Chapitre 5

La plate-forme J2EE

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

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

5.1 Introduction

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

5.1.1 Historique

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

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

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

5.1.2 J2EE, pour quoi faire ?

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

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

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

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

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

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

5.1.3 Principes d'architecture

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

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

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

L'étage client

L'étage client est représenté par le terminal et prend en charge l'interaction avec l'utilisateur. Du point de cette interaction, deux architectures sont possibles :
- Architecture client léger : dans ce cas, le terminal embarque un mécanisme qui permet d'interpréter les informations de présentation et d'interaction avec l'utilisateur produite par le serveur. Cela peut être un navigateur qui interprète des pages XML (XHTML, WML, ou VoiceXML suivant la modalité utilisée) produite par le serveur

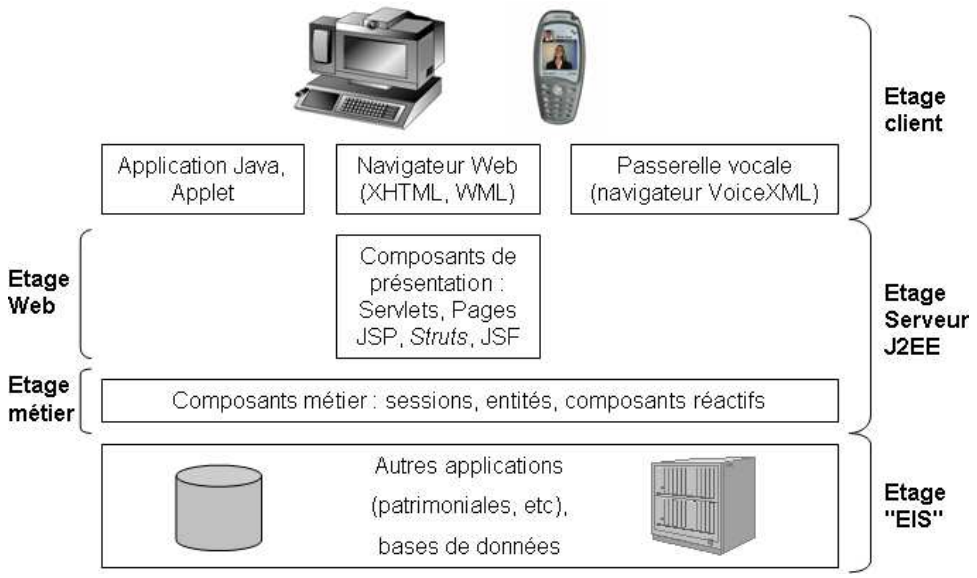

Figure 5.1 – Architecture multiétages dans l'environnement J2EE

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

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

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

L'étage serveur

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

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

L'étage information

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

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

5.1.4 Modèles de programmation et conteneurs

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

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

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

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

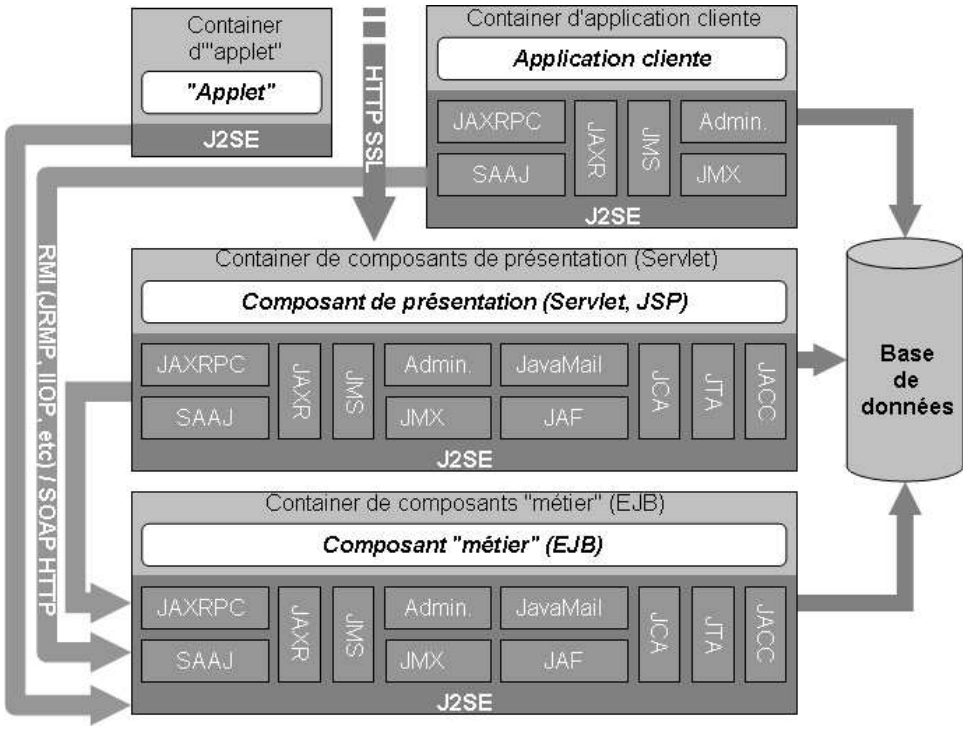

Figure 5.2 – Conteneurs J2EE

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

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

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

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

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

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

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

L'environnement J2EE prend en charge une grande partie du cycle de vie d'une application. Il propose des solutions pour le développement de l'application, pour son assemblage puisqu'il se base sur la notion de composants, pour son déploiement, et enfin pour son exécution et son administration. A un premier niveau, nous distinguons deux types d'acteurs : les fournisseurs de technologies et leurs utilisateurs. Voici une liste des fournisseurs ainsi que leur rôle :
- Le fournisseur de la plate-forme serveur : il fournit la plate-forme logicielle implantant les différents services proposés par l'environnement J2EE. Cette plate-forme implante aussi les mécanismes permettant de déployer et d'exécuter les composants

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

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

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

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

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

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

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

5.1.6 Références utiles

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

5.2 Approche à composants

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

5.2.1 Modèle de composants

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

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

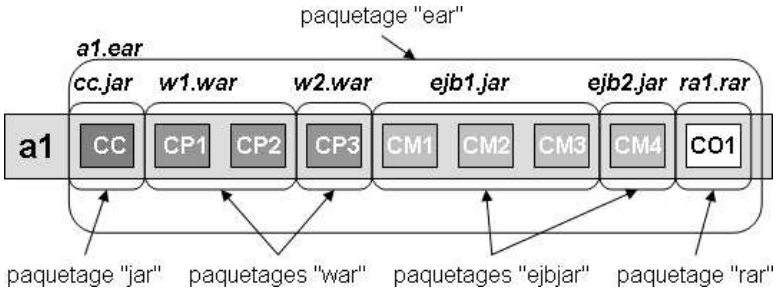

Figure 5.3 – Empaquetage de composants J2EE

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

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

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

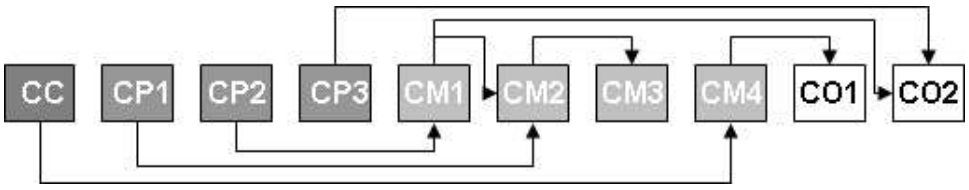

Figure 5.4 – Assemblage de composants J2EE

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

5.2.2 Empaquetage de composants et d'applications

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

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

Application : `application.xml .ear`
Application cliente : `client.xml .jar`
Module Web : `web.xml .war`

Module métier : `ejb-jar.xml .jar`
Connecteur : `ra.xml .rar`

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

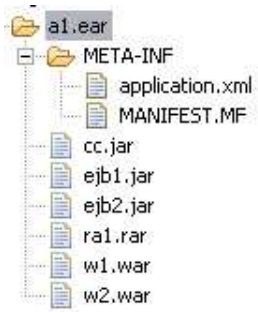

Figure 5.5 – Une archive d'application J2EE

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

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

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

5.2.3 Principes d'assemblage

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

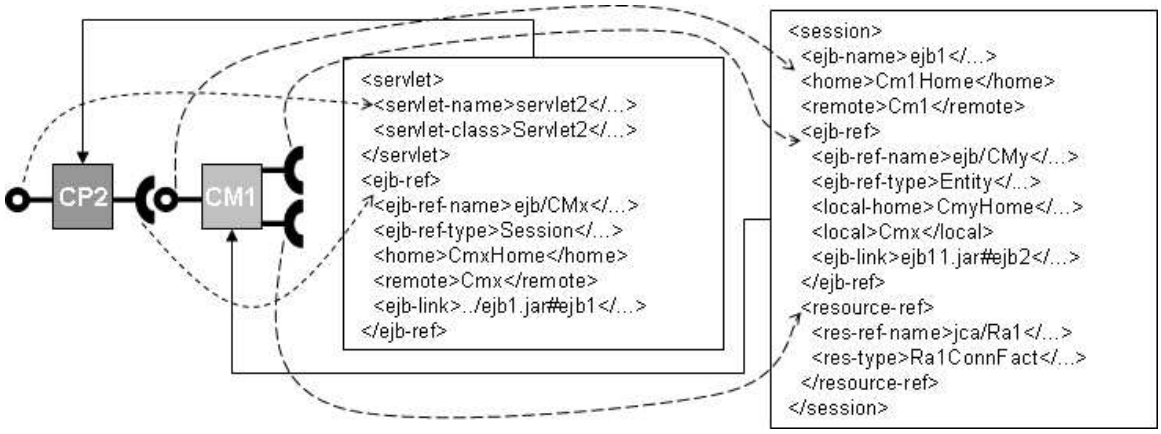

Figure 5.6 – Description d'un assemblage entre deux composants

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

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

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

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

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

- Le nom de la référence : c'est le nom qui va être utilisé dans le code du composant pour récupérer la liaison effective vers le composant référencé. Il est bien souvent préfixé par le type de composant ou de ressource auquel on se lie. La figure 5.6 montre deux types de références : deux références vers des composants métier dont le nom est préfixé par `ejb/` et une référence vers un connecteur préfixée par `jca/`.
- Le type du composant auquel on cherche à se lier : il permet de vérifier la validité des liaisons définies à l'assemblage (cas d'un lien vers un composant métier). Il suffit pour cela de vérifier que le type du composant référencé, par exemple `Cm1Home` de `CM1`, est bien conforme au type du composant attendu par le référençant, ici `CmxHome` dans la référence de `CP2` (c'est à dire `Cm1Home` est `CmxHome` ou bien l'étend (relation d'héritage)).
- Le lien vers le composant utilisé dans le cas d'une référence vers un composant métier : l'élément `ejb-link` utilisé dans ce cas exprime une mise en liaison effective de deux composants. Il utilise le nom identifiant le composant dans le cadre de l'application en question. L'exemple de la figure 5.6 montre le lien du composant `CP2` vers le composant métier `CM1`. Ce lien est défini par `../ejb1.jar/ejb1` de façon à identifier complètement le composant dans le paquetage de l'application à partir du paquetage dans lequel la référence est définie (ici le paquetage `w1.war`).
  Dans le cas des références vers d'autres ressources comme une queue JMS, une source de données JDBC, ou encore une usine à connection JCA, la définition de la liaison effective n'est pas couverte par le standard J2EE. Elle est donc spécifique à chaque produit. Elle est bien souvent définie dans un descripteur supplémentaire propre au produit.

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

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

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

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

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

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

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

5.2.4 Déploiement d'application

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

J2EE au niveau du serveur et au niveau des clients.

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

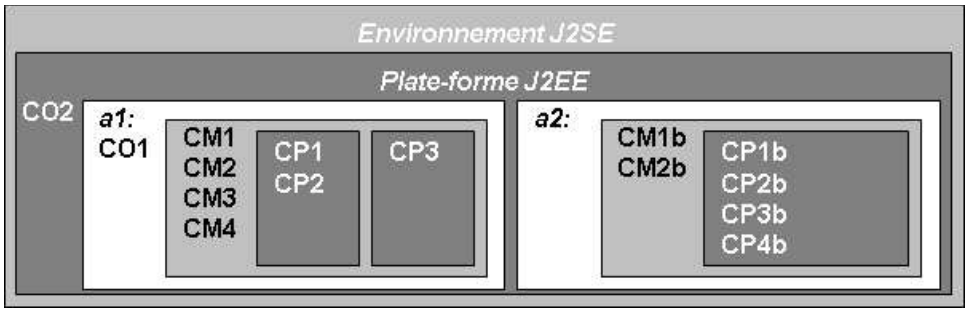

Figure 5.7 – Hiérarchie des chargeurs de classes dans un serveur J2EE

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

Les premiers contiennent le code commun à toutes les applications déployées dans un serveur J2EE. Au niveau de ces chargeurs, on a généralement une hiérarchie distinguant le code de J2SE à la base, puis celui du serveur J2EE avec ses services de base, puis un dernier contenant le code des ressources mises à disposition des applications (dans l'exemple, le chargeur en question contient le code du composant `CO2`).

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

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

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

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

5.3 Applications réparties et intégration

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

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

5.3.1 Appel de procédure à distance

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

Java RMI

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

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

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

Web Services

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

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

5.3.2 Communication asynchrone : JMS

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

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

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

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

5.3.3 Connecteurs au standard JCA

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

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

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

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

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

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

5.3.4 Connecteurs de base

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

Connecteur JDBC d'accès aux bases de données relationnelles

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

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

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

Connecteur d'accès au *mail*

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

5.4 Gestion des transactions

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

5.4.1 Notion de transaction

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

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

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

5.4.2 Manipulation des transactions dans un serveur J2EE

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

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

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

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

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

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

5.4.3 Mise en relation des composants J2EE avec le système transactionnel

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

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

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

5.5 Programmation de l'étage de présentation Web

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

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

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

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

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

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

5.5.1 Principes d'architecture des composants de présentation Web

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

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

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

5.5.2 *Servlets* : organisation de la pile protocolaire

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

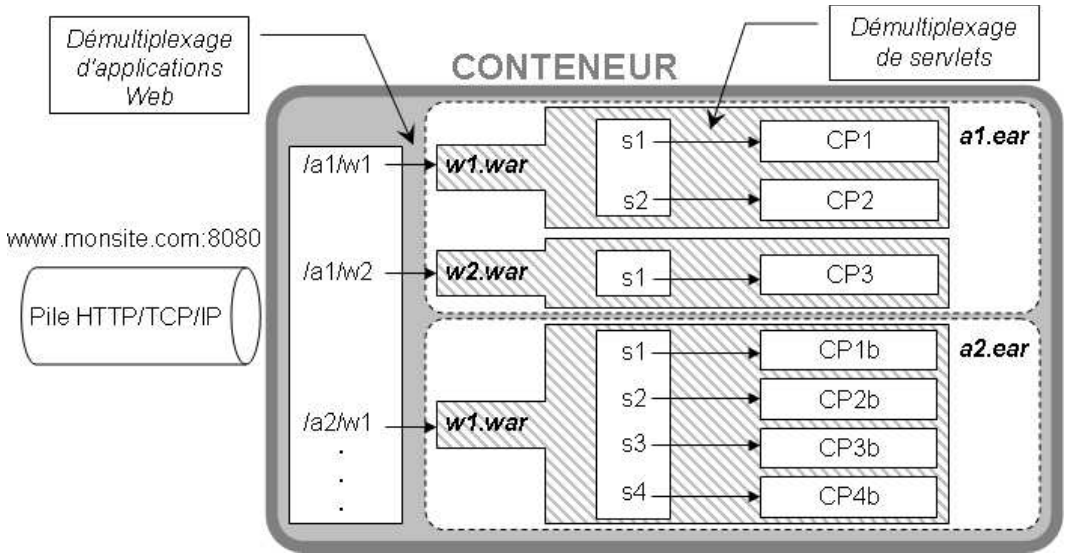

Figure 5.8 – Le démultiplexage des composants de présentation (*servlets*)

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

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

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

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

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

sommes en train de décrire.

- Le contenu de la requête
  Il est représenté par plusieurs informations. La première est la taille du contenu du message. Une autre information donne le type de ce contenu qui est un type MIME, ou qui est nul dans le cas où il n'est pas connu. Enfin, on donne le moyen d'accéder à ce contenu, soit à travers la lecture d'un flux binaire, soit à travers la lecture d'un flux de caractères. Dans le cas d'un contenu de type texte, le type d'encodage de l'information est aussi défini.
- Des attributs associés à la requête
  Au cours des phases de traitement de la requête, il est possible de gérer des attributs permettant de caractériser la requête vis-à-vis des phases suivantes. Cette gestion autorise la création, la modification, la destruction d'attributs, ou encore leur énumération.

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

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

- Le contenu de la réponse
  Il est représenté par plusieurs informations de façon symétrique à une requête. La première est la taille du contenu de la réponse. Une autre information spécifie le type de cette réponse qui est un type MIME. Enfin, on donne le moyen de produire cette réponse, soit à travers l'écriture d'un flux binaire, soit à travers l'écriture d'un flux de caractères. Dans le cas d'un contenu de type texte, le type d'encodage de l'information est aussi défini.
- Le contrôle du flux d'informations renvoyés au client
  Un point important est la possibilité de gérer le flux de données émis vers le client auquel la *servlet* répond. Dans le cas de réponse de grande taille, comme par exemple le renvoi d'une image, cela permet de ne pas saturer la mémoire du serveur pour le seul bénéfice de la requête en question (un serveur peut gérer un nombre important de requêtes simultanées). Pour cela, il est possible de définir la taille du tampon utilisé pour gérer les données de réponse. Les données du tampon sont alors émises vers le client soit lors d'une demande explicite (opération `flush`), soit lorsque ce tampon est plein. Il est important de noter que toutes les méta-informations associées à la réponse (cf. point précédent) doivent avoir été définie avant la première émission ; en effet, l'en-tête de la réponse (contenant notamment la taille de la réponse) est évidemment émis à ce moment là.

Organisation du conteneur et cycle de vie des *servlets*

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

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

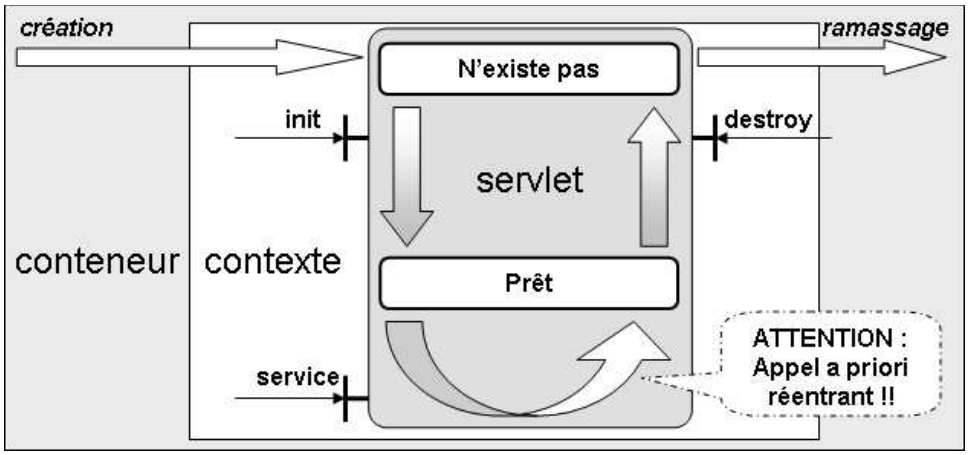

Figure 5.9 – Le cycle de vie des *servlets*

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

– Le contexte de *servlets*
  Un contexte contient un certain nombre de *servlets* qui y ont été déployées auxquelles un nom est associée et qui permet d'identifier celle-ci dans le contexte. Au contexte lui-même est associé un nom qui sert de préfixe lors du processus de démultiplexage. Il offre aussi à ces *servlets* de gérer un ensemble d'attributs qui leur permettent de partager des informations (communication entre *servlets*). Il permet aussi de de manipuler le système de démultiplexage (par exemple, retrouver une *servlet* d'un nom donné ou le contexte auquel elle appartient).
– La *servlet*
  Une fois qu'une *servlet* est prête (initialisation terminée), le conteneur peut l'activer au rythme qu'il juge bon (invocation de `service` en parallèle). Comme au niveau contexte, la *servlet* gère aussi un ensemble d'attributs accessible aux requêtes traitées par `service`.

Filtres de *servlets*

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

Contrôle du démultiplexage

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

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

5.5.3 *Servlets* HTTP

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

Démultiplexage de requêtes HTTP

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

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

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

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

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

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

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

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

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

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

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

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

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

Gestion de session HTTP

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

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

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

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

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

5.5.4 JSP : construction de pages HTML dynamiques

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

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

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

```
<%@ page contentType="text/html; charset=UTF-8" %>
<html>
   <head><title>La date</title></head>
   <body bgcolor="white">
      <jsp:useBean id="date" class="monorg.MaDate"/>
      <jsp:setProperty name="date" property="localisation" value="French"/>
      <b>Nous sommes le : </b>${date.date}
   </body>
</html>
```

Nous observons aussi les échanges avec le monde Java par l'utilisation d'un *bean*. La directive JSP `useBean` crée un *bean* de la classe donnée en paramètre et l'affecte à la variable `date`. La directive suivante affecte la propriété `localisation` du même *bean* (appel de la méthode `setLocalisation` de la classe `MaDate` définie ci-dessous). Enfin, on récupère la chaine de caractères correspondant à la date à afficher par l'instruction `${date.date}`.

```
package monorg;

class MaDate {
   public setLocalisation(String loc) {
      ...
   }
   public String getDate() {
      return ...
   }
}
```

Notons que le code d'une page JSP peut accéder à toutes les informations de l'environnement de *servlet* et a donc accès à toutes les APIs de ce dernier.

Les pages JSP sont déployées comme des *servlets*. En effet, au déploiement d'une page JSP, le conteneur génére la classe *servlet* correspondante, la compile et la charge. Lors de chaque interaction avec cette *servlet*, le conteneur vérifie que la *servlet* est plus jeune que la page JSP associée. Si ce n'est pas le cas, le conteneur met à jour la classe *servlet*, supprime les anciennes instances et en recrée de nouvelles pour traiter les requêtes. Cela offre une grande souplesse notamment en phase de développement.

D'autres mécanismes sont aussi offerts par JSP, et notamment les *tags* et les librairies de *tags* ainsi que le langage d'expression. Cela sort du sujet du présent chapitre dont l'objectif est de présenter les principes des technologies fournies par l'environnement J2EE. Nous ne les présentons donc pas.

5.6 Programmation de l'étage métier

L'étage métier de l'environnement J2EE fournit le moyen de programmer la logique métier d'une application orientée serveur en lui assurant les propriétés généralement nécessaires aux applications d'entreprise, ou plus précisément aux applications dites critiques (en anglais *mission-critical applications*). Il est définit par la spécification EJB (*Entreprise Java Bean*) [Sun JSR-000153 2003], la version 2.1 de la spécification servant de référence pour la description faite dans cette section.

L'environnement EJB fournit l'ensemble des services systèmes nécessaire au support d'un premier niveau de sûreté de fonctionnement (par exemple la persistance ou les transactions), ou de propriétés de sécurité (par exemple l'authentification ou le contrôle d'accès).

Il fournit d'autres mécanismes systèmes permettant d'optimiser la gestion des ressources mises en œuvre pour l'exécution d'une application. L'objectif est ici de pouvoir régler l'allocation de ressources pour optimiser les performances d'une application.

Enfin, il offre des moyens de communication de haut niveau permettant d'intégrer des applications J2EE réparties. Ces moyens permettent d'une part des communications dites *synchrones* (émission d'une requête avec attente de réponse), ou de type appel de procédures à distance (en anglais *Remote Procedure Call (RPC)*). Il peut s'agir ici soit de RMI (*Remote Method Invocation* ou support d'objets répartis pour Java), soit de *Web Services* (appel de procédures distantes basé sur XML). D'autre part, ils permettent des communications dites *asynchrones* (envoi d'une requête sans attente de réponse). C'est généralement le support JMS (*Java Messaging Service*) qui est mis en œuvre à cette fin. Tous ces mécanismes de communication sont décrits plus en détail dans la section 5.3.

L'objectif du modèle de programmation est de rendre la mise en œuvre de ces différents mécanismes aussi transparente que possible pour le programmeur. La description de ce modèle fait l'objet de cette section dans laquelle nous allons étudier les trois éléments principaux de ce modèle : composants de session, composants de données (appelés aussi composants entité), et composants réactifs.

5.6.1 Principes d'architecture des composants métier

Comme pour les composants de présentation (cf. section 5.5.1), les principes d'architecture des composants J2EE se déclinent dans le cadre des composants métier (cf. figure 5.10). On retrouve un conteneur EJB qui prend en charge le déploiement et l'exécution des composants métier. Pour tous les types de composants métier, la gestion du cycle d'allocation et de libération d'instances est sous le contrôle du conteneur. C'est là un point majeur puisque c'est à partir de cet ancrage que le serveur va pouvoir optimiser la gestion des ressources.

Au déploiement, le conteneur prend en charge des paquetages spécifiques qui sont des `.jar` avec une structure et des informations précises (cf. section 5.2.2). Il charge le code de ces composants métier et les rend aptes à être liés à d'autres composants afin qu'ils puissent être activés (appels synchrones ou asychrones). Cette phase de mise en aptitude à la liaison procède de deux actions et met en œuvre le patron d'architecture *export/bind* comme utilisé dans [Dumant et al. 1998] :

- La première action consiste à exporter les références à ces composants déployés sous un nom discriminant dans un espace de noms associé à l'application en cours de

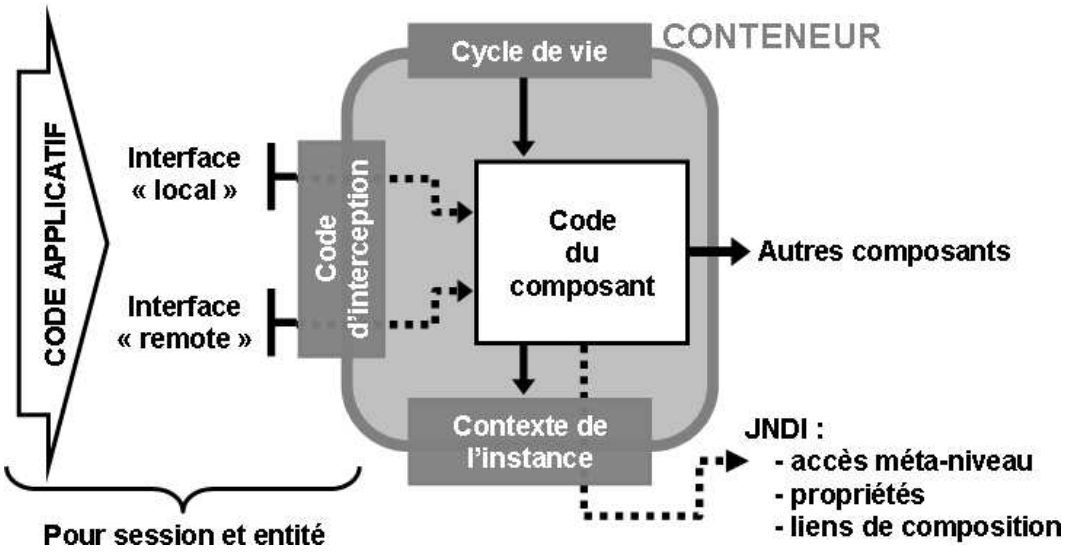

Figure 5.10 – Les principes d'architecture d'un composant métier EJB

déploiement. Cela se fait à l'aide d'un annuaire dédié accessible par JNDI.

– La deuxième action consiste à exporter les dépendances de ces composants dans un espace de noms associé au composant lui-même et à y associer la référence vers le composant auquel se lier grâce à son nom dans l'espace de nommage de l'application.

Rappelons que le découplage en deux actions permet de rendre le code indépendant du processus d'assemblage (pas de lien dans le code avec un composant particulier). A partir de là, les composants vont pouvoir accéder aux composants auxquels ils sont liés en effectuant la phase de liaison effective (phase de *bind*) dans le code applicatif (voir code utilisant *lookup* dans la section 5.2.3 équivalent au *bind*). Notons que cette phase est prise en charge par le conteneur dans la version 3 des EJB (injection des références par le conteneur dans une variable du code du composant) et devient donc transparente pour le code applicatif.

A l'exécution, le conteneur prend en charge le fait que le composant est activé de façon locale (appel à l'intérieur de la même JVM), de façon distante (appel à distance), ou sur arrivée d'un événement (messagerie asynchrone). Il prend aussi en charge les aspects techniques associés à ces composants comme l'activation de transactions, la vérification de contraintes de sécurité, ou les échanges de données avec un espace de sécurisation des données (en général une base de données relationnelles). Nous détaillons cela dans les sections qui suivent pour chaque catégories de composants métier.

La figure 5.10 nous montre que le conteneur met en œuvre du code (pour gérer les aspects techniques qu'il supporte de façon transparente au code applicatif) par interception des appels aux interfaces métier des composants (seulement pour les composants de session et les composants entité). Malheureusement, cette interception n'est pas transparente dans le modèle de programmation. La spécification fait l'hypothèse architecturale qu'il existe un objet d'interception indépendant de l'instance du composant métier, cet objet implantant l'interface métier. Dans ce cas, le comportement fonctionnel est différent suivant qu'on appelle une opération métier sur l'objet d'interposition ou sur l'instance du composant.

Pour se prémunir de mauvaises manipulations entre objet d'interception et instance de composant, l'instance du composant n'implante pas l'interface métier. Cela évite de diffuser une instance d'un composant (diffusion de `this`) comme un objet supportant l'interface métier du composant. Cette contrainte est levée dans la version 3 de la spécification des EJB.

Outre le code d'interposition définie dans le paragraphe précédent, d'autres interactions exsitent entre le conteneur et les composants qu'il gère. Le conteneur a en effet la charge du cycle de vie des instances de composant et signale à ces instances tout événement relatif à ce cycle. Enfin, un composant accède au contexte dans lequel une de ces instances s'exécute (cf. *Contexte de l'instance* dans la figure 5.10). Cela lui permet d'accéder par exemple au contexte de sécurité dans lequel elle s'exécute ou encore d'accéder à certains services (service transactionnel ou service d'échéancier).

5.6.2 Composants de session

Comme leur nom l'indique, d'un point de vue conceptuel, les composants de session ont pour objectif de représenter une session d'utilisation de l'application au niveau du serveur. Etant donné que l'utilisation de l'environnement J2EE ambitionne pour l'essentiel le support des applications Web, de telles sessions impliquent un utilisateur unique. En effet, le modèle d'application Web correspond à l'interaction entre un utilisateur unique et l'application qu'il utilise, généralement à travers un navigateur Web.

Dans le cadre des applications Web, les composants session sont souvent utilisés pour mettre en œuvre le patron d'architecture FAÇADE introduit dans [Gamma et al. 1994]. En effet, parmi les règles de bonne utilisation de J2EE, on retrouve souvent ce patron qui permet de présenter un point d'entrée unique aux applications utilisant ce code métier. Cela simplifie notamment l'utilisation de ce code dans un contexte réparti, en limitant le nombre d'objets qui peuvent être engagés dans des interactions à distance.

L'unicité de l'utilisateur dans le cadre d'une session applicative est un des défauts majeurs du modèle car d'autres formes de sessions existent où peuvent interagir plusieurs utilisateurs. C'est le cas par exemple des sessions de discussion interactives de type *chat* ou de téléphonie impliquant deux utilisateurs, voir des sessions de conférence, de travail coopératif ou de jeux en réseau impliquant potentiellement plus de deux utilisateurs. Cette contrainte spécifiée par le modèle de composant session des EJB sert de garde-fou pour la mise en œuvre de session Web (contexte d'exécution mono-programmé dont le comportement correspond à un enchainement séquentiel d'une page HTML dynamique à une autre). Elle n'est pas contraignante vis-à-vis des autres aspects techniques pris en charge (il supporte généralement la multi-programmation) par ces composants et pourrait donc être facilement levée pour traiter d'autres types de session.

Sessions sans état

Les spécifications EJB distinguent deux types de composants de session : les composants de session sans état et ceux avec état. Concernant la première catégorie, elle sert à supporter des processus métier sans mémoire. Ces processus peuvent être mis en œuvre localement ou à distance à l'aide de RMI ou de Web Services.

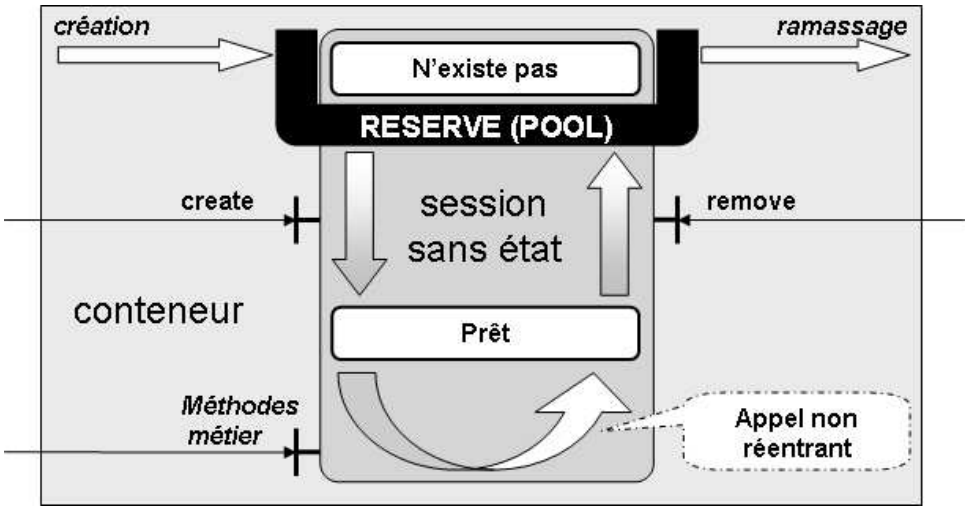

Figure 5.11 – Le cycle de vie d'un composant métier EJB de session sans état

Comme le montre la figure 5.11, un composant de session sans état est relativement primitif dans son fonctionnement. Des instances sont allouées pour effectuer des opérations dans le cadre d'une session et libérées ensuite. Leur cycle de vie contient deux états : l'état non alloué ou non existant et l'état prêt dans lequel les opérations de son interface métier peuvent être activées.

Deux aspects techniques peuvent être pris en charge pour ces composants : le support des transactions et le support de l'accès distant. Pour les transactions, le niveau de transparence offert permet de caler la démarcation transactionnelle sur l'activation d'une opération de l'interface d'un composant : le démarrage de la transaction s'opère à l'appel de l'opération et la terminaison à la sortie de l'opération. Différents comportements transactionnels peuvent être définis :

- `NotSupported` : si le *thread* appelant l'opération avec ce comportement transactionnel est associé à un contexte transactionnel, celui-ci est suspendu le temps d'exécuter cette opération.
- `Required` : si le *thread* appelant l'opération avec ce comportement transactionnel n'est associé pas à un contexte transactionnel, un nouveau contexte est activé le temps d'exécuter cette opération. Dans le cas contraire, l'opération s'exécute dans le contexte transactionnel courant sans toucher à l'association entre le *thread* et le contexte transactionnel.
- `RequiresNew` : le *thread* appelant l'opération avec ce comportement transactionnel suspend le contexte transactionnel courant s'il existe et associe un nouveau contexte transactionnel le temps d'exécuter cette opération.
- `Mandatory` : si le *thread* appelant l'opération avec ce comportement transactionnel n'est pas associé à un contexte transactionnel, une exception est levée par le conteneur.
- `Supports` : si le *thread* appelant l'opération avec ce comportement transactionnel est associé à un contexte transactionnel, cette opération s'exécute dedans sans toucher à l'association entre le *thread* et le contexte transactionnel.

– `Never` : si le *thread* appelant l'opération avec ce comportement transactionnel est associé à un contexte transactionnel, une exception est levée par le conteneur.

Concernant le support des accès distants, il est explicite dans le modèle de programmation : l'interface métier doit étendre l'interface `java.rmi.Remote`, les méthodes définies par l'interface métier devant supporter l'exception `java.rmi.RemoteException`. Cette distinction faite dans le modèle de programmation entre interface métier locale et distante est très contraignante. Elle est supprimée dans la version 3 de la spécification des EJB.

Le coût d'allocation d'une instance de composant de session peut être important, notamment dans le cas où le composant supporte les accès distants (mise en place de chaîne de liaison complexe faisant intervenir des ressources réseau). Par ailleurs, les instances de ce type de composant sans état ont tendance à avoir une durée de vie courte. Pour pallier le problème lié à ces deux propriétés contradictoires en terme de performance, le conteneur EJB interpose généralement une réserve d'instances de composant dans le cycle d'allocation/libération de celles-ci 2.3.3. Un autre intérêt de cette technique est qu'elle permet d'opérer du contrôle d'admission sur l'application, notamment lorsque ce type de composant est utilisé avec le patron d'architecture Façade. Cette approche est possible parce que ces composants sont sans état et qu'il n'ont donc pas d'état initial implicite, comme ça peut être le cas pour les composants de session avec état décrit dans la section suivante.

Sessions avec état

Les composants de session avec état supportent exactement les mêmes fonctions et aspects techniques que ceux sans état. La différence est qu'ils possèdent une mémoire liée à l'activité d'une session. De ce fait, ces instances ont tendance à avoir une durée de vie plus longue que celles sans état. De plus, les instances d'un tel composant ont une identité propre

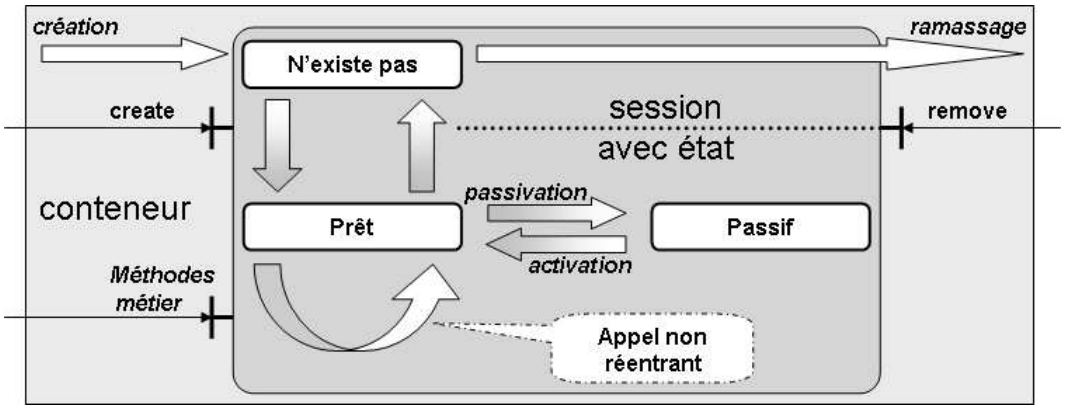

Figure 5.12 – Le cycle de vie d'un composant métier EJB de session avec état

Etant données les propriétés que doivent supporter ces composants, il est proposé de pouvoir les *passiver* de façon à éviter d'emcombrer la mémoire du serveur dans le cas d'un nombre important de sessions à gérer en parallèle. Cette étape, correspondant à un nouvel état dans le cycle de vie du composant de session (cf. figure 5.12), implique que lors d'une

saturation de la mémoire du serveur, certaines instances de composants de session avec état peuvent être vidée dans un espace mémoire secondaire, le choix de ces instances dépendant de politiques de gestion de ressources mises en œuvre par le serveur, par exemple choix des instances les moins récemment utilisées (en anglais *Least Recently Used* ou LRU).

5.6.3 Composants réactifs

Les composants EJB réactifs (en anglais *Message Driven Bean* ou MDB) sont très proches des composants de session sans état. En effet, ils sont eux-mêmes sans état (neutralité de l'instance traitant un message) et leur cycle de vie présente les deux mêmes états. Dans l'état prêt, le composant réactif ne fait que réagir à des événements gérés par le conteneur qui les fait traiter par une instance du composant à l'aide de l'opération `onMessage`.

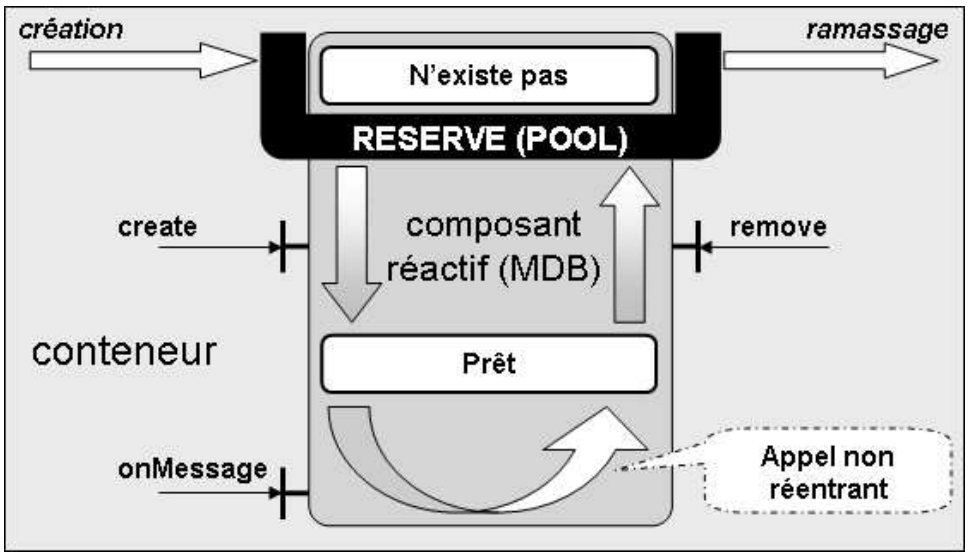

Figure 5.13 – Le cycle de vie d'un composant EJB réactif

Comme pour les composants de présentation (les *servlets*), les composants réactifs ne peuvent pas être invoqués de l'extérieur du conteneur, ce qui n'est pas le cas des composants de session pour lesquels l'allocation et la libération d'instance ou encore l'activation du code métier sont pilotées de l'extérieur du conteneur. Nous verrons que c'est aussi le cas pour les composants entité.

Les composants réactifs supportent des comportements transactionnels plus limités que les composants de session. En fait, seuls deux comportements transactionnels peuvent être spécifiés pour l'opération `onMessage` :

- `Required` : un contexte transactionnel est alloué et est associé au *thread* utilisé pour exécuter `onMessage` le temps de son exécution. Le message est normalement consommé de manière transactionnelle (dépendant des capacité technique de l'émetteur de message). Cela signifie que si la transaction exécutée par `onMessage` échoue, une nouvelle tentative de consommation du message sera effectuer ultérieurement (nouvelle activation de `onMessage`).

– `NotSupported` : l'opération `onMessage` est exécutée hors de tout contexte transactionnel et consomme le message qui a été délivré.

5.6.4 Composants entité

Les composants entité sont des composants qui représentent des informations persistantes d'une application d'entreprise, c'est-à-dire des informations dont la durée de vie est supérieure à la session qui les a créée ou même à l'application elle-même (l'information persiste à l'arrêt de l'application). Ces informations sont généralement stockées dans une base de données relationnelle et manipulées à travers l'API JDBC (pour *Java DataBase Connectivity*). Cette interface fournit le niveau d'abstraction correspondant aux fonctions d'une base de données relationnelle. Or, l'environnement EJB ambitionne de fournir une vue objet métier de ces données relationnelles au niveau du modèle de programmation. C'est l'objectif des composants entité que de fournir cette couche d'adaptation des objets applicatifs métier vers leur stockage relationnel.

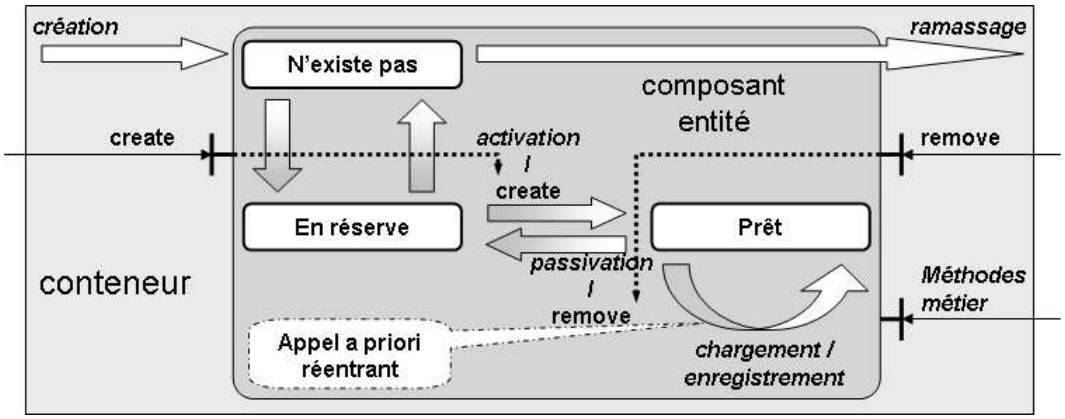

Figure 5.14 – Le cycle de vie d'un composant EJB entité

Le cycle de vie d'un composant entité est assez semblable aux précédents. La principale différence consiste à exhiber le passage par une réserve d'instances pour leur allocation. Etant donné que ces instances peuvent être manipuler en grand nombre et de manière récurrente d'une transaction à l'autre, il est important d'optimiser leur processus d'allocation. En fait, les instances qui sont dans la réserve ont a priori exécuté tout leur cycle de configuration (mise en liaison avec les ressources dont elles dépendent). Lors de l'allocation d'une instance, le conteneur en cherche une disponible dans la réserve avant d'allouer un nouvel objet Java à configurer.

Pour passer dans l'état *Prêt*, il faut qu'une instance soit liée à un son représentant dans la base (en général un n-uplet dans une table). Cette liaison est définie par le nom dans la base de ce représentant, c'est à dire sa clé primaire. Cette liaison peut être créée de plusieurs manière :

– Création d'une nouvelle instance : la création d'une nouvelle instance de composant entité crée implicitement un représentant dans la base et donc provoque la création d'un nouveau nom définissant la liaison.

– Navigation à travers une référence vers un objet : lorsque le conteneur traverse une référence vers une autre objet persistant, il doit alors activer une instance de composant entité si elle n'existe pas déjà. Il va alors allouer cette instance et lui associer le nom de l'objet à atteindre en récupérant le nom de son représentant dans l'état de l'objet référençant, définissant ainsi la liaison.
– Parcours du résultat d'un *finder* : un *finder* permet de récupérer le résultat d'une requête EJB-QL qui est généralement une collection d'instances d'un composant entité. La mécanique sous-jacente émet vers la base de données une requête SQL qui remonte la collection de noms des représentants sélectionnés. A partir de là le conteneur peut définir les liaisons entre les instances et leur représentant lorsque l'utilisateur parcourt sa collection d'instances.

Une fois une liaison établie d'une des manières précédentes, le conteneur prend en charge la synchronisation entre l'instance et son représentant en base de données de façon transparente pour le programmeur (cf. actions *chargement* et *enregistrement* dans la figure 5.14 équivalent en anglais des actions *load* et *store*). L'action de *chargement* est généralement exécutée après la mise en liaison et avant tout autre appel vers des opérations métier. L'opération inverse d'*enregistrement* intervient quant à elle soit lors de la validation de transaction, soit lors d'opérations de passivation (voir les paragraphes qui suivent).

Les instances ne retournent dans la réserve ou ne sont détruites qu'après une des deux opérations suivantes :

– Exécution de `remove` déclenchée depuis le code applicatif : cette opération provoque la destruction du représentant de la base de données et donc du nom associé. La liaison avec l'instance de composant est rompue et cette instance retourne alors dans la réserve.
– Opération de passivation déclenchée par le conteneur : lorsqu'il considère que la mémoire est trop encombrée, le conteneur peut décider d'en éliminer certaines instances de composants entité, suivant des politiques qui lui sont propres (par exemple *LRU* comme précédemment pour les composants de session avec état). Dans ce cas, l'instance est généralement synchronisée avec son représentant si nécessaire, déliée et renvoyée dans la réserve, voire ramassée par la JVM.

Outre la transparence de projection entre composants entité et base de donnée, les transactions sont supportées de la même manière que pour les composants de session au niveau des opérations métier. Cette fonction est en principe peu utilisée dans ce cadre car les transactions sont des séquences d'opérations opérant à une autre granularité (processus métier). Néanmoins, les composants entité peuvent aussi être utilisés pour supporter des sessions persistantes (cas de sessions particulièrement longues).

5.6.5 Gestion des événements relatifs aux composants métier

Les composants métier ont la possibilité de récupérer un certain nombre d'événements qui peuvent être utiles à leur logique propre. Trois catégories d'événements sont décrites dans les sous-sections qui suivent. Dans tous les cas, les événements sont gérés/émis par le conteneur vers des instances de composants métier. Le même patron architectural est utilisé : les composants intéressés par des événements doivent implanter une interface spécifique qui permet au conteneur d'activer l'instance de composant intéressée pour qu'elle traite l'événement en question. Dans le cas de la version 3 de la spécification, il n'y a

pas besoin d'implanter des interfaces. Il suffit de définir une correspondance entre des opérations du composant et les événements qui peuvent être traités.

Evénements relatifs au cycle de vie des instances de composant

Les premiers événements qui peuvent être traités par les instances de composants sont ceux relatifs leur cycle de vie qui est géré par le conteneur. Ils dépendent du type de composants. Ils sont spécifiés par des interfaces associées à chacun de ces types : `javax.ejb.SessionBean` pour les composants de session, `javax.ejb.MessageDrivenBean` pour les composants réactifs et `javax.ejb.EntityBean` pour les composants entité. Comme ces interfaces typent le composant, l'une d'entre elles doit obligatoirement être implantée par un composant métier, même si aucun traitement n'est effectué pour ces événements. Cette contrainte est levée dans la version 3 de la spécification EJB.

Pour les composants de session, quatre événements sont émis par le conteneur. Seulement deux d'entre eux sont émis pour les sessions sans état, à savoir `setSessionContext` et `ejbRemove` :

- `setSessionContext` : cet événement est émis à la création d'une instance de composant de session. Il permet à l'instance de récupérer le contexte qui lui est associé.
- `ejbRemove` : cet événement est émis à la destruction d'une instance de composant de session. L'étape suivante est généralement la prise en charge de cette instance par le ramasse-miette.
- `ejbPassivate` : cet événement est émis lorsqu'une instance de composant de session avec état va être rendu passif (stockage dans un espace mémoire secondaire).
- `ejbActivate` : cet événement est émis lorsqu'une instance de composant de session avec état est réactivée, c'est ramenée en mémoire principale avec restauration de son suivant les règles définies par la spécification EJB (un composant de session avec état doit être `serializable` et les variables définies comme `transient` peuvent aussi être sauvegardées et restaurées sous certaines conditions.

Pour les composants réactifs, deux événements seulement sont émis par le conteneur :

- `setMessageDrivenContext` : similaire à l'événement `setSessionContext` des composants de session.
- `ejbRemove` : similaire à l'événement `ejbRemove` des composants de session.

Pour les composants entité, sept événements sont émis par le conteneur :

- `setEntityContext` : similaire à l'événement `setSessionContext` des composants de session.
- `unsetEntityContext` : similaire à l'événement `ejbRemove` des composants de session.
- `ejbRemove` : l'événement est émis lorsque le représentant de stockage associé à cette instance est éliminé de la base de données.
- `ejbLoad` : l'événement est émis lorsque l'état de l'instance de composant est chargé de la base de données vers la mémoire principale.
- `ejbStore` : l'événement est émis lorsque l'état de l'instance de composant est enregistré dans la base de données à partir de la mémoire principale.
- `ejbPassivate` : similaire à l'événement `ejbPassivate` des composants de session, mise à part que l'espace secondaire vers lequel est envoyé l'état de l'instance est la base de données sous-jacente, et notamment le représentant de stockage de l'instance en question.

– `ejbActivate` : similaire à l'événement `ejbActivate` des composants de session, mise à part que l'espace secondaire depuis lequel l'état de l'instance est réactivé est la base de données sous-jacente, et notamment le représentant de stockage de l'instance en question.

La définition de ces événements de même que celle des cycles de vie aurait méritée un traitement plus systématique. En effet, certains événements communs sont définis avec le même patron de nommage (pourquoi pas le même nom...) comme `set...Context`. D'autres événement communs changent de nom suivant le type de composant (par exemple `ejbRemove` des composants de session et des composants réactifs devient `unsetEntityContext` pour les composants entité, pour lesquels `ejbRemove` prend une autre sémantique. Concernant les cycles de vie, ceux des composants réactifs et des composants de session sans état sont exactement les mêmes, sachant que le cycle de vie des composants de session avec état étend le précédent, de même que celui des composants entité étant lui-même celui des sessions avec état.

Evénements relatifs aux transactions

Il est possible d'être averti des événements relatifs aux démarcations transactionnelles, notamment lorsque celles-ci sont prises en charge de façon transparente par le conteneur. Le composant doit pour cela implanter l'interface `javax.ejb.SessionSynchronization` qui définit les événements sont les suivants :

– `afterBegin` : l'événement est émis avant l'exécution de n'importe qu'elle opération métier lorqu'une instance de composant s'exécute dans un nouveau contexte transactionnel.
– `beforeCompletion` : l'événement est émis lorsqu'une instance de composant s'exécute dans un contexte transactionnel qui est sur le point d'être validé ou annulé.
– `afterCompletion` : l'événement est émis lorsqu'une instance de composant s'exécute dans un contexte transactionnel qui vient d'être validé ou annulé.

Ces événements peuvent être émis seulement dans le cadre des composants de session avec état. Là encore, le manque de vision systématique est dommageable car de tels événements pourraient aussi bien intéressés d'autres types de composants qui supportent la démarcation gérée par le conteneur (en fait, tous les composants EJB ! !). Il est toujours possible d'obtenir cette fonction en interagissant directement avec le service transactionnel pour qu'une instance de composant s'enregistre comme une *Synchronization*.

Evénements temporels (horloges / échéanciers)

Il est possible d'associer des événements temporels aux composants EJB. Pour que les composants supportent de tels événements, il faut qu'ils implantent l'interface `javax.ejb.TimedObject`. Lors de l'échéance d'un événement temporelle, l'opération `ejbTimeout` de l'interface en question est invoquée par le conteneur, avec en paramètre l'événement temporel arrivé à échéance.

Deux types d'échéancier peuvent être définis. Le premier type permet de définir une échéance unique, c'est à dire qu'un seul événement est émis à l'échéance. Le second type permet de définir des échéanciers périodiques, c'est à dire qu'un événement est émis à période fixe.

Le service d'échéanciers (en anglais *Timer Service*) est accessible aux composants (sauf pour les composants de session avec état qui ne sont pas pris en compte dans la version 2.1 de la spécification EJB) ou plus précisément à leurs instances à travers leur contexte. Il permet donc de créer de nouveaux échéanciers qui sont associés aux instances de composants de la manière suivante :

- Pour les composants de session sans état et les composants réactifs : les échéanciers créés ne sont en fait pas associés à une instance particulière puisque dans le cas de ces composants, il n'y a pas de distinction entre les instances. Lorsqu'un événement temporel est émis par le conteneur, celui-ci active une instance quelconque du composant pour le traiter.
- Pour les composants entité : les échéanciers sont effectivement associés à l'instance qui les crée. En fait, l'association est faite avec l'identifiant (basé sur la clé primaire) de l'instance. Cela permet au conteneur d'activer la bonne instance de composant pour traiter les échéances d'événements.

Dans tous les cas, les échéanciers ainsi définis sont persistants. En effet, ils doivent survivre à l'arrêt ou à la panne du serveur qui les héberge. Si des échéances sont intervenues pendant de tels arrêts, tous les événements correspondants doivent être émis lors du redémarrage.

On comprend bien que cette approche ne peut pas être utilisée pour les composants de session avec état car les instances de ceux-ci ont une identité propre mais ne survive pas à l'arrêt du serveur qui les héberge. Des échéanciers persistants n'ont donc pas de sens pour de tels composants. Leur support dans de futures versions de la spécification ne doit néanmoins pas poser de problème.

5.7 Conclusion et perspectives

Ce chapitre présente une synthèse des principaux principes et des principales fonctionnalités de l'environnement J2EE. Cette synthèse se concentre sur les fonctions qui permettent de développer et d'intégrer des applications d'entreprise avec des garanties de sûreté de fonctionnement (support des transactions et notamment des transactions réparties) et de sécurité (support de l'authentification et du contrôle d'accès).

La synthèse insiste aussi sur les principes de d'architecture et de construction du code proposés par l'environnement J2EE. Le support de composant est omniprésent dans la solution proposée même s'il n'est pas appliqué de façon homogène et systématique (approche ad hoc avec différentes notions de composant suivant les spécifications considérées telles que *servlet* ou EJB).

On peut néanmoins affirmer que la notion de composant J2EE est un relatif échec. En effet, il n'existe pratiquement pas d'environnement de développement permettant d'effectuer de l'assemblage d'applications à partir de bibliothèques de composants J2EE existant.

Par ailleurs, la mise en œuvre des modèles de composants J2EE est considérée comme lourde par de nombreux développeurs, même si les environnements de développement tels que Eclipse (cf. `http://www.eclipse.org`) ou NetBeans (cf. `http://www.netbeans.org`) exhibent des fonctions avancées pour faciliter ces développements. La facilité de programmation est d'ailleurs un des principaux axes d'amélioration à l'œuvre dans la version 1.5 de J2EE qui doit être finalisée bientôt. C'est

notamment le cas pour les EJB où la version 3.0 des spécifications simplifie largement le développement.

Bibliographie

[Alur et al. 2003] Alur, D., Malks, D., and Crupi, J. (2003). *Core J2EE Patterns: Best Practices and Design Strategies, Second Edition*. Prentice Hall. 650 pp.

[B. Stearns 2000a] B. Stearns (2000a). JavaBeans 101 Tutorial, Part I. http://java.sun.com/developer/onlineTraining/Beans/bean01/index.html.

[B. Stearns 2000b] B. Stearns (2000b). JavaBeans 101 Tutorial, Part II. http://java.sun.com/developer/onlineTraining/Beans/bean02/index.html.

[Dudney et al. 2003] Dudney, B., Asbury, S., Krozak, J. K., and Wittkopf, K. (2003). *J2EE AntiPatterns*. Wiley. 600 pp.

[Dumant et al. 1998] Dumant, B., Horn, F., Tran, F. D., and Stefani, J.-B. (1998). Jonathan: an Open Distributed Processing Environment in Java. In Davies, R., Raymond, K., and Seitz, J., editors, *Proceedings of Middleware'98: IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing*, pages 175–190, The Lake District, UK. Springer-Verlag.

[Gamma et al. 1994] Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (1994). *Design Patterns: Elements of Reusable Object Oriented Software*. Addison-Wesley. 416 pp.

[Gray and Reuter 1993] Gray, J. and Reuter, A. (1993). *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann.

[Handley et al. 2000] Handley, M., Schulzrinne, H., Schooler, E., and Rosenberg, J. (2000). SIP: Session Initiation Protocol. http://www.ietf.org/internet-drafts/draft-ietf-sip-rfc2543bis-02.txt.

[J2EE 2005] J2EE (2005). Java 2 Enterprise Edition. http://java.sun.com/products/j2ee.

[J2SE 2005] J2SE (2005). Java 2 Standard Edition. http://java.sun.com/products/j2se.

[JBoss Group 2003] JBoss Group (2003). JBoss Application Server. http://www.jboss.com/products/jbossas.

[Objectweb Enhydra 2005] Objectweb Enhydra (2005). Enhydra XMLC. http://xmlc.objectweb.org/.

[Sun 2005] Sun (2005). J2EE Application Validation Kit. http://java.sun.com/j2ee/avk/.

[Sun JSR-000116 2003] Sun JSR-000116 (2003). SIP Servlet API. http://jcp.org/aboutJava/communityprocess/final/jsr116/index.html.

[Sun JSR-000127 2004] Sun JSR-000127 (2004). JavaServer Faces Specification. http://jcp.org/aboutJava/communityprocess/final/jsr127/index2.html.

[Sun JSR-000152 2003] Sun JSR-000152 (2003). JavaServer Pages 2.0 Specification. http://jcp.org/aboutJava/communityprocess/final/jsr152/index.html.

[Sun JSR-000153 2003] Sun JSR-000153 (2003). Java Entreprise Bean 2.1 Specification. http://www.jcp.org/aboutJava/communityprocess/final/jsr153/.

[Sun JSR-000154 2003] Sun JSR-000154 (2003). Java Servlet 2.4 Specification. http://jcp.org/aboutJava/communityprocess/final/jsr154/index.html.

[Sun JSR-000904 2000] Sun JSR-000904 (2000). JavaMail Specification.
http://jcp.org/aboutJava/communityprocess/maintenance/jsr904/index.html.

[Sun Microsystems 2003] Sun Microsystems (2003). Java Remote Method Invocation (RMI).
http://java.sun.com/products/jdk/rmi/.

[Szyperski and Pfister 1997] Szyperski, C. and Pfister, C. (1997). Component-oriented programming: WCOP'96 workshop report. In Mühlhäuser, M., editor, *Workshop Reader of 10th Eur. Conf. on Object Oriented Programming ECOOP'96, Linz, July 8–12*, Special Issues in Object-Oriented Programming, pages 127–130. Dpunkt, Heidelberg.

[The Apache Software Foundation 2005] The Apache Software Foundation (2005). The Struts Framework.
http://struts.apache.org/.

[The Apache Software Foundation 2006] The Apache Software Foundation (2006). Apache Geronimo.
http://geronimo.apache.org/.

[The Objectweb Consortium 2000] The Objectweb Consortium (2000). JOnAS: Java (TM) Open Application Server.
http://jonas.objectweb.org/.