RENPAR'16 / CFSE'4 / SympAAA'2005 / Journées Composants
Le Croisic, France, 5 au 8 avril 2005

Passage à l'échelle de serveur J2EE : le cas des EJB

Sylvain Sicard, Noel De Palma, Daniel Hagimont

INRIA Rhônes-Alpes,
Zirst - 655 avenue de l'Europe -
Montbonnot 38334 Saint Ismier Cedex - France
Prenom.Nom@inrialpes.fr

Résumé
Le Web d'aujourd'hui nécessite la programmation de serveurs d'applications de plus en plus évoluées. La norme J2EE permet de concevoir des serveurs d'applications. Ces serveurs doivent assurer le passage à l'échelle et la haute disponibilité des services proposés. Le passage à l'échelle et la haute disponibilité peuvent être atteints en dupliquant ces serveurs sur des grappes de machines (clusters). CMI (Cluster Method Invocation) est une solution existante permettant de dupliquer le tiers EJB dans un serveur J2EE. Dans un premier temps, cet article étudie les performances de la solution fondée sur CMI. Dans un deuxième temps, nous proposons une solution alternative et nous montrons les gains significatifs obtenus comparés à CMI.

Mots-clés : Passage à l'échelle, J2EE, EJB, grappe.

---

1. Introduction

Le Web d'aujourd'hui nécessite la programmation de serveurs d'applications de plus en plus évoluées. La norme J2EE [12] permet de concevoir des serveurs d'applications multi-tiers (ou multi-niveaux). Un serveur J2EE, présenté Figure 1, est généralement composé de quatre tiers chacun pouvant s'exécuter sur une machine différente :

– Un tiers Web qui exécute les requêtes des clients (e.g. Apache [14]) : une référence à une page statique est traitée par le serveur web, une référence à une page dynamique est déléguée au serveur de Servlet.
– Un tiers serveur de Servlet (e.g. Tomcat [5]) qui génère une page web à la volée à partir des données fournies par le tiers EJB (la création de la page Web est dynamique).
– Un tiers serveur EJB (e.g. JonAS [6]) qui contient la logique fonctionnelle de l'application et qui assure les propriétés non-fonctionnelles de l'application. Le serveur EJB interagit principalement avec le tiers base de données qui stocke les données applicatives.

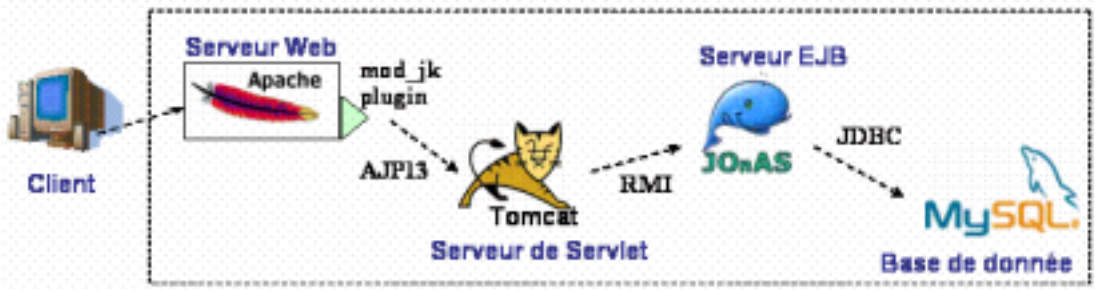

FIG. 1 – Architecture classique d'un serveur J2EE

Ces serveurs doivent assurer, entre autre, le passage à l'échelle des services proposés. Le passage à l'échelle d'un serveur peut se définir comme étant la capacité d'un serveur à traiter simultanément un nombre quelconque de requêtes clientes dans un délais raisonnable.

Le passage à l'échelle peut être atteint en dupliquant ces serveurs sur des grappes de machines (ou clusters). La mise en grappe d'un serveur d'application est appelée *clusterisation*. La démarche employée dans les solutions actuelles consiste à dupliquer "entièrement" les tiers de façon à pouvoir disposer d'un ensemble de clones. Les requêtes clientes sont aiguillées indifféremment vers l'un des clones afin de répartir la charge, en général en suivant un algorithme de type Round-Robin (tourniquet).
Cette solution pose un problème de cohérence important. En effet, un tiers dupliqué (en particulier un serveur EJB) peut contenir un état modifiable qui doit être maintenu en cohérence. Ceci a de forte implication sur la mise en grappe d'un serveur d'application comme nous le verrons par la suite.
Cet article étudie le passage à l'échelle d'un serveur J2EE par duplication du tiers EJB et propose une solution alternative, plus performante, basée sur un clonage partiel du serveur EJB. Dans un premier temps nous présentons le contexte et les motivations pour nos travaux (Section 2). Après une description de l'environnement d'expérimentation utilisé (Section3), nous rapportons les performances de différentes configurations montrant les avantages et lacunes de la solution actuelle (Sections 4 et 5). Nous décrivons ensuite la conception et les performances d'une solution améliorant cette solution (Sections 6 et 7).

2. Contexte et motivations

L'objectif des serveurs J2EE en grappe est d'assurer le passage à l'échelle et la tolérance aux fautes des applications Web. Dans le cas du passage à l'échelle, une application Web doit pouvoir servir des millions de requêtes par jour. Ceci est généralement assuré par une réplication massive de tous les tiers J2EE. Un exemple d'une telle architecture est présenté Figure 2.

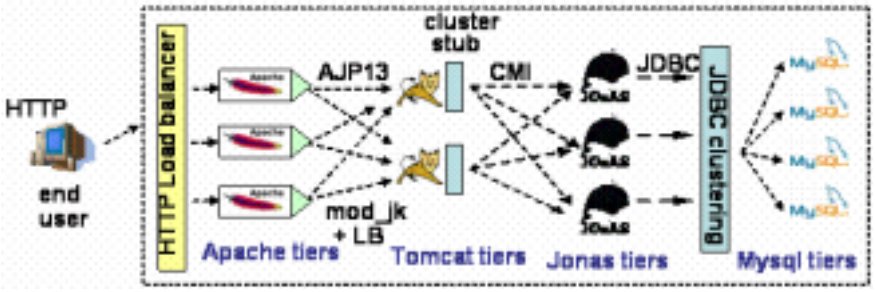

FIG. 2 – Architecture J2EE en grappe

Dans cet exemple, le tiers Web est composé d'un ensemble de serveurs Web Apache dupliqués. Les serveurs Apache ont un état en lecture seule, ils ne posent donc pas de problèmes de cohérence. La duplication est masquée aux clients grâce à des techniques matérielles ou logicielles, telles que : Level-4 switch (où un routeur dédié peut distribuer simultanément jusqu'à 700000 connexions TCP vers différents serveurs), TCP handoffs (où un serveur frontal établit les connexions TCP et délègue le reste du travail à des serveurs esclaves), ou Round-Robin DNS (où un serveur DNS change périodiquement les adresses IP associées au nom du site Web).
Le tiers Servlet est composé de serveurs de Servlet Tomcat dupliqués. Le tiers Apache répartit les requêtes vers les Tomcats grâce au connecteur AJP13 et au plugin Apache mod_jk qui implante une stratégie de répartition de charge Round-Robin entre tous les Tomcats. Les Tomcats peuvent maintenir un état modifiable. Cet état est maintenu en cohérence entre tous les Tomcats grâce à un protocole de groupe.
Le tiers EJB est composé de serveurs EJB JOnAS dupliqués. Les serveurs EJB gèrent un ensemble de beans (objets Java) qui contiennent un état modifiable. La duplication et la cohérence sont assurées dans cet exemple par CMI (Cluster Method Invocation, l'outil de clusterisation fourni par JOnAS), un outil de répartition des appels de méthode de type Round-Robin. Les Tomcats interagissent avec les serveurs EJB via des cluster-stubs (dans CMI, l'équivalent des stubs RMI) qui assurent une répartition de charge entre tous les serveurs.
Le serveur d'EJB, au cours de ses traitements, est amené à interagir avec la base de données, qui stocke les beans de façon persistante. Ces interactions étant très coûteuses en ressource, le serveur d'EJB conserve dans un cache une copie locale des beans ramenés de la base. De cette façon, lors des traitements sui-

vants, ces beans sont présents localement et le serveur évite ainsi un accès coûteux à la base. Si on duplique le serveur d'EJB, une copie d'un même bean peut se trouver copié sur plusieurs serveurs en même temps (voir Figure 3). Le problème de la cohérence se pose alors. CMI évite ce problème en désactivant la fonction de cache sur le serveur d'EJB. De cette façon, le mécanisme transactionnel de la base de données assure la gestion de la cohérence, toute modification étant systématiquement répercutée sur la base de données et relue par un serveur d'EJB depuis la base de données.

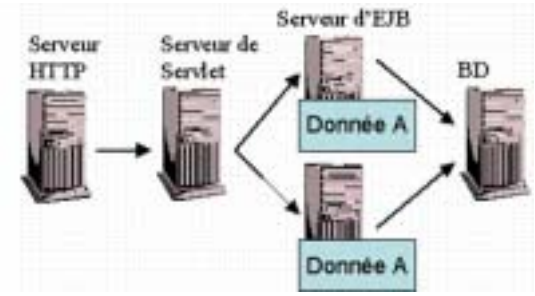

FIG. 3 – Problème de cohérence des données

Le tiers Base de Données est composé d'un ensemble de base de données Mysql dupliquées. Les données sont maintenues en cohérence à cet étage grâce à C-JDBC [2]. Les serveurs EJB sont connectés aux bases de données via à un contrôleur frontal C-JDBC, permettant de passer à l'échelle tout en maintenant la cohérence entre les clones de la base de données. A notre connaissance, CMI est la seule technologie permettant de la duplication de serveur EJB dans une architecture J2EE. Dans ce contexte nous nous intéressons au problème du passage à l'échelle des serveurs EJB dans une architecture J2EE. Nous proposons dans le travail relaté ici, d'explorer une solution alternative à CMI permettant de préserver l'usage du cache sur les serveurs d'EJB.

3. Environnement

Lors de nos expérimentations nous avons utilisé le serveur http Apache (version 1.3.31), le serveur de servlet Tomcat (version 3.3.1a), le serveur d'EJB JOnAS (version 1.4.2), le système de gestion de base de données MySQL (version 4.0.20). Afin de garantir que l'étage base de données ne soit pas un facteur limitant dans nos expérimentations (la base de données pouvant être saturée avant le serveur d'EJB) nous avons utilisé l'outil C-JDBC, Cluster-Java DataBase Conectivity (version 1.0).
Nous avons utilisé l'application RUBiS [1] comme banc d'essai de notre plateforme J2EE. C'est avec cette application que sont effectués tous les tests dont il sera question par la suite. RUBiS est un prototype de site Web d'enchère modélisé d'après Ebay.com. RUBiS implémente toutes les fonctions de base de ce type de site avec notamment la possibilité pour un utilisateur, de vendre, enchérir, rechercher et naviguer entre les objets mis en vente. Dans RUBiS, trois types d'utilisateurs ont été modélisés :

– les acheteurs : ils doivent s'enregistrer. Ensuite ils peuvent faire des enchères, laisser un commentaire sur un vendeur ou encore consulter la liste de leurs enchères.
– les vendeurs : ils doivent s'enregistrer et verser un droit d'inscription. Ensuite ils peuvent mettre des objets en vente.
– les visiteurs : ils n'ont pas besoin de s'enregistrer mais peuvent seulement naviguer sur le site, sans faire d'enchère ni de mise en vente.

RUBiS est une application J2EE complète qui une fois déployée sur la plate-forme de test permet de mesurer les performances du serveur en charge. RUBiS fournit un certain nombre d'implémentations différentes du même site afin de tester les différents étages d'un serveur J2EE. Nous avons utilisé une implémentation avec EJB utilisant le design pattern Session Façade Bean avec une gestion de la persistance de type CMP car il s'agit du modèle hébergeant sur JOnAS le plus de logique métier, par opposition à d'autres implémentations sollicitant plus fortement l'étage Tomcat. RUBiS comporte également un injecteur de charge. Cet injecteur émule des clients se connectant sur le site à l'aide de leur navigateur web. Ces clients sont modélisés par des chaînes de Markov. L'émulateur de clients qu'intègre RUBiS assure également le monitoring des différents noeuds (client et serveur). Dans les expérimentations dé-

crites dans la suite, nous nous intéressons principalement au débit du serveur en requêtes par seconde, lorsque qu'il est soumis à une forte charge. Nous augmentons la charge en créant un grand nombre de clients (éventuellement sur plusieurs machines clientes).

4. Server EJB non dupliqué

Dans cette section, nous nous intéressons au mécanisme de cache implémenté sur le serveur EJB JOnAS . On rappelle que, dans la norme EJB, les composants applicatifs persistants sont appelés *bean entité*. Une usine à composant permettant de créer des beans est appelée une *factory*. Une factory permet également de récupérer une référence à un bean à partir d'un nom. Les fonctions de cache d'un serveur EJB ne concernent que les beans entité. Chaque factory déployée dans un serveur JOnAS gère son propre cache. Il existe donc autant de cache que de factory et chaque cache ne contient que des beans issus de la même Factory.
Les applications J2EE intègrent un descripteur de déploiement permettant de spécifier, entre autre, pour chaque EJB s'il est partagé ou non. Si l'EJB est déclaré comme partagé, il ne sera pas conservé dans le cache de la factory et sera lu / écrit systématiquement depuis la base de données à chaque accès. S'il est déclaré comme non partagé, il ne sera chargé depuis la base de données que lors de la première invocation et ne sera réécrit que lors de la fin de la transaction qui l'a chargé.

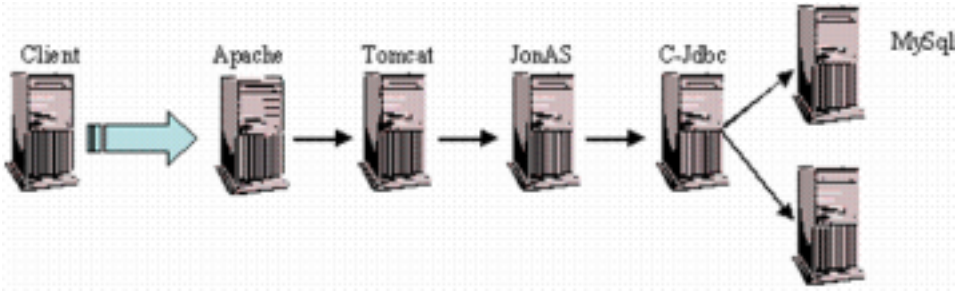

FIG. 4 – Architecture de la plateforme d'évaluation du cache

4.1. Performances avec cache

Dans cette section, nous mesurons les performances obtenues avec RUBiS lorsque le cache du serveur EJB est activé. Pour cela, l'application est déployée avec un descripteur de déploiement spécifiant les beans comme n'étant pas partagés. La plate-forme de test que nous avons mise en place pour cette mesure comprend 7 machines différentes comme indiqué sur la Figure 4.

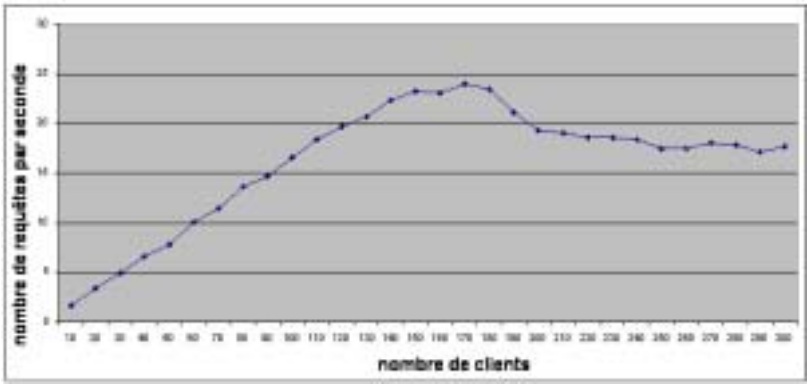

FIG. 5 – Débit avec un serveur EJB et cache activé

Dans cette expérience, la charge sur les bases de données ne rend pas indispensable la présence de l'étage C-JDBC, mais il a été mis en place pour des raisons de cohérence entre cette expérience et celles à venir. De cette façon, les charges et temps de latence observés ici seront plus facilement comparables

si les architectures mises en oeuvre ne varient pas.

La Figure 5 présente les résultats de l'évaluation de performance. La courbe montre le débit en nombre de requêtes par seconde sur le serveur J2EE en fonction du nombre de clients. On constate que tant qu'aucune ressource ne sature dans les serveurs, le débit en requêtes croît linéairement en fonction de la charge injectée jusqu'à atteindre un maximum au-delà duquel le débit décroît. Ce maximum correspond à la saturation de la ressource CPU sur la machine hébergeant le serveur JOnAS. Ce maximum est atteint pour un nombre de clients voisin de 160.

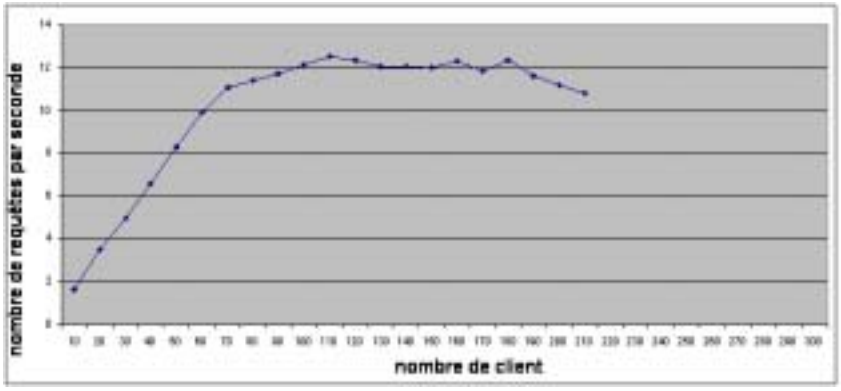

FIG. 6 – Débit avec un serveur EJB et cache désactivé

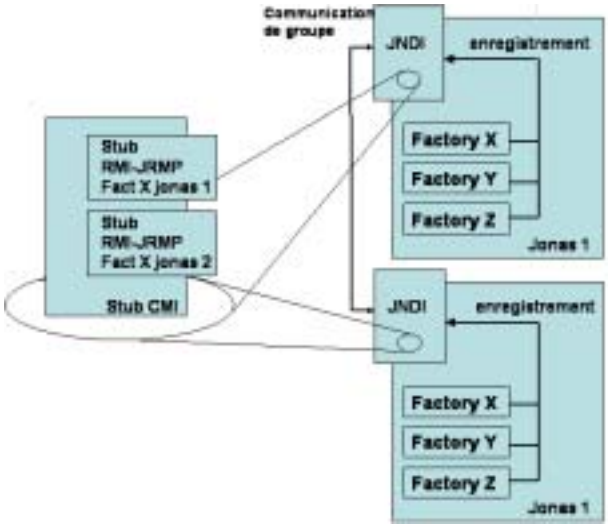

FIG. 7 – Principe de fonctionnement de CMI

4.2. Performances sans cache

Le but ici est de reproduire exactement la même expérience que précédemment à ceci près que l'on interdira aux factory de conserver l'état des beans dans le cache. De cette façon, nous seront à même de quantifier l'impact du cache sur les performances. Dans cette configuration, chaque accès à un bean génèrera un accès à la base de données. On s'attend donc naturellement à observer des temps de latence supérieurs à ceux observés précédemment.

La Figure 6 présente les résultats de l'évaluation. On observe un comportement général similaire à l'expérience précédente, ce qui est conforme à nos attentes. Ici, la ressource CPU est saturée avec un nombre de clients proche de 80. On remarque qu'on ne présente pas de résultats pour des charges allant au delà de 210 clients. En effet, passé cette charge, les performances étaient tellement dégradées que toutes sortes d'erreurs se produisaient, ce qui rendait les résultats ininterprétables. On considère alors que le système

est au-delà de ses capacités.
Comme le montre des deux expériences précédentes, le cache influe sur les performances dans le sens que l'on attendait. Les performances sont meilleures lorsque l'état persistant des beans est conservé localement. Le nombre d'interactions avec la base de données est alors bien moins important.
Pour qu'un cache soit efficace, il faut que d'une part son accès soit très nettement moins coûteux que l'accès original, ce qui est donc notre cas, mais il faut en plus que le nombre de défauts dans le cache (accès à une donnée ne se trouvant pas dans le cache) soit le plus faible possible. Pour évaluer ceci, nous avons effectué des expériences complémentaires qui ont permis de montrer que le taux d'accès (*hit*) dans le cache était proche de 86%. C'est un très bon taux, mais il convient de garder à l'esprit que c'est une donnée entièrement conditionnée par le niveau applicatif. Une application accédant de façon homogène à une très grande quantité de donnée aura tendance à faire diminuer ce taux, mais ce sont là des phénomènes connus liés au concept de cache.
Les résultats obtenus ici montrent que le fait de cacher des beans au niveau de JOnAS s'avére extrêmement utile surtout au vu du gain de performances obtenus. Dans un contexte où l'on souhaite dupliquer le serveur JOnAS, il est donc probable que la mise en place de politique d'équilibrage autorisant le cache usage aura un impact certain sur les performances de l'ensemble du système.

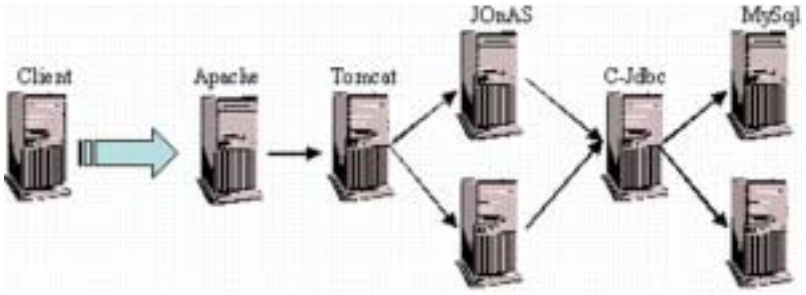

FIG. 8 – Architecture de la plateforme de test de CMI

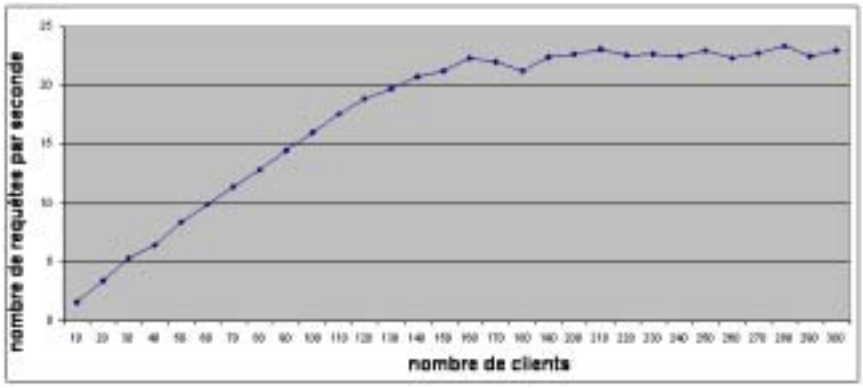

FIG. 9 – Débit avec deux serveurs EJB, CMI et cache désactivé

5. Duplication du serveur EJB avec CMI

5.1. CMI sans cache

Cette expérience a pour objectif de mesurer les performances de la solution CMI en comparaison avec celle d'un seul JOnAS avec son cache activé. L'objectif de cette comparaison est de montrer qu'avec cette configuration, le gain de puissance apporté par la duplication du serveur JOnAS est contrebalancé par la perte de performance lié à la désactivation du cache. Cette expérience nous servira également de performance de référence lorsque nous testerons les performances de la politique que nous proposons, basée sur le partitionnement des factory (Section 6).

CMI est une partie de JOnAS proposant une évolution de RMI-JRMP (Java Remote Message Protocol) conçue pour permettre un équilibrage de charge sur un ensemble de serveur JOnAS. Cette solution apporte certes le passage à l'échelle, mais au détriment des performances apportées par le cache que l'on ne peut utiliser dans ce contexte. Nous verrons par la suite les raisons de cette impossibilité.

Le schéma d'appel de méthode sur un bean depuis une Servlet cliente est le suivant. Le client consulte un serveur de nom JNDI ?[13] pour récupérer une référence (stub) à une factory à partir d'un nom. Cette factory joue le rôle de serveur de nom pour les beans qu'elle gère, donc le client consulte cette factory pour récupérer une référence (stub) à un bean. Le client peut alors appeler une méthode sur le bean.

Le principe de fonctionnement de CMI est illustré Figure 7. Chaque serveur JOnAS déploie toutes les factory de beans de l'application et son propre serveur de nommage JNDI. Les serveurs JNDI sont donc dupliqués et se synchronisent via des communications de groupe afin de tous enregistrer les mêmes stub des factory. Ces stubs de factory sont des stubs CMI. On peut voir un stub CMI comme une enveloppe contenant tous les stubs RMI des réplicats de factory. Lorsqu'une factory est interrogée via un stub CMI, le stub CMI élit un de ses stubs RMI et l'utilise ensuite comme pour une invocation RMI conventionnelle. L'équilibrage de la charge est fourni par la phase d'élection du stub RMI. La factory retourne quand à elle des stub RMI pour les appels de méthodes sur les beans.

La conséquence immédiate du fait de répliquer les factory est qu'elles vont accéder de façon concurrente aux beans et c'est pour cette raison que les états des beans ne peuvent être conservés dans le cache (non distribué) de JOnAS. L'usage de CMI oblige donc à désactiver cette optimisation.

L'effet immédiat est donc que le nombre d'interactions vers la base de données est beaucoup plus important. A chaque accès, l'état du bean est chargé depuis la base, puis réécrit sur le disque s'il a été modifié. Fonctionnellement CMI fournit une sorte de RAID-1 applicatif où la ressource est clonée et les requêtes réparties sur chaque clone. Notre architecture de test de CMI (Figure 8) est la même que lors du test du cache (Figure 4) à l'exception près que nous avons dupliqué l'étage JOnAS.

5.2. Performances

La Figure 9 présente les résultats de cette expérience. Comme nous pouvons le constater, dans cette configuration, le système atteint un rendement maximum pour un nombre de clients voisin de 160 avec un débit de 23 requêtes par secondes.

Cette expérience montre qu'en doublant la puissance disponible sur l'étage JOnAS, les performances de CMI arrivent tout juste à égaler celle d'un seul serveur disposant de son cache. Ces résultats sont tout à fait cohérents avec ceux obtenus lors de notre évaluation des performances du cache JOnAS. En effet, le débit maximum enregistré ici est environ égal à deux fois celui enregistré lors de notre expérience avec un JOnAS sans cache (Figure 6). Ce test est donc positif dans la mesure où il montre que le passage à l'échelle de CMI est plutôt bon.

6. Le partitionnement par factory

6.1. Description

Dans cette section nous décrivons une solution à base de partitionnement par factory et nous la comparons à CMI. Avec CMI chaque JOnAS déploie toute l'application, l'idée ici est de ne déployer que des fragments de l'application sur chaque serveur de façon à ce que l'union de ces fragments compose l'application entière. Pour cela, la solution que nous proposons consiste à partitionner l'application en plusieurs ensembles de beans. Chaque bean ne sera que dans une partition et une seule et chaque partition sera déployée sur un serveur d'EJB unique. De cette façon les beans ne seront pas dupliqués et donc la gestion de leur cohérence ne posera pas de problème. En conséquence l'usage du cache ne sera pas prohibé. Dans cette expérimentation, le partitionnement est par factory, une factory n'étant déployée que sur un seul serveur.

Pour que cette solution fonctionne, il faut disposer d'une information permettant aux Servlets de connaître la répartition des factory sur les serveurs JOnAS. Cette information existe de façon implicite dans les stubs que les Factory enregistrent dans le service de nommage JNDI. En effet, un stub est capable de communiquer à travers le réseau avec le skeleton qui lui est associé. En déployant une architecture où tous les serveurs JOnAS partagent le même service JNDI (Figure 10), le routage des requêtes depuis les Servlets sera donc fait de façon automatique et implicite. Les Servlets n'ont besoin de connaître que la

façon d'interroger le service JNDI et en fonction de la factory demandée, la requête sera envoyée à la machine qui aura enregistré cette factory.

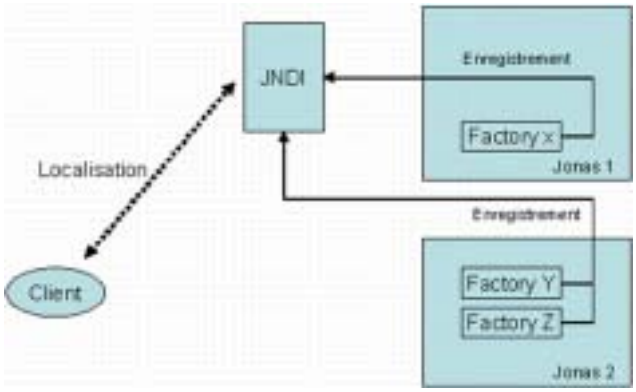

FIG. 10 – Principe du partitionnement des factory

D'un point de vue fonctionnel, cette approche s'apparente à du RAID-0 applicatif. La ressource JOnAS est dupliquée mais tous les réplicats ne sont pas équipotents. Dans ce procédé, la tolérance aux pannes est assurée à l'aide du mécanisme transactionnel utilisé. En cas de panne de l'un des serveurs JOnAS, toutes les transactions non validées sont annulées. Les beans déployés sur la machine en panne doivent alors être simplement redéployés sur une autre machine en réserve et l'application est à nouveau opérationnelle.

6.2. Performances

Les tests ont été effectués sur la même plate-forme (Figure 8) que ceux effectués pour CMI afin de fournir une base équitable aux deux solutions.

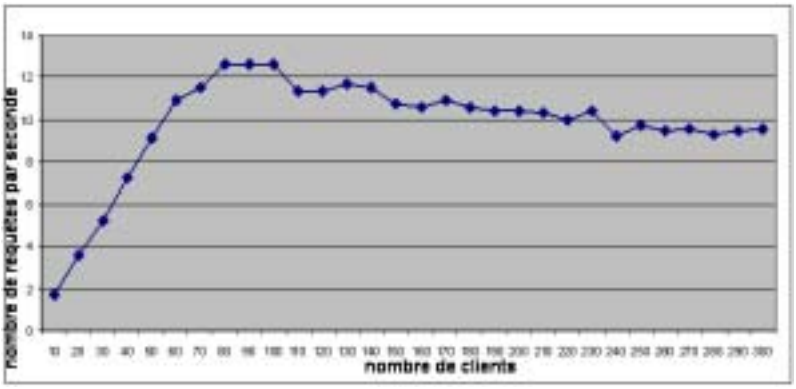

FIG. 11 – Débit avec deux serveurs EJB, partitionnement et cache activé

Comme on peut le voir sur la Figure 11, avec cette configuration le rendement maximum est atteint pour 90 clients avec un débit de 13 requêtes par secondes. Les performances sont assez loin de celles escomptées. Le débit maximum admissible par le système est presque moitié moins important que pour CMI.

Pour expliquer cet écart, examinons les facteurs variants entre CMI et notre solution. Le coût de la localisation d'un EJB depuis une Servlet doit être sensiblement équivalent puisque dans les deux cas il s'agit d'un appel distant. Seul l'implémentation du service JNDI varie et nous prendrons comme hypothèse que ce facteur est négligeable. L'écart de performance doit donc se situer au niveau de l'exécution sur JOnAS.

Dans le cas de CMI, chaque serveur JOnAS intègre son propre service JNDI alors que dans notre solution, ce service est séparé sur une autre machine. La conséquence est qu'à chaque appel sur ce service

depuis un bean (dans JOnAS), notre solution effectue une communication sur le réseau alors que dans le cas de CMI toutes ces communications sont locales.
Comme le montrent les Figures 12 et 13, le nombre de communications sur le réseau est nettement plus important dans le cas où les EJB sont partitionnés. Avec CMI, dès qu'une exécution a démarré sur un serveur EJB, tous les appels (à JNDI, à une factory ou à un bean) sont locaux.

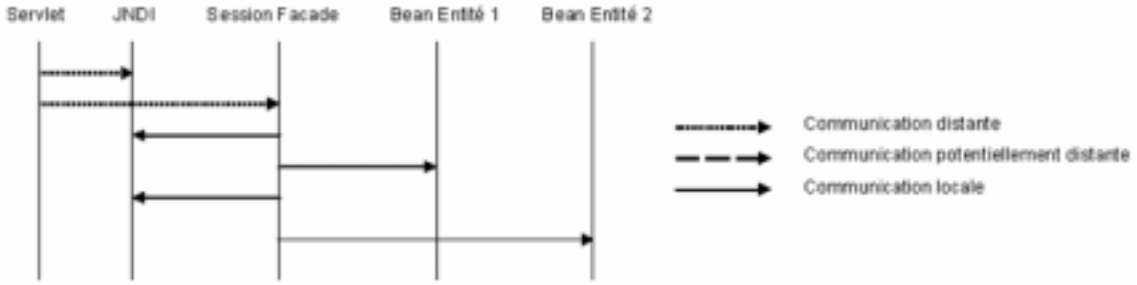

FIG. 12 – Séquence des communications d'une requête avec CMI

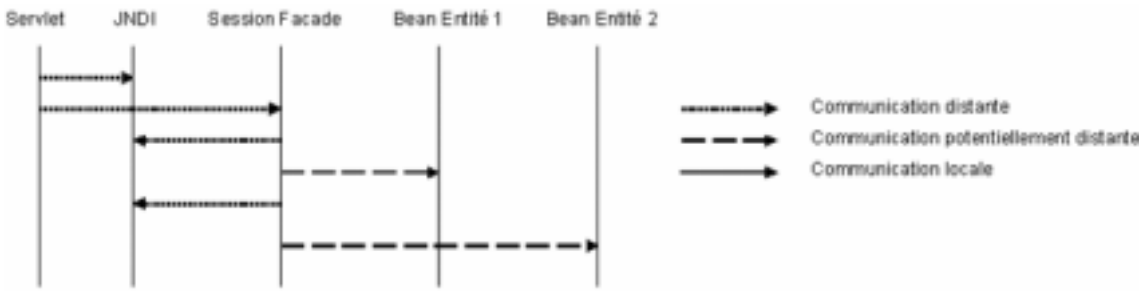

FIG. 13 – Séquence des communications d'une requête avec une partition des EJBs

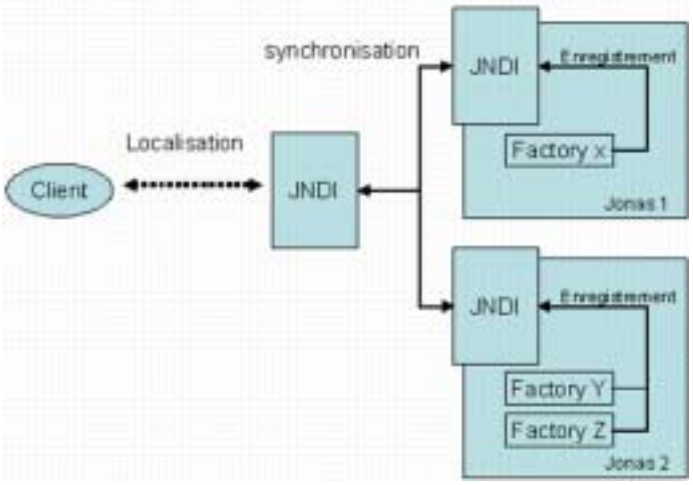

FIG. 14 – Partitionnement des factory et réplication de JNDI

6.3. Evolution de la solution

L'évolution en question consiste à éviter autant que possible de passer par le réseau (élément le plus coûteux en temps et en CPU). Pour cela nous avons répliqué l'état du service de nommage afin que chaque serveur JOnAS déploie son propre service JNDI. De cette façon, à chaque fois qu'un bean localise une factory, l'appel à JNDI est local (voir Figure 14). Lors de la localisation d'un bean, le nombre de

communications sur le réseau avec cette évolution est ramené à celui de CMI.

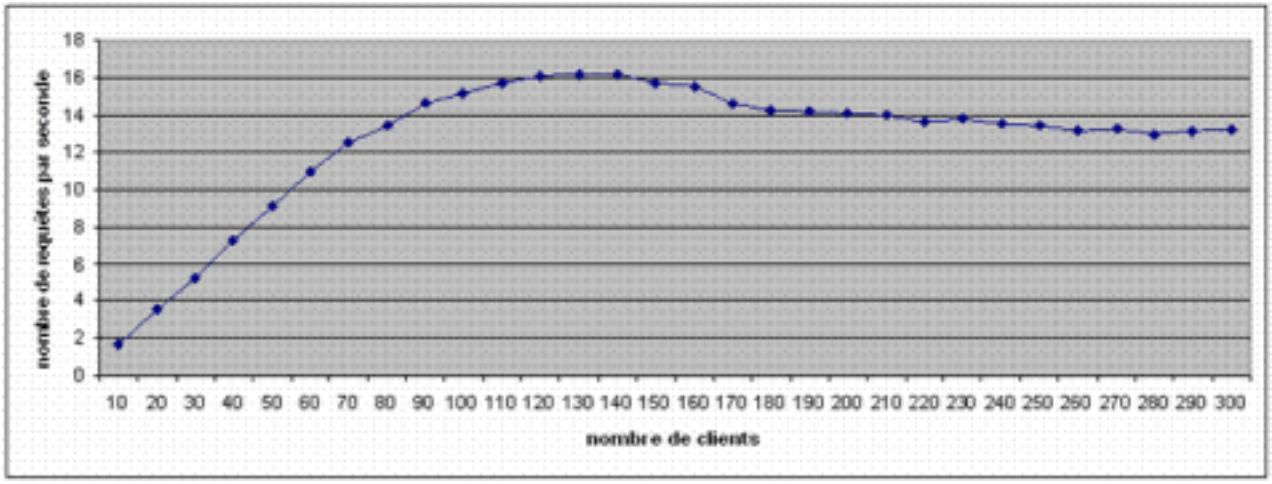

FIG. 15 – Débit avec deux serveurs EJB, partitionnement, JNDI dupliqué et cache activé

La figure 15 montre les performances obtenues avec cette solution sur notre plate-forme de test. On obtient un débit maximum pour un nombre de client proche de 130 avec 16 requêtes par secondes. Ce résultat reste encore inférieur à CMI. La raison est que le partitionnement par factory engendre un grand nombre d'appels distants entre les deux serveurs JOnAS, ce qui est très pénalisant. Cette pénalité n'est pas compensé par le fait d'avoir le cache activé.
Une meilleure stratégie de partitionnement (par bean et non par factory) permettrait d'améliorer ces résultats. Cependant, nous avons préféré étudier une piste bien plus prometteuse, à savoir une solution hybride entre CMI et le partitionnement.

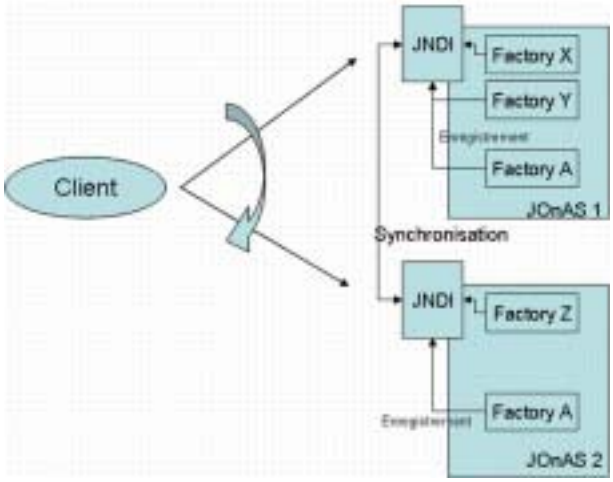

FIG. 16 – Partitionnement des données accédées en écriture et duplication des données accédées en lecture seule

7. Méthode hybride

7.1. Description

Le principe de cette solution est de distinguer quelles sont les données accédées par l'application en lecture, et celles accédées en écriture. Pour ce faire, une connaissance de l'application est indispensable. Toutes les données accédées uniquement en lecture sont dupliquées dans les caches tandis que les données accédées en lecture / écriture sont partitionnées. Cette solution superpose donc les deux politiques

décrites précédemment. La figure 16 montre une telle architecture. Les factory X, Y et Z sont partitionnées car accédés en écriture tandis que la factory A est dupliquée car accédée uniquement en lecture.
Le service JNDI retourne un stub CMI pour les factory en lecture seule et un stub RMI pour les factory en lecture / écriture. De cette façon toutes les requêtes accédant à un EJB en lecture seule sont réparties selon un algorithme de type round robin. A l'inverse une requêtes accédant un EJB en écriture est routée vers le serveur l'hébergeant. Le cache peut être activé.

7.2. Performances

Nous avons répliqué tout les beans session sans état qui par définition ne posent pas de problème au niveau de la gestion de leur cohérence. Ensuite nous avons analysé l'application RUBiS afin de déterminer quels sont les beans entités pouvant être répliqués. Nous avons identifié les beans entités Query, Category, Region, OldItem, Comment, Bid et BuyNow comme étant accédés en lecture seule. Nous avons partitionné les beans entités User, Item et IDManager. La figure 17 montre les performances obtenues avec cette solution sur notre plateforme de test.
On constate un débit maximum pour un nombre de clients proche de 180 avec 30 requêtes par secondes. Cette solution améliore entre 20 et 30du serveur dupliqué par rapport à CMI. L'inconvénient de cette solution vient du fait quelle n'est pas complètement générique. Cette duplication intelligente est dépendante des applications. Le programmeur de l'application doit marquer les beans qui sont en lecture seuls de ceux qui sont en lecture / écriture.

FIG. 17 – Débit avec deux serveurs EJB, CMI en lecture, partitionnement en écriture et cache activé

8. Comparaison aux travaux existants

La duplication de ressource (à tous les niveaux : disque, processus, machine, etc.) a été étudiées beaucoup plus extensivement dans le cadre de la tolérance aux fautes que dans le cadre du passage à l'échelle. Toutefois, passage à l'échelle et tolérance aux fautes sont étroitement liées (notamment dans les systèmes distribués) et se confondent dans la notion de *performabilité*. Ainsi les mécanismes mis en jeu dans la réplication pour la tolérance aux fautes amène souvent, par "effet de bord" ou par essence, à un meilleur passage à l'échelle. Ainsi, la réplication des disques durs en RAID1 [10] autorise une panne sur un disque mais améliore dans le même temps les performances globales du support. Ce constat a inspiré les travaux du projet C-JDBC [2] dont l'objectif est de permettre le passage à l'échelle des SGBDs Relationnels tout en assurant leur tolérance aux fautes. C-JDBC introduit la notion de disques redondants (Redundant Array of Inexpensive DataBase) calquée sur la définition du RAID. Le RAIDb1 permet de répliquer une base de données sur plusieurs machines. Les multiples répliquas sont ensuite maintenus en cohérence par un contrôleur C-JDBC dont les fonctions reprennent celle d'un contrôleur RAID.
La réplication de ressource ne permet d'obtenir le passage à l'échelle que dans la mesure où a) d'une part les répliquas sont idempotents, et d'autre part le traitement de requêtes lui-même n'est pas répliqué

(les requêtes sont aiguillées plutôt que dupliquées et leur traitement reste centralisé sur un seul serveur), b) le traitement d'une requête est distribué sur plusieurs serveurs, auquel cas les serveurs ne sont plus nécessairement idempotents. Ainsi les modèles de " réplication active " ou encore de " primaire - sauvegarde " apportent la tolérance aux fautes mais pas le passage à l'échelle des serveurs [4]. La duplication de serveur *sans état* pose peu de problème et est déjà mis en oeuvre avec succès par exemple dans les serveurs de pages Web statiques. Ces derniers peuvent être dupliqués simplement, la répartition de la charge entre les différents répliquas étant ensuite assurée par un Round Robin DNS ou un switch L4 [7][11]. Dans le cas de serveur *avec état*, il convient de gérer convenablement le partage de cet état entre les multiples instances de serveurs, les solutions actuelles procèdent : i) par diffusion de l'état sur tous les répliquas [5][8]. Dans ces cas la l'approche consiste à utiliser un protocole de communication de groupe (multicast IP, JGroup, etc.) permettant de synchroniser les serveurs entre eux. ii) par externalisation de l'état sur un support transactionnel accessible depuis toutes les instances de serveur [6]. Le choix du support en question influe énormément sur les performances du système [3]. L'approche la plus courante mais également la moins performante consiste à stoker l'état dans une base de données. D'autres solutions, exploitant la réplication en mémoire volatile, sont explorées afin d'améliorer les performances liées à la manipulation de cet état [9].

Notre solution propose une troisième alternative qui consiste à segmenter l'état afin de le partitionner sur un ensemble de noeuds et ainsi spécialiser les répliquas de serveur. Par conséquent l'état est distribué mais non répliqué, ce qui évite les surcoût liés à la gestion de la mise en cohérence d'un état dupliqué ou les surcoûts liés à un accès distant systématique.

9. Conclusion

La norme J2EE permet de concevoir des serveurs d'applications multi-tiers (où multi-niveaux). Un serveur J2EE est généralement composé de quatre tiers chacun pouvant s'exécuter sur des machines différentes : un tiers Web, un tiers serveur de Servlet , un tiers serveur EJB et un tiers serveur de base de données. L'évolution croissante de la charge (en terme de nombre de requête) à laquelle les serveurs J2EE doivent faire face amène à se poser la question du passage à l'échelle de telles architectures.

La démarche employée généralement consiste à dupliquer "entièrement" les différents étages d'un serveur J2EE. Cette démarche permet de ne pas avoir à distinguer les clones et simplifie ainsi l'aiguillage des requêtes. Cependant le problème posé par cette solution est le maintien en cohérence de l'état dupliqué dans un étage lorsqu'il est modifiable, ce qui est le cas pour les EJB. CMI, qui est la solution fournie dans le serveur EJB JOnAS, résout le problème de maintien de la cohérence en se synchronisant à travers la base de données (en forçant systématiquement les lectures et les écritures à se propager vers la base de données). La fonction de cache dans JOnAS est donc désactivée.

Dans le cadre d'une architecture J2EE en grappe, nous avons comparé les performances d'un unique serveur EJB à un serveur EJB dupliqué à l'aide de CMI. Nous avons observé et analysé le gain lié à l'utilisation de CMI. Nous avons présenté ensuite une première solution basée sur le partitionnement des beans de l'application sur différents serveurs EJB. L'évaluation de performance a montré que cette solution était moins intéressante que CMI. Différentes améliorations nous ont conduit à construire une solution hybride entre CMI et le partitionnement. Dans cette solution, les beans sont dupliqués intelligemment en fonction de l'application. Tous les beans accédés uniquement en lecture sont dupliquées avec CMI (et peuvent être mis en cache) tandis que les beans accédés en lecture / écriture sont partitionnées et gérés en exemplaire unique (et peuvent également être mis en cache). Cette solution améliore entre 20 et 30performances du serveur dupliqué par rapport à CMI.

Les perspectives de poursuite de ce travail concernent deux axes :

– Nous souhaitons affiner ces mesures en (1) évaluant ces stratégies avec un nombre plus grand de machines et en (2) étudiants différentes stratégies de partitionnement (notamment par beans au lieu de par factory).
– Nous avons étudié le passage à l'échelle au niveau des autres tiers, mais l'association des stratégies de chaque tiers au sein d'une architecture J2EE reste difficile et complexe à gérer. Il ne semble pas possible de combiner n'importe quelles stratégies aux différents étages.

Bibliographie

1. E. Cecchet, J. Marguerite and W. Zwaenepoel. Performance and Scalability of EJB Applications. In Proceedings of OOPSLA, Seattle (USA), November 4th–8th, 2002.
2. E. Cecchet, J. Marguerite and W. Zwaenepoel. C–JDBC : Flexible Database Clustering Middleware. In Proceedings of USENIX Annual Technical Conference, Freenix track, Boston, MA, USA, june 2004.
3. G. Gama, K. Nagaraja, R. Bianchini, R. P. Martin, W. Meira Jr.and T. D. Nguyen. State Maintenance and Its Impact on the Performability of Multi–tiered Internet Services. In Proceedings of the 23rd Symposium on Reliable Distributed Systems (SRDS), Florianopolis, Brazil, October 2004
4. R. Guerraoui, A. Schiper. Fault–Tolerance by Replication in Distributed Systems. Département d'Informatique Ecole Polytechnique Fédérale de Lausanne, 1996.
5. Jakarta Tomcat Servlet Engine – http : / / jakarta.apache.org / tomcat /
6. JOnAS Open Source EJB Server – http : / / www.objectweb.org
7. E. Katz, M. Butler, and R. McGrath, A scalable http server : The ncsa prototype. Computer Networks and ISDN systems, 27 :155–164, 1994.
8. S. Labourey. Load Balancing and Failover in the JBoss Application Server. Sept 2003 http : / / www.clustercomputing.org / index.jsp ?pag 5-2-labourey.shtml
9. B. C. Ling, Emre Kiciman, Armando Fox. Session State : Beyond Soft State. In Proceedings of the 1st USENIX / ACM Symposium on Networked Systems Design and Implementation (NSDI 2004), San Francisco, CA, March 29–31, 2004
10. D. A. Patterson, G. Gibson and R.H. Katz. A Case for Redundant Arrays of In–expensive Disks (RAID). In Proceedings of the ACM SIGMOD International Conference onManagement of Data, Chicago, IL, USA, 109-116, 1988.
11. S. Sudarshan, R. Piyush. Link level Load Balancing and Fault Tolerance in NetWare 6. NetWare Cool Solutions Article, March 2002.
12. Sun Microsystems – Enterprise Java Beans Specifications – http : / / java.sun.com / j2ee /
13. Sun Microsystems – Java Naming and Directory Interface (JNDI) – http : / / java.sun.com / products / jndi /
14. The Apache Software Foundation – http : / / www.apache.org /