qc.start();
```

On vérifiera alors que si le message est persistant et envoyé en mode publication/abonnement, le client pourra le recevoir ultérieurement sur reconnexion.

Pour supprimer l'abonnement, il faut se connecter avec le même `clientID`, et utiliser la méthode `unsubscribe` avec le nom d'abonnement :

```
qc.setClientID("MonAppliJMS");
...
s.unsubscribe("Abonnement1") ;
```

8.4.2 Séquence des messages

Les messages transitent par un nombre quelconque de courtiers. De plus, ils peuvent être associés à des priorités diverses, et le courtier peut être plus ou moins chargé. Les implémentations vont autant que possible respecter la chronologie des messages, mais certains ordinateurs peuvent avoir des horloges non synchronisées, etc. Il est donc possible qu'en bout de chaîne il y ait inversion de l'ordre des messages. Il est important dans la conception d'envisager le cas d'inversion des messages et soit de s'assurer que cela n'a pas d'impact, soit d'y remédier.

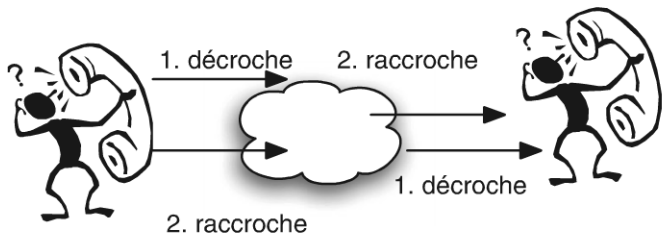


Figure 8.14 — Inversion temporelle des messages

L'inversion des messages ne peut arriver dans un mode RPC pour un même producteur, car il doit attendre le retour de son appel avant d'effectuer le suivant.

La figure 8.14 illustre le cas d'un système de facturation de communications qui ne respecterait pas la chronologie des messages.

Copyright © 2015 Dunod.
© Dunod – Toute reproduction non autorisée est un délit.

8.4.3 Acquittements de messages

Envoi d'un message une fois et une seule

Dans certaines applications, il est très important que le message soit envoyé une fois et une seule, par exemple pour un retrait bancaire. La figure 8.15 illustre cette sémantique. Si lors d'un retrait d'argent, le serveur de banque renvoie une erreur de connexion, il peut être impossible de savoir si elle a eu lieu avant ou après la prise en compte de la transaction. Si on renvoie le message de débit, cela peut être au détriment de l'utilisateur qui sera furieux d'être débité deux fois, et si on ne le renvoie pas, cela peut porter préjudice à la banque. Obtenir que le message soit envoyé 1 fois, et non 0 ou 2 peut être très important.

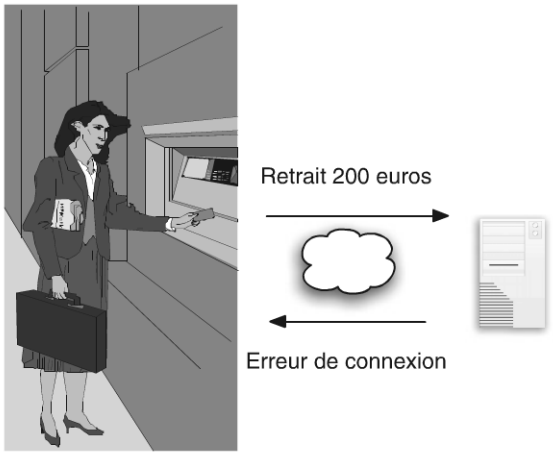


Figure 8.15 — La transaction a-t-elle été prise en compte ?

Pour cela, on utilisera des mécanismes sophistiqués d'acquittement.

Acquittements gérés par le système

Une des manières d'assurer la fiabilité est de pouvoir gérer les accusés de réception. Les accusés de réception ne sont gérés que pour les `Queue` ou pour les `Topic` avec des abonnés durables. Par défaut, on utilise l'attribut `Session.AUTO_ACKNOWLEDGE`, mais il existe deux autres possibilités.

```
connection.createSession(false, Session.CLIENT_ACKNOWLEDGE);
```

`AUTO_ACKNOWLEDGE` signifie que respectivement lors du `send` le client JMS sera en attente d'un accusé de réception du courtier, et inversement lors du message `receive`. Si un incident survient, le message n'est pas considéré comme reçu et peut être renvoyé à nouveau par le courtier. La figure 8.16 illustre ce concept.

En mode automatique, l'acquittement est envoyé après chaque message. Si le message est renvoyé à nouveau, il portera un drapeau `JMSRedelivered` dans son en-tête. Le courtier utilise l'attribut `JMSMessageId` pour vérifier que les messages sont envoyés une fois et une seule.

Copyright © 2015 Dunod.

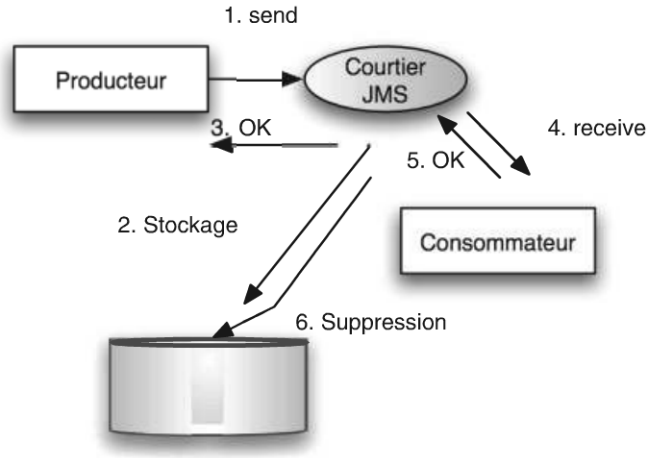


Figure 8.16 — Le message est supprimé après accusé de réception

En mode `CLIENT_ACKNOWLEDGE`, le consommateur a la responsabilité d'acquitter le message par programme à tout moment de la session :

```
message.acknowledge() ;
```

Il peut ainsi acquitter un paquet de messages d'un coup (figure 8.17), ce qui peut optimiser les performances. Si jamais il y a une exception avant l'acquittement global, tous les messages seront renvoyés au prochain `receive`, même ceux qui avaient déjà été reçus.

En mode `DUPS_OK_ACKNOWLEDGE`, on spécifie au courtier qu'il est possible de recevoir plusieurs fois le même message. Ce mode n'est pas toujours implémenté par les fournisseurs de JMS, car il suppose que vérifier que le message a été reçu au plus une fois est plus pénalisant, et il s'agit d'une simple optimisation.



Figure 8.17 — Réception de plusieurs messages avant acquittement

Copyright © 2015 Dunod.
© Dunod – Toute reproduction non autorisée est un délit.

Il n'aura pas échappé à la sagacité du lecteur, que si le message est détruit à la lecture, il est impossible d'obtenir un navigateur de messages ! En effet, l'observateur détruit alors ce qu'il est censé observer. Heureusement il existe des classes `QueueBrowser` qui permettent d'énumérer les messages sans les détruire du magasin à messages.

Acquittement de messages applicatif

Le modèle de communication JMS est asynchrone, par envoi de requêtes sans réponses. Il peut être souhaitable de recevoir une réponse de manière différée, mais comment alors l'apparier avec la requête initiale (figure 8.18) ?

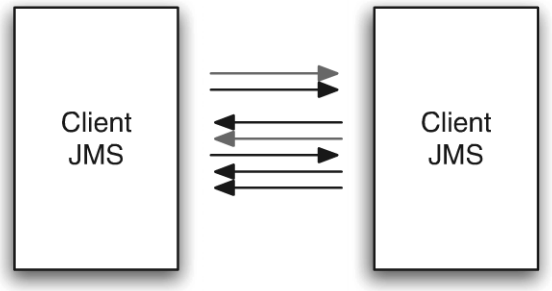


Figure 8.18 — Comment apparier les requêtes et les réponses ?

Il existe plusieurs éléments de réponses à ce problème :

- utiliser l'attribut `JMSCorrelationID` dans l'en-tête du message ;
- utiliser des objets `QueueRequestor` ;
- utiliser des `Queues` temporaires de réponse (`TemporaryQueue`).

L'utilisation des `QueueRequestor` est simple d'emploi : cet objet se substitue au `MessageProducer` et au `MessageConsumer`, et va rester bloqué en attente sur la méthode `request`.

```
requestor = new QueueRequestor(session, queue);
connection.start();}
TextMessage requete = session.createTextMessage();
msg.setText("bonjour" );
Message reponse = requestor.request(requete);
```

Pendant ce temps, son interlocuteur va devoir répondre de manière particulière. Il va extraire de l'en-tête du message l'adresse de retour, et également apparier les numéros de corrélation de la requête et de la réponse.

```
String requete = messageRequete.getText();
Queue replyQueue = (Queue) messageRequete.getJMSReplyTo();
if (replyQueue != null) {
// Renvoie la réponse.
```

Copyright © 2015 Dunod.

```
TextMessage reponse = session.createTextMessage(("Reponse");
reponse.setJMSCorrelationID(aMessage.getJMSMessageID());
MessageProducer producteur...
producteur.send (replyQueue, reponse);
```

Si on ne peut utiliser de `QueueRequestor`, on peut également créer une `Queue` temporaire manuellement et l'affecter lors de l'envoi de la requête :

```
Message.setJMSReplyTo(session.createTemporaryQueue()) ;
```

Tout se passe exactement comme si vous demandiez un document officiel à une administration en incluant dans votre courrier une enveloppe de retour à votre adresse (figure 8.19). Le correspondant peut récupérer la destination de retour dans le champ `JMSReplyTo` du message.

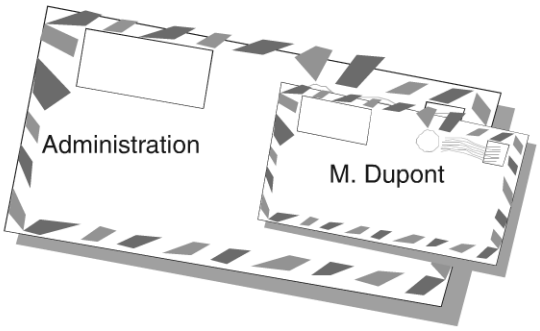


Figure 8.19 — Utilisation d'une enveloppe de retour à l'expéditeur

8.4.4 Utilisation des transactions

Les transactions ne sont pas réservées aux bases de données. La motivation d'une transaction en JMS est de permettre de rendre des messages solidaires au sein d'une session afin de s'assurer qu'ils sont tous envoyés ou reçus, ou aucun d'entre eux. Dans l'exemple de requête-réponse précédent, deux opérations interviennent sur l'interlocuteur :

- la réception de la requête,
- l'envoi de la réponse.

Il est souhaitable que la réponse n'ait lieu que si la requête a été bien reçue, et de même que la requête bien reçue comporte une réponse. On peut donc lier ces deux opérations dans une même transaction (figure 8.20). La délimitation de la transaction sera le début et la fin de la session.

Concrètement, dans le flux de circulation d'information, on voudra par exemple s'assurer de déclencher une livraison de produit uniquement si l'on a reçu le paiement, et inversement mettre obligatoirement en livraison toutes les commandes payées.

Copyright © 2015 Dunod.
© Dunod – Toute reproduction non autorisée est un délit.

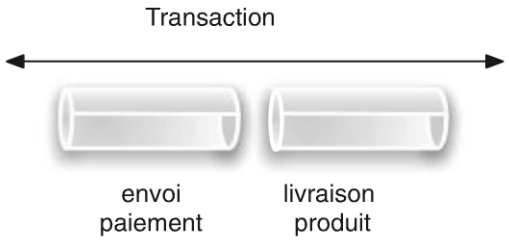


Figure 8.20 — Une transaction permet de lier solidairement deux opérations

Concrètement, la transaction est attachée à la session lors de sa création (booléen `true` pour indiquer la présence d'une transaction) :

```
connexion.createSession(true, Session.AUTO_ACKNOWLEDGE) ;
```

La session peut alors procéder à des émissions et réceptions de messages. La transaction sera validée lors du `commit` et invalidée lors du `rollback`.

```
1. messagePaiement =consommateur.receive() ;
2. producteur.send(messageLivraison) ;
3. session.commit() ;
```

Si une erreur survient entre 1 et 2, le message de paiement ne sera pas reçu. En général le `rollback` est utilisé dans une clause d'exception :

```
try{ producteur.send(messsageFacturation) ;
producteur.send(messageLivraison) ;}
    catch(Exception e) {session.rollback() ;}
```

8.5 JMS 2.0

JMS 2.0[1] apporte une simplification significative à JMS 1.0 et 1.1.

L'API simplifiée se résume à trois nouvelles interfaces : `JMSContext`, `JMSProducer` et `JMSConsumer`.

- `JMSContext` fusionne les deux objets `Connection` et `Session` en une seule classe.
- `JMSProducer` simplifie l'utilisation de `MessageProducer`. Il permet de choisir des options de livraison des messages, de définir des en-têtes et des propriétés.
- `JMSConsumer` remplace `MessageConsumer` et s'utilise de la même manière.

L'API simplifiée offre toutes les caractéristiques de l'API classique plus quelques ajouts. L'API classique n'est pas obsolete mais reste partie intégrante de JMS. Les

1. http://www.oracle.com/technetwork/articles/java/jms20-1947669.html

Copyright © 2015 Dunod.

développeurs gardent le choix entre l'utilisation des classes historiques (`Connection`, `Session`, `MessageProducer`, `MessageConsumer`) et les nouvelles classes (`JMSContext`, `JMSProducer et JMSConsumer`).

8.5.1 Utilisation de l'API simplifiée pour envoyer un message

La nouvelle API exige moins de code. Les deux codes nécessitent tous deux l'obtention d'une `ConnectionFactory`.

```
static Context ictx = null;
ictx = new InitialContext();
Destination dest = (Destination) ictx.lookup(args[0]);
ConnectionFactory cf = (ConnectionFactory) ictx.lookup("cf");
ictx.close();
```

Voici un exemple de code JMS 1.1. pour envoyer des messages sur le nom de la destination (Queue ou Topic) passée en argument.

```
Connection cnx = cf.createConnection();
Session sess = cnx.createSession(false, Session.AUTO_ACKNOWLEDGE);
MessageProducer producer = sess.createProducer(dest);

int i;
for (i = 0; i < 10; i++) {
  TextMessage msg = sess.createTextMessage("Test numero " + i);
  producer.send(msg);
}
cnx.close();
```

Et voici le même code en utilisant l'API simplifiée :

```
JMSContext context = cf.createContext();
JMSProducer producer = context.createProducer();
int i;
for (i = 0; i < 10; i++) {
     producer.send(dest, "Test numero " + i);
}
```

On observe les modifications suivantes :

- JMSContext remplace les objets `Connection` et `Session`.
- Le mode `AUTO_ACKNOWLEDGE` est automatique, mais peut être modifié en passant un paramètre supplémentaire, par exemple `CLIENT_ACKNOWLEDGE` au constructeur de Session.
- `JMSContext` implémente une méthode `close` tout comme `Connection`, mais il n'est pas nécessaire de l'appeler puisqu'il implémente `AutoCloseable`, qui n'est défini qu'à partir de Java SE 7.
- Inutile de créer un objet `TextMessage`, cela fait est fait automatiquement lors du `send`.

Copyright © 2015 Dunod.
© Dunod – Toute reproduction non autorisée est un délit.

- JMS 1.1 lève des exceptions `JMSException`, alors que JMS 2.0 lève une exception `JMSRuntimeException`.
- `JMSRuntimeException` est une sous-classe de `RuntimeException`, ce qui rend facultatif la capture dans le contexte appelant.

8.5.2 Utilisation de JMS 2.0 pour la réception synchrone des messages

Voici le code pour la version 1.1 :

```
Connection cnx = cf.createConnection();
Session session = cnx.createSession(false, Session.AUTO_ACKNOWLEDGE);
MessageConsumer receiver = session.createConsumer(queue);
cnx.start();

for (int i = 0; i < 10; i++) {
     Message msg = receiver.receive();
     if (msg instanceof TextMessage)
        System.out.println("Msg reçu: " + ((TextMessage) msg).getText());
     else if (msg instanceof ObjectMessage)
        System.out.println("Msg reçu: "
                           + ((ObjectMessage) msg).getObject());
      else
        System.out.println("Message reçu: " + msg);
    }
cnx.close();
```

Et le code pour la version 2.0 :

```
    JMSContext context = cf.createContext();
    JMSConsumer recv = context.createConsumer(dest);
    String leMessage = null;
    for (int i = 1; i < 10; i++) {leMessage=recv.receiveBody(String.class);
    System.out.println("Message reçu :" + leMessage);}
```

On observe les mêmes modifications que pour l'envoi, mais également l'omission du `start()`.

8.5.3 Utilisation de JMS 2.0 pour la réception asynchrone des messages

Voici le code pour la version 1.1 :

```
Connection cnx = cf.createConnection();
Session session = (Session) cnx.createSession(false,
                                              Session.AUTO_ACKNOWLEDGE);
session.setQueueMessageReadMax(100);
MessageConsumer consumer = (MessageConsumer) session.createConsumer(queue);
consumer.setMessageListener(new MyListener());
cnx.start();
System.in.read();
session.close();
cnx.close();
```

Copyright © 2015 Dunod.

La méthode `System.in.read()` permet une attente bloquante d'une touche tapée au clavier sinon le code terminerait sans nécessairement avoir reçu les messages. Dans le paragraphe précédent la réception synchrone était bloquante, et cette méthode était inutile.

Code pour la version 2.0 :

```
JMSContext context = cf.createContext();
JMSConsumer recv = context.createConsumer(dest);
recv.setMessageListener(new MsgListener("Ecouteur sur " + nomFile));
System.in.read();
```

Même remarque que précédemment, plus de `start()` et un code plus compact.

8.5.4 Autres simplifications en JMS 2.0

La réception est également simplifiée. Au lieu de :

```
Message message = consumer.receive(1000); // renvoie un TextMessage
String body = ((TextMessage) message).getText()
```

On peut écrire :

```
Message message = consumer.receive(1000); // renvoie un TextMessage
String body = message.getBody(String.class);
```

Mais aussi selon le type du message :

```
byte[] bytes = message.getBody(byte[].class)
Map body = message.getBody(Map.class);
```

Ou pour un type d'objet personnalisé et `Serializable` :

```
MyObject body = message.getBody(MyObject.class);
```

8.5.5 Nouvelle fonctionnalité : plusieurs consommateurs sur le même abonnement

En JMS 1.1, il est possible à plusieurs consommateurs de recevoir les messages d'un même Topic en utilisant des abonnements, mais il n'est pas possible de faire de l'équilibrage de charge en alternant par exemple la réception sur deux récepteurs. En Java EE c'est possible en utilisant des Message-Driven Beans (MDB), mais pas en Java SE.

JMS 2.0 offre une nouvelle fonctionnalité pour cela : les consommateurs partagés. Lors de la création de l'abonnement,on spécifie un nom :

```
JMSConsumer conso1 = context.createSharedConsumer(topic,"abonnementPartage");
```

Copyright © 2015 Dunod.
© Dunod – Toute reproduction non autorisée est un délit.

Et l'on peut implementer un autre consommateur sur une autre machine ou dans une autre JVM pour l'équilibrage de charge :

```
JMSConsumer conso2 = context.createSharedConsumer(topic,"abonnementPartage");
```

En mettant précisément le même nom d'abonnement. En définitive, il y a quatre sortes d'abonnement :

- Non durable et non partagé : `createConsumer`
- Durable non partagé : `createDurableConsumer`
- Partagé non durable : `createSharedConsumer`
- Partagé durable : `createSharedDurableConsumer`

8.5.6 Nouvelle fonctionnalité : délai de livraison

Il est possible de spécifier un délai de livraison :

- Soit avec le nouveau style :

```
JMSContext context = connectionFactory.createContext();){
context.createProducer().setDeliveryDelay(20000).send(queue,"Bonjour");
```

On notera que `setDeliverayDelay` renvoie le producteur lui-même afin d'enchaîner les méthodes.

- Soit avec l'ancien style :

```
Connection connection = connectionFactory.createConnection();){
Session session = con.createSession();
MessageProducer messageProducer = session.createProducer(queue);
messageProducer.setDeliveryDelay(20000);
TextMessage textMessage = session.createTextMessage("Bonjour");
messageProducer.send(textMessage);
```

8.5.7 Nouvelle fonctionnalité : envoi asynchrone

Cette caractéristique n'est pas disponible pour les applications dans un conteneur EJB. Dans un envoi asynchrone, la méthode `send` n'est pas bloquante.

Il suffit d'attacher un `CompletionListener` au producteur pour spécifier quelle action va être faite sur réponse du serveur.

Code du CompletionListener

Il doit implémenter l'interface `CompletionListener`. Il utilise ici une primitive de synchronisation `CountDownLatch`. Cette primitive permet de déclencher une action lorsque son compteur atteint zéro.

Copyright © 2015 Dunod.

```
class MyCompletionListener implements CompletionListener {

   CountDownLatch latch;
   Exception exception;

   public MyCompletionListener(CountDownLatch latch) {
      this.latch=latch;
   }
   public void onCompletion(Message message) {
      latch.countDown();
   }

   @Override
   public void onException(Message message, Exception exception) {
      latch.countDown();
      this.exception=exception;
   }

   public Exception getException(){
      return exception;
   }
}
```

On peut alors écrire le code du récepteur avec un compteur à un :

```
JMSContext context = connectionFactory.createContext();){
CountDownLatch latch = new CountDownLatch(1);
MyCompletionListener myCompletionListener = new MyCompletionListener(latch);
context.createProducer().setAsync(myCompletionListener).send(queue,"Bonjour le
monde");
System.out.println("Message envoye, attente de réponse");

      // À cet instant on peut faire autre chose en attendant la réponse

latch.await();
if (myCompletionListener.getException()==null){
         System.out.println("Réponse recue du serveur");
      } else {
         throw myCompletionListener.getException();
      }
```

8.5.8 Nouvelle fonctionnalité : comptage de livraison

Il est à présent possible de savoir combien de fois un message est livré. Ceci est très important par exemple en comptabilité, si l'on souhaite qu'une écriture soit passée une fois et une seule.

Cette information est obtenue à l'aide de la propriété `JMSXDeliveryCount`. Dans l'exemple suivant, on simule une erreur sur réception du message jusqu'à 10 fois afin de tester les itérations de livraison :

Copyright © 2015 Dunod.
© Dunod – Toute reproduction non autorisée est un délit.

```
    public void onMessage(Message message) {
       try {
          int compteurLivraison = message.getIntProperty("JMSXDeliveryCount");
     if (compteurLivraison <10){
        // on génère une RuntimeException
        // pour simuler un problème de reception de message
        // le message va être livré à nouveau
        throw new RuntimeException("Exception pour simuler un message erroné");
          } else {
        // le message a été livre 10 fois,
        // pour éviter de boucler infiniment on peut penser
        // à l'envoyer sur une dead message queue
     }
...
```

En résumé

JMS est une API définie dans J2EE qui permet deux modes de communications : par messages ou par événements. Le mode par événement permet une diffusion fiable d'une information vers plusieurs interlocuteurs. Elle permet l'implémentation d'une qualité de service sophistiquée (transactions, acquittement, priorités) mais il faut valider chaque implémentation pour savoir quel est le niveau de qualité de service mis en œuvre. Les communications transitent en général par un stockage intermédiaire qui peut induire des aléas de performance en fonction de la charge du courtier. Certains éditeurs introduisent des solutions hautement disponibles pour augmenter la fiabilité en couplant plusieurs courtiers.

JMS 2.0 a été introduit en avril 2013 et simplifie l'API de JMS tout en rajoutant des fonctionnalités telles que le comptage de livraison des messages.

Copyright © 2015 Dunod.

9

SOAP et services web

Objectif

Ce chapitre fait une courte présentation des services web qui demanderaient un tome entier pour une description détaillée. De plus les standards sont en perpétuelle redéfinition, et il est difficile de présenter autre chose qu'un cliché instantané des technologies actuelles. On s'attachera à montrer les aspects de communication en point à point, en soulignant les évolutions depuis les modèles CORBA et RMI.

9.1 COMMUNICATION ENTRE APPLICATIONS SUR INTERNET

9.1.1 Enjeux du « B to B »

Avec les services web, l'enjeu de la communication entre applications devient planétaire : il s'agit en effet de faire communiquer par exemple un service de réservation aérienne en un endroit donné avec un service de réservation d'hôtel à plusieurs milliers de kilomètres de là. Loin de la vision limitée de la communication d'entreprise, on veut pouvoir effectuer des transactions ou des échanges de données entre entreprises (B to B *Business to Business*). Ceci est essentiel à noter pour les choix d'architectures, car on souhaite ici faire communiquer de gros ensembles en couplage lâche, avec le minimum de connaissance sur l'interlocuteur, et sans pouvoir imposer aucune contrainte de service : si le site distant n'est pas disponible, il n'y a rien à faire.

Copyright © 2015 Dunod.
© Dunod – Toute reproduction non autorisée est un délit.

9.1.2 Intérêt d'un mode RPC sur Internet

Les services web sont des applications utilisant Internet pour interagir dynamiquement avec d'autres programmes en s'appuyant sur des standards issus le plus souvent du W3C (*World Wide Web Consortium*)[1] :

- **XML** (*Extensible Markp Language*)[2] pour le formatage des données,
- **HTTP** (*Hypertext Transfer Protocol*)[3] pour le protocole de transport,
- **SOAP** (*Simple Object Access Protocol*)[4] pour le protocole de communication.

XML est un standard du W3C qui permet de structurer des données sous forme textuelle à l'aide de balises, tout comme le langage HTML permet de décrire des pages web avec des balises. Voici un exemple de XML :

```
<personne>
    <nom>Dupont</nom>
    <prenom>Jean</prenom>
</personne>
```

Les balises sont des termes entre « <> » qui définissent la syntaxe de la structure de données. La syntaxe d'un document XML peut être décrite avec XML Schema[5], un autre standard du W3C. Le schéma décrira le type de données utilisées (`xs:string`, `xs:double`...), les types de données composites, la cardinalité et la séquence des types de données.

HTTP est un protocole de communication au-dessus de TCP/IP qui permet d'effectuer des requêtes sur un serveur web.

SOAP est un protocole de type RPC qui utilise XML pour sérialiser les méthodes et leurs arguments, ainsi que les valeurs de retour. Avec SOAP, XML transite en général au-dessus du protocole HTTP.

Les interfaces des services web sont décrites en WSDL (*Web Services Description Language*), qui joue le même rôle que l'IDL de CORBA (voir chapitre 6 et chapitre 4). Cette description est suffisante pour utiliser le service sans connaître son implémentation, y compris à distance par Internet : en effet le service est couplé à un serveur web qui le met à disposition sur Internet. L'interface WSDL est elle aussi publiée sur Internet.

Le but est de permettre à une application de communiquer sur Internet avec le service dont elle a besoin et d'échanger des données avec lui. Si le service est payant, la procédure de règlement sera comprise dans la description du service.

1. *http://www.w3c.org*
2. *http://www.w3.org/XML/*
3. *http://www.w3.org/Protocols/*
4. *http://www.w3.org/TR/soap/*
5. *http://www.w3.org/XML/Schema*

Copyright © 2015 Dunod.

On remarquera que l'infrastructure de mise en œuvre est plus lourde que pour les solutions telles que CORBA et RMI, puisqu'il faut utiliser un serveur web permettant de répondre à des requêtes HTTP. Par contre, cette solution est incontournable dès que l'on souhaite faire transiter des informations à travers un pare-feu, car aucun des protocoles tels que RMI ou CORBA ne peut le faire[1].

Il ne faut pas surestimer le coût de mise en œuvre d'un serveur HTTP : la version 6 de Java en intègre un simplifié de manière native. Un serveur HTTP peut également tourner sur un équipement embarqué avec quelques classes Java, et le domaine des services web pour les mobiles est potentiellement illimité.

L'utilisation de SOAP a été également rendue populaire par Microsoft, alors que .Net Remoting très proche de RMI n'a jamais été poussé en avant. Le protocole SOAP 1.2 est une recommandation du W3C depuis juin 2003[2].

9.1.3 Service et... qualité de service

Le nom de **service** vient du monde du *mainframe* où un « service » est lié à une transaction et définit une suite d'opérations cohérentes qui rend un résultat. De manière plus générale, le concept de service réintroduit l'aspect *workflow* et la décomposition fonctionnelle dans un monde où le tout objet l'avait rendu hérétique, contrairement aux méthodes de conception type SADT des années quatre-vingt.

On peut donc voir la notion de « service » comme un retour du fonctionnel.

Dans le monde des services web, on conservera également l'ambiguïté entre des concepts de documents XML et des concepts d'objets.

La notion de service est souvent liée à celle de « **qualité de service** ». Si la spécification initiale de SOAP est tout à fait minimaliste, la vision IBM de SOA (*Service Oriented Architecture*) inclut en premier lieu la qualité de service, au sens des services CORBA (voir chapitre 6). Dans cette perspective, il s'agit de gérer des aspects non fonctionnels transversaux aux différents intergiciels :

- la gestion des transactions,
- la gestion de la sécurité,
- les annuaires, etc.

Toutefois, les spécifications récentes de SOAP permettent une interopérabilité de la qualité de service, notamment en ce qui concerne les transactions et la sécurité. Ceci est en forte évolution et ne sera pas développé dans ce livre.

1. On signalera toutefois le produit ICE (*http://www.zeroc.com/ice.html*) qui ressemble à CORBA mais permet de traverser des pare-feu. On peut également utiliser des techniques de *tunneling*, non décrites ici.
2. *http://www.w3.org/TR/soap12-part1/*

Copyright © 2015 Dunod.
© Dunod – Toute reproduction non autorisée est un délit.

9.1.4 Standardisation des services web

Les services web sont l'enjeu d'une lutte commerciale féroce et d'une activité de publicité intense. De nombreuses normes s'affrontent, en particulier celles issues du W3C[1] et celles issues du consortium OASIS[2]. SOAP 1.2 est une recommandation du W3C depuis juin 2003, mais de nombreux autres standards tentent de définir une interopérabilité entre services web y compris sur les aspects non fonctionnels (transactions, sécurité, etc.). On notera par exemple les travaux importants du consortium WS-I[3] (*Web Services Interoperability Organization*) qui ont posé les fondements de l'interopérabilité entre web services avec le *Basic Profile* et font de même avec le *Basic Security Profile* publié en mars 2007. Par contre les standards de l'OASIS et du W3C sont parfois en compétition par exemple sur la sécurité avec *WS-SecurityPolicy* pour le W3C et *WS-Security* pour Oasis. L'actualité est changeante et très controversée dans tous ces domaines, et ce livre n'abordera pas ces aspects en détail.

Oracle n'est pas en reste avec une activité importante de normalisation des services web et de leur intégration à Java, notamment avec le JSR 181 qui permet de représenter un service web à l'aide d'annotations.

Le langage WSDL (*Web Service Description Language*) permet tout comme l'IDL CORBA de décrire les interfaces de services web, et il a été normalisé comme note du W3C sous la version 1.1 en mars 2001. La version 2.0 est une recommandation du 26 juin 2007.

Ce livre ne parlera pas d'une extension importante des services web décrivant leur coordination et la description des processus métier, que ce soit ebXML, BPEL ou d'autres langages de chorégraphie.

De manière générale l'utilisation des services web va reposer sur une multitude de standards dans des versions spécifiques à chacun (XML 1.0 avec HTTP 1.1, SOAP 1.2, WSDL 2.0, etc.) et l'utilisateur devra redoubler d'attention pour mettre en œuvre une solution réellement interopérable.

9.2 ARCHITECTURE DE SOAP

9.2.1 Trois composants fondamentaux

Tout comme on retrouve dans l'architecture CORBA l'ORB, l'IDL, le service de noms, on retrouve trois composants équivalents dans les services web :

- **Invocation distante des services web** : SOAP – l'équivalent de l'ORB, ou bus logiciel.
- **Description des services web** : WSDL l'équivalent de l'IDL ou description de l'interface.

1. *http://www.w3c.org*
2. *http://www.oasis-open.org*
3. *http://www.ws-i.org/*

Copyright © 2015 Dunod.

- **Enregistrement et découverte de services web** (figure 9.1) : UDDI (*Universal Description, Discovery, and Integration*) est l'équivalent du service de noms CORBA. On notera toutefois que UDDI est relativement peu utilisé, bien que normalisé depuis 2005 sous sa version 3 par l'Oasis. Sa complexité initiale n'a pas permis un développement large d'annuaires de services. De plus, la description d'un service web à usage planétaire pose des problèmes de sémantique, comme le montre la figure 9.1. Comment rechercher un service web dans le monde entier ? Quelle est la nature des résultats ? La qualité de service attendue ? Mais comme l'URL du service suffit à pouvoir l'utiliser, UDDI n'est pas indispensable à l'utilisation des services web. L'utilisation des URL a fortement influencé les autres annuaires qui ont défini des pseudos-URL, comme nous l'avons vu avec RMI et CORBA (`corbaloc`).

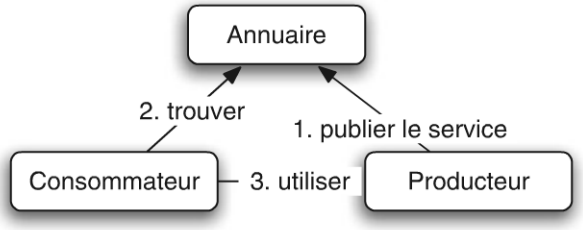


Figure 9.1 — Un monde idéal où le consommateur pourrait trouver le service qu'il souhaite sur Internet

9.2.2 Requêtes HTTP et messages SOAP

Une requête HTTP est tout simplement une chaîne de caractères répondant au format HTTP.

- Exemple de requête GET :

```
GET /site/page.html HTTP/1.1
```

- Exemple de requête POST :

```
POST /site/page.html HTTP/1.1
Content-Type: text/plain
Content-Length: 7
Bonjour
```

Une réponse HTTP est tout aussi simple :

```
200 OK
Content-Type: text/plain
Content-Length: 12
Hello, World
```

Copyright © 2015 Dunod.
© Dunod – Toute reproduction non autorisée est un délit.

On va donc pouvoir en SOAP coder l'appel de méthode distant dans une requête HTTP en utilisant XML. SOAP consiste en l'envoi de messages à des objets distants avec du XML inclus dans des requêtes et réponses HTTP. On verra donc passer sur le réseau des requêtes SOAP telles que celles-ci :

```
POST /StockQuote HTTP/1.1
Host: www.stockquoteserver.com
Content-Type: text/xml; charset="utf-8"
Content-Length: nnnn
SOAPAction: "Some-URI"
<SOAP-ENV:Envelope
    xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
    SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
    <SOAP-ENV:Body>
        <m:GetLastTradePrice xmlns:m="Some-URI">
        <symbol>DIS</symbol>
        </m:GetLastTradePrice>
    </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Dans l'exemple précédent, on remarquera la structure du message comme l'indique la figure 9.2.

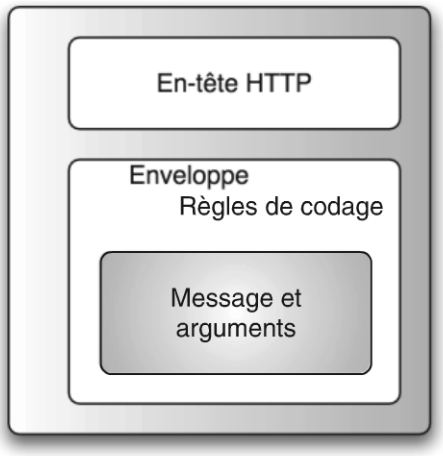


Figure 9.2 — Structure d'un message SOAP

La réponse va être glissée dans une enveloppe SOAP et transiter par le réseau comme ceci :

```
HTTP/1.1 200 OK
Content-Type: text/xml; charset="utf-8"
Content-Length: nnnn
<SOAP-ENV:Envelope xmlns:SOAP-ENV=
    "http://schemas.xmlsoap.org/soap/envelope/"
    SOAP-ENV:encodingStyle= "http://schemas.xmlsoap.org/soap/encoding/"/>
    <SOAP-ENV:Body>
```

Copyright © 2015 Dunod.

```
        <m:GetLastTradePriceResponse xmlns:m="Some-URI">
            <Price>34.5</Price>
        </m:GetLastTradePriceResponse>
    </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

9.2.3 Versalitité de SOAP

SOAP est un protocole ambivalent :

- On peut le considérer comme un nouvel outil **RPC** (*Remote Procedure Call* ou appel de méthode distant) entre objets : les appels de méthodes et leurs arguments sont sérialisés en XML.
- On peut le considérer comme un protocole **d'envoi de messages** : une requête comporte un seul message, une réponse un seul message.
- On peut le considérer comme un simple protocole d'échange de **documents** XML, utile à l'EDI (Échange de données informatisées ou *Electronic Data Interchange*).

Ce qui différencie SOAP de XML-RPC est la présence de l'enveloppe, ainsi que la description des interfaces avec un langage spécifique, WSDL, qui sera décrit au paragraphe 9.3. Cette enveloppe sert à préciser des informations de service, de codage des arguments, mais de plus en plus s'y ajoutent des informations d'acheminement et de qualité de service. Certains pensent que la complexification de l'enveloppe va à terme condamner SOAP, mais le principe d'interopérabilité subsistera certainement.

SOAP dans sa version 1.2 est en fait plus complet qu'un simple RPC avec un mécanisme de requête et de réponse, dans la mesure où il permet également une communication asynchrone, et des envois de messages dans un seul sens.

9.2.4 Interopérabilité des protocoles : les attaches

À leur début les services web ont été abondamment critiqués car ils n'apportaient pas la qualité de service des autres intergiciels : HTTP est en effet un protocole sans état qui ne permet pas de gérer une session ni de gérer l'état d'un dialogue. C'est pour cela qu'HTTP est considéré comme un protocole non fiable. IBM avait lancé l'idée d'un HTTP fiable (RHTTP, *Reliable HTTP*), mais HTTP est à la fois la force et le talon d'Achille des services web.

Avec la version SOAP 1.2, il est possible d'utiliser plusieurs protocoles pour implémenter une communication SOAP : on appellera cela une attache (*binding*). On notera en particulier l'utilisation possible de JMS[1] au lieu de HTTP afin d'obtenir des communications fiables avec SOAP (figure 9.3).

1. Par exemple avec Synapse : *http://synapse.apache.org*

© Dunod – Toute reproduction non autorisée est un délit.
Copyright © 2015 Dunod.

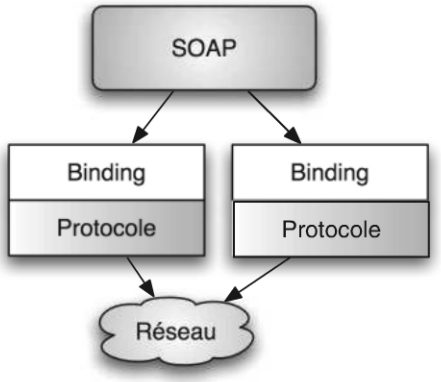


Figure 9.3 — SOAP peut se lier avec l'un ou l'autre protocole réseau

9.3 WSDL

9.3.1 Un langage d'interface

WSDL[1] permet de définir l'interface d'un service web tout comme un IDL CORBA (voir chapitre 6). Toutefois, ce descripteur est plus riche que l'IDL car il décrit à la fois l'interface et une partie de l'implémentation (point d'accès) dans deux parties séparées.

Un fichier WSDL est un document au format XML qui décrit :

- **Ce que fait le service** : la liste des opérations, leur sens (requête seule ou requête-réponse), leur signature (le type des arguments). Cette partie est très analogue à un IDL CORBA.
- **Où le trouver (nommage)** : le nom du service, et le point d'attache (*endpoint*) le plus souvent l'URL du serveur web.
- **Comment l'invoquer** : les règles d'encodage des arguments, le protocole de transport utilisé.

On voit ici que WSDL est plus riche que l'IDL, puisqu'il comporte non seulement la description de l'interface à proprement parler, mais aussi des indications d'implémentation, notamment le port de connexion au travers de l'URL du point d'attache (*endpoint*). Nous allons voir que WSDL contrairement à l'IDL n'a pas été conçu pour être écrit à la main, mais généré par un outil de développement. Il devient à peu près illisible pour un programmeur non averti.

On peut toutefois discuter pour savoir s'il faut combiner au même endroit la description abstraite (types de données et signatures des méthodes) avec les aspects de déploiement (port de connexion).

1. *http://www.w3.org/TR/wsdl.html* et pour la version 2 *http://www.w3.org/TR/wsdl20/*.

Copyright © 2015 Dunod.

9.3.2 Structure d'un document WSDL

Un document WSDL peut couramment comporter plusieurs pages. Contrairement à l'IDL il sera généré par un outil et non manuellement.

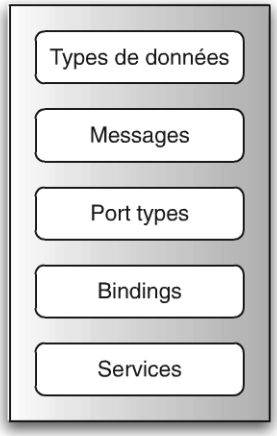


Figure 9.4 — Structure d'un document WSDL

En WSDL, un service est défini en plusieurs parties distinctes (figure 9.4).

1) La description des types de données

La description des types de données est optionnelle et permet de définir des structures de données spécifiques à l'application. Les types peuvent être tirés de la spécification XML Schema (comme `xsd:string`) ou être définis par l'utilisateur par composition de types simples.

2) Les messages

Les messages sont unidirectionnels (sens entrant, `in`, ou sortant, `out`). Une opération RPC comportera donc en général deux messages, la requête et la réponse. Une opération « `oneway` » ne comportera qu'une requête. Un message comporte plusieurs arguments (*parts*), et chaque argument a un type défini par un schéma pouvant figurer dans le cartouche des types de données.

Exemple de messages liés (requête et réponse) :

```
<message name="getRateRequest">
    <part name="country1" type="xsd:string"/>
    <part name="country2" type="xsd:string"/>
</message>
<message name="getRateResponse">
    <part name="Result" type="xsd:float"/>
</message>
```

Ces deux messages définissent une opération qui prendra en argument d'entrée deux chaînes de caractères représentant des pays et renverra en résultat un nombre de

© Dunod – Toute reproduction non autorisée est un délit.
Copyright © 2015 Dunod.

type `float` représentant le cours de change : le WSDL définit la signature des messages, tout comme l'IDL. Cette signature peut comporter des types prédéfinis (tirés ici de XML Schema comme `xsd:float`). Mais il est également possible de définir des types utilisateurs en tête du WSDL (cf. typedef CORBA).

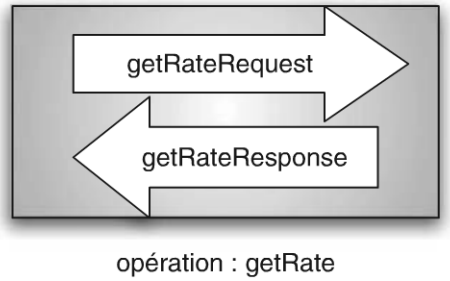


Figure 9.5 — Une opération est composée de deux messages

Voici la définition de l'opération composée des deux messages précédents (figure 9.5) :

```
<operation name="getRate">
    <input message="tns:getRateRequest" />
    <output message="tns:getRateResponse" />
</operation>
```

3) Le PortType

Le **PortType** (ou interface) : ceci définit l'interface abstraite du service comportant un ensemble d'opérations (NB : en WSDL version 2 le `PortType` est rebaptisé interface). Chaque opération peut être soit *in-out* (appel-réponse RPC classique), *in-only* (pas de réponse ou *oneway* à la CORBA), *out-only* et *out-in* (sollicitation de la réponse). Chaque opération définit les messages d'entrée (*In*) ou de sortie (*Out*) ou les deux, selon le mode.

Le PortType définit un ensemble d'opérations, ici la seule opération `getRate` :

```
<portType name="CurrencyExchangePortType">
    <operation name="getRate">
        <input message="tns:getRateRequest" />
        <output message="tns:getRateResponse" />
    </operation>
</portType>
```

4) L'attache

L'**attache** (*binding*) **:** une attache définit un lien entre le `PortType` abstrait et un service réel associé à un protocole et un format. Par exemple, le binding SOAP définit le style d'encodage, l'en-tête de l'action SOAP, le namespace utilisé pour le corps XML (`targetURI`), et ainsi de suite.

Copyright © 2015 Dunod.

Disons sommairement qu'un port type est une liste d'opérations : une « interface » à la CORBA ou mieux une facette à la CORBA 3. Un *binding* est un lien entre cette facette et un protocole (HTTP ou SNMP ou...).

En SOAP, il existe deux styles d'envoi : RPC et document. On peut également spécifier si les arguments sont codés ou littéraux. Ceci définit quatre modes d'appels (RPC ou document, encodé ou littéral, voir paragraphe 9.3.3) à l'origine de beaucoup de problèmes d'interopérabilité dans les années passées. Dans l'exemple suivant, il s'agit d'un binding SOAP en mode rpc/encodé au-dessus du protocole HTTP (paramètre « transport »).

```
<binding name="CurrencyExchangeBinding" type="tns:CurrencyExchangePortType">
    <soap:binding style="rpc"
        transport="http://schemas.xmlsoap.org/soap/http"/>
    <operation name="getRate">
      <soap:operation soapAction=""/>
      <input >
        <soap:body use="encoded" namespace="urn:xmethods-CurrencyExchange"
        encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"/>
      </input>
      <output >
        <soap:body use="encoded" namespace="urn:xmethods-CurrencyExchange"
        encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"/>
      </output>
    </operation>
</binding>
```

5) Le port

Le **port** définit la localisation concrète (`endpoint`) du service, typiquement l'URL de la servlet mettant le service à disposition.

Dans l'exemple suivant, la servlet `http://services.xmethods.net:80/soap/` (le *endpoint*) met à disposition le service `CurrencyExchangeService` avec l'attache décrite précédemment (référence par nom).

```
<service name="CurrencyExchangeService">
    <documentation>Renvoie le taux de change entre les deux monnaies
    </documentation>
    <port name="CurrencyExchangePort"
      binding="tns:CurrencyExchangeBinding">
    <soap:address location="http://services.xmethods.net:80/soap"/>
    </port>
</service>
```

Pour l'appel du service, il est nécessaire de connaître à la fois le point d'attache (*endpoint*) et le nom du service.

En WSDL aujourd'hui, chaque `portType` n'a qu'un seul *binding*, et chaque *binding* n'a qu'un seul `PortType`.

Copyright © 2015 Dunod.
© Dunod – Toute reproduction non autorisée est un délit.

Inversement (et ceci est plus important), chaque service peut avoir plusieurs `portTypes`, chacun avec une adresse et un protocole différent : on peut donc publier un service sous plusieurs protocoles !

9.3.3 Plusieurs styles WSDL : lien avec l'interopérabilité

Un *binding* WSDL peut avoir plusieurs styles : RPC ou document[1]. Ces termes prêtent à confusion, et il ne faut pas croire que l'on ne peut faire communiquer deux applications en mode document. Il faut donc les prendre tels qu'ils sont, sans inférer de sémantique précise si ce n'est celle du mode de communication.

Le passage des informations peut être codé (*encoded*) ou littéral (*literal*). Ceci donne au total quatre méthodes de communication :

1. RPC/encoded
2. RPC/literal
3. Document/encoded
4. Document/literal

Comment reconnaître le style ?

Ces différentes méthodes sont spécifiées dans le WSDL dans l'attribut `type` du *binding* ainsi que dans l'attribut `use` de `soap:binding` :

```
<binding name="TemperatureBinding" type="tns:TemperaturePortType">
    <soap:binding style="rpc"
        transport="http://schemas.xmlsoap.org/soap/http"/>
    <operation name="getTemp">
        <soap:operation soapAction=""/>
        <input>
          <soap:body use="encoded" namespace="urn:xmethods-Temperature"
          encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"/>
        </input>
    ...
</binding>
```

On a donc ici un style `rpc/encoded`.

L'exemple suivant par contre est `document/literal` (doc/lit), souvent utilisé dans le monde Microsoft :

```
<binding name="ServiceSMSSoap" type="s0:ServiceSMSSoap">
    <soap:binding transport="http://schemas.xmlsoap.org/soap/http"
        style="document" />
    ...
    <operation name="SendSmsText">
      <soap:operation
```

1. *http://www.ibm.com/developerworks/webservices/library/ws-whichwsdl/*

Copyright © 2015 Dunod.

```
            soapAction="http://SMSServer.dotnetISP.com/SendSmsText"
            style="document" />
        <input>
            <soap:body use="literal" />
        </input>
        <output>
            <soap:body use="literal" />
        </output>
    </operation>
</binding>
```

Exemple

Supposons que l'on ait une méthode :

```
public void myMethod(int x);
```

Le mode RPC

Si l'on spécifie le style RPC/encoded, le message aura le contenu suivant :

```
<soap:envelope>
    <soap:body>
        <myMethod>
            <x xsi:type="xsd:int">5</x>
        </myMethod>
    </soap:body>
</soap:envelope>
```

Plusieurs raisons pour utiliser le style RPC/encoded :

- Le polymorphisme (plusieurs méthodes avec des signatures différentes).
- Les graphes de données (utilisation de `href` en XML pour pointer d'un élément sur un autre).

Si l'on précise au contraire RPC/literal, le message aura le contenu suivant :

```
<soap:envelope>
    <soap:body>
        <myMethod>
            <x>5</x>
        </myMethod>
    </soap:body>
</soap:envelope>
```

On voit donc que l'on économise le passage de l'information de type. Par contre, il n'est plus possible de valider le contenu du message sans avoir le WSDL décrivant le type comme ceci :

```
<message name="myMethodRequest">
    <part name="x" type="xsd:int"/>
</message>
```

Copyright © 2015 Dunod.
© Dunod – Toute reproduction non autorisée est un délit.

Il est à noter que le WS-I a préconisé dans son *Basic Profile*[1] l'utilisation de SOAP en mode `RPC/literal` ou `document/literal` afin d'éviter les problèmes d'interopérabilité.

Le mode document

Si l'on précise un mode document, le message aura cette forme :

```
<soap:envelope>
    <soap:body>
        <xElement>5</xElement>
    </soap:body>
</soap:envelope>
```

Par contre, l'élément `xElement` doit avoir été défini dans le WSDL :

```
<types>
    <schema>
        <element name="xElement" type="xsd:int"/>
    </schema>
</types>
```

Cette fois, il n'y a pas d'information d'encodage et il est possible de valider le contenu du message, décrit par un schéma, au prix d'une plus grande complexité du WSDL.

Les restrictions de Microsoft .Net

Le framework .Net de Microsoft ne supporte que le style *Wrapped Document-Literal* de WSDL :

- Le message d'entrée a une seule partie (part).
- La partie est un élément.
- L'élément a le même nom que l'opération.
- Le type complexe de l'élément n'a aucun attribut.

Types de données

Le tableau 9.1 précise la traduction des types de données entre WSDL et SOAP.

1. *http://www.ws-i.org/Profiles/BasicProfile-1.0-2004-04-16.html*

Copyright © 2015 Dunod.

Tableau 9.1 — Typage des données en SOAP

Spécification WSDL	Encodage SOAP

Types primitifs
`<element name="price" type="float"/>`
 	
`<element name="greeting" type="xsd:string"/>`
	`<price>15.57</price>`
	
	`<greeting id="id1">Hello</greeting>`

Structures
`<element name="Book"><complexType>`
`  <element name="author"`
`  type="xsd:string"/>`
`  <element name="title"`
`  type="xsd:string"/>`
`  </complexType></element>`
	`<e:Book>`
	`<author>J.R.R Tolkien</author>`
	`<title>A hobbit story</title>`
	`</e:Book>`

Énumérations
`<element name="color">`
`  <simpleType base="xsd:string">`
`  <enumeration value="Green"/>`
`  <enumeration value="Blue"/>`
`  </simpleType>`
`</element>`
	`<color>Blue</color>`

Tableaux	`<SOAP-ENC:Array id="id3" SOAP-ENC:arrayType= xsd:string[2,2]>`
	`  <item>r1c1</item>`
	`  <item>r1c2</item>`
	`  <item>r2c1</item>`
	`  <item>r2c2</item>`
	`</SOAP-ENC:Array>`

9.4 BIJECTIONS ENTRE JAVA ET XML

9.4.1 Modèle objet ou modèle de document

Une des grandes questions pour utiliser des services web est de savoir si l'on souhaite préserver un modèle de document ou un modèle objet.

XML permet de représenter des documents, et l'on peut rester à ce niveau d'analyse en travaillant avec les concepts de documents, de nœud d'arborescence et d'attributs. Par exemple, le listing suivant :

```
<livre>
<auteur>Goscinny</auteur>
<titre langue="francais">Asterix en Hispanie</titre>
<livre>
```

Copyright © 2015 Dunod.
© Dunod – Toute reproduction non autorisée est un délit.

Pourra être vu comme un document de racine `livre`, avec deux éléments, `auteur` et `titre`. L'élément `titre` a un attribut `langue`.

On peut également lui faire correspondre un objet Java de classe `Livre`, comportant les champs auteur et titre comme l'illustre la figure 9.6.

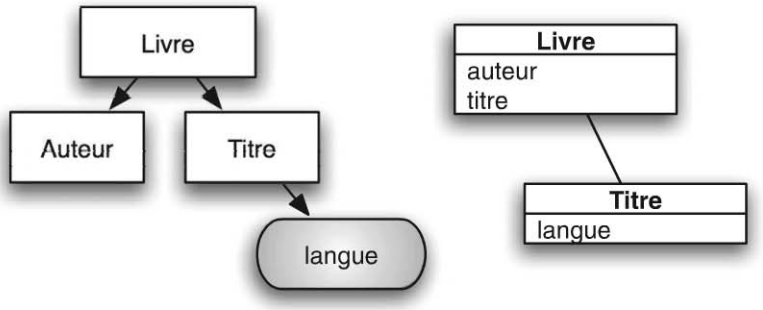


Figure 9.6 — Modèle XML et modèle Java

Doit-on créer une classe `Titre` pour rajouter l'attribut `langue` ? Un champ Java correspond-il à un élément XML ou à un attribut ? Toutes ces questions se compliquent encore lorsque l'on prolonge l'arborescence XML. Si le livre contient des éléments chapitres sous forme de séquence, comment traduire cela en Java ? Est-ce une `Collection` ou une `Hashmap` ? Si la séquence comporte des éléments de balises différentes, faut-il créer une nouvelle classe pour cette séquence ?

9.4.2 Bibliothèques Java-XML

Cette réflexion rapide montre pourquoi il y a tant de manières d'écrire des bijections entre XML et Java. On peut citer par exemple :

- **JAXB**[1] : la bibliothèque d'Oracle, qui est intégrée à Java depuis Java 6. Il s'agit d'un réel mapping objet entre Java et XML.
- **JIXB**[2] : l'équivalent dans le monde Apache.
- **XMLBeans**[3] : une méthode simple à mettre en œuvre.
- **ADB** : *Axis databinding Framework*, une alternative à JIXB dans le monde Apache.

1. *http://jaxb.java.net/*
2. *http://jibx.sourceforge.net/*
3. *http://xmlbeans.apache.org/*

Copyright © 2015 Dunod.

Depuis Java 6, la conversion Java vers XML utilise des annotations comme ci-après :

```
@XmlRootElement
class Person {
    private int _id;
    private String _name;
public Person(){}
public Person(int id, String name){
_id = id;
_name = name;
}
@XmlAttribute
public int getId(){ return _id; }
public void setId(int id){ _id = id; }
@XmlAttribute
public String getName(){ return _name; }
public void setName(String name){ _name = name; }
}
```

9.5 CLIENT SOAP EN JAVA

9.5.1 Choix d'implémentation

Une des difficultés de SOAP n'est pas d'implémenter le code, mais de choisir une des multiples façons de le faire. Il existe plusieurs implémentations de SOAP en Java :

- **Oracle** depuis Java 6 a la sienne propre avec JAX-WS
- **Axis**[1] dans le monde Apache. Notons que la version 2 a été complètement redéfinie par rapport à la version 1. Le modèle de données n'est plus imposé, mais doit être choisi parmi ATOM, JAXB, JIXB, ADB, et XMLBeans.
- **CXF**[2], une alternative à Axis2 sur le site Apache.

Pour la suite, nous nous intéressons à l'implémentation d'Apache et d'Oracle.

9.5.2 Apache Axis

Il est possible d'utiliser la bibliothèque Apache Axis pour communiquer en mode client avec un serveur web. L'application de test choisie permet de récupérer le cours d'une action en temps différé d'une vingtaine de minutes. Apache Axis nécessite l'utilisation d'un serveur web. On peut par exemple télécharger le serveur Tomcat[3] dans le répertoire `$TOMCAT_HOME`. On télécharge ensuite l'archive de l'application Axis (`axis.war`) que l'on copie sous `$TOMCAT_HOME/webapps`.

1. *http://ws.apache.org/axis2/*
2. *http://cxf.apache.org/*
3. *http://tomcat.apache.org/*

© Dunod – Toute reproduction non autorisée est un délit.
Copyright © 2015 Dunod.

On vérifie l'installation en ouvrant un navigateur sur l'URL : *http://localhost:8080/axis*.

9.5.3 Téléchargement du fichier d'interface

La première chose est de charger le fichier WSDL à l'adresse : *http://ws.cdyne.com/delayedstockquote/delayedstockquote.asmx?wsdl*

On nommera ce fichier `delayedStockQuote.wsdl`. On constate que ce fichier offre des opérations telles que `GetQuickStockQuote` et `GetQuickQuoteResponse`.

La requête prend les arguments suivants :

- `StockSymbol` de type `xsd:string`
- `LicenseKey` de type `xsd:string`

Et la réponse renvoie un paramètre de nom `GetQuickQuoteResult` et de type `xsd:decimal`.

Le WSDL fournit également les protocoles utilisables et les points d'accès (*end-points*), ici les protocoles :

- SOAP
- SOAP 1.2
- HTTP GET
- HTTP POST

Ainsi que le point d'accès : *http://ws.cdyne.com/delayedstockquote/delayedstockquote.asmx*

Génération de la souche pour ADB

ADB signifie *Axis Databinding Framework* et permet de générer des classes Java à partir de l'analyse des schémas XML.

Par exemple le schéma XML suivant qui comporte un élément et un type complexe générera deux classes Java, `MyElement` et `SOAPStruct` :

```
<schema xmlns="http://www.w3.org/2001/XMLSchema"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:tns="http://soapinterop.org/types"
    targetNamespace="http://soapinterop.org/types"
    elementFormDefault="qualified" >
<import namespace="http://schemas.xmlsoap.org/soap/encoding/"/>
  <complexType name="SOAPStruct">
    <sequence>
      <element name="varString" type="xsd:string"/>
      <element name="varInt" type="xsd:int"/>
      <element name="varFloat" type="xsd:float"/>
    </sequence>
  </complexType>
<element name="myElement" type="tns:SOAPStruct"/>
</schema>
```

Copyright © 2015 Dunod.

On génère alors de manière automatique le code Java pour cette application :

- Soit en utilisant par exemple le plugin Eclipse pour la génération de code Java (voir sur le site d'Axis).
- Soit en utilisant la commande en ligne de Axis[1] :

```
%AXIS2_HOME%/bin/WSDL2Java -uri DelayedStockQuote.wsdl -p mon.package -d adb
-s -ss -sd -ssi -o build/service
```

On obtient alors une classe `DelayedStockQuoteStub`, qui représente encore une fois une souche pour l'application cliente. Cette classe n'offre pas les méthodes escomptées comme RMI nous en a donné l'habitude ! En effet chaque requête est implémentée comme une classe interne avec une opération de requête et une opération de réponse, ainsi que des méthodes pour remplir les arguments. On voit ici que l'on perd la transparence d'appel de méthodes.

Le code d'utilisation de cette souche est alors le suivant :

```
public static void main(String[] args) {
    DelayedStockQuoteStub stub;
    stub = new DelayedStockQuoteStub();
    getQuickQuote(stub); }
```

Et la requête proprement dite :

```
public static void getQuickQuote(DelayedStockQuoteStub stub){
    DelayedStockQuoteStub.GetQuickQuote req = new
        DelayedStockQuoteStub.GetQuickQuote(); // classe interne
    // on remplit les arguments
    req.setStockSymbol("ORCL"); // le sigle d'Oracle
    req.setLicenseKey("0"); // licence d'essai
    DelayedStockQuoteStub.GetQuickQuoteResponse res =stub.GetQuickQuote(req);
    System.out.println(res.getGetQuickQuoteResult());
```

Nous voyons dans ce code que la souche donne des guides précis pour saisir les arguments et appeler le service, mais on est loin de la transparence du mode RPC (vu aux chapitres précédents).

9.6 CRÉATION D'UN SERVICE WEB AVEC JAVA 6

Java 6 permet à présent de déployer facilement un service web en utilisant des annotations :

1. `%AXIS-HOME%` sous Windows ou `$AXIS_HOME` sous Unix/Linux désigne le répertoire d'installation de Axis.

Copyright © 2015 Dunod.
© Dunod – Toute reproduction non autorisée est un délit.

- `@WebService` : toutes les implémentations de service web doivent posséder cette annotation. Elle peut accepter des paramètres tels que `serviceName` ou `portName`.
- `@SOAPBinding` : définit le style document ou RPC.
- `@WebParam` : définit le nom d'un paramètre.
- `@WebResult` : définit l'argument de retour.
- `@OneWay` : optionnel pour les appels asynchrones.

La plate-forme met à disposition un mini serveur web avec la classe `Endpoint.publish()`.

Écriture du client

Le client s'écrit comme un POJO (*Plain Object Java Object* ou objet Java simple) avec des annotations.

```
package demo ;
import javax.jws.*;
@WebService(name="AFCWebServices")
@SOAPBinding(style = SOAPBinding.Style.RPC)
public class MyWebServices
{
    @WebMethod
    public String hello( String nom )
    {
    return "Bonjour " + nom + "!";
    }
    @WebMethod(operationName="indexCorporel")
    @WebResult(name = "votre-index")
    public double indexCorporel( @WebParam(name="taille") double taille,
    @WebParam(name="poids") double poids )
    {
    return poids / (hauteur * hauteur) / 100 * 100;
    }
}
```

Publication du service

On peut utiliser une classe java pour publier sur le serveur comme ceci :

```
public class PublierSurServeur
{
    public static void main( String[] args )
    {
       Endpoint endpoint = Endpoint.publish( "http://localhost:8080/services",
                                             new MyWebServices() );
       JOptionPane.showMessageDialog( null, "Eteindre le serveur" );
       endpoint.stop();
    }
}
```

Copyright © 2015 Dunod.

On peut alors vérifier le déploiement en inspectant le WSDL sous : *http://localhost:8080/services?wsdl*.

Écriture d'un client

Le client doit importer les classes générées à partir de l'interface. On utilise pour cela l'outil `wsimport`. Si le source est dans le répertoire `src`, on pourra utiliser la commande :

```
wsimport -d src -keep -p demo.gen http://localhost:8080/services?wsdl
```

Les classes générées seront dans le package `demo.gen`.

- `demo.gen.AFCWebServices.java`
- `demo.gen.MyWebServicesService.java`

On peut alors écrire le code d'appel du client :

```
AFCWebServices port = new MyWebServicesService().getAFCWebServicesPort();
system.out.printf( "%s Votre index est %.1f%n",
                        port.hello( "Annick" ),
                        port.indexCorporel( 160, 60 ) );
}
```

Ici l'utilisation d'une souche rend l'appel transparent, contrairement à l'exemple Axis avec ADB.

9.7 CRÉATION D'UN SERVICE WEB AVEC JAX-WS

9.7.1 JAX-WS

JAX-WS est le modèle de programmation standard pour les services webs en Java. Il fait partie de Java depuis la version 6.

Il est indépendant du serveur web de déploiement (Glassfish, Axis2 ou CXF supportent JAX-WS), et l'on peut ainsi porter un service d'une plate-forme à l'autre.

Il fait usage des annotations telles que :

- `@WebService`
- `@WebMethod`

Ainsi cela évite d'avoir un fichier central pour le déploiement de chaque service, et l'on peut maintenir une correspondance étroite entre le code et le déploiement.

JAX-WS utilise le concept de POJO (*Plain Old Java Object*), c'est-à-dire de classes en Java sans caractéristique ni hiérarchie particulière.

Il remplace JAX-RPC qui consistait en quelque sorte en un RMI transitant sur HTTP.

Copyright © 2015 Dunod.
© Dunod – Toute reproduction non autorisée est un délit.

9.7.2 Téléchargement du logiciel nécessaire

Nous allons tester cette fois un déploiement industriel, et non une simple mise à disposition comme précédemment. Nous avons besoin pour cela d'un vrai serveur web Java. La démarche suivante mentionne plusieurs outils de développement en open source, mais le code est suffisamment général pour que d'autres outils puissent être utilisés :

- **Eclipse** est un environnement de développement pour Java, et peut être remplacé par l'environnement NetBeans
- **Ant** est un outil de contruction de logiciel et de déploiement peut être remplacé par Maven
- **Tomcat** est un serveur web supportant les *servlets* Java, qui font le lien entre les requêtes http et le code Java, et peut être remplacé par exemple par Jetty

Ant doit être lancé dans un répertoire contenant un fichier build.xml (nom réservé). Le fichier `build.xml` de Ant comporte toujours une en-tête telle que celle-ci :

```
<?xml version="1.0" encoding="UTF-8"?>
<project name="session" basedir="." default="wsgen">
```

où `wsgen` dans ce cas est la tâche lancée par défaut et le répertoire de travail est la répertoire de lancement. Le fichier comporte des tâches appelées **target** (cible) :

```
<target name="compile.serveur"/>
```

et se termine par :

```
</project>
```

On lance une tâche Ant à l'aide de la commande :

```
ant nomDeLaTache
```

On commence par télécharger l'environnement de développement Eclipse sur *http://www.eclipse.org*, en prenant soin de sélectionner la version J2EE.

On télécharge alors l'implémentation de référence de JAX-WS depuis le site : *http://jax-ws.java.net*.

Attention : Java 6 intègre la version 2.1, alors que Java 7 intègre la version 2.2. On peut toutefois utiliser la dernière version en Java 6 en utilisant le mécanisme d'endossement. Il faut copier les fichiers `jaxws-api.jar` et `jaxb-api.jar` dans le répertoire `JAVA_HOME/lib/endorsed` pour éviter les conflits, où `JAVA_HOME` désigne le répertoire de Java (*Java Development Kit* ou Java pour le développement et non *Java Runtime Environment*, Java pour l'exécution).

Copyright © 2015 Dunod.

On télécharge ensuite le serveur web Tomcat depuis l'adresse : *http://tomcat.apache.org*.

Le site JAX-WS mentionne comment installer JAX-WS dans Tomcat, il s'agit principalement de copier les fichiers jar du répertoire `lib` de JAX-WS dans le répertoire `shared\lib` de Tomcat et les fichiers `jaxws-api.jar` et `jaxb-api.jar` dans le répertoire `common/endorsed`.

On peut utiliser pour cela l'utilitaire Ant, disponible également en open source à l'adresse : *http://ant.apache.org*.

Apache est un consortium de logiciel Open Source qui publie bien d'autres logiciels que le fameux serveur web HTTP.

Ant fonctionne principalement comme un outil de commande en ligne permettant d'effectuer toutes sortes de tâches d'installation, de copie de fichiers, de compilation...

Ces tâches sont décrites dans un fichier appelé `build.xml`, qui décrit des cibles (*target*) associées à chacune des tâches. Ant permet l'automatisation de tâches répétitives de construction de logiciel. Toutefois certains préfèrent utiliser Maven, un autre outil de la galaxie Apache, qui associe la mise à jour du code et les tests aux tâches de construction de logiciel.

Pour installer JAX-WS sous Tomcat, on peut définir la variable d'environnement `CATALINA_HOME` en la faisant pointer sur le répertoire de Tomcat, puis se placer dans le répertoire de JAX-WS où se trouve le fichier `build.xml`, et lancer la tâche d'installation :

```
ant install.
```

Si l'on dispose de Java 6, on doit alors également lancer :

```
ant install-api
```

Pour copier les jars dans le répertoire `JAVA-HOME/lib/endorsed`.

Nous verrons également que pour déployer un service web avec JAX-WS on peut utiliser des tâches Ant.

9.7.3 Lancement du serveur web Tomcat et arborescence de fichiers

On peut alors lancer le serveur Tomcat à l'aide de la commande se situant dans le répertoire `bin` :

```
startup.bat
```

sous Windows, ou sous Linux

```
startup
```

Copyright © 2015 Dunod.
© Dunod – Toute reproduction non autorisée est un délit.

en ayant pris soin de définir la variable d'environnement `JAVA_HOME` qui doit pointer sur le répertoire d'installation de Java.

Pour vérifier que Tomcat est lancé correctement, on peut ouvrir la page web d'accueil dans un navigateur à l'adresse : *http://localhost:8080*.

Naturellement, si Tomcat est installé sur une autre machine, il faut changer le nom `localhost`, de même s'il tourne sur un autre port que le `8080`.

Les applications web d'un serveur web Java sont toujours placées dans le répertoire `webapps`.

Ce répertoire contient les différentes applications web disponibles, et sa structure reflète l'arborescence des différentes URL.

Typiquement le sous-répertoire `examples` de `webapps` va correspondre à une URL : *http://localhost:8080/examples*.

Pour déployer une application web, on génère en général un fichier `war` qui contient de manière compressée, un peu à la manière d'un `jar`, l'arborescence de l'application web à déployer. Le nom de ce fichier `war` sera par défaut le nom de l'URL racine de l'application.

Si l'on déploie `services.war`, l'URL racine sera : *http://localhost:8080/services*.

L'arborescence de fichiers d'un `war` pour un service web va comporter les éléments suivants (attention à la casse pour un déploiement sous Linux) :

- `META-INF` : répertoire comportant les méta données de l'application web
- `WEB-INF` : répertoire contenant le code compilé soit sous forme de `.class` (répertoire `class`) soit sous forme de `.jar` (répertoire `lib`).

`WEB-INF` contient également le descripteur `web.xml` qui décrit les paramètres de l'application web, notamment le nom de la servlet et l'URL utilisée. Une servlet est un code java qui tourne directement dans le serveur web pour faire le lien entre des requêtes http et du code Java. C'est dans le fichier `web.xml` que l'on fait le lien entre le code et l'URL du service.

Voici un exemple pour un service web en JAX-WS : le service web va utiliser la servlet prédéfinie dans JAX-WS `com.sun.xml.ws.transport.http.servlet.WSServlet`, référencée par l'URL relative `/jaxws`.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE web-app
        PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
        "http://java.sun.com/j2ee/dtds/web-app_2_3.dtd">
<web-app>
    <listener>
        <listener-class>
            com.sun.xml.ws.transport.http.servlet.WSServletContextListener
        </listener-class>
    </listener>
    <servlet>
```

Copyright © 2015 Dunod.

```
        <servlet-name>jaxws</servlet-name>
        <servlet-class>com.sun.xml.ws.transport.http.servlet.WSServlet
        </servlet-class>
        <load-on-startup>1</load-on-startup>
    </servlet>
    <servlet-mapping>
        <servlet-name>jaxws</servlet-name>
        <url-pattern>/jaxws</url-pattern>
    </servlet-mapping>
    <session-config>
        <session-timeout>60</session-timeout>
    </session-config>
</web-app>
```

Pour déployer un service web en JAX-WS à l'URL relative `/meteo` avec une classe nommée `serveur.ServiceMeteo`, on a également besoin dans le répertoire `WEB-INF` d'un fichier `sun-jaxws.xml` tel que celui-ci :

```
<?xml version="1.0" encoding="UTF-8"?>
<endpoints xmlns='http://java.sun.com/xml/ns/jax-ws/ri/runtime'
        version='2.0'>
    <endpoint name='meteo'
            implementation='serveur.ServiceMeteo'
            url-pattern='/meteo'/>
</endpoints>
```

Si l'on met cette classe dans un fichier `war services.war` l'URL d'accès du service sera la suivante : *http://localhost:8080/services/meteo*[1].

Cette URL affiche une page d'information telle que la suivante (figure 9.7) :



Figure 9.7 — Page d'accueil des services JAX-WS

1. On remarquera que l'URL de la servlet JAX-WS n'intervient pas dans cette adresse.

© Dunod – Toute reproduction non autorisée est un délit.
Copyright © 2015 Dunod.

9.7.4 Code de l'application serveur

On souhaite réaliser un service météo permettant de donner la température en fonction de données d'entrée en Java.

Pour implémenter un service, on peut partir soit du descripteur d'interface WSDL, soit du code Java en générant automatiquement le descripteur d'interface, et c'est cette approche qui va être décrite ici.

On va s'appuyer sur les annotations en Java pour décrire le service web en écrivant le code, ainsi la mise à jour est forcément synchronisée entre le code et ses méta-données.

Voici donc le code de notre service météo :

```
@WebService(
        serviceName = "ServiceMeteo",
targetNamespace="http://www.afceurope.com")
@SOAPBinding(parameterStyle=SOAPBinding.ParameterStyle.BARE)
public class ServiceMeteo {
   @WebMethod
   @WebResult(name="result", targetNamespace="http://www.afceurope.com")
   public String traiteRequete(
         @WebParam(name="requete", targetNamespace="www.afceurope.com")
         Requete requete){
System.out.println("Je traite la requete");
      System.out.println("Pays "+requete.getPays());
      System.out.println("Ville "+requete.getVille());
      if( requete.isCelsius()) return "20.0" ;
      else return "65.0";
   }
}
```

Il s'agit d'une classe Java classique appelée `ServiceMeteo` que l'on peut mettre dans un package appelé `serveur`.

Au nom de la classe, on doit rajouter un nom de service dans l'annotation `@WebService`. Ce nom de service utilise un nom d'URL qui a pour but de créer un espace de nommage séparé. Il faut donc remplacer *http://www.afceurope.com* par votre URL personnalisée qui permet d'éviter toute collision de noms sur le web : votre espace de jeu est planétaire à présent !

On remarquera au passage l'annotation décrivant le style de passage des paramètres :

```
SOAPBinding.ParameterStyle.BARE
```

Copyright © 2015 Dunod.

L'annotation @SOAPBinding

Nous avons vu plus haut qu'il existait plusieurs modes de sérialisation pour les services web. L'annotation `@SOAPBinding` va permettre de préciser cette sérialisation[1].

Tableau 9.2 — L'annotation `SOAPBinding`

Propriété	Valeur	Description
Style	`Style.DOCUMENT` (par défaut) `Style.RPC`	Décrit le mode de passage des arguments par document ou par appel RPC.
Use	`Use.LITERAL` (défaut) `Use.ENCODED`	Décrit le pliage des arguments SOAP.
ParameterStyle	`ParameterStyle.BARE` `ParameterStyle.WRAPPED` (défaut)	Spécifie comment les paramètres de la méthode sont placés dans le corps du message SOAP. Un style de paramètre `BARE` (ou tel quel) signifie que chaque paramètre est placé dans le corps de message comme un enfant de l'élément message. Alors que `WRAPPED` (ou enveloppé) signifie que les paramètres sont enveloppés dans un élément unique dans la requête du message, de même pour les paramètres de retour. Si le style est RPC le style de paramètres doit être `WRAPPED`.

On voit ainsi que la sérialisation par défaut est `Document Literal Wrapped` qui est le modèle utilisé par les applications Microsoft.

Ce service comporte une seule méthode décrite par l'annotation :

```
@WebMethod
```

Tout comme ses paramètres sont décrits par une annotation :

```
@WebParam
```

Et son résultat par l'annotation :

```
@WebResult
```

Les noms donnés à chaque fois se retrouveront dans le descripteur d'interface web WSDL lors de sa génération à partir du code et des annotations.

1. Voir aussi : *http://www.ibm.com/developerworks/webservices/library/ws-whichwsdl/*

Copyright © 2015 Dunod.
© Dunod – Toute reproduction non autorisée est un délit.

9.7.5 Passage des paramètres

Dans un service web, les données sont nécessairement sérialisées sous forme XML. On remarque au passage qu'il n'est pas possible de passer des références sur des objets.

L'ordre des paramètres dans un service web peut être parfois déroutante, et notamment l'option WRAPPED/BARE, et afin de s'assurer que tout est correct, on préfère pour la simplicité et la généralité implémenter une classe contenant tous les paramètres d'appel du service.

Cette classe peut être générée à partir d'une description d'un schéma XML, afin de s'assurer que les types de données sont passés correctement et dans l'ordre.

On va s'appuyer sur les outils de l'interface JAXB (*Java Architecture for XML Binding*) de Java qui permet la correspondance de données entre un schéma XML et une classe Java.

Voici donc la description de la requête pour le service météo sous forme de schéma XML :

```
<xsd:schema targetNamespace="http://www.afceurope.com"
xmlns:tns="http://www.afceurope.com"
xmlns:xsd="http://www.w3.org/2001/XMLSchema">
   <xsd:element name="Requete" type="tns:Requete"/>
   <xsd:complexType name="Requete">
      <xsd:complexContent>
         <xsd:restriction base="xsd:anyType">
              <xsd:sequence>
                  <xsd:element name="pays" type="xsd:string"/>
                  <xsd:element name="ville" type="xsd:string"/>
                  <xsd:element name="codePostal" type="xsd:int"/>
                  <xsd:element name="Celsius" type="xsd:boolean"/>
              </xsd:sequence>
         </xsd:restriction>
      </xsd:complexContent>
   </xsd:complexType>
</xsd:schema>
```

La requête porte le nom `Requete` dans l'espace de nommage *http://www.afceurope.com*, à remplacer par votre propre espace.

Elle comporte dans un ordre imposé quatre paramètres de types différents, des chaînes de caractères, un entier et un booléen. Les types de données de XML schéma correspondent à des types Java que l'on devine facilement.

JAXB va générer une classe `Requete` à partir de ce schéma à l'aide de l'outil `xjc`.

Voici la tâche Ant permettant cette génération :

```
<property name="jaxws.lib.home" value="D:\Book\jaxws-ri\lib" />
<target name="xjc">
      <taskdef name="xjc" classname="com.sun.tools.xjc.XJCTask">
         <classpath>
            <fileset dir="${jaxws.lib.home}" includes="*.jar" />
```

Copyright © 2015 Dunod.

```
            </classpath>
        </taskdef>
        <xjc schema="resources/Requete.xsd" destdir="src" package="serveur.xjc"
/>
    </target>
```

Et voici la classe `Requete` obtenue :

```
@XmlAccessorType(XmlAccessType.FIELD)
@XmlType(name = "Requete", propOrder = {
    "pays",
    "ville",
    "codePostal",
    "celsius"
})
public class Requete {
    @XmlElement(required = true)
    protected String pays;
    @XmlElement(required = true)
    protected String ville;
    protected int codePostal;
    @XmlElement(name = "Celsius")
    protected boolean celsius;
public String getPays() {return pays; }
public void setPays(String value) {this.pays = value;}
public String getVille() {return ville;}
public void setVille(String value) {this.ville = value;}
public int getCodePostal() {return codePostal;}
public void setCodePostal(int value) {this.codePostal = value; }
public boolean isCelsius() {return celsius; }
public void setCelsius(boolean value) {this.celsius = value; }
}
```

On remarquera la génération de tous accesseurs aux champs de la classe, mais également des annotations spécifiques à JAXB indiquant :

- le nom du type XML associé à la classe :

```
@XmlType(name = "Requete"
```

- si le paramètre est obligatoire ou non :

```
@XmlElement(required = true)
```

- l'ordre de passage des arguments :

```
propOrder = {"pays","ville","codePostal","celsius"}
```

À défaut de cette indication, on peut recevoir les arguments dans le désordre. On constate également la création d'une classe `ObjectFactory` qui va faire le lien entre la classe `Requete` et sa sérialisation JAXB.

Copyright © 2015 Dunod.
© Dunod – Toute reproduction non autorisée est un délit.

```
@XmlRegistry
public class ObjectFactory {
private final static QName _Requete_QNAME = new
QName("http://www.afceurope.com", "Requete");
public Requete createRequete() {
        return new Requete();
    }
@XmlElementDecl(namespace = "http://www.afceurope.com", name = "Requete")
    public JAXBElement<Requete> createRequete(Requete value) {
        return new JAXBElement<Requete>(_Requete_QNAME, Requete.class, null,
value); }
}
```

9.7.6 Mise en œuvre

Arborescence de fichiers

On choisit pour la mise en œuvre d'utiliser au maximum les tâches Ant, afin de garantir la portabilité du code. Il existe des outils permettant de simplifier le travail, mais qui peuvent masquer des incompatibilités de versions. À titre indicatif, on montrera l'arborescence sous l'environnement de développement Eclipse. On peut créer sous Eclipse un projet de service web, mais l'outil fait toujours des suppositions sur la *servlet* et la version d'implémentation, et afin de présenter une application dont on maîtrise les paramètres, il est préférable de créer un projet agnostique à l'aide de la commande par menus successifs : `File/ New/ Other/ Dynamic Web Project`.

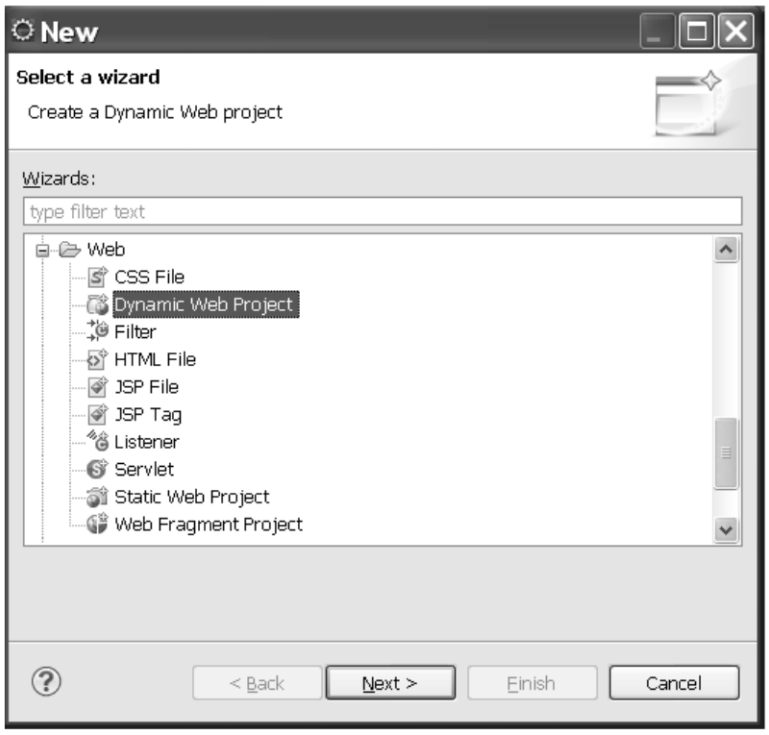


Figure 9.8 — Menu de création d'un service web sous Eclipse

Copyright © 2015 Dunod.

On voit alors l'arborescence de répertoire à peu près similaire à l'arborescence du `war` décrit plus haut, mais avec en plus un répertoire `src` pour contenir les sources en Java du projet.

On crée également un répertoire `resources` pour définir les schémas XML des objets sérialisés, un répertoire `jar` pour contenir le code et un répertoire `build` pour le code compilé.

On place le code de la classe `serveur.ServiceMeteo` dans le package `serveur` du répertoire `src`.



Figure 9.9 — Arborescence des fichiers pour le fichier `war`

Utilisation de Ant pour les tâches de construction du logiciel

On commence par définir les variables d'environnement ou les paramètres des tâches que vous devez paramétrer selon votre installation :

```
<property name="java.home" value="C:\Program Files\Java\jdk1.6.0_11" />
<property name="jaxws.lib.home" value="D:\Book\jaxws-ri\lib" />
<property name="tomcat.home" value="D:\Book\apache-tomcat-7.0.26" />
<property name="build.dir" value="build" />
<property name="jar.dir" value="jar" />
<property name="jar.name" value="meteo" />
<property name="war.name" value="services.war" />
<property name="resources.home" value="${basedir}/resources" />
<property name="classe.service" value="serveur.ServiceMeteo" />
```

On définit ensuite le CLASSPATH pour compiler l'application avec les bibliothèques JAX-WS.

```
<path id="jaxws.classpath">
      <fileset dir="${jaxws.lib.home}">
        <include name="*.jar" />
      </fileset>
</path>
```

Copyright © 2015 Dunod.
© Dunod – Toute reproduction non autorisée est un délit.

Puis voici la tâche de compilation du serveur pour créer un jar utilisé par la tâche de génération du service `wsgen` :

```
<target name="compile.serveur">
   <delete dir="${jar.dir}" />
   <mkdir dir="${jar.dir}" />
   mkdir dir="${jar.dir}/classes" />
   <javac debug="on" fork="true" destdir="${jar.dir}/classes"
srcdir="${basedir}/src"/>
   <jar destfile="${jar.dir}/${jar.name}.jar">
         <fileset excludes="**/Test.class" dir="${jar.dir}/classes" />
   </jar>
</target>
```

Et voici la tâche de génération du service `wsgen`, qui utilise les outils de la bibliothèque `jaxws-tools.jar`. Cette bibliothèque doit figurer dans le `classpath` de Ant.

```
<!– il faut un jar pour wsgen–>
<path id="wsgen.path">
            <pathelement location="${jar.dir}/${jar.name}.jar" />
</path>
<target name="wsgen">
      <taskdef name="wsgen" classname="com.sun.tools.ws.ant.WsGen">
                  <classpath path="jaxws.classpath" />
      </taskdef>
            <mkdir dir="${basedir}/resources" />
            <mkdir dir="${build.dir}/classes" />
            <wsgen destdir="${build.dir}/classes"
sourcedestdir="${basedir}/src" genwsdl="true" verbose="true"
resourcedestdir="${basedir}/resources" keep="true" sei="${classe.service}">
                  <classpath refid="wsgen.path" />
            </wsgen>
</target>
```

On observe lors de l'exécution de cette tâche la création d'un fichier `ServiceMeteo.wsdl` dans le répertoire `resources`, ainsi que des schémas XML correspondant au passage des arguments et aux paramètres de retour, ici le paramètre `result` de type `String`. On constate avec satisfaction que le paramètre `Requete` est semblable à sa spécification en JAXB.

Copyright © 2015 Dunod.

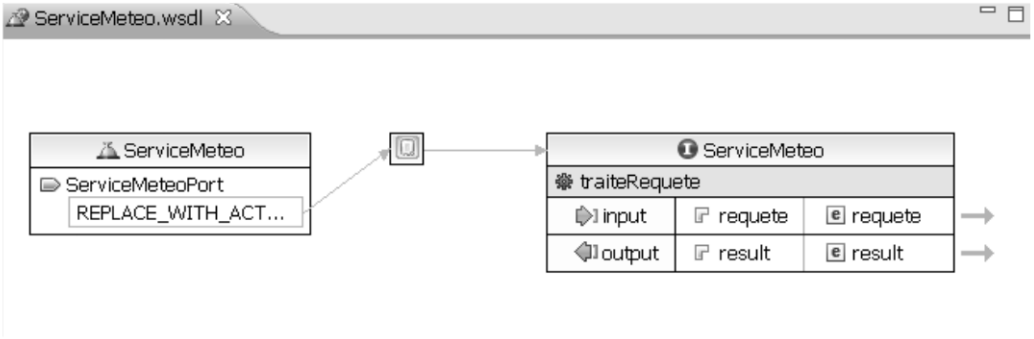


Figure 9.10 — Aperçu graphique du fichier WSDL sous Eclipse

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<xs:schema version="1.0" targetNamespace="http://www.afceurope.com"
xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="result" nillable="true" type="xs:string"/>
  <xs:complexType name="Requete">
    <xs:sequence>
      <xs:element name="pays" type="xs:string"/>
      <xs:element name="ville" type="xs:string"/>
      <xs:element name="codePostal" type="xs:int"/>
      <xs:element name="Celsius" type="xs:boolean"/>
    </xs:sequence>
  </xs:complexType>
</xs:schema>
```

Vient ensuite la compilation du code :

```
<!– on doit rajouter les classes generees –>
   <target name="compile.wsgen">
      <delete dir="${build.dir}" />
      <mkdir dir="${build.dir}" />
      <mkdir dir="${build.dir}/classes" />
      <javac debug="on" fork="true" destdir="${build.dir}/classes"
srcdir="${basedir}/src" />
      <copy todir="${build.dir}/classes">
         <fileset dir="${jar.dir}/classes">
            <include name="*.class" />
         </fileset>
      </copy>
   </target>
```

Puis la création du fichier `.war`[1] :

```
<target name="war">
      <war destFile="${war.name}" webxml="./WEB-INF/web.xml"
basedir="${basedir}">
         <classes dir="${build.dir}/classes" />
```

1. Attention qu'il ne reste pas de fichier war existant dans le répertoire

© Dunod – Toute reproduction non autorisée est un délit.
Copyright © 2015 Dunod.

```
            <lib dir="${jaxws.lib.home}" />
            <fileset dir="${basedir}/WEB-INF" includes="*.xml" excludes="web.xml"
/>
        </war>
    </target>
```

Et finalement le déploiement par copie du fichier `war` dans le répertoire `webapps` de Tomcat.

```
    <target name="deploy">
        <copy file="${war.name}" todir="${tomcat.home}/webapps" />
        <delete file="${war.name}" />
    </target>
```

Sous Eclipse, on peut lancer des tâches Ant en cliquant à droite sur le fichier `build.xml` puis en lançant la commande « Ant Run as configuration... »

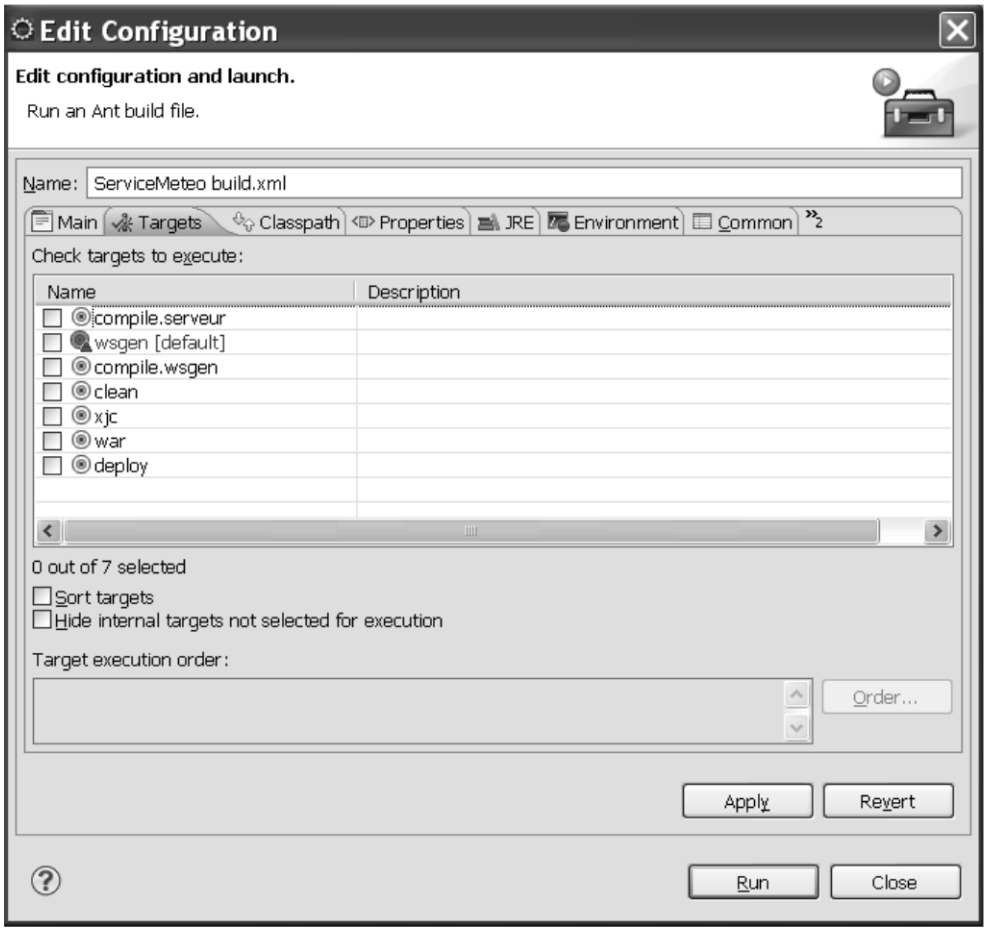


Figure 9.11 — Tâches Ant sous Eclipse

Attention toutefois à vérifier dans l'onglet JRE de Ant précédent que l'on pointe bien vers le JDK et non le JRE, sinon il serait impossible de compiler le code. Il faut aussi rajouter dans l'onglet Classpath les librairies `jaxws-tools.jar`.

Copyright © 2015 Dunod.

9.7.7 Test de l'application et code du client

On peut alors vérifier que l'application a été correctement déployée en ouvrant dans un navigateur l'URL : *http://localhost:8080/services/meteo*.

Si le serveur a utilisé l'outil `wsgen`, le client lui doit utiliser l'outil `wsimport` de JAX-WS.

On utilise également une tâche Ant pour l'outil `wsimport` de JAX-WS :

```
<?xml version="1.0" encoding="UTF-8"?>
<project name="session" basedir="." default="wsimport">
<property name="jaxws.lib.home" value="D:\Book\jaxws-ri\lib" />
<property name="build.dir" value="build" />
<property name="client.classes.dir" value="${client.dir}/classes" />
<property name="wsdl.location"
value="http://localhost:8080/services/meteo?wsdl" />
   <path id="jaxws.classpath">
      <fileset dir="${jaxws.lib.home}">
         <include name="*.jar" />
      </fileset>
      <pathelement location="${build.dir}/**/*.class" />
   </path>
   <target name="wsimport">
      <mkdir dir=".\build" />
      <mkdir dir=".\build\classes" />
      <taskdef name="wsimport" classname="com.sun.tools.ws.ant.WsImport"
classpathref="jaxws.classpath" />
      <wsimport wsdl="${wsdl.location}" destdir=".\build\classes"
package="client.jaxws" verbose="true" keep="true"
sourceDestDir="${basedir}/src" xendorsed="true">
      </wsimport>
   </target>
</project>
```

On lance l'unique tâche `wsimport`, en prenant soin de renseigner correctement l'URL ainsi que les variables d'environnement.

Ici l'URL est la suivante : *http://localhost:8080/services/meteo?wsdl*.

On observe alors de nouvelles classes générées dans le répertoire `client.jaxws`. Parmi elles on observera les classes :

- `Requete` : la même classe utilisée par le serveur pour recevoir les arguments.
- `ServiceMeteo` : la classe Java qui implémente le service.
- `ServiceMeteo_Service` : la classe Java qui comporte toutes les informations d'accès au service, notamment le lien sur l'URL.

La `classe ServiceMeteo_Service` va servir de Factory pour obtenir une instance du service avec tous ses paramètres d'initialisation.

Ensuite on peut utiliser le service comme un objet distant.

Copyright © 2015 Dunod.
© Dunod – Toute reproduction non autorisée est un délit.

On écrit tout simplement le code suivant :

```
public class ClientMeteo {
   public static void main(String[] args) {
       ServiceMeteo_Service endpoint = new ServiceMeteo_Service();
       ServiceMeteo meteo = endpoint.getServiceMeteoPort();
       Requete requete= new Requete();
       requete.setCelsius(true);
       requete.setVille("Nice");
       requete.setPays("France");
       requete.setCodePostal(06270);
       System.out.println(meteo.traiteRequete(requete));
   }
}
```

9.7.8 Variations d'implémentation

Utilisation de l'attribut WRAPPED pour le passage des paramètres

Si dans le code du serveur on utilise l'option de sérialisation :

```
SOAPBinding.ParameterStyle.WRAPPED
```

qui est l'option par défaut, on constatera lors du lancement de `wsgen` la création de deux classes `TraiteRequete` et `TraiteRequeteResponse` qui vont envelopper les paramètres d'appel et de retour.

Lors du lancement de `wsimport` dans le client, on observera également la génération de ces deux classes, par contre le code principal du client et du serveur ne change pas.

Voici le commentaire généré dans `TraiteRequete` pour son association à un schéma XML :

```
<complexType name="traiteRequete">
 <complexContent>
<restriction base="{http://www.w3.org/2001/XMLSchema}anyType">
<sequence>
<element ref="{www.afceurope.com}requete" minOccurs="0"/>
<sequence>
<restriction>
<complexContent>
</complexType>
```

`TraiteRequete` est donc une structure de données comportant un seul élément `Requete`.

Copyright © 2015 Dunod.

Plusieurs services déployés en même temps

On peut naturellement utiliser le même `war` pour développer deux services différents.

Il suffit pour cela de définir deux `endPoints` dans le fichier `sun-jaxws.xml` :

```
<?xml version="1.0" encoding="UTF-8"?>
<endpoints xmlns='http://java.sun.com/xml/ns/jax-ws/ri/runtime'
        version='2.0'>
    <endpoint name='meteo'
            implementation='serveur.ServiceMeteo'
            url-pattern='/meteo'/>
     <endpoint name='meteoLuxe'
            implementation='serveur.ServiceMeteoLuxe'
            url-pattern='/meteoLuxe'/>
</endpoints>
```

Attention à bien changer les noms des services dans les annotations du code. Dans la cible `wsgen`, on rajoute une deuxième tâche identique à la première. On obtiendra alors deux WSDL différents, avec chacun son URL :
http://localhost:8080/services/meteo
http://localhost:8080/services/meteoLuxe

Rajout d'exceptions et de paramètres de sortie

On suppose que l'on veut déployer un autre service `ServiceMeteoLuxe` qui renvoie un objet de type `Bulletin`, et qui lève une erreur.

On définira le schéma XML de `Bulletin` comme paramètre de retour tout comme on a défini `Requete` en paramètre d'entrée :

```
<xsd:schema targetNamespace="http://www.afceurope.com"
xmlns:tns="http://www.afceurope.com"
xmlns:xsd="http://www.w3.org/2001/XMLSchema" >
   <xsd:element name="Bulletin" type="tns:Bulletin"/>
   <xsd:complexType name="Bulletin">
      <xsd:complexContent>
         <xsd:restriction base="xsd:anyType">
            <xsd:sequence>
               <xsd:element name="pays" type="xsd:string"/>
               <xsd:element name="ville" type="xsd:string"/>
               <xsd:element name="Temperature" type="xsd:double"/>
               <xsd:element name="Situation" type="xsd:string"/>
            </xsd:sequence>
         </xsd:restriction>
      </xsd:complexContent>
   </xsd:complexType>
</xsd:schema>
```

Sans oublier de compléter la tâche `xjc` pour générer la classe associée à ce schéma avec JAXB.

On peut également définir une exception, `TempeteException`, comme une exception classique en rajoutant un champ information et ses accesseurs :

Copyright © 2015 Dunod.
© Dunod – Toute reproduction non autorisée est un délit.

```
public class TempeteException extends Exception {
private String information;
   public TempeteException(String aMessage, String anInformation){
      super(aMessage);
      information = anInformation; }
   public void setInformation(String information) {this.information =
information; }
   public String getInformation() {return information;} }
```

Voici à présent la définition complète de notre nouveau service :

```
@WebService(
        serviceName = "ServiceMeteoLuxe",
targetNamespace="http://www.afceurope.com")
@SOAPBinding(parameterStyle=SOAPBinding.ParameterStyle.BARE)
public class ServiceMeteoLuxe {
   @WebMethod
   @WebResult(name="bulletin", targetNamespace="http://www.afceurope.com")
   public Bulletin traiteRequete (
         @WebParam(name="requete", targetNamespace="www.afceurope.com")
         Requete requete) throws TempeteException{
      Bulletin bulletin = new Bulletin();
      bulletin.setVille(requete.getVille());
      if( requete.isCelsius()) bulletin.setTemperature(20.5) ;
      else bulletin.setTemperature(68.9);
      bulletin.setSituation("Ensoleille");
      if(requete.getPays().equalsIgnoreCase("Islande"))throw new
TempeteException("Mauvais temps", "Islande");
      return bulletin;
   }
}
```

Et le client associé :

```
public class ClientLuxe {
public static void main(String[] args) {
ServiceMeteoLuxe_Service endpoint = new ServiceMeteoLuxe_Service();
       ServiceMeteoLuxe meteo = endpoint.getServiceMeteoLuxePort();
       Requete requete = new Requete();
       requete.setPays("France");
       requete.setVille("Nice");
       requete.setCelsius(true);
       requete.setCodePostal(06000);
       try {
         Bulletin bulletinNice = meteo.traiteRequete(requete);
         System.out.println("Temperature : "+bulletinNice.getTemperature());
         System.out.println("Situation : "+bulletinNice.getSituation());
         //assurons-nous de l'ordre de passage des arguments
         System.out.println("Ville : "+bulletinNice.getVille());
      } catch (TempeteException_Exception e) {
                e.printStackTrace();
      }
        try {
         requete.setPays("Islande");
```

Copyright © 2015 Dunod.

```
            Bulletin bulletinIslande = meteo.traiteRequete(requete);
    } catch (TempeteException_Exception e) {
                        System.out.println(e.getMessage());
                        System.out.println(e.getFaultInfo().getInformation());}
       }}
```

Dans la première requête, on va récupérer un objet `Bulletin` avec lequel on va travailler comme un objet Java ordinaire.

Dans la deuxième requête sur l'Islande, on obtiendra une exception qui pourra être gérée comme une exception classique en Java.

On remarquera que lors du `wsimport`, on obtient deux classes pour l'exception : `TempeteException_Exception` et `TempeteException`.

Et la classe `TempeteException_Exception` pourra accéder à l'objet sérialisé `TempeteException`, qui à son tour pourra livrer son information : Islande.

On remarquera dans la classe générée `TempeteException` le commentaire donnant le schéma de sérialisation :

```
<complexType name="TempeteException">
<complexContent>
<restriction base="{http://www.w3.org/2001/XMLSchema}anyType">
 <sequence>
 <element name="information" type="{http://www.w3.org/2001/XMLSchema}string"
minOccurs="0"/>
<element name="message" type="{http://www.w3.org/2001/XMLSchema}string"
minOccurs="0"/>
 <sequence>
<restriction>
<complexContent>
</complexType>
```

9.8 CRÉATION D'UN SERVICE WEB AVEC AXIS

Démarche de mise en œuvre

Afin d'illustrer la simplicité de mise à disposition d'un objet Java comme service web, nous allons considérer un exemple simple de météo. Il s'agit d'un POJO (*Plain Old Java Object*), c'est-à-dire un objet qui n'a besoin d'aucun héritage ni d'aucune interface particulière, si ce n'est simplement les opérations et attributs nécessaires à son fonctionnement.

La mise en œuvre comporte plusieurs étapes :

1. L'écriture de l'objet.
2. L'écriture d'un service de mise à disposition de l'objet (un peu semblable au rôle du POA en CORBA).
3. Un descriptif du service.
4. Un déploiement de l'application sous le serveur web Tomcat.

Copyright © 2015 Dunod.
© Dunod – Toute reproduction non autorisée est un délit.

On suppose que l'on a téléchargé Tomcat comme indiqué précédemment, et téléchargé le fichier `axis2.war` sous `$TOMCAT_HOME/webapps`.

On vérifie l'installation en lançant Tomcat avec la commande `startup.bat` ou `startup.sh` et en tapant dans un navigateur web la commande suivante :

```
http://localhost:8080/axis2
```

On crée par ailleurs la structure de répertoire suivante. Les fichiers seront détaillés ci-après :

```
Axis_pojo (répertoire de travail)
    build.xml (fichier ant de construction de l'application)
    src (répertoire source)
        pojo/data/Meteo.java (fichier Meteo.java dans le package pojo.data)
        pojo/service/ServiceMeteo.java (fichier ServiceMeteo.java
            dans le package pojo.service)
        META-INF/services.xml (fichier de déploiement
            dans le répertoire META_INF))
build (répertoire de compilation)
```

Implémentation du service

Voici donc la classe `Meteo.java` pour implémenter un service météo. Cet objet n'a aucune contrainte particulière, ni en termes d'interface, ni en termes d'héritage. Il implémente essentiellement des accesseurs sur des variables d'instance.

```
package pojo.data;
public class Meteo{
    float temperature;
    String bulletin;
    boolean pluie;
    float hygrometrie;
    public void setTemperature(float temp){temperature = temp;}
    public float getTemperature(){return temperature;}
    public void setBulletin(String texte){ bulletin = texte;}
    public String getBulletin(){return bulletin;}
    public void setPluie(boolean r){pluie = r;}
    public boolean getPluie(){return pluie;}
    public void setHygrometrie(float combien){hygrometrie = combien;}
    public float getHygrometrie(){return hygrometrie;}
}
```

Classe de service : servant

Nous avons à présent besoin d'une classe de service qui met cet objet à disposition :

```
package pojo.service;
import pojo.data.Meteo;
public class ServiceMeteo{
    Meteo meteo;
```

Copyright © 2015 Dunod.

```
    public void setMeteo(Meteo meteo){
        this.meteo = meteo;
    }
public Meteo getMeteo(){
    return this.meteo;
    }
}
```

Fichier de déploiement

On doit ensuite créer un fichier descriptif du service `services.xml` dans le répertoire `META-INF`. Ce fichier décrit l'application `ServiceMeteo`.

```
<service name="ServiceMeteo" scope="application">
    <description>
        Service Meteo POJO
    </description>
    <messageReceivers>
        <messageReceiver
            mep="http://www.w3.org/2004/08/wsdl/in-only"
            class="org.apache.axis2.rpc.receivers.RPCInOnlyMessageReceiver"/>
        <messageReceiver
            mep="http://www.w3.org/2004/08/wsdl/in-out"
            class="org.apache.axis2.rpc.receivers.RPCMessageReceiver"/>
        </messageReceivers>
        <parameter name="ServiceClass">
            pojo.service.ServiceMeteo
    </parameter>
</service>
```

Déploiement

Pour le déploiement, on a besoin d'un utilitaire Java en Open Source qui s'appelle Ant[1] et doit être téléchargé.

On déploie le service à l'aide de Ant en écrivant le fichier `build.xml` suivant[2] :

```
<project basedir="." default="generate.service">
<!—les différentes variables du projet –>
    <property name="service.name" value="ServiceMeteo" />
    <property name="dest.dir" value="build" />
    <property name="dest.dir.classes" value="${dest.dir}/${service.name}"
/>
    <property name="dest.dir.lib" value="${dest.dir}/lib" />
    <property name="axis2.home" value="c:\apps\axis2-1.2" />
    <property name="repository.path" value="${axis2.home}/repository" />
< !—la génération du service –>
<target name="generate.service" depends="clean,prepare">
    <copy file="src/META-INF/services.xml"
    tofile="${dest.dir.classes}/META-INF/services.xml"
```

1. *http://ant.apache.org/*
2. Modifier le paramètre `axis2.home` en fonction de votre configuration.

Copyright © 2015 Dunod.
© Dunod – Toute reproduction non autorisée est un délit.

```
    overwrite="true" />
    <javac srcdir="src" destdir="${dest.dir.classes}"
        includes="pojo/service/**,pojo/data/**">
        <classpath refid="build.class.path" />
    </javac>
    <jar basedir="${dest.dir.classes}"
        destfile="${dest.dir}/${service.name}.aar" />
    <copy file="${dest.dir}/${service.name}.aar"
    tofile="${repository.path}/services/${service.name}.aar"
        overwrite="true" />
</target></project>
```

Pour générer le fichier de déploiement `ServiceMeteo.aar`, il suffit de lancer la commande `ant` dans le répertoire où se trouve le fichier `build.xml`. Les fichiers `.aar` sont des archives de déploiement de services web, tout comme les fichiers `.jar`.

```
ant
```

On copie ensuite le fichier `ServiceMeteo.aar` sous `$TOMCAT_HOME/webapps/axis2`, où `$TOMCAT-HOME` désigne le répertoire d'installation de Tomcat.

On peut alors vérifier dans un navigateur la présence du service en partant de l'URL : *http://localhost:8080/axis2/services*.

Le WSDL généré est accessible en ligne à l'URL : *http://localhost:8080/axis2/services/ServiceMeteo?wsdl*.

On observe dans ce WSDL la création d'un schéma XML correspondant à l'objet Météo :

```
<xs:element name="Meteo" type="ax21:Meteo"/>
    <xs:complexType name="Meteo">
        <xs:sequence>
            <xs:element name="bulletin" nillable="true" type="xs:string"/>
            <xs:element name="hygrometrie" type="xs:float"/>
            <xs:element name="pluie" type="xs:boolean"/>
            <xs:element name="temperature" type="xs:float"/>
        </xs:sequence>
    </xs:complexType>
</xs:element>
```

On observe également la création de trois messages :

```
<wsdl:message name="setMeteoMessage">
    <wsdl:part name="part1" element="ns0:setMeteo"/>
</wsdl:message>
<wsdl:message name="getMeteoMessage"/>
<wsdl:message name="getMeteoResponse">
    <wsdl:part name="part1" element="ns0:getMeteoResponse"/>
</wsdl:message>
```

`setMeteoMessage` et `getMeteoResponse` ont des arguments, `getMeteoMessage` n'a pas d'arguments.

Copyright © 2015 Dunod.

On observe la création de deux opérations :

- `setMeteo` qui utilise le message `setMeteoMessage`.
- `getMeteo` qui utilise les messages `getMeteoMessage` et `getMeteoResponse`, un pour la requête et un pour la réponse.

```
<wsdl:operation name="setMeteo">
    <wsdl:input message="axis2:setMeteoMessage" wsaw:Action="urn:setMeteo"/>
</wsdl:operation>
<wsdl:operation name="getMeteo">
    <wsdl:input message="axis2:getMeteoMessage" wsaw:Action="urn:getMeteo"/>
    <wsdl:output message="axis2:getMeteoResponse"/>
</wsdl:operation>
```

On observe finalement que le mode par défaut est **document/literal** comme le montre le morceau de WSDL suivant :

```
<wsdl:operation name="setMeteo">
    <soap12:operation soapAction="urn:setMeteo" style="document"/>
<wsdl:input>
    <soap12:body use="literal"/>
</wsdl:input>
</wsdl:operation>
```

Écriture du client RPC

On peut alors écrire le client RPC. Le point de départ est l'URL du service (*endpoint*), dans notre cas : *http://localhost:8080/axis2/services/ServiceMeteo*.

`localhost` est une convention réseau et désigne naturellement l'ordinateur sur lequel on travaille. Mais pour un appel à distance il suffit de le remplacer par le nom ou l'adresse IP de la machine distante.

On initialise d'abord un client RPC avec ce *endpoint*[1] :

```
RPCServiceClient serviceClient = new RPCServiceClient();
Options options = serviceClient.getOptions();
EndpointReference targetEPR = new EndpointReference(
    "http://localhost:8080/axis2/services/ServiceMeteo");
options.setTo(targetEPR);
```

On peut ensuite utiliser ce client pour exécuter la méthode `setMeteo` et initialiser les valeurs du service. On remarquera que l'opération est définie par un `Qname` (*Qualified Name*) et qu'elle utilise à l'envers le nom du package de la classe `ServiceMeteo`, `pojo.service`. Le `Qname` permet d'utiliser des espaces de nommage (*namespaces*) afin d'éviter les collisions de noms entre plusieurs applications. On construit les arguments comme une requête dynamique, ce qui nous permet de comprendre pourquoi cette fois on n'a pas besoin de souche spécifique.

1. Le port 8080 est le port par défaut de Tomcat.

Copyright © 2015 Dunod.
© Dunod – Toute reproduction non autorisée est un délit.

```
private static void setMeteo(RPCServiceClient serviceClient)
        throws AxisFault {
    javax.xml.namespace.QName opSetWeather =
        new QName("http://service.pojo", "setMeteo");
    Meteo w = new Meteo();
    w.setTemperature((float)25.3);
    w.setBulletin("Beau temps");
    w.setPluie(true);
    w.setHygrometrie((float)70);
    Object[] opSetWeatherArgs = new Object[] { w };
    serviceClient.invokeRobust(opSetWeather, opSetWeatherArgs);
}
```

De même, on peut invoquer la méthode `getMeteo` pour obtenir le bulletin :

```
private static Meteo getMeteo(RPCServiceClient serviceClient)
        throws AxisFault
{
    QName opGetWeather =
        new QName("http://service.pojo", "getMeteo");
    Object[] opGetWeatherArgs = new Object[] { };
    Class[] returnTypes = new Class[] { Meteo.class };
    Object[] response = serviceClient.invokeBlocking(opGetWeather,
        opGetWeatherArgs, returnTypes);
    Meteo result = (Meteo) response[0];
    if (result == null) {
        System.out.println("Le service meteo n'est pas initialisé!");
        return null;
}
return result;}
```

Et finalement on peut afficher les résultats :

```
Meteo result=getMeteo(serviceClient);
System.out.println("Temperature : " +
        result.getTemperature());
System.out.println("Bulletin : " +
        result.getBulletin());
System.out.println("Pluie : " +
        result.getPluie());
System.out.println("Hygrometrie : " +
        result.getHygrometrie());
```

Nous voyons donc qu'une des grandes limitations de l'utilisation de POJO est le fait de ne pas pouvoir initialiser l'objet du service web.

9.9 COMPARAISON DE DÉPLOIEMENT

Selon que la génération du stub dépend de l'interface ou non, cela induit des architectures très différentes. Dans l'approche descendante, on définit d'abord l'interface,

Copyright © 2015 Dunod.

puis les outils du bus logiciels vont générer le code pour l'utilisation du bus. C'est le cas historique de CORBA (figure 9.12).

Dans l'approche descendante l'interface précède l'implémentation et permet de générer le code souche pour le client. Dans l'approche montante, l'interface est extraite de l'implémentation, et n'a pas besoin d'être définie à l'avance (figure 9.13).

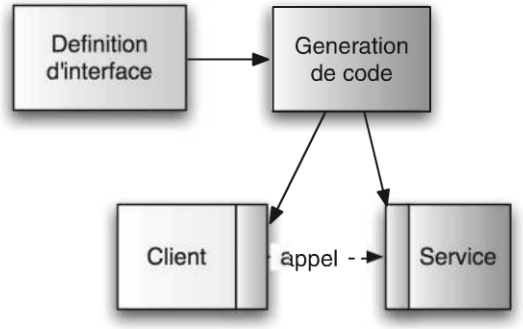


Figure 9.12 — Génération de code à partir de l'interface.

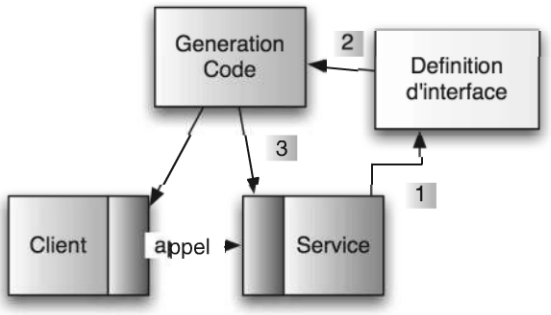


Figure 9.13 — Génération de l'interface à partir du code

En CORBA, la souche est générée à partir de l'IDL, ce qui permet lors du déploiement de n'exporter que l'IDL sur la partie cliente, qui peut générer son propre code d'accès au bus.

En RMI, la souche est générée à partir de la classe compilée. Il n'est pas possible au client de générer le code de son côté à moins de disposer de l'implémentation des objets serveurs. Ceci est important pour l'utilisation des EJB (*Entreprise JavaBeans*), puisque l'implémentation de l'objet est générée de manière cachée par le serveur. Le conteneur d'EJB doit fournir des outils pour exporter la souche sans révéler l'implémentation.

En RMI, la souche peut être également téléchargée mais cela induit des problèmes de sécurité.

En SOAP, l'interface peut être générée à partir de l'implémentation, et génère elle-même la souche. L'interface peut-être téléchargée depuis un site web sur Internet, et n'importe quel client peut générer sa propre souche pour se connecter au serveur.

Copyright © 2015 Dunod.
© Dunod – Toute reproduction non autorisée est un délit.

9.10 REST – REPRESENTATIONAL STATE TRANSFER

REST est un nouveau style d'architecture de services web qui s'appuie sur les différentes méthodes du protocole HTTP. Il a été décrit pour la première fois par Roy Fielding en 2000.

Dans une architecture REST, tout est ressource. Une ressource peut être appelée par une méthode `http` pour effectuer différentes actions de création, modification ou destruction. Les ressources ont des identifiants globaux, typiquement des URI web avec paramètres, par exemple :

```
http://localhost:8080/MaBoutique/visiteurs/1
```

Cette notation fait penser aux URI de CORBA (voir chapitre 6), sauf que dans le cas de CORBA il s'agit d'une référence sur un objet unique, alors que REST ne renvoie qu'une copie d'une représentation d'un objet. Le programme doit assurer la gestion des doublons et des appels répétés (idempotence).

REST permet aux ressources d'offrir plusieurs représentations, par exemple sous forme de texte, de XML, de JSON...Le client REST accède aux ressources par les méthodes HTTP. La servlet qui gère les requêtes maintient un lien entre les formats demandés et le code de la classe.

9.10.1 Les méthodes HTTP et les services REST

Les méthodes sont les suivantes : `PUT`, `GET`, `POST` et `DELETE`. Elles et font partie du standard HTTP.

Dans une architecture REST, elles prennent la signification suivante :

- `GET` définit un accès en lecture sur une ressource sans effet de bord. La ressource est inchangée.
- `PUT` crée une nouvelle ressource.
- `DELETE` supprime une ressource.
- `POST` met à jour une ressource existante.

Les services web sur architecture REST vont donc travailler uniquement avec ces quatre méthodes. Ils sont adaptés à des opérations de traitement de données (CRUD : *Create*, *Read*, *Update*, *Delete*) : création, lecture, mise à jour, destruction.

Copyright © 2015 Dunod.

9.10.2 Mise en œuvre sous Eclipse

Java définit un support REST avec la spécification JSR 311. Elle est appelée JAX-RS (Java API for API RESTful Web Services). JAX-RS utilise des annotations dans les codes Java.

Il a plusieurs implémentations possibles de JAX-RS. Les plus courantes sont Jersey (*https://jersey.java.net/*) qui est à présent intégré dans Glassfish d'Oracle et RestEeasy (*http://resteasy.jboss.org/*) de JBoss. On choisit ici RestEasy.

On commence par télécharger les bibliothèques RestEasy, puis les bibliothèques du serveur web Tomcat (*http://tomcat.apache.org/*) et on crée ensuite un projet dynamique, `MaBoutique`.

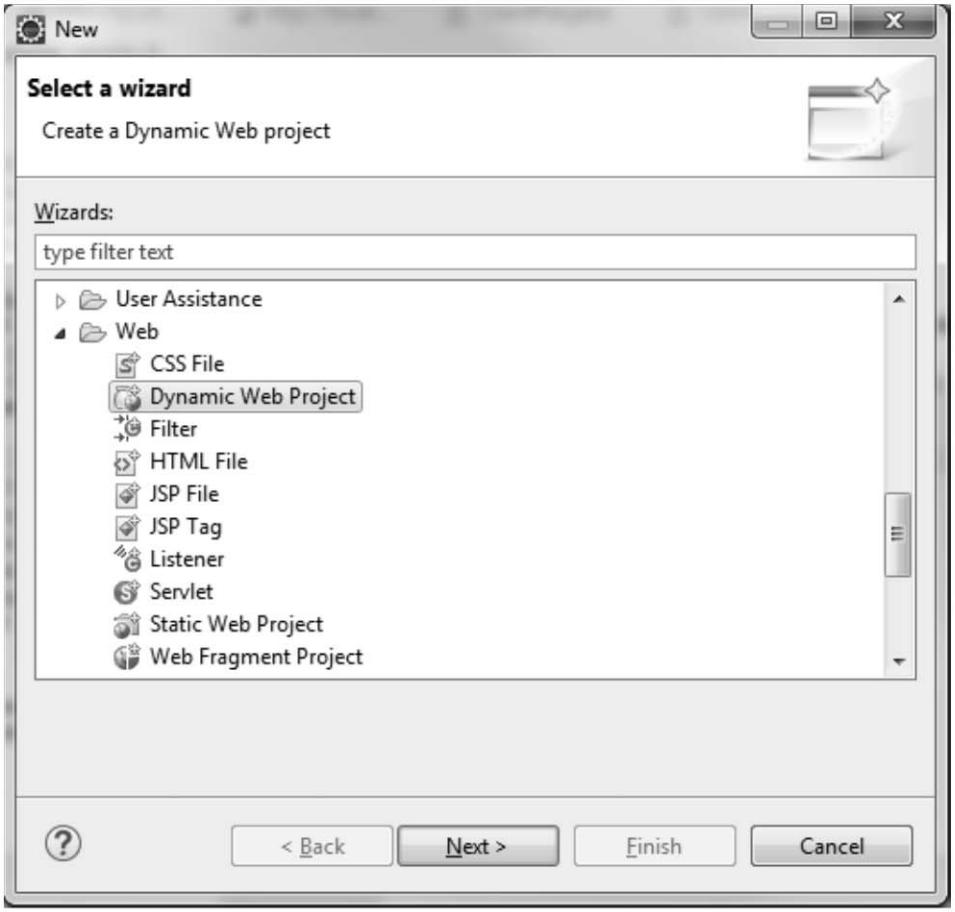


Figure 9.14

Copyright © 2015 Dunod.
© Dunod – Toute reproduction non autorisée est un délit.

On obtient ceci dans la perspective J2EE :

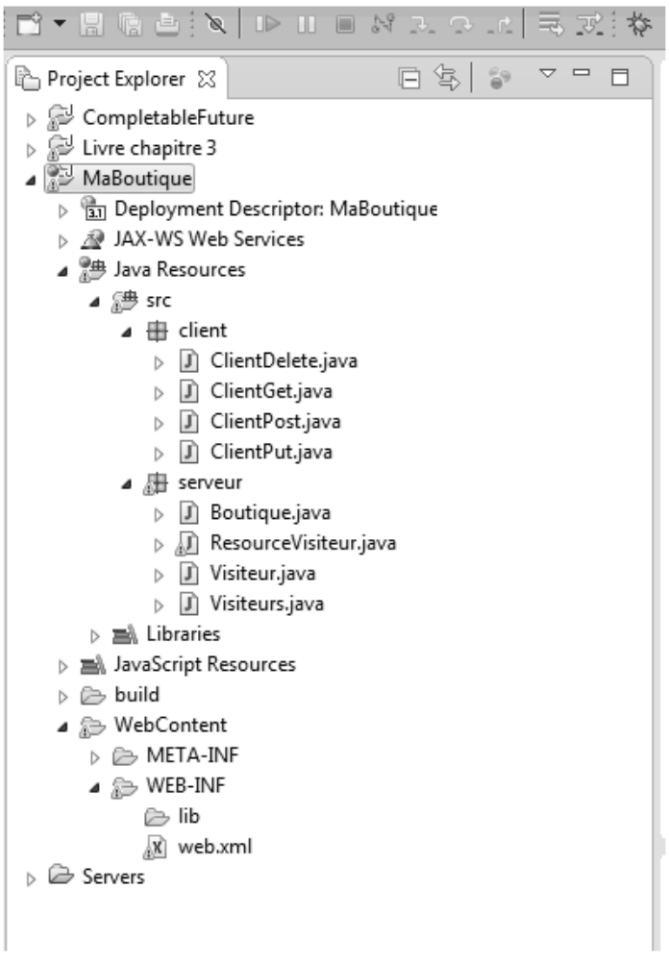


Figure 9.15

On ajoute les jars de RestEasy dans le répertoire **lib** et dans le CLASSPATH, et on crée un fichier **web.xml** sous WEB-INF de cette manière :

```
<web-app>
    <display-name>Archetype Created Web Application</display-name>

    <servlet>
        <servlet-name>Resteasy</servlet-name>
        <servlet-class>
            org.jboss.resteasy.plugins.server.servlet.HttpServletDispatcher
        </servlet-class>
        <init-param>
            <param-name>javax.ws.rs.Application</param-name>
            <param-value>serveur.Boutique</param-value>
        </init-param>
    </servlet>

    <servlet-mapping>
        <servlet-name>Resteasy</servlet-name>
```

Copyright © 2015 Dunod.

```
        <url-pattern>/*</url-pattern>
    </servlet-mapping>
</web-app>
```

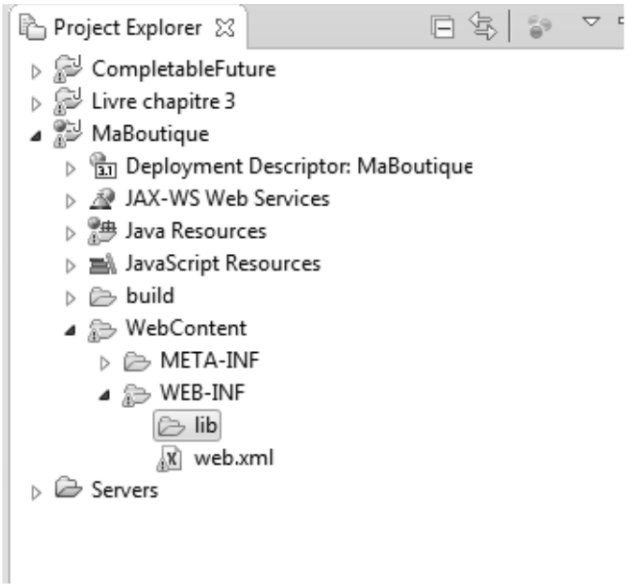


Figure 9.16

Ce fichier va permettre d'utiliser la servlet de dispatch de `RestEasy`.

9.10.3 Implémentation du code du domaine

Afin d'illustrer les fonctions CRUD, nous allons prendre un exemple d'une classe simple, `Visiteur`, qui possède seulement un nom et un courriel. Toutefois, nous allons lui rajouter des annotations JAXB afin de permettre une sérialisation/désérialisation transparente en XML.

```
@XmlRootElement(name = "visiteur")
@XmlAccessorType(XmlAccessType.NONE)
public class Visiteur implements Serializable {

        private static final long serialVersionUID = 1L;
        @XmlAttribute(name = "nom")
        private String nom;
        @XmlAttribute(name = "courriel")
        private String courriel;
        private int id;
        public int getId() {
                return id;
        }
        public void setId(int id) {
                this.id = id;
        }
        public String getNom() {
                return nom;
```

© Dunod – Toute reproduction non autorisée est un délit.
Copyright © 2015 Dunod.

```
        }
        public void setNom(String nom) {
               this.nom = nom;
        }
        public String getCourriel() {
               return courriel;
        }
        public void setCourriel(String courriel) {
               this.courriel = courriel;
        }
}
```

On remarquera l'utilisation de la déclaration explicite des attributs :

```
@XmlAttribute(name = "nom")
```

En combinaison avec l'utilisation de l'annotation :

```
@XmlAccessorType(XmlAccessType.NONE)
```

Afin de choisir quels attributs de la classe sont sérialisés en XML.

On doit également définir un conteneur pour les listes de visiteurs, que l'on appelle `Visiteurs` :

```
@XmlRootElement(name = "visiteurs")
@XmlAccessorType(XmlAccessType.NONE)
public class Visiteurs {
     @XmlElement(name="visiteur")
     private ArrayList<Visiteur> visiteurs;

     public ArrayList<Visiteur> getVisiteurs() {
      return visiteurs;
     }
     public void setVisiteurs(ArrayList<Visiteur> visiteurs) {
      this.visiteurs = visiteurs;
     }
}
```

9.10.4 Mise en œuvre du serveur

Le serveur va utiliser deux classes : la première classe va faire le lien avec les opérations `http`, la deuxième classe va faire le lien entre la classe de dispatch de l'application et la servlet `RestEasy`.

La première classe utilise les annotations REST en Java (tableau 1.3).

Copyright © 2015 Dunod.

Tableau 9.3

`@Path`	Spécifie le pattern d'URL associé à la méthode
`@GET`, `@PUT`, `@DELETE`, `@POST`	Spécifient la méthode HTTP utilisée
`@Consumes`	Spécifie le format attendu en entrée
`@Produces`	Spécifie le format attendu en sortie
`@PathParam`	Spécifie un paramètre de l'URL

La classe de gestion des visiteurs

La classe de gestion des visiteurs `ResourceVisiteur` va tout d'abord être idéntifiée par son path.

```
@Path("/visiteurs")
```

Elle va utiliser un dictionnaire des visiteurs numérotés par leur `id`, et un compteur de visiteur pour attribuer de nouveaux `id`. On remarquera l'utilisation d'un compteur atomique pour gérer les problèmes d'accès concurrents.

```
private Map<Integer, Visiteur> baseVisiteurs = new ConcurrentHashMap<Integer,
Visiteur>();
private AtomicInteger idCounter = new AtomicInteger(1);
```

On initialise la liste des visiteurs lors du constructeur de la classe :

```
public ResourceVisiteur() {
    Visiteur visiteur1 = new Visiteur();
    visiteur1.setNom("Lara");
    visiteur1.setCourriel("lara@croft");
    visiteur1.setId(1);
    baseVisiteurs.put(1,visiteur1);
    }
```

On peut ensuite écrire les correspondances entre les URL, le type de méthode HTTP et le type de données d'une part, les méthodes de la classe d'autre part.

Pour la méthode `GET`, on implémente deux URLs possibles, l'une pour obtenir tous les visiteurs avec le préfixe « / » et l'autre pour obtenir un visiteur particulier repéré par son id. Le format utilisé est XML puisque la classe `Visiteur` est en correspondance avec JAXB.

```
@GET
@Path("/")
@Produces("application/xml")
public Visiteurs getAllVisiteurs() {
       Visiteurs visiteurs = new Visiteurs();
       visiteurs.setVisiteurs(new
ArrayList<Visiteur>(baseVisiteurs.values()));
```

Copyright © 2015 Dunod.
© Dunod – Toute reproduction non autorisée est un délit.

```
        return visiteurs;
}
@GET
@Path("{id}")
@Produces("application/xml")
public Visiteur getVisiteurParId(@PathParam("id") int id) {
Visiteur visit = baseVisiteurs.get(id);
        if (visit == null) {
            throw new WebApplicationException(Response.Status.NOT_FOUND);
        }
        return visit;}
```

On peut à présent implémenter la méthode `PUT`, qui va modifier les attributs d'un visiteur de la base.

```
@PUT
@Path("{id}")
@Consumes("application/xml")
@Produces("application/xml")
public Visiteur updateVisiteur(@PathParam("id") int id, Visiteur update) {

   Visiteur current = baseVisiteurs.get(id);
   if (current == null) throw new
WebApplicationException(Response.Status.NOT_FOUND);
   if ( update.getNom() != null) current.setNom(update.getNom());
   if ( update.getCourriel() != null)
current.setCourriel(update.getCourriel());
   return current;
  }
```

Puis la méthode `POST`, qui va créer un nouvel enregistrement :

```
@POST
@Path("/")
@Consumes("application/xml")
public Response createVisiteur(Visiteur visit) {
    idCounter.set(idCounter.get()+1);
    int numero = idCounter.get();
      visit.setId(numero);
      baseVisiteurs.put(numero, visit);
    System.out.println("Visiteur créé " + visit.getId());
    return Response.created(URI.create("/visiteurs/" +
visit.getId())).build();
}
```

Et enfin la méthode `DELETE`, qui va supprimer un enregistrement identifié par son `id` :

```
@DELETE
@Path("{id}")
@Produces("text/plain")
public String deleteVisiteurParId(@PathParam("id") int id) {
Visiteur visit = baseVisiteurs.get(id);
```

Copyright © 2015 Dunod.

```
    if (visit == null) {
      return "Pas trouvé";
    }
    baseVisiteurs.remove(id);
    return "OK";}
```

La classe de lien avec la servlet RestEasy

Cette classe `Boutique` hérite de la classe `Application` de JAX-RS.

```
import javax.ws.rs.core.Application;
//@ApplicationPath("/MaBoutique")
public class Boutique extends Application {
       private Set<Object> singletons = new HashSet<Object>();
          public Boutique() {
             singletons.add(new ResourceVisiteur());
          }
          @Override
          public Set<Object> getSingletons() {
             return singletons;
          }
}
```

Elle va mettre à disposition notre application `ResourceVisiteur`, mais pourrait en gérer d'autres associées à d'autres URL.

9.10.5 Déploiement et test

Nous pouvons à présent déployer l'application et la tester. Pour cela il suffit de cliquer sur le projet `MaBoutique` et sélectionner par clic droit « Run as/Run on server ».

Un dialogue apparaît pour choisir le type de serveur utilisé, ici Tomcat 8 (voir figure 9.17).

On doit alors spécifier l'emplacement du serveur, et ensuite une fenêtre de navigateur web apparaît.

On peut vérifier la trace de Tomcat dans la console pour déceler s'il n'y a pas d'erreur.

Le chemin du projet est `MaBoutique`, le même que le nom du projet eclipse.

L'URL *http://localhost:8080/MaBoutique* renvoie une erreur, puisque l'on n'a pas défini de page d'accueil.

Par contre on peut à présent tester l'URL :

```
http://localhost:8080/MaBoutique/visiteurs/
```

Et on obtient l'affichage de la sérialisation XML de l'unique visiteur créé dans la base :

```
<?xml version="1.0" encoding="UTF-8" standalone="true"?>
<visiteurs><visiteur courriel="lara@croft" nom="Lara"/></visiteurs>
```

© Dunod – Toute reproduction non autorisée est un délit.
Copyright © 2015 Dunod.

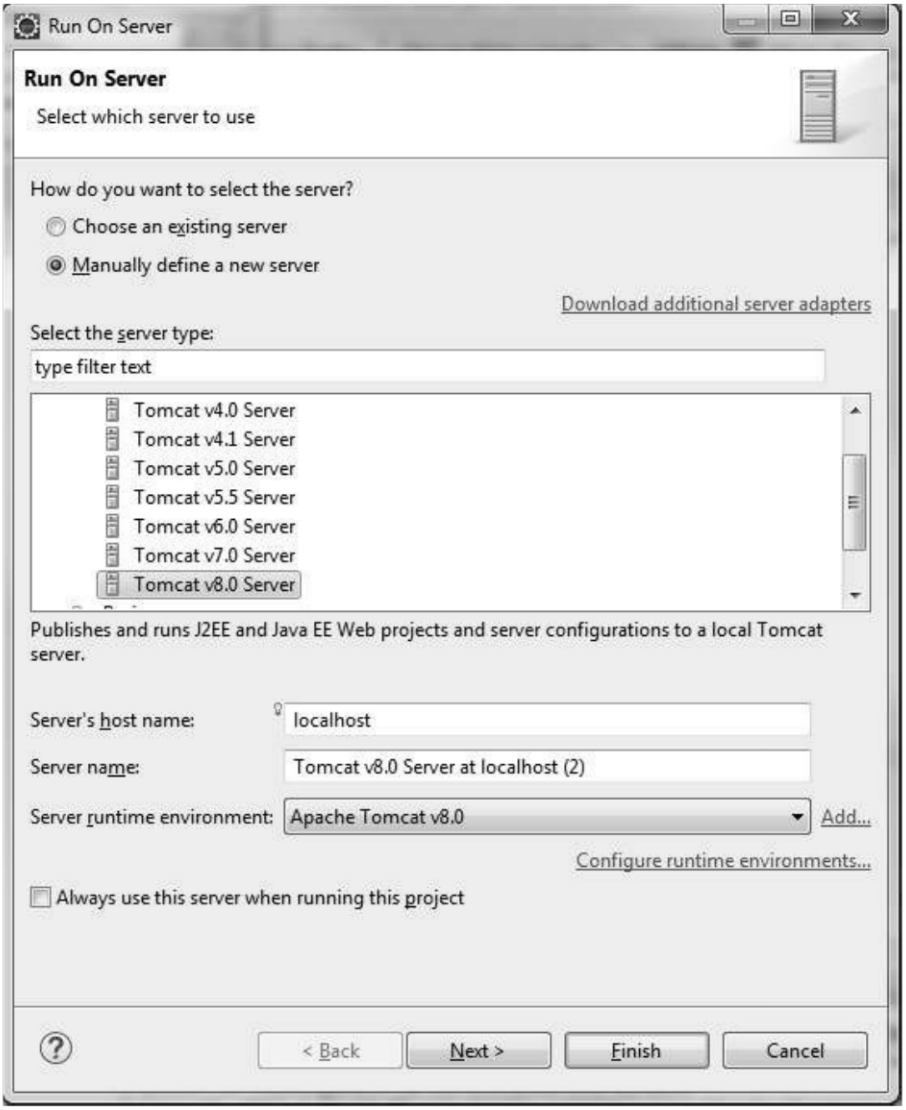


Figure 9.17

On peut également faire une requête get avec l'identifiant du visiteur, ici 1 :

```
http://localhost:8080/MaBoutique/visiteurs/1
```

et on obtient la sérialisation du premier enregistrement :

```
<?xml version="1.0" encoding="UTF-8" standalone="true"?>
<visiteur courriel="lara@croft" nom="Lara"/>
```

Par contre, il n'est pas possible de tester dans un navigateur classique les autres méthodes `PUT`, `POST`, `DELETE`. Nous allons utiliser pour cela un client Java.

Copyright © 2015 Dunod.

9.10.6 Client Java de test

JAX-RS offre une classe `Client` qui permet le test des méthodes HTTP.

Commençons par un client de la méthode GET, qui va effectuer une requête `http` sur l'URL du premier visiteur :

```
Client  client = ClientBuilder.newClient();
                //on regarde le premier visiteur
WebTarget target1 =
client.target("http://localhost:8080/MaBoutique/visiteurs/1");
Response response = target1.request().get();
Visiteur value = response.readEntity(Visiteur.class);
response.close();  // Fermons les connexions !
System.out.println(value.getNom());
```

On remarquera que l'on obtient directement un objet de type `Visiteur` !

Naturellement on peut obtenir tous les visiteurs :

```
Client  client = ClientBuilder.newClient();
                //on regarde tous les visiteurs
WebTarget target1 =
client.target("http://localhost:8080/MaBoutique/visiteurs/");
Response response = target1.request().get();
Visiteurs value = response.readEntity(Visiteurs.class);
response.close();  // Fermons les connexions !
for (Visiteur visit : value.getVisiteurs())
System.out.println(visit.getNom());
```

Créons à présent un nouvel enregistrement avec `POST`

```
Client  client = ClientBuilder.newClient();
         // on va créer un visiteur
WebTarget target =
client.target("http://localhost:8080/MaBoutique/visiteurs");
Visiteur visit = new Visiteur();
visit.setNom("Fred");
visit.setCourriel("fr@orange.fr");
 Response response =
target.request().post(Entity.entity(visit,"application/xml"));
       System.out.println(response.readEntity(String.class ));
       response.close();
```

On remarque que l'on passe directment au `POST` notre objet `Visiteur` Java, et la sérialisation est implicite du fait du `@Consumes( application/xml )`.

Ici notre implémentation renvoie un message pour dire si tout s'est bien passé, on va donc décoder le résultat avec `readEntity(String.class)`.

Si l'on regarde à nouveau notre navigateur, on vérifie que Fred a bien été créé (voir figure 9.18).

Copyright © 2015 Dunod.
© Dunod – Toute reproduction non autorisée est un délit.

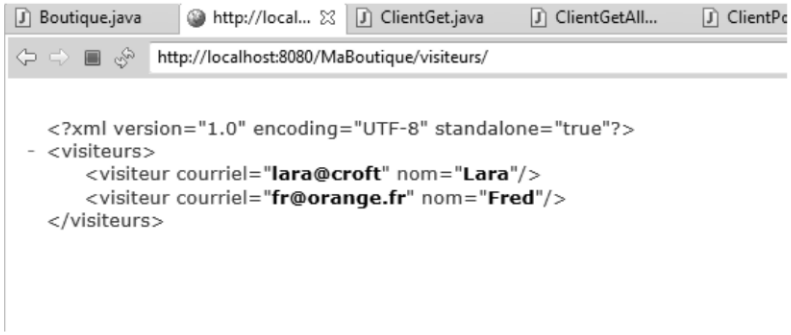


Figure 9.18

Et modifions le courriel de Lara avec `PUT` :

```
Client  client = ClientBuilder.newClient();
              //on regarde le premier client
WebTarget target1 =
client.target("http://localhost:8080/MaBoutique/visiteurs/1");
Visiteur visit= new Visiteur();
visit.setCourriel("lara@gmail.com");
Response response = target1.request().put(Entity.entity(visit,
"application/xml"));
Visiteur visiteurRetour = response.readEntity(Visiteur.class);
  response.close();  // Fermons les connexions !
System.out.println(visiteurRetour.getNom()+ " "+visiteurRetour.getCourriel());
```

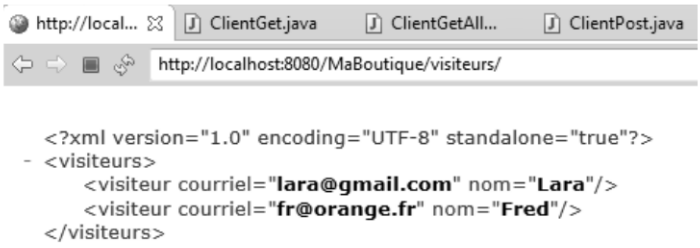


Figure 9.19

Et supprimons Fred avec DELETE :

```
Client client = ClientBuilder.newClient();
              // on regarde le premier client
WebTarget target = client
                  .target("http://localhost:8080/MaBoutique/visiteurs/2");
Response response = target.request().delete();
System.out.println(response.readEntity(String.class));
response.close(); // Fermons les connexions !
```

Copyright © 2015 Dunod.

REST permet donc d'éviter la création de nombreuses servlets Java pour les actions des formulaires web ordinaires. La mise en œuvre dans JAX-RS est quasiment transparente, mais nous ne dirons rien ici des connexions possibles aux bases de données. Il faut toutefois faire attention à bien gérer l'idempotence des requêtes afin d'éviter les doublons ou les copies.

En résumé

En résumé les services web permettent de faire communiquer des applications au travers du réseau internet en utilisant un encodage XML et le protocole SOAP. Ces services web sont interopérables avec d'autres langages, tels que C# de Microsoft. Les services publient une interface d'appel dans un fichier de description d'interface WSDL publié sur le web. Très souvent ce fichier est généré par un outil à partir du code d'implémentation. Un client peut générer ses propres souches pour appeler le service à partir du document WSDL publié par le web.

Les services web sont mis à disposition dans un serveur web, ce qui alourdit l'infrastructure, mais permet l'accès au travers de pare-feu. Autour des services web gravitent de nombreux groupes de travail visant à définir des standards pour la sécurité, les transactions, etc.

XML reste un encodage très verbeux, et de nombreux travaux (XOP[1], etc.) visent à le compresser et l'optimiser.

Par contre, les services web offrent une vue planétaire des échanges entre applications informatiques.

Copyright © 2015 Dunod.
© Dunod – Toute reproduction non autorisée est un délit.

1. http://www.w3.org/TR/xop10/

Copyright © 2015 Dunod.

10

JMX, test et déploiement

Objectif

L'objectif de ce chapitre est de sensibiliser le lecteur aux problèmes d'administration des applications réparties, ainsi qu'à la génération de journaux, en montrant que Java fournit en standard des interfaces prêtes à l'emploi. Là encore il s'agit d'un sujet très vaste dont seulement quelques éléments seront donnés ici.

10.1 ARCHITECTURE DE JMX

10.1.1 JMX est une API de J2SE

JMX (*Java Management Extension*) est une extension Java permettant la gestion et la configuration de serveurs Java en réseau, qui s'inspire du protocole SNMP (*Simple Network Management Protocol*) largement utilisé dans les télécommunications et la gestion d'appareils en réseau (imprimantes notamment).

Réservé initialement aux applications d'entreprise, il est à présent en standard dans J2SE 5. Il a été modifié légèrement dans Java 6 puis Java 7, mais ce livre en parlera brièvement.

JMX est à la fois une API et une architecture de surveillance de réseau Java avec des services spécifiques. Il est important dans la conception d'une architecture de prendre en compte la surveillance et la configuration très en amont du projet comme un élément à part entière de l'architecture.

Copyright © 2015 Dunod.
© Dunod – Toute reproduction non autorisée est un délit.

JMX est intéressant à plusieurs titres pour les applications réparties, d'une part parce que la configuration est une part importante dans une application répartie, d'autre part parce que JMX présente un modèle de communication de haut niveau qui a bénéficié de l'expérience de plusieurs années en Java, et a contourné certaines des erreurs de jeunesse de RMI. Notamment, il présente une abstraction du protocole de communication entre les objets contrôlés et les consoles de surveillance (clients JMX) permettant d'offrir des connecteurs soit en HTTP, soit en RMI, soit avec WBEM (*Web Based Entreprise Management*) du DTMF (*Distributed Management Task Force*[1]).

10.1.2 Motivation de JMX

L'administration d'un système réparti peut avoir plusieurs raisons d'être :

- gestion des configurations,
- gestion des performances,
- gestion des anomalies,
- gestion de la sécurité,
- gestion des informations comptables.

JMX permet de mettre en place une infrastructure d'administration et de surveillance et permet de gérer à la fois les ressources matérielles et logicielles en Java sur des applications en cours d'exécution. On peut gérer ou surveiller par exemple de manière dynamique :

- des ressources systèmes utilisées par Java : le nombre de threads, l'occupation mémoire ;
- du code embarqué possédant une JVM Java ;
- des objets Java simples : des piles, des EJB, des serveurs, etc. ;
- des identités de sécurité ;
- des niveaux de trace ou de debug.

Le code est instrumenté à partir d'objets de type `MBean` (pour *Management Bean*) qui vont exposer une interface de configuration et d'administration des objets réels.

JMX est une architecture par composants, ce qui permet une souplesse de mise en œuvre.

On conseille pour la mise en œuvre de JMX d'implémenter un design pattern de délégation, où on ne va pas directement administrer des objets d'application, mais envoyer les requêtes au travers d'un objet dédié à cette tâche.

JMX peut ainsi être rajouté par-dessus une architecture existante, sans remettre en cause celle-ci.

1. http://www.dmtf.org

Copyright © 2015 Dunod.

10.1.3 Architecture de JMX

Une architecture à niveaux

L'architecture de JMX comporte plusieurs niveaux comme le montre la figure 10.1.

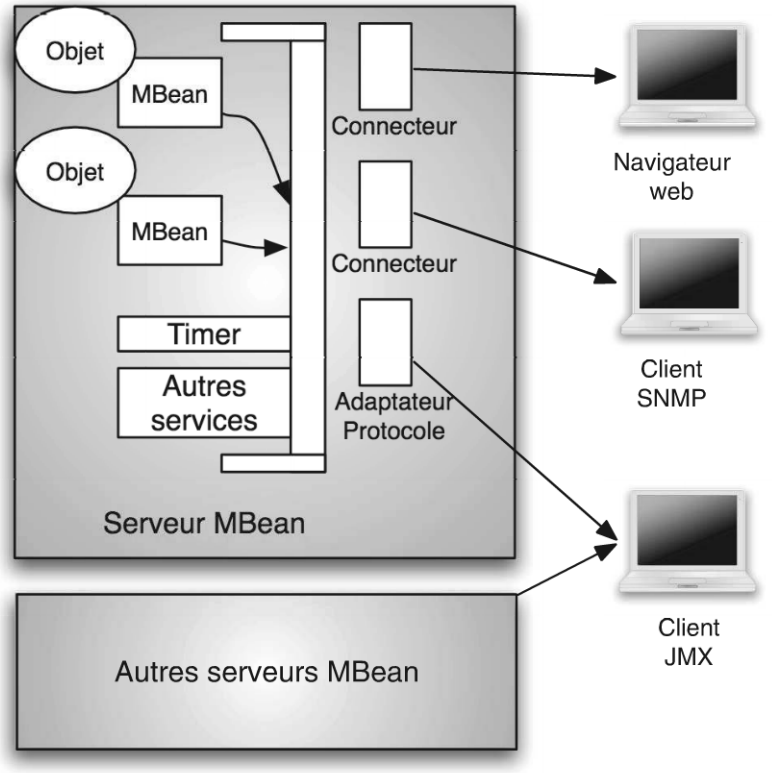


Figure 10.1 — L'architecture de JMX

Cette architecture repose sur des serveurs de `MBean` locaux à une machine qui vont administrer un ensemble de `MBean` : on appelle ce niveau l'instrumentation. Ces `MBean` en général ne sont pas les objets réels mais seulement des adaptateurs. Le serveur de `MBean` fournit des connecteurs pour permettre à des consoles d'administration de se connecter sous différents protocoles, notamment avec HTTP pour les navigateurs web et RMI pour les clients lourds JMX.

Niveau Instrumentation

Les ressources administrées par JMX peuvent être de natures très différentes. Les `MBean` exposent les attributs et les interfaces qui vont être administrés.

- Les **attributs** peuvent être en lecture/écriture ou lecture seule.
- Les **opérations** doivent pouvoir être appelées à distance, ce qui restreint les types d'arguments.
- Les **constructeurs** ne peuvent être appelés qu'après l'instanciation d'un `MBean`, ce qui les rend à peu près inutilisables. Il aurait été intéressant de pouvoir

Copyright © 2015 Dunod.
© Dunod – Toute reproduction non autorisée est un délit.

instancier un objet à distance. On doit donc se ramener au schéma classique d'utilisation des fabriques d'objets (*factories*).

- Les **notifications** sont les événements que peuvent envoyer les objets pour refléter des changements d'état (franchissement de seuil, etc.), ou bien pour refléter des erreurs dans le système.

Agent JMX

Le composant principal de l'architecture JMX est le `MBeanServer`, qui peut lui-même être administré. Il permet d'enregistrer les objets sous contrôle. Ce serveur va communiquer à l'aide de connecteurs ou d'adaptateurs de protocole soit avec un navigateur web, soit avec une console de surveillance standard ou propriétaire. Le J2SE fournit une console RMI et il est possible de rajouter une bibliothèque pour obtenir un adaptateur HTTP. Le serveur possède également des services particuliers, tels que des timers. C'est l'ensemble de ce serveur et des services associés que l'on appellera par la suite l'agent JMX. L'agent est au plus près des ressources sous son contrôle.

Le `MBeanServer` peut être invoqué en local ou à distance. Il permet d'effectuer les opérations suivantes :

- créer des `MBean`,
- enregistrer ces `MBean` dans son annuaire,
- invoquer des opérations sur ces `MBean`, ou lire/écrire des valeurs d'attributs,
- effectuer des requêtes sur les propriétés des `MBean`.

Il existe par défaut un `MBeanServer` par JVM. On peut y accéder par la commande suivante :

```
MBeanServer.getPlatformMBeanServer() ;
```

Par défaut, ce serveur gère quelques éléments essentiels de la JVM tels que le nombre de threads, la version de JDK, et la mémoire.

Connecteurs du MBeanServer

Le `MBeanServer` peut être administré à distance. Il sert alors de relais pour acheminer les requêtes vers les `MBean` : c'est en quelque sorte un point d'entrée unique.

Pour cela, il définit plusieurs types de connecteurs :

- connexion par RMI ou RMI/IIOP,
- connexion en HTTP.

La connexion en HTTP va limiter aux chaînes de caractères le type des arguments pouvant être affiché dans un browser.

Copyright © 2015 Dunod.

La définition d'un connecteur pour web services ou SOAP est en cours, et adhérera vraisemblablement au standard WS-Man[1] défini par le DMTF. Ceci permettra d'ouvrir l'administration de programmes Java à bien d'autres plates-formes dans des langages différents.

JMX peut être utilisé selon différents modes :

- Soit à l'initiative du client JMX qui va rechercher des informations (mode *pull*) sur l'état de l'objet ou modifier sa configuration.
- Soit à l'initiative de l'objet administré qui va émettre des événements (mode *push*) à destination des consoles de surveillance (clients JMX).

10.2 MISE EN ŒUVRE DE JMX

10.2.1 Plusieurs types de MBeans

Il existe essentiellement deux types de `MBean` : **standards** et **dynamiques**.

Les `MBean` **standards** doivent implémenter une interface dont le nom se termine nécessairement par `MBean`. Ce sont les opérations de cette interface qui sont exposées.

Les `MBean` **dynamiques** implémentent l'interface `javax.management.DynamicMBean`.

Pour les `MBean` dynamiques, il ne s'agit pas de modifier l'interface de manière dynamique, mais de programmer la publication des méthodes exposées à l'administration. Pour tous les types de beans, JMX utilise la réflexion (voir chapitre 2) pour générer dynamiquement l'interface enregistrée dans le `MBean` serveur.

Il existe deux autres types de `MBean` dynamiques : les `OpenMBean` et les `ModelMBean`.

Exemple de StandardMBean

Le nom de l'interface d'un `StandardMBean` doit nécessairement se terminer par le suffixe `MBean`. Par contre il peut s'agir d'une classe Java quelconque, à condition qu'elle implémente l'interface. Ici on considère un serveur web que l'on souhaite pouvoir réinitialiser avec la méthode `resetServer` et administrer deux attributs :

- Le nombre maximum de connexions `MaxConnexions` en lecture et écriture, ce qui donne une méthode `get` et une méthode `set`.
- Le nombre de hits sur le serveur en lecture seule, `IpHit` :

```
public interface ServeurWebMBean {
  public Integer getMaxConnexions() ;
  public void setMaxConnexions(Integer i) ;
  public IPHitItem[] getIPHit()
  public void resetServer(DateTime t, Integer m); }
```

1. http://www.dmtf.org/standards/wsman

© Dunod – Toute reproduction non autorisée est un délit.
Copyright © 2015 Dunod.

L'implémentation du `MBean` correspondant sera par exemple :

```
public class ServeurWeb implements ServeurWebMBean
{
public ServeurWeb(){
    maxConn = 5;
    maTable = new IPHitTable();
};
public Integer getMaxConnexions(){return maxConn; };
public void setMaxConnexions(Integer I) {
    maxConn = I;
    if (connexionsCourantes >= maxConn) // fermer des connexions ... } };
public IPHitTable getIPHit() { return maTable; };
public void resetServer(DateTime t, Integer m) {
// planification de réinitialisation ...}
};
```

Interface de MBean dynamique

L'interface de `MBean` dynamique est prédéfinie de manière générique et utilise au maximum la réflexion (voir chapitre 1). L'objet `MBeanInfo` regroupe toutes les informations de réflexion d'un `MBean` (voir chapitre 2), la liste de ses opérations, attributs et notifications. Naturellement ces opérations peuvent être invoquées de manière dynamique :

```
interface DynamicMBean {
Object getAttribute(String attribute); // valeur d'un attribut
void setAttribute(Attribute attribute); // affectation d'un attribut
AttributeList getAttributes(String[] attributes); // liste des attributs
MBeanInfo getMBeanInfo(); // objet regroupant toutes informations
Object invoke(String actionName, Object[] params, String[] signature);
// invocation dynamique d'une opération
AttributeList setAttributes(AttributeList attributes);
// affectation multiple
}
```

Implémentation de MBean dynamique

```
public class ServeurWeb implements DynamicMBean {
public ServeurWeb() {
        buildDynamicMBeanInfo();
}
public Object getAttribute(String attribute_name)
        throws AttributeNotFoundException, MBeanException,
            ReflectionException {
        if (attribute_name.equals("MaxConnections")) {
            return getMaxConnections();}
        if (attribute_name.equals("IpHit")) { return getIPHit(); }
        throw new AttributeNotFoundException("attribute_name +
        " introuvable dans" + dClassName);}
```

Copyright © 2015 Dunod.

```
public Object invoke(String operationName, Object params[],
       String signature[]) throws MBeanException, ReflectionException {
    if (operationName.equals("resetServer")) {resetServer();
    return null; ...
....}
```

10.2.2 Nommage des MBeans

Afin d'administrer les `MBean`, il est nécessaire de les nommer de manière unique dans le système. Le nom d'un `MBean` répond à une codification très stricte. Il appartient à un domaine, et possède une liste d'attributs valeurs. Ainsi on pourra avoir des noms constitués de la manière suivante :

```
Cache:type=CacheManager
Cache:type=CacheManager,name="Permanent Cache "
java.lang:type=GarbageCollector,name=MarkSweep
```

Ces attributs permettront au gestionnaire d'effectuer des requêtes. Il est également souhaitable de créer des hiérarchies de composants par exemple :

```
• domain:type=Server,name=server5
• domain:type=Server.Application,Server=server5,name=app1
• domain:type=Server.Application.WebModule,Server=server5,
    Application=app1,name=module3
• domain:type=Server.Application.WebModule.Servlet,Server=server5,
    Application=app1,WebModule=module3,name=default
```

10.2.3 Mise à disposition d'un MBean dans un serveur

Implémentation du MBean

Dans l'exemple suivant, on va assimiler l'objet administré au `MBean`, et décrire son interface d'administration en suivant la convention du suffixe `MBean`. Cet objet expose un attribut `Message` accessible en lecture et écriture et une opération `bonjour`.

```
public interface HelloMBean {
    public void setMessage(String message);
    public String getMessage();
    public void bonjour();
}
```

On remarquera que l'interface n'hérite d'aucune interface particulière, contrairement aux objets RMI.

On définit ensuite l'implémentation de cette interface, la classe `Hello` :

```
public class Hello implements HelloMBean {
    private String message = null;
    public Hello() {message = "Bonjour le monde";}
```

Copyright © 2015 Dunod.
© Dunod – Toute reproduction non autorisée est un délit.

```
    public Hello(String message) {this.message = message;}
    public void setMessage(String message) {this.message = message;}
    public String getMessage() {return message;}
    public void bonjour() {System.out.println(message);}}
```

Mise à disposition de l'objet dans le serveur

Par défaut, toute JVM possède un `MBeanServer`.

```
MBeanServer mbs = MBeanServerFactory.createMBeanServer("SimpleAgent");
```

Il est possible d'enregistrer une instance de `Hello` dans ce serveur, afin de pouvoir l'administrer en lui donnant le nom `bonjourlabas` :

```
Hello helloBean = new Hello();
ObjectName helloName = null;
helloName = new ObjectName("SimpleAgent:name=bonjourlabas");
mbs.registerMBean(helloBean, helloName);
```

Un `MBean` peut être enregistré sous plusieurs noms et dans plusieurs serveurs.

10.2.4 Ajout d'un connecteur dans le serveur

Il est alors possible de créer un connecteur afin d'avoir accès à distance au serveur. Plusieurs connecteurs sont fournis en standard, notamment un connecteur RMI et un connecteur HTTP.

Connecteur RMI

L'ajout du connecteur RMI nécessite la définition d'une URL. Cette URL donne un nom propre au serveur dans le registre RMI, ici `server`. On reconnaîtra le port 1099 de `rmiregistry`.

```
JMXServiceURL url = new
    JMXServiceURL("service:jmx:rmi:///jndi/rmi://localhost:1099/server");
        JMXConnectorServer cs =
JMXConnectorServerFactory.newJMXConnectorServer(url, null, mbs);
        cs.start();
```

Dans le cas d'un connecteur RMI, le JDK fournit un outil de visualisation et d'administration simple, `jconsole`. On peut alors tester le résultat en lançant successivement les éléments suivants :

- Le serveur de noms RMI sur le port 1099 par une commande en ligne : `rmiregistry`.
- Le programme principal.
- L'outil `jconsole` : `jconsole`.

Copyright © 2015 Dunod.

Dans l'outil `jconsole`, on peut alors dans l'onglet « *Advanced* », taper l'URL du code suivant :

```
service:jmx:rmi:///jndi/rmi://localhost:1099/server
```

On voit alors apparaître la fenêtre qui apparaît en figure 10.2.

On voit clairement apparaître le nom de domaine de l'objet : `SimpleAgent`, ainsi que le nom de l'objet : `bonjourlabas`.

On remarque également un autre domaine, `JMImplementation`, qui est enregistré d'office et permet d'aller administrer certains éléments de la JVM, en particulier d'observer la taille mémoire, le nombre de threads, etc.

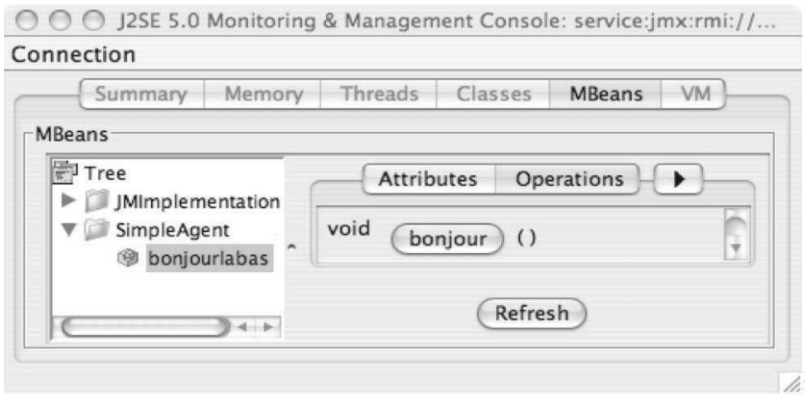


Figure 10.2 — Console d'administration JMX avec un connecteur RMI

Contrairement à `rmiregistry` qui nécessite de disposer des souches des objets dans son `classpath`, la `jconsole` n'a besoin d'aucune classe pour tourner. Par contre, ceci limite le type des arguments des méthodes administrées. Les `OpenMBean` permettent de définir des types de données plus complexes, mais par souci de concision il n'en sera pas question ici. Le JDK 6 permet d'enrichir les types de données pouvant être échangées entre la `jconsole` et les objets administrés.

La version 6 rajoute également des informations sur les domaines de valeur des attributs, qui permettent de définir de beaux graphiques sur l'évolution des paramètres.

Connecteur HTTP

Cette restriction est encore plus forte pour un connecteur de type HTTP. Les seuls arguments qui peuvent alors transiter sont de type `String`.

On reprend la création et l'enregistrement de l'objet `Hello` comme précédemment, mais cette fois on utilise l'adaptateur suivant :

```
HtmlAdaptorServer adapter = new HtmlAdaptorServer();
ObjectName adapterName =
    new ObjectName("SimpleAgent:name=htmladapter,port=8000");
```

© Dunod – Toute reproduction non autorisée est un délit.
Copyright © 2015 Dunod.

```
adapter.setPort(8000);
mbs.registerMBean(adapter, adapterName);
adapter.start();
```

On notera que l'adaptateur est lui aussi un MBean. Cet adaptateur n'est pas standard mais se trouve dans l'implémentation de référence de JMX[1]. dans la bibliothèque `jmxtools.jar`. Cet adaptateur comporte un mini-serveur web.

Il suffit alors de lancer le programme puis d'ouvrir un navigateur internet à l'adresse suivante : `http://localhost:8000`. On observe alors la fenêtre de la figure 10.3.



Figure 10.3 — Navigateur web de visualisation d'un `MBean`

Il existe d'autres types de connecteurs et d'adaptateurs, dont certains proviennent de bibliothèques tierces parties :

- Connecteurs :
  - RMI et RMI/IIOP
  - JMX Messaging Protocol (JMXMP) (JSR 160)
  - Services web (JSR 262)
  - XML/RPC, JMS (Lingo)...

1. http://www.oracle.com/technetwork/java/javase/tech/javamanagement-140525.html

Copyright © 2015 Dunod.

- Adaptateurs :
  - SNMP (JSR163)
  - Web (HTML/HTTP)
  - CORBA...

10.2.5 Notifications

Il est possible à un `MBean` d'envoyer des notifications à la console, afin d'avertir l'administrateur de certains événements (passage de seuil, erreurs, etc.). Dans ce cas c'est le `MBean` qui prend l'initiative (mode *push*), contrairement au mode normal ou la console lui demande des informations (mode *pull*).

Les notifications utilisent un mode publication/abonnement, tout comme JMS (voir chapitre JMS). Il existe plusieurs types de `Notification` :

- `AttributeChangeNotification`, `JMXConnectionNotification`,
- `MBeanServerNotification`, `MonitorNotification`,
- `RelationNotification`,
- `TimerAlarmClockNotification`, `TimerNotification`...

Pour qu'un `MBean` puisse envoyer des notifications, il suffit qu'il implémente l'interface `NotificationBroadcaster` ou sa fille `NotificationEmitter`.

Tout objet qui souhaite recevoir des notifications doit implémenter l'interface `NotificationListener`.

Exemple

Reprenons l'exemple de `HelloMBean`. Dans l'implémentation de la classe `Hello`, on va seulement modifier le code en rajoutant une notification sur la modification de l'attribut `message`.

On va également rajouter une variable d'instance pour stocker le numéro de séquence du message. En effet la notification de modification d'attribut prend les arguments suivants :

```
public AttributeChangeNotification(Object source,
long sequenceNumber, // numéro de séquence de l'événement
long timeStamp, // date de l'événement
String msg, // texte d'information
String attributeName, // nom de l'attribut modifié
String attributeType, // type de l'attribut
Object oldValue, // ancienne valeur
Object newValue) // nouvelle valeur
```

Ainsi le code modifié sera le suivant :

```
public class Hello extends NotificationBroadcasterSupport
    implements HelloMBean {
private String message = null;
```

© Dunod – Toute reproduction non autorisée est un délit.
Copyright © 2015 Dunod.

```
private int numeroSequence;
public Hello() {message = "Bonjour le monde";}
public Hello(String message) {this.message = message;}
public void setMessage(String nouveauMessage) {
    String ancienMessage=message;
    this.message = nouveauMessage;
    Notification n = new AttributeChangeNotification(this,
        numeroSequence++,
        System.currentTimeMillis(),
        "Message modifie",
        "Message", // nom attribut
        "String", // type attribut
        ancienMessage,
        nouveauMessage);
        sendNotification(n); }
@Override
public MBeanNotificationInfo[] getNotificationInfo() {
    String[] types = new String[] {
    AttributeChangeNotification.ATTRIBUTE_CHANGE };
    String name = AttributeChangeNotification.class.getName();
    String description = "Un attribut de ce MBean a change";
    MBeanNotificationInfo info = new MBeanNotificationInfo(types, name,
    description);
    return new MBeanNotificationInfo[] {info};
}
    public String getMessage() {return message;}
    public void bonjour() {System.out.println(message);}}
```

Pour tester cet exemple, il suffit de réenregistrer le `MBean` sur le serveur avec un connecteur RMI, d'ouvrir la console et de modifier l'attribut message. Dans l'onglet « *Notifications* » on voit alors apparaître le texte d'information de la variable `description`.

10.2.6 Requêtes d'un serveur MBeanServer

Le `MBeanServer` est capable d'effectuer des requêtes, afin de sélectionner la liste des objets satisfaisant à certaines propriétés. Typiquement il peut faire une recherche par nom, par domaine, ou sur les valeurs de l'une des propriétés du `ObjectName`.

On peut tout d'abord obtenir la liste des domaines sur un adaptateur RMI de référence `jmxconnector` comme ceci :

```
JMXServiceURL url = new JMXServiceURL(
"service:jmx:rmi:///jndi/rmi://localhost:1099/jmxconnector");
JMXConnector jmxc = JMXConnectorFactory.connect(url, null);
// Get an MBeanServerConnection
mbsc = jmxc.getMBeanServerConnection();
String domains[] = mbsc.getDomains();
```

Et obtenir la liste des objets pour le domaine `Mantaray` :

```
Set names=mbsc.queryNames(ObjectName.getInstance("MantaRay:*"),null);
for (Object name : names) System.out.println(name);
```

Copyright © 2015 Dunod.

On peut également faire des requêtes sur un attribut du `MBean`, ici ceux dont le nom de domaine commence par « Mantaray » :

```
Set<ObjectInstance> objects;
objects = mbsc.queryMBeans(ObjectName.getInstance("MantaRay:*"),null);
```

Il existe un langage de requête sophistiqué sur les attributs de `MBean` greffé sur la classe `Query` que le lecteur pourra approfondir par lui-même. Par exemple, pour rechercher les `MBean` de nom Dupont et d'âge > 5 :

```
QueryExp exp = Query.and(Query.gt(Query.attr("age"),Query.value(5)),
                         Query.match(Query.attr("nom"),
                                     Query.value("Dupont")));
```

10.2.7 Découvertes d'agents JMX et annuaires

Il existe de nombreux protocoles de découverte de services sur le réseau : UpNP, promu par Microsoft, Bonjour d'Apple, SLP (*Service Location Protocol*) utilisé par de nombreux Linux et Jini, le framework de découverte Java. Les agents JMX ont la possibilité de s'interfacer avec de tels services de découvertes, afin de permettre de découvrir de manière dynamique tous les objets Java administrés sur le réseau. La plupart des services de découverte permettent l'enregistrement et le retrait dynamique des agents, ainsi que les requêtes sur le réseau en mode page jaune, selon des valeurs d'attributs. Ils utilisent des techniques de multicast pour synchroniser les enregistrements d'agents, ainsi que des mécanismes de baux renouvelables, afin de ne pas garder trace des agents disparus.

Bien entendu, il est aussi possible de répertorier statiquement des agents JMX dans un annuaire LDAP (voir chapitre 7).

10.3 JOURNALISATION

10.3.1 La journalisation partie prenante du développement et de l'administration

L'administration des applications réparties nécessite souvent la journalisation des événements pour une raison simple : le flux d'entrée standard est réparti sur chaque machine ! Il est donc nécessaire de collecter les informations en un endroit centralisé, afin de recouper le déroulement des différents programmes clients et serveurs.

La **journalisation** (*logs*) consiste à garder les traces sur un support sûr des événements survenus dans un système ou dans une application. Un ou plusieurs fichiers de log au format prédéfini sont générés en cours d'exécution et conservent des messages informant sur la date et l'heure de l'événement, la nature de l'événement et sa gravité par un code ou une description sémantique, éventuellement d'autres informations : utilisateur, classe, etc.

Copyright © 2015 Dunod.
© Dunod – Toute reproduction non autorisée est un délit.

Les journaux peuvent utilement être réutilisés par :

- Un administrateur afin de produire des statistiques sur l'utilisation d'un système (par exemple les logs du serveur web Apache).
- Un agent de sécurité pour réaliser un audit de contrôle d'accès.
- Un développeur afin de détecter des défaillances et de corriger les bugs qui en sont responsables, il est plus facile de repérer la source d'une défaillance si le journal est dense en informations (fonctions appelées, valeurs des paramètres passés...).
- Un développeur pour éviter de polluer son code avec des `println`().
- Un utilisateur afin de revenir sur un crash et refaire les opérations qui auraient été perdues (transactions).

10.3.2 La bibliothèque Log4J

Java SE fournit une API pour la journalisation dans le package `java.util.logging`. Toutefois cette API est très peu utilisée. `Log4j`[1] est un framework très répandu et extensible qui permet de logger et débugger des applications Java. Une nouvelle version `Log4j 2` est à présent disponible et offre davantage de performance. Il devient très important pour des applications réparties puisqu'il va permettre de centraliser et de synchroniser des traces d'exécution d'événements ou d'applications éparpillées sur le réseau. Il joue un rôle important également pour la sécurité pour retrouver par exemple la trace d'un intrus dans les machines. Si au contraire on se fie à des journaux stockés en local sur chaque machine, et que celles-ci ont une dérive d'horloge légère, on ne pourra pas retrouver la causalité entre un accès à 13h40:31 et un accès à 13h40:29 dont l'estampille aura été inversée !

`Log4J` est en accès libre et incontournable en Open Source.

10.3.3 Les différents types de composants

Le framework utilise trois types de composants : `Logger`, `Appender` et `Layout`.

Ces composants communiquent entre eux pour déterminer : quels messages doivent être journalisés, la façon de journaliser les messages et la présentation des messages.

Logger

Le `Logger` est le point d'entrée d'un message. C'est pratiquement une des seules classes que nous allons utiliser.

`Log4j` gère les `Logger` de façon hiérarchique. C'est-à-dire qu'un `Logger` peut avoir des enfants et des parents. La hiérarchie est gérée par le nom des `Logger`. Les niveaux sont définis par des points, un peu comme la structure des packages de classes.

1. http://logging.apache.org/log4j

Copyright © 2015 Dunod.

Par exemple :

- `com`
- `com.afceurope`
- `com.afceurope.layout`

`com` est le parent de `com.afceurope` et l'ancêtre de `com.afceurope.layout`.

Log4j possède par défaut un `Logger` à la racine de la hiérarchie. Il se nomme `rootLogger`.

Lorsque l'on demande à Log4j de créer un `Logger`, il vérifie dans la configuration s'il y a quelque chose de défini pour celui-ci. S'il ne trouve rien, il remonte dans la hiérarchie jusqu'à ce qu'il trouve quelque chose. C'est pourquoi le `rootLogger` qui est à la racine est très important.

Appender

`Appender` est une interface dont le rôle est de filtrer les messages et de déterminer le mode de sortie des messages selon l'implémentation (fichier, serveurs de sockets, etc.).

Un `Logger` peut avoir un nombre quelconque d'`Appender`. Pour ajouter un `Appender` manuellement il suffit de faire :

```
monLogger.addAppender(monAppender);
```

Un `Logger` utilise les `Appender` de ses ancêtres. Donc les `Appender` définis pour le `rootLogger` seront utilisés par tous les `Logger`, quel que soit leur niveau.

Log4j fournit de base les `Appender` utilisés pour les tâches courantes :

- `ConsoleAppender` : envoie les messages dans la console du système.
- `FileAppender` : envoie les messages dans un fichier.
- `SocketAppender` : envoie les messages dans une socket.
- `SMTPAppender` : envoie les messages par mail.

Il en existe d'autres pour les utilisateurs plus avancés et il est évidemment possible de créer son propre `Appender` en implémentant l'interface `org.apache.log4j.Appender`. La figure 10.4 illustre comment centraliser des journaux de plusieurs machines en utilisant des sockets (`SocketAppender`). On se reportera au chapitre 3 pour l'utilisation de sockets.

Layout

Les `Layout` définissent le format du message. On peut par exemple avoir la date, l'heure, le nom de la classe qui envoie le message et le niveau de journalisation au début de chaque message. C'est le rôle du `Layout` de faire cela et non celui du développeur qui envoie le message.

Copyright © 2015 Dunod.
© Dunod – Toute reproduction non autorisée est un délit.

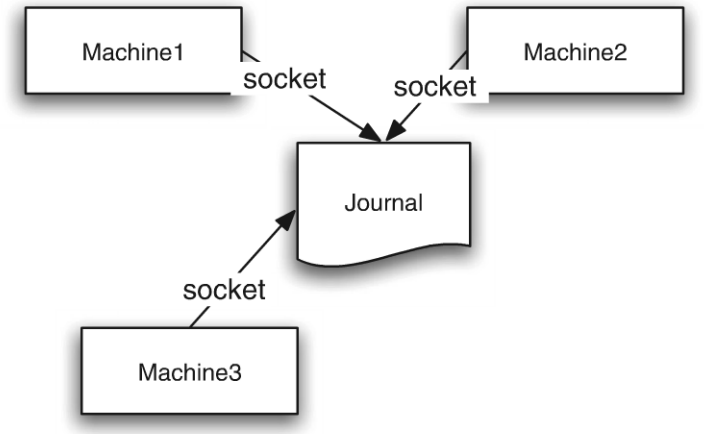


Figure 10.4 — Centralisation et synchronisation des journaux en utilisant un protocole par socket

Lorsqu'on envoie un message à un logger, il faut spécifier un niveau. Par défaut Log4j offre les niveaux suivants :

- `DEBUG`
- `INFO`
- `WARN`
- `ERROR`
- `FATAL`

Les niveaux sont gérés par ordre de priorité, `DEBUG` étant le moins important.

Par exemple, durant la période de développement, on utilise le niveau `DEBUG`. Une fois que l'application est déployée en production, on change le niveau pour `WARN` afin d'ignorer les messages de déverminage, sans modifier une seule ligne dans notre code.

10.3.4 Utilisation de Log4J

Fichier de configuration

En plus de la bibliothèque `log4j.jar` à mettre dans le `classpath`, il faut disposer d'un fichier de configuration de nom `log4j.properties` tel que celui-ci :

```
#définition du niveau et des Appender du rootLogger
log4j.rootLogger=DEBUG, monAppender
#configuration de "monAppender"
#nous allons envoyer les messages dans la console de Tomcat
log4j.appender.monAppender=org.apache.log4j.ConsoleAppender
#définition du Layout pour "monAppender"
log4j.appender.monAppender.layout=org.apache.log4j.PatternLayout
#définition du pattern d'affichage pour "monAppender"
#voici un exemple de sortie que l'on va obtenir :
```

Copyright © 2015 Dunod.

```
#2005-06-18 14:53:37 DEBUG [Main] Hello World
log4j.appender.monAppender.layout.ConversionPattern=%d{yyyy-MM-dd HH:mm:ss}
%-5p [%c{1}] %m%n
```

Utilisation du framework

Prenons à présent un exemple de code simple :

```
import org.apache.log4j.Logger;
public class HelloService {
// déclaration et création du Logger
private final static Logger logger = Logger.getLogger(HelloService.class);
    public String helloWorld() {
        logger.debug("Le service helloWorld est invoqué");
        return "Hello, world";
    }
}
```

Dans l'exemple précédent le texte « le service ... est invoqué » n'apparaîtra dans les journaux que si le niveau est supérieur ou égal à DEBUG. Typiquement, il n'apparaîtra pas en mode de production d'une application. Avec le fichier de configuration, il est possible de changer de manière centralisée le niveau de trace de toutes les applications.

Si la création de la trace est trop coûteuse, on pourra également la mettre dans une conditionnelle.

En résumé

Java fournit en standard une interface d'administration sur des objets de la JVM et sur la JVM elle-même appelée JMX. On peut utiliser les consoles d'administration de Java SE `jconsole`, utiliser des adaptateurs ou des connecteurs dans de nombreux protocoles répartis et écrire son propre client d'administration à distance.

La journalisation est un point important des applications réparties dans la mesure où elle permet le déverminage ou la surveillance du fonctionnement d'une application.

Copyright © 2015 Dunod.

Copyright © 2015 Dunod.

11

Courriels et messagerie

Objectif

Le but de ce chapitre est d'introduire les techniques d'envoi de courriels et de messagerie instantanée. L'offre de messagerie est très large, et l'on a donc choisi un protocole très répandu, XMPP, et une implémentation en open source, afin de permettre au lecteur de faire ses propres expérimentations.

11.1 UTILISATION DE COURRIELS EN JAVA

11.1.1 Introduction aux serveurs de courriel

La première chose à comprendre pour manipuler des courriels en Java est que le serveur d'envoi n'est pas identique au serveur de réception. Le serveur d'envoi utilise le protocole SMTP (*Simple Mail Transfer Protocol*), alors que la réception utilise soit le protocole IMAP (*Internet Message Access Protocol*), soit le protocole POP (*Post Office Protocol*). POP a été conçu pour récupérer le courriel de manière occasionnelle lorsqu'on n'est pas connecté en permanence. La boîte est bloquée lors de l'accès POP, et il n'est pas possible à une autre boîte ou un autre appareil d'y accéder. À l'inverse avec IMAP tous les messages restent sur le serveur, et l'on ne relève plus sa boîte mais on se contente de la synchroniser avec le serveur. Il est ainsi possible d'y accéder de n'importe quel appareil.

Le protocole SMTP est un protocole assez simple. On commence par spécifier l'expéditeur du message, puis le ou les destinataires, l'objet du message et son contenu, avant de l'envoyer.

Copyright © 2015 Dunod.
© Dunod – Toute reproduction non autorisée est un délit.

La deuxième chose est de bien comprendre la structuration du message en corps et pièces attachées.

11.1.2 L'interface JavaMail

Il est possible d'utiliser une interface optionnelle de Java, JavaMail[1], pour envoyer ou recevoir des courriels en Java. Cette interface est optionnelle avec J2SE mais est incluse dans J2EE.

Nous prendrons ici la version 1.5 avec J2SE. Nous allons commencer par télécharger le jar existant `javax.mail.jar` sur le site :
https://java.net/projects/javamail/pages/Home.

La documentation de l'API est disponible sur le site :
https://javamail.java.net/nonav/docs/api/

Pour réaliser les exemples ci-après, il faut disposer de l'adresse d'un serveur d'envoi sur lequel on est identifié, tel que `smtp.orange.fr`.

11.1.3 Les principales classes

La classe `Session` permet de gérer l'envoi.

La classe `Store` permet de gérer les boîtes aux lettres. Le nom de la boîte par défaut est `INBOX`.

La classe `Message` permet de gérer le message avec son corps, son objet, ses destinataires et son émetteur et ses pièces attachées.

La classe `Transport` effectue les connexions réseau de bas niveau.

11.1.4 Le code d'envoi d'un message

Le protocole SMTP (*Simple Mail Transfer Protocol*) est un standard de l'Internet pour l'envoi de messages. Le port par défaut de SMTP est 25, mais il devient 587 avec authentification et 487 avec SSL. On commence par définir les paramètres de connexion :

```
private static final String SMTP_HOST_NAME = "smtp.orange.fr";
private static final String SMTP_AUTH_USER = "moi@orange.fr";
private static final String SMTP_AUTH_PWD  = "****";
private static final String MAIL_ADDRESS = "moi@orange.fr";
```

On peut ensuite créer la session de connexion, créer le message, et envoyer le message.

1. https://java.net/projects/javamail/pages/Home

Copyright © 2015 Dunod.

```
Properties props = new Properties();
        props.put("mail.transport.protocol", "smtp");
        props.put("mail.smtp.host", SMTP_HOST_NAME);
        props.put("mail.smtp.auth", "true");
        Authenticator auth = new SMTPAuthenticator();
        Session mailSession = Session.getDefaultInstance(props, auth);
        // commenter pour éviter la trace
          mailSession.setDebug(true);
        Transport transport = mailSession.getTransport();
        MimeMessage message = new MimeMessage(mailSession);
        message.setContent("Ceci est un test", "text/html");
        message.setSubject("Essai avec pièce attachée");
        message.setFrom(new InternetAddress(MAIL_ADDRESS));
        message.addRecipient(Message.RecipientType.TO,
             new InternetAddress(MAIL_ADDRESS));
        transport.connect();
        transport.sendMessage(message,
            message.getRecipients(Message.RecipientType.TO));
        transport.close();
```

La plupart des serveurs SMTP modernes exigent une identification pour éviter les pourriels. Ici, l'identification est la même que celle du compte de courriel[1].

```
  private class SMTPAuthenticator extends javax.mail.Authenticator {
        public PasswordAuthentication getPasswordAuthentication() {
           String username = SMTP_AUTH_USER;
           String password = SMTP_AUTH_PWD;
           return new PasswordAuthentication(username, password);
        }
```

11.1.5 Le code de réception des messages

La réception des messages utilise ici un serveur IMAP. Pour les tests, l'intérêt est que IMAP va laisser intacte votre boîte de réception, en laissant les messages sur le serveur. Le port standard de IMAP est le 143, IMAP avec SSL est le 993.

```
      private static String IMAP_HOST= "imap.orange.fr";
      private static int IMAP_PORT = 143; // sans SSL
      private static String IMAP_USER = "moi@orange.fr";
      private static String IMAP_PASSWORD = "*****";

     // Propriétés du serveur
     Properties properties = new Properties();
     properties.put("mail.imap.host", IMAP_HOST);
     properties.put("mail.imap.port", IMAP_PORT);

   //  properties.put("mail.imap.ssl.enable", "true");

     // Identification
```

1. http://www.oracle.com/technetwork/java/javamail-1-149769.pdf

Copyright © 2015 Dunod.
© Dunod – Toute reproduction non autorisée est un délit.

```
Session emailSession = Session.getInstance(properties,
   new javax.mail.Authenticator() {
      protected PasswordAuthentication getPasswordAuthentication() {
         return new PasswordAuthentication(
                  IMAP_USER, IMAP_PASSWORD);
      }
   });
// enlever le commentaire pour debug
// emailSession.setDebug(true);

// création de la boîte IMAP
Store store = emailSession.getStore("imap");
store.connect();

// création du dossier de réception
Folder emailFolder = store.getFolder("INBOX");
emailFolder.open(Folder.READ_ONLY);

// récupère les messages dans la boîte de réception
// et affiche les en-têtes
Message[] messages = emailFolder.getMessages();
System.out.println("messages.length—" + messages.length);

for (int i = 0, n = messages.length; i < n; i++) {
   Message message = messages[i];
   System.out.println("- - - - - - - - - - - - - - - - - - - -");
   System.out.println("Message numéro " + (i + 1));
   System.out.println("Objet: " + message.getSubject());
   System.out.println("De: " + message.getFrom()[0]);
   System.out.println("Texte: " + message.getContent().toString());
}

// Fermeture du compte
emailFolder.close(false);
store.close();
```

Pour IMAP est également possible d'utiliser les connexions sécurisées en utilisant la propriété :

```
mail.imap.ssl.enable
```

Pour un compte POP3, il suffirait de modifier le type du store :

```
emailSession.getStore("pop3");
```

ainsi que les paramètres du serveur. Le port de connexion de POP est le 110, le port avec SSL le 995.

11.1.6 Pièces attachées

Il est possible d'envoyer des pièces attachées en utilisant la classe `MimeMultiPart`.

Copyright © 2015 Dunod.

MIME signifie *Multipurpose Internet Mail Extensions*. C'est un standard Internet qui élargit le type de données que l'on peut envoyer par courriel pour supporter les textes en ASCII ou HTML, les images, les fichiers vidéos. Les formats acceptés sont définis dans les RFC 4288 et 4289.

Voici un exemple pour envoyer un fichier texte et un fichier image. Il ne faut pas oublier de mettre tout de même un texte dans le contenu du message, sinon une erreur est levée.

```
MimeMessage message = new MimeMessage(mailSession);
        message.setContent("Ceci est un test", "text/plain");
        message.setSubject("Essai avec pièces attachées");
        message.setFrom(new InternetAddress(MAIL_ADDRESS));
        message.addRecipient(Message.RecipientType.TO,
             new InternetAddress(MAIL_ADDRESS));
     // création d'une pièce de message
              MimeBodyPart messageBodyPart = new MimeBodyPart();
              messageBodyPart.setContent(message, "text/html");
              messageBodyPart.setText("Contenu du message");

              // création d'un multipart
              Multipart multipart = new MimeMultipart();
              multipart.addBodyPart(messageBodyPart);
              // pièces attachées
              String[] attachFiles = new String[2];
              String repertoireCourant = ".";
              attachFiles[0] = repertoireCourant+"photo.png";
              attachFiles[1] = repertoireCourant+"texte.txt";

              // ajout des pièces attachées
              if (attachFiles != null && attachFiles.length > 0) {
                     for (String filePath : attachFiles) {
                          MimeBodyPart attachPart = new MimeBodyPart();

                              try {
              // System.out.println(new File(filePath).exists());
                                  attachPart.attachFile(filePath);
                              } catch (IOException ex) {
                                     ex.printStackTrace();
                              }

                              multipart.addBodyPart(attachPart);
                         }
              }

             // met le multipart dans le contenu du message
              message.setContent(multipart);
```

11.1.7 Autres fonctionnalités

Il est possible avec JavaMail de gérer la complexité des courriels qui provient des éléments suivants :

© Dunod – Toute reproduction non autorisée est un délit.
Copyright © 2015 Dunod.

- Les différents ports de connexion selon que l'on est identifié ou non et chiffré ou non (utilisation de SSL).
- Les pièces attachées et leurs types multiples.
- Les mécanismes d'identification (identification du poste, identification pas SASL, identifiant de messagerie, etc.).
- La suppression des mails et leur synchronisation entre plusieurs postes.
- La gestion des quotas et des dossiers.
- Le workflow des mails : réponse aux destinations, faire suivre.

Nous laisserons le lecteur explorer ces possibilités par lui-même.

11.2 XMPP

11.2.1 Une norme

XMPP (*Extensible Messaging and Presence Protocol*) est un protocole de communication pour les logiciels de messagerie fondé sur XML. Les différentes spécifications du protocole peuvent être trouvées sur http://www.xmpp.org.



XMPP est constitué d'un protocole TCP/IP basé sur architecture client-serveur permettant les échanges décentralisés de messages instantanés ou non entre clients.

Le protocole s'appelait à l'origine Jabber, et a été développé en open source à partir de 1999 pour la messagerie instantanée (IM, *Instant Messaging*), la gestion de la présence en ligne et la gestion des contacts. Conçu pour être extensible, le protocole a été utilisé notamment pour les systèmes de messagerie instantanée et de clavardage, pour la voix sur IP, pour les réseaux sociaux, les outils de collaboration, le partage de fichiers, les jeux en ligne...

XMPP est un standard ouvert, qui a été repris par l'IETF (*Internet Engineering Task Force*) dans plusieurs spécifications (RFC 3920, 3921, 3922, 3923) approuvées comme normes en 2004. D'autres spécifications ont suivi. En plus de l'IETF, la XMPP Standards Foundation continue à développer des extensions.

Google a utilisé XMPP pour Google Talk en 2005 mais l'a abandonné en 2014. Cisco a racheté Jabber en 2008. En 2010, Facebook a ouvert ses fonctions de clavardage aux développeurs tiers *via* XMPP. Il n'implémente pas exactement un serveur XMPP mais fournit seulement une interface, et la gestion de la liste de présence (*roster*) n'est pas implémentée.

En 2011, Microsoft a fourni une interface XMPP à Microsoft Messenger. Skype, son successeur fournit partiellement une interface XMPP.

Copyright © 2015 Dunod.

Forces et faiblesses

XMPP est similaire à JMS (voir chapitre 8) et peut être classifié comme un MOM (*Message Oriented Middleware*). Les clients communiquent au travers d'un serveur en s'échangeant des messages. Un type de message joue un rôle particulier, le message de présence. Ce message permet de garder à jour la liste de tous les participants présents en temps réel (*roster* en anglais).

C'est un mécanisme décentralisé comme le mail, aucun serveur n'a de rôle central. Sa normalisation est un aspect important.

Tout comme JMS et le courriel, il ne garantit pas la livraison des messages.

Tout comme SOAP, XMPP utilise XML, qui nécessite un codage sur des caractères étendus, et qui de plus est verbeux. Tout cela fait un protocole lourd à transférer. Une optimisation a été tentée avec EXI, *XML Efficient Format*. Remarquons au passage que l'on retrouvera de telles tentatives pour les web services, et le fort développement de JSON (*JavaScript Object Notation* voir chapitre 2) plus compact.

Les données binaires doivent être codées sur 64 bits, ce qui augmente le volume de données échangées.

Les réseaux de serveurs peuvent être privés ou publics, comme Jabber.

Adressage

Pour l'adressage, XMPP utilise un identificateur unique au réseau de serveurs connectés, le *Jabber Id* ou JID. Ce JID comporte en suffixe la ressource de l'utilisateur connecté, par exemple *username@example.com/mobile* pour l'utilisation depuis un mobile. Chaque resource peut avoir une priorité numérique. Un message adressé à *username@example.com* sera envoyé à la ressource de l'utilisateur avec la plus haute priorité, un message adressé à *username@example.com/mobile* ne parviendra qu'à son mobile.

XMPP implémente un mécanisme de découverte utilisant les messages de présence, qui permet aux utilisateurs de rentrer et sortir du réseau de manière dynamique. Ceci est particulièrement adapté aux réseaux ouverts, et peut être comparé au mécanisme USB qui permet de découvrir dynamiquement les appareils connectés. Ce mécanisme de présence peut couvrir plusieurs domaines de réseau et traverse les pare-feux.

Transports

De nombreux serveurs Jabber offrent un service de transports (également appelés passerelles) permettant à l'utilisateur de dialoguer avec les utilisateurs d'autres systèmes de messagerie instantanée.

L'utilisateur peut s'enregistrer à ces transports en fournissant ses informations spécifiques au réseau cible (par exemple, pour ICQ, il faut fournir l'UIN, *User Identification Name*, et le mot de passe). Une fois cette opération effectuée, l'utilisateur peut ajouter des contacts utilisant cet autre réseau comme il le ferait pour un contact utilisant Jabber. À l'heure actuelle, des transports sont disponibles pour les protocoles les plus utilisés, comme IRC, MSN Messenger, Yahoo! Messenger et ICQ/AOL Instant

Copyright © 2015 Dunod.
© Dunod – Toute reproduction non autorisée est un délit.

Messenger (AIM). Il existe également d'autres transports comme par exemple les services de courrier électronique.

En utilisant un transport, l'utilisateur se limite aux capacités du client et du transport utilisés ; il n'aura probablement pas toutes les fonctions du client officiel du réseau cible.

11.2.2 OpenFire

Nous allons expérimenter avec une mise en œuvre de XMPP en Java et pour cela nous choisirons OpenFire, un serveur de messagerie instantané en open source. Il utilise un format de sérialisation XMPP. Il dispose également de greffons permettant des fonctionnalités supplémentaires, mais nous nous limiterons à l'échange de messages.

Téléchargement et installation

Openfire peut être téléchargé à l'adresse suivante :
http://www.igniterealtime.org/projects/openfire/

Sous Windows on peut alors lancer l'exécutable et obtenir la fenêtre suivante :



Figure 11.1

Copyright © 2015 Dunod.



Figure 11.2

On doit alors lancer la configuration à l'aide de **Launch Admin**.

Une fenêtre de navigateur s'ouvre pour compléter le processus d'installation. On choisira par simplicité ici une base de données embarquée et des paramètres de profil par défaut.



Figure 11.3

© Dunod – Toute reproduction non autorisée est un délit.
Copyright © 2015 Dunod.

On peut installer des plug in d'Openfire en copiant simplement le fichier .jar dans le répertoire plug in. Voici un exemple de la liste des plug in : http://www.igniterealtime.org/projects/openfire/plugins.jsp

On remarquera des plug in de service de présence, des passerelles vers d'autres messageries, des plug in pour créer des réunions, pour téléphoner sur Internet,etc.

On s'intéressera pour la suite uniquement au plug in Smack, qui permet de faire de la messagerie instantanée. La documentation se trouve ici : http://www.igniterealtime.org/builds/smack/docs/latest/documentation/

On rajoute le plug in de gestion de présence des utilisateurs. Et l'on rajoute également les utilisateurs dans la console d'administration, ici Annick et Bob.

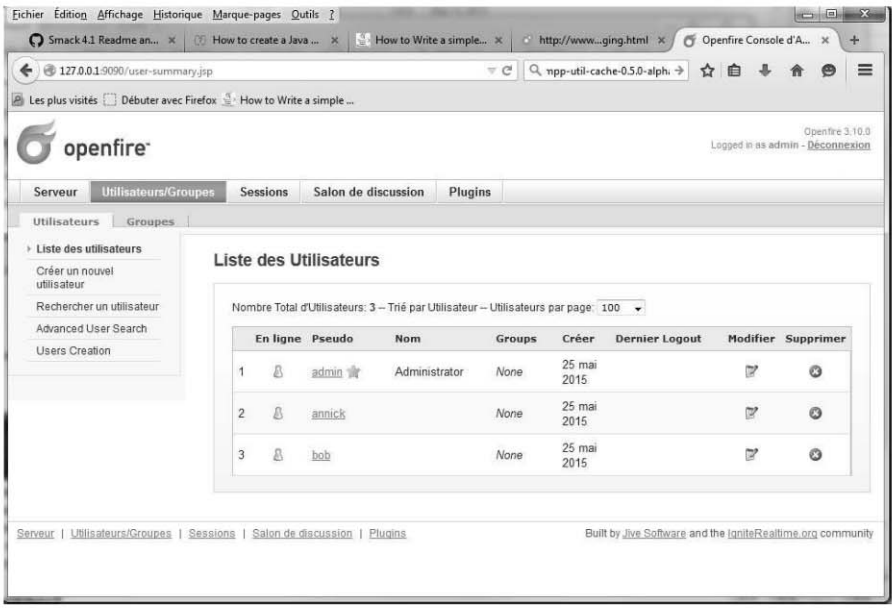


Figure 11.4

On commence par télécharger JXMPP : http://jxmpp.sourceforge.net/.

On écrit alors le programme suivant.

```
import java.util.*;
import java.io.*;

import org.jivesoftware.smack.Chat;
import org.jivesoftware.smack.ConnectionConfiguration;
import org.jivesoftware.smack.MessageListener;
import org.jivesoftware.smack.Roster;
import org.jivesoftware.smack.RosterEntry;
import org.jivesoftware.smack.XMPPConnection;
import org.jivesoftware.smack.XMPPException;
import org.jivesoftware.smack.packet.Message;

public class JabberSmackAPI implements MessageListener{

    XMPPConnection connection;
```

Copyright © 2015 Dunod.

```
    public void login(String userName, String password) throws XMPPException
    {
    ConnectionConfiguration config = new
ConnectionConfiguration("127.0.0.1",5222, "Conference");
    connection = new XMPPConnection(config);

    connection.connect();
    connection.login(userName, password);
    }

    public void sendMessage(String message, String to) throws XMPPException
    {
    Chat chat = connection.getChatManager().createChat(to, this);
    chat.sendMessage(message);
    }

    public void displayBuddyList()
    {
    Roster roster = connection.getRoster();
    Collection<RosterEntry> entries = roster.getEntries();

    System.out.println("\n\n Nombre de personnes en ligne :" +
entries.size());
    for(RosterEntry r:entries)
    {
    System.out.println(r.getUser());
    }
    }

    public void disconnect()
    {
    connection.disconnect();
    }

    public void processMessage(Chat chat, Message message)
    {
    if(message.getType() == Message.Type.chat)
    System.out.println(chat.getParticipant() + " répond: " +
message.getBody());
    }

    public static void main(String args[]) throws XMPPException, IOException
    {
    // declare variables
    JabberSmackAPI c = new JabberSmackAPI();
    BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
    String msg;


    // turn on the enhanced debugger
    XMPPConnection.DEBUG_ENABLED = true;


    // Enter your login information here
    c.login("annick", "annick");
```

Copyright © 2015 Dunod.
© Dunod – Toute reproduction non autorisée est un délit.

```
        c.displayBuddyList();

        System.out.println("- - - - -");

        System.out.println("A qui voulez-vous parler:");
        String talkTo = br.readLine();

        System.out.println("- - - - -");
        System.out.println("Tous les messages seront envoyés à " + talkTo);
        System.out.println("Taper votre message dans la console:");
        System.out.println("- - - - -\n");

        while( !(msg=br.readLine()).equals("bye"))
        {
            c.sendMessage(msg, talkTo);
        }

        c.disconnect();
        System.exit(0);
        }

}
```

On lance alors le programme une fois avec le login du premier utilisateur (ici `annick` et mot de passe `annick`), une fois avec le login du deuxième utilisateur. On obtient le message sur le flux d'entrée/sortie :

```
Nombre de personnes en ligne :0
- - - - -
A qui voulez-vous parler:
annick@port-fron/Smack
- - - - -
Tous les messages seront envoyés à annick@port-fron/Smack
Taper votre message dans la console:
- - - - -
Bonjour c'est moi Bob
```

Le nom d'utilisateur est suffixé par le nom de l'ordinateur (ici port-fron) suivi de Smack. On vérifie que le message a bien été reçu dans le debugger de Bob et d'Annick.

On remarquera les messages de présence, qui sont la particularité de XMPP dans la fenêtre de debug qui apparaît telle que ci-après.

Copyright © 2015 Dunod.

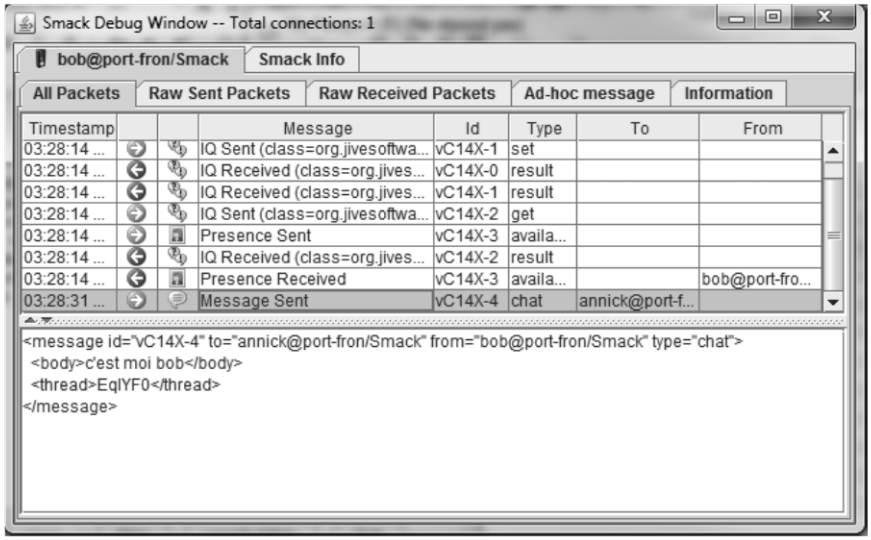


Figure 11.5

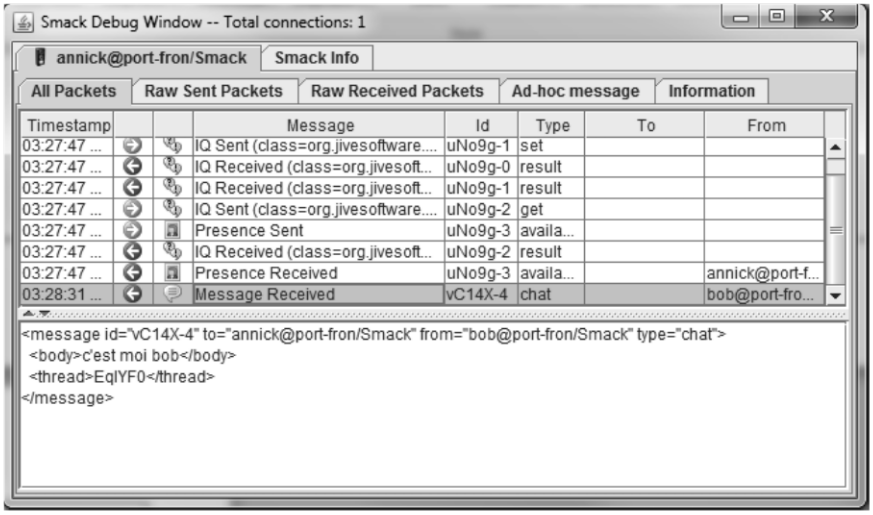


Figure 11.6

On vérifie la présence des sessions utilisateurs dans l'interface d'administration.

Copyright © 2015 Dunod.
© Dunod – Toute reproduction non autorisée est un délit.

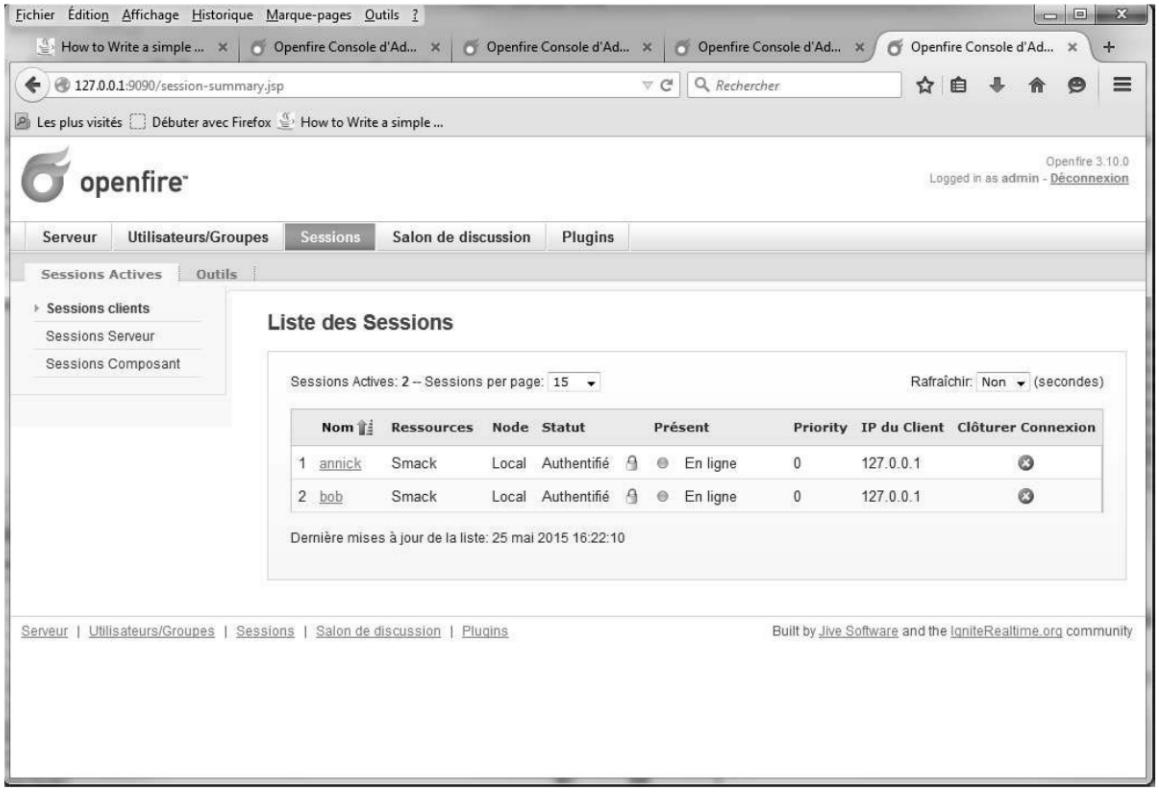


Figure 11.7

En résumé

Il est possible de faire des expérimentations de messagerie instantanée avec des outils open source utilisant XMPP. Toutefois, les fonctionnalités sont réduites par rapport aux plates-formes couramment utilisées. Les messages de présence qui permettent la découverte des clients présents sont adaptés à des architectures ouvertes et de topologie dynamique.

Copyright © 2015 Dunod.

Bibliographie

Ouvrages généraux

[Bloch2002] Joshua BLOCH, *Java efficace*, Vuibert, Paris, 2002.

[Gof4] Erich GAMMA, Richard HELM, Ralph JOHNSON, John VLISSIDES, *Design Patterns : Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1994.

[Lea2003] Doug LEA, *Concurrent Programming in Java*, 2e édition, Addison-Wesley, The Java Series, 2003.

[Meyer1991] Bertrand MEYER, *Conception et programmation par objets*, 2e édition, Intereditions, Paris, 1991.

[General1] David REILLY, Michael REILLY, *Java Network Programming and Distributed Computing*, Addison Wesley Professional, 2002.

[General2] Jim FARLEY, *Java Distributed Computing*, O'Reilly, 1998.

[General3] Jan GRABA, *An Introduction to Network Programming with Java*, Springer, 2006.

[General4] Jim FARLEY, William CRAWFORD, Prakash MALANI, John G. NORMAN, Justin GETHLAND, *Java Enterprise in a Nutshell*, O'Reilly, 2005.

Services web

[Web1] Steve GRAHAM, Doug DAVIS, Simeon SIMEONOV, Glen DANIELS, Peter BRITTENHAM, Yuichi NAKAMURA, Paul FREMANTLE, Dieter KÖNIG, Claudia ZENTNER, *Building Web Services with Java: Making Sense of XML, SOAP, WSDL, and UDDI*, 2e édition, Sams, 2004.

[Web2] Dave CHAPPELL, Tyler JEWELL, *Java Web Services*, O'Reilly, 2002.

[Web3] Kim TOPLEY, *Java Web Services in Nutshell*, O'Reilly, 2003.

[Web4] Mark D. HANSEN, *SOA Using Java™ Web Services*, Prentice Hall, 2007.

Copyright © 2015 Dunod.

© Dunod – Toute reproduction non autorisée est un délit.

[Web5] Robert BRUNNER, Frank Cohen, Francisco CURBERA, Darren GOVONI, Steven HAINES, Matthias KLOPPMANN, Benoît MARCHAL, K. Scott MORRISON, Arthur RYMAN, Joseph WEBER, Mark WUTKA, *Java*tm *Web Services Unleashed*, Sams, 2002.

[Web6] Eric ARMSTRONG, Stephanie BODOFF, Debbie CARSON, Maydene FISHER, Dale GREEN, Kim HAASE, *Java*tm *Web Services Tutorial*, Prentice Hall, 2002.

[Web7] Martin KALIN, *Java Web Services : Up and Running*, 2e edition, O'Reilly, 2013.

[REST1] Leonard RICHARDSON, Mike AMUNDSEN, Sam RUBY, *RESTful Web APIs*, O'Reilly, 2013.

[REST2] Bill BURKE, *RESTful Java with JAX-RS 2.0*, O'Reilly, 2013.

CORBA

[Daniel2002] Jérôme DANIEL, *Au Cœur de CORBA*, Vuibert, Paris, 2002.

[CORBA1] Michi HENNING, Steve VINOSKI, *Advanced CORBA® Programming with* C++, Addison Wesley Professional, 1999.

[CORBA2] Jon SIEGEL, *CORBA 3 : Fundamentals and Programming*, 2e édition, John Wiley & Sons, 1999.

[CORBA3] Gerald BROSE, Andreas VOGEL, Keith DUDDY, *JavaTM Programming with CORBATM : Advanced Techniques for Building Distributed Applications*, 3e édition, Wiley, 2001.

[CORBA4] Markus ALEKSY, Axel KORTHAUS, Martin SCHADER, *Implementing Distributed Systems with Java and CORBA*, Springer, 2010.

JNDI LDAP

[JNDI1] Marcel RIZCALLAH, *Annuaires LDAP*, Eyrolles, 2004.

[JNDI2] Rosanna LEE, *Jndi Api Tutorial And Reference*, Addison Wesley, 2000.

[JNDI3] Pierre-Yves CLOUX et Rafael CORVALAN, *Les annuaires LDAP*, Dunod, 2005.

Copyright © 2015 Dunod.

RMI

[RMI1] Esmond PITT, Kathleen MCNIFF, *java.rmi : The Remote Method Invocation Guide*, Addison Wesley Professional, 2001.

[RMI2] William GROSSO, *Java RMI*, O'Reilly, 2001.

NIO, Sockets, Réseau, Concurrence

[NIO1] Brian GOETZ, Tim PEIERLS, Joshua Bloch, Joseph BOWBEER, David HOLMES, Doug LEA, *Concurrency and NIO*, Addison Wesley Professional, 2006.

[NIO2] Elliotte Rusty HAROLD, *Java Network Programming*, 4e édition, O'Reilly Media, 2013.

[NIO3] Ron HITCHENS, *Java NIO*, O'Reilly, 2002.

[NIO4] Brian GOETZ, Tim PEIERLS, Joshua BLOCH, Joseph BOWBEER, David HOLMES, Doug LEA, *Java Concurrency in Practice*, Addison-Wesley Professional, 2006.

[NIO5] Kenneth L. CALVERT, Michael J. DONAHOO, *TCP/IP Sockets in Java, 2nd Edition: Practical Guide for Programmers (The Practical Guides)*, Morgan Kaufmann, 2008.

JMS

[JMS1] Shaun TERRY, *Entreprise JMS Programming*, Wiley, 2002.

[JMS2] Richard MONSON-HAEFEL, David A. CHAPPELL, *Java Message Service*, O'Reilly, 2001.

[JMS3] Eric BRUNO, David A. CHAPPELL, *Java Messaging*, Charles River Media, 2005.

[JMS4] Kareem YUSUF, *Enterprise Messaging Using JMS and IBM® WebSphere®*, IBM Press, 2004.

[JMS5] Mark HAPNER, Rich BURRIDGE, Rahul SHARMA, Joseph FIALLI, Kim HAASE, *Java Message Service API Tutorial and Reference: Messaging for the J2EE Platform*, Prentice Hall, 2002.

[JMS6] Scott GRANT, Michael P. KOVACS, Meeraj KUNNUMPURATH, Silvano MAFFEIS, K. SCOTT MORRISON, Gopalan SURESH RAJ, Paul GIOTTA, James MCGOVERN, *Professional JMS*, Wrox, 2001.

[JMS7] Mark RICHARDS, *Java Message Service*, O'Reilly, 2009.

Copyright © 2015 Dunod.

© Dunod – Toute reproduction non autorisée est un délit.

Messagerie instantanée et courriel

[XMPP1] Peter SAINT-ANDRE, Kevin SMITH, Remko TRONCON, *XMPP : The Definitive Guide : Building Real-Time Applications with Jabber Technologies*, O'Reilly, 2009.

[XMPP2] Peter DAUGHERTY, *XMPP 253 Success Secrets - 253 Most Asked Questions On XMPP - What You Need To Know*, Emereo Publishing, 2014.

[Mail1] Elliotte RUSTY HAROLD, *JavaMail API*, O'Reilly, 2013.

JMX et logs

[LOG1] Samudra GUPTA, *XMPP : Pro Apache Log4j*, 2e édition, Apress, 2005.

[JMX1] Heather KREGER, Ward HAROLD, Leight WILLIAMSON, *Java¿ and JMX: Building Manageable Systems*, Addison-Wesley Professional, 2003.

Copyright © 2015 Dunod.

Index

Symboles

.Net 214
    Remoting 203

A

abonnement durable 187
abonnés durables 174, 188, 190
accusé de réception 173, 174, 182
Active Directory 157
ADB 218
administration 8
agent JMX 262
annotations 226
annuaire
    CORBA 111
    d'applications 162
    d'entreprise 161
    de matériel 162
    de personnes 162
    interopérable 135
    LDAP 163
    `rmiregistry` 157, 159
Ant 222, 228, 241
appel
    asynchrone 220
    dynamique 133
`Appender` 273
applications réparties 1
architecture répartie 2
`ASN1` 184
asynchrone 173
    communication 178
    consommateurs 180
    mode 6
    programmation 30
    socket 71
attache SOAP (*binding*) 207
autoboxing 51

B

B to B 1, 201
balise XML 202
Basic Security Profile 204
battement de cœur 175
binding 207
buffer, tampon 69
bus logiciel 114, 117

C

CCM 88
`Channel`, canal 68, 70
clé de sélection 72
commande Java
    `idlj` 128, 129
    `jconsole` 266
    `orbd` 111, 117, 126

Copyright © 2015 Dunod.
© Dunod – Toute reproduction non autorisée est un délit.

rmic 96
rmid 143
rmiregistry 150, 160, 163
wsimport 221
connecteur JMX 263, 268
ConnectionFactory 47, 167, 180
connexion JMS 174
conteneur 119
contrat 80
CORBA 113, 203, 245
annuaire 111
architecture 116
BOA 119
références objets 124
services 125
souche 118
Tie 132, 133
types primitifs 122
corbaloc 137, 205
corbaname 137
couplage 4
courriel 277
courtier 151
CRUD 246
cycle de vie d'une référence 148

D

DataSource 162, 165
DDS 144
debug 8
découverte de services 271
déploiement 8
design pattern 45
délégation 132, 260
observateur 179
design patternfactory 157, 176, 262
Destination 171
DGC 109, 148
DN 163
DNS 157, 161, 162
document/literal 212
DynamicMBean 264

E

Eclipse 222, 230
EDI 207
EJB 166, 245
encodage des caractères 69
encoded 212
endPoint 237
endpoint 208
exceptions 102, 123, 175
RMI 103
Executors 29
EXI 283
Externalizable 15

F

factory 29, 46, 181, 235
fichier de propriétés 158
filtrage 185
filtrer 185
fork/join 33
futur 30, 31

G

garbage collector 148
grain 4

H

heartbeat 175
Helper 124
Holder 124
HTTP 202, 262
connecteur 267

I

idempotence 246
IDL 81, 116, 122, 202, 208
idlj 126
IIOP 110, 116, 262
IMAP 277

Copyright © 2015 Dunod.

implémentation
de JMS 169
de JNDI 157
intégration d'applications 174
interblocage 178
interface
exceptions 82
réseau 56
syntaxe 81
intergiciel, *middleware* 2
Internet des objets 1
invocation dynamique 118
IOR 124
IPv6 56

J

Jabber Id 283
Java
JMS 169
JMX 259
NIO 55, 68
types primitifs 122
JavaMail 278
JAX-WS 221
JAXB 216, 228, 229, 237
JCA 175
JID 283
JIXB 216
JMS 162, 167, 169, 170
consommateur 170
producteur 170
JMS 2.0 194
JMX 51, 259
adaptateur 268
connecteur 263, 268
JNDI 111, 156, 176, 181
journalisation 271
JRMP 110
JSR 181 204
JTA 126
JXMPP 286

L

lambdas 35
`Layout` 273
LDAP 161
fédération d'annuaires 164
literal 212
`localhost` 55
`Log4j` 272
`Logger` 272

M

Management Bean, *MBean* 260
marshalling 80, 83, 151, 183
`MBeanServer` 262
MDA 113
message
acquittement, accusé de réception 190
en-tête 187
filtrage 185
SOAP 206, 209
messagerie instantanée 277
MIME 281
`MimeMultiPart` 280
mode
pull 263
push 263
modèle
de document 215
objet 215
modules 123
MOM 6, 169
multicast 67

N

Naming Service 125, 134, 148, 166
NIO 68
niveau de journalisation 275
nommage 85, 265
notifications 269

© Dunod – Toute reproduction non autorisée est un délit.
Copyright © 2015 Dunod.

O

objets distants 93
OMG 113
`oneway` 142
oneway 210
Open Source 115, 169, 272
OpenFire 284
opération SOAP 209
ORB 116, 117
`orbd` 111, 117, 126

P

pages
    blanches 148, 156
    jaunes 148, 156, 168
parallélisme 2
pare-feu 203
passage d'arguments
    par référence 84, 137
    par valeur 84, 140
passerelles 283
pattern`Factory` 153
patternproxy 155
POA 118
POAManager 120
point
    à point 173
    d'entrée 149
POJO 220, 239
polling 71, 179
pool de *threads* 63
POP 277
port de communication 57
processus de développement 8
Protobuffs 21
Protocol Buffer Compiler 21
Protocol Buffers 21
proxy 47
    dynamique 49, 155
pseudo-URL 168, 205
    DNS 161
    IIOP 160
    LDAP 165
    LDAP-search 168
    `rmi` 159
publication/abonnement 173, 188, 269

Q

`Qname` 243
qualité de service 203
    en JMS 175
`Queue` 171, 177
`QueueBrowser` 192
`QueueRequestor` 192

R

RDN 163
références distantes 148
referral 164
réflexion 50, 151
répartition 2
REST 246
RHTTP 207
RMI 91, 203, 245, 262, 265
    IIOP 160
`rmic` 96
`rmiregistry` 97, 164, 166, 267
router 185
RPC 4, 91, 114, 170, 174, 175, 189, 202
`rpc/encoded` 212
`Runnable` 27

S

sécurité 154
sélecteur 71
sérialisation, `Serializable` 11, 185
`serialver` 15
servant 94, 118, 149, 150, 240
`servertool` 143

Copyright © 2015 Dunod.