Tolérance aux fautes dans les applications J2EE: Une solution scalable et générique

Sara Bouchenak, Sacha Krakowiak, Noël de Plama

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Tolérance aux fautes dans les applications J2EE : Une solution scalable et générique

Sara Bouchenak, Sacha Krakowiak, Noël De Plama

N° 5463

Janvier 2005

THÈME 1

ISSN 0249-6399 ISRN INRIA/RR--5463--FR

Tolérance aux fautes dans les applications J2EE : Une solution scalable et générique

Sara Bouchenak, Sacha Krakowiak, Noël De Plama

Thème 1 — Réseaux et systèmes
Projet Sardes

Rapport de recherche n° 5463 — Janvier 2005 — 26 pages

Résumé : La plate-forme J2EE définit un modèle pour développer des applications réparties dans une architecture multi-tiers, telles que les applications Web de commerce électronique. Avec la multiplication des couches logicielles et intergicielles, les applications Internet sont encore plus sujettes aux défaillances ; ces défaillances étant, pour la majorité, causées par des fautes logicielles ou des fautes de l'opérateur. Dans cet article, nous proposons un système de tolérance aux fautes qui traite les fautes logicielles et les fautes de l'opérateur dans les applications J2EE, en masquant l'occurrence de ces fautes. L'approche proposée est à la fois *scalable*, *automatique* et *générique* dans le sens où elle : (i) supporte la montée en charge des applications, (ii) s'applique automatiquement à toute application J2EE, (iii) ne modifie pas manuellement le code de l'application, (iv) ne requiert pas d'aide du programmeur de l'application, (v) ne modifie pas la mise en œuvre du middleware sous-jacent.

Mots-clés : tolérance aux fautes, applications J2EE, grappes de serveurs, programmation par aspects

Fault-tolerance in J2EE applications: A scalable and generic solution

Abstract: The J2EE platform defines a model for developing multi-tier distributed applications, such as e-commerce applications. With the multiplication of software and middleware levels, Internet services are becoming more sensitive to outages. In this context, a large portion of outages in Internet services is due to software and operator faults; those types of faults being rarely tackled. In this paper, we propose a fault-tolerance system that tackles software and hardware faults in J2EE applications, using fault masking mechanisms. The advantages of the proposed approach are threefold, *scalability*, *automatism* and *genericity*, in the sense that: (i) it supports an increasing load of applications, (ii) it can be automatically applied to any J2EE application, (iii) it does not require a manual modification of the application code, (iv) it does not require a help from the application programmer, (v) it does not modify the underlying middleware.

Key-words: fault-tolerance, J2EE applications, server clusters, aspect-oriented programming

1 Introduction

Un des modèles standards pour le développement d'applications Internet est l'architecture multi-tiers (multi-couches) où : (i) une partie présentation est représentée par un serveur Web, (ii) suivie d'une partie application qui se charge d'exécuter la fonctionnalité propre à l'application, puis (iii) une partie persistance qui se charge de stocker les données non éphémères dans une base de données [9]. Avec la multiplication des couches logicielles et le fait que ces couches deviennent de plus en plus complexes, les applications Internet sont encore plus sujettes aux défaillances [21, 22]. Ces défaillances peuvent être des pannes de durée plus ou moins longue [14, 19], et peuvent avoir pour origine diverses causes telles que des fautes de l'opérateur (erreurs de configuration) ou des fautes logicielles (erreurs transitoires dépendant de l'environnement, manque de ressources physiques telles que la mémoire) [22, 21].

J2EE est une plate-forme qui permet de concevoir et de mettre en œuvre des applications Java dans des architectures multi-tiers [26]. Pour garantir une plus haute disponibilité des applications J2EE, chaque *tier* (ou couche de l'architecture multi-tiers) est dupliqué dans une grappe de serveurs. Dans ce contexte, les solutions existantes de tolérance aux fautes des applications J2EE sont principalement basées sur deux techniques : le masquage de pannes franches ou le redémarrage de composants logiciels fautifs.

La première technique consiste à masquer une panne franche (arrêt d'un serveur de la grappe) de la manière suivante : lorsqu'une requête est envoyée à une grappe de serveurs dupliqués, les serveurs en pannes sont ignorés et la requête est traitée par un des serveurs disponibles [24, 16, 10]. Cette technique ne traite que les pannes franches de serveurs et ne considère que les pannes qui surviennent avant l'envoi de requêtes aux serveurs ; elles ne traite pas les autres types de pannes qui peuvent survenir au cours du traitement d'une requête sur un serveur (par exemple une faute temporelle due à un dépassement de délai - *timeout* -, une erreur transitoire dépendant de l'environnement).

La seconde technique de tolérance aux fautes des applications J2EE traite, plus particulièrement, les fautes logicielles qui causent la dégénérescence du système et qui le mettent dans un état "irrécupérable" (manque de mémoire, inter-blocage de processus). Dans ce cas, l'arrêt puis le redémarrage (*reboot*) des composants logiciels fautifs reste l'unique alternative pour retrouver un état cohérent [7].

Entre l'approche qui consiste à ne traiter que les fautes de type pannes franches et l'approche qui consiste à traiter les fautes logicielles en arrêtant puis redémarrant les composants fautifs, nous proposons une solution intermédiaire qui :

i traite les fautes logicielles *et* les fautes de l'opérateur, puisque ces deux types de fautes représentent la majorité des fautes dans les applications Internet [21, 22],

ii masque l'occurrence d'une faute aux clients de l'application,

iii traite une faute qui survient lors de l'exécution d'une requête sur un serveur en tentant de *rejouer* la même requête sur d'autres serveurs de la grappe ; ceci reviendrait, en quelque sorte, à mimer le comportement d'un utilisateur humain qui, face à une requête qui échoue sur un serveur, tente de re-exécuter cette même requête sur un autre serveur.

Nous avons conçu le système JSR (*Java Server Recovery*), un système de tolérance aux fautes d'applications J2EE. Les principales contributions scientifiques de cet article sont :

- La *scalabilité* du système de tolérance aux fautes proposé dans le sens où le système JSR supporte la montée en charge dans une grappe de serveurs, à la fois en termes de nombre de requêtes traitées par les serveurs et en termes de nombre de machines constituant la grappe.
- La *généricité* du système de tolérance aux fautes proposé, dans le sens où le système JSR peut être automatiquement intégré à toute application J2EE :
  - sans modifier manuellement le code de l'application,
  - sans avoir besoin de connaître la sémantique de l'application (ce qui requerrait une aide du programmeur de l'application),
  - sans modifier la mise en œuvre du middleware sous-jacent (serveurs Web, serveurs d'application Servlet/EJB, serveurs de base de données).

Pour garantir la scalabilité et la généricité du système JSR, nous avons combiné une gestion décentralisée de la tolérance aux fautes à l'utilisation de techniques de programmation par aspect [13, 18]. Pour valider l'approche proposée, nous considérons en particulier deux applications J2EE de benchmark : l'application TPC-W qui met en œuvre une librairie en ligne de type *Amazon.com* [28] et l'application RUBiS qui modélise un site Web de vente aux enchères tel que *eBay.com* [1]. Une première analyse de l'application TPC-W montre, par exemple, que des fautes de l'opérateur telles qu'un mauvais déploiement de l'application ou une erreur de configuration de l'application génèrent des fautes qui sont visibles par les clients. L'intégration automatique du système JSR à cette application permettra donc de masquer ces fautes aux clients.

Le reste de l'article est organisé comme suit. La section 2 rappelle le contexte dans lequel se situent nos travaux et quelques notions nécessaires à la compréhension de l'article. Les sections 3 et 4 décrivent respectivement les principes de conception et de mise en œuvre du système de tolérance aux fautes proposé. La section 5 présente des études de cas et la section 6 décrit l'état de l'art. Finalement, la section 7 présente nos conclusions et perspectives.

2 Contexte

Dans cette section, nous rappelons quelques notions nécessaires à la compréhension du reste de l'article, en particulier : les applications J2EE et la tolérance aux fautes dans de telles applications.

2.1 Architecture J2EE

La plate-forme *Java 2 Platform, Enterprise Edition* (J2EE) définit un modèle pour développer des applications réparties dans une architecture multi-tiers [26], telles que les applications Web de commerce électronique. L'architecture de ces applications suit généralement le schéma défini dans la Figure 1 : des clients envoient des requêtes à un serveur Web qui héberge des documents Web statiques (par exemple des pages HTML), ces requêtes sont ensuite transférées à un serveur d'application qui se charge d'exécuter la fonctionnalité de l'application et de générer des documents Web dynamiquement, les informations persistantes utilisées par le serveur d'application sont quant à elles stockées sur une base de données.

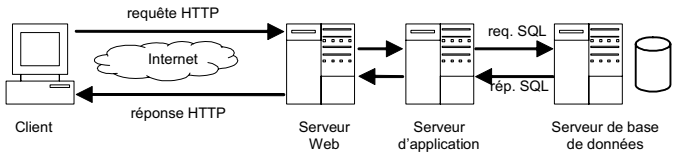


FIG. 1 – Architecture d'applications J2EE

Dans une telle architecture, le traitement d'une requête passe par les étapes suivantes. Lorsqu'un client envoie une requête qui désigne un document statique, le serveur Web retourne directement le document au client. Si la requête fait référence à un document dynamique, le serveur Web transmet cette requête au serveur d'application. Le serveur d'application exécute ensuite un ou plusieurs composants logiciels (par exemple des Servlets, EJB) qui effectuent des requêtes SQL sur la base de données ; un document Web sera alors dynamiquement généré puis retourné au client. La Figure 2 donne un exemple de programme principal d'un composant logiciel Servlet.

2.2 Tolérance aux fautes

Dans le domaine de la tolérance aux fautes, une *faute* est définie comme étant une cause qui provoque une erreur ; et une *erreur* correspond à un état corrompu dans lequel se trouve le système ou l'application, un tel état pouvant engendrer alors une défaillance. Une *défaillance*, quant à elle, se manifeste sous la forme d'une déviation du comportement de l'application par rapport au comportement correct défini par la spécification de l'application. La construction

```
public void doGet(HttpRequest  request, HttpResponse response)
    throws IOException, ServletException {

    // Execute SQL queries on the back-end database (or via EJBs)
    ...

    // Generate a web document
     ...

    // Return the web document to the client
    ...
}
```

FIG. 2 – Exemple de composant logiciel Web

d'un système de tolérance aux fautes est principalement basée sur deux mécanismes : la détection de l'occurrence d'une faute, puis le traitement de la faute détectée.

Il existe différents types de fautes : (i) des fautes matérielles causant, par exemple, l'arrêt d'un serveur (panne franche), (ii) des fautes logicielles telles que des fautes transitoires dues à un état particulier dans lequel se trouve un serveur à un moment donné, des fautes temporelles dues à un dépassement de délai ou un bug dévoilé dans un cas particulier, (iii) et finalement des fautes de l'opérateur humain (programmeur, administrateur) telles qu'une erreur de configuration du serveur ou un mauvais déploiement de l'application sur les serveurs. Par ailleurs, des études récentes ont montré que les fautes logicielles et les fautes de l'opérateur représentaient une grande partie des fautes dans les services Internet [21, 22].

Dans le contexte particulier des applications J2EE, une des techniques utilisées pour fournir la tolérance aux fautes est la duplication des serveurs dans une grappe de machines [1], ceci est illustré par la Figure 3. Cette solution définit généralement un composant particulier que nous appellerons *proxy* et qui se trouve en entrée de la grappe de serveurs dupliqués. Ce proxy a un premier rôle de répartition de charge : il réceptionne les requêtes des clients et les aiguille vers le bon serveur en utilisant un algorithme de répartition de charge approprié (aléatoire, tourniquet ou *Round-Robin*, etc.). Le composant proxy a également pour rôle de gérer les pannes franches (exemple : fautes matérielles) d'un serveur. Ici, la détection d'une faute de type panne franche est réalisée en utilisant des mécanismes apparentés au *ping* ; quant au traitement de la faute, il consiste tout simplement à ignorer le serveur "fautif" et à aiguiller les requêtes vers un des autres serveurs disponibles dans la grappe. Ce traitement

[1] Une autre motivation de la duplication de serveurs dans une grappe de machines est la montée en charge (*scalabilité*) en termes de nombre de requêtes traitées par les serveurs et en termes de nombre de machines constituant la grappe.

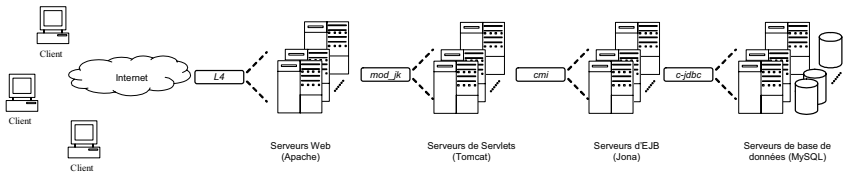


FIG. 3 – Architecture multi-tiers J2EE sur grappes de serveurs dupliqués

de fautes revient donc à masquer la panne d'un serveur à l'utilisateur en redirigeant, automatiquement, sa requête vers un autre serveur. Parmi les solutions existantes de gestion de pannes franches dans des grappes de serveurs dupliqués, nous pouvons citer le système c-jdbc pour une grappe de serveurs de bases de données [10], cmi pour une grappe de serveurs d'EJB Jonas [16], le module mod_jk pour une grappe de serveurs de Servlets Tomcat [24], le switch L4 pour une grappe de serveurs tels que des serveurs Web Apache dans notre exemple [25] (voir Figure 3).

3 Conception d'un système de tolérance aux fautes pour grappes de serveurs J2EE

3.1 Objectifs

Notre objectif est de concevoir et de mettre en œuvre un système de détection et de traitement de fautes logicielles et de fautes de l'opérateur dans les serveurs d'applications J2EE. Un tel système viendrait en complément aux solutions existantes qui, elles, ne traitent que les fautes de type panne franche, comme par exemple le module mod_jk pour une grappe de serveurs de Servlets Tomcat [24] et le système cmi pour une grappe de serveurs d'EJB Jonas [16]). Deux propriétés principales doivent être garanties par le système de tolérance aux fautes proposé :

- **Scalabilité.** Le système de tolérance aux fautes proposé doit être *scalable*, c'est-à-dire supporter la montée en charge dans la grappe de serveurs, à la fois en termes de nombre de requêtes traitées par les serveurs et en termes de nombre de serveurs constituant la grappe. En effet, la scalabité est une des motivations de la mise en grappes des serveurs ; il ne faudrait pas que la prise en compte de la tolérance aux fautes viole cette propriété des grappes de serveurs.
- **Généricité.** Le système de tolérance aux fautes proposé doit être *générique* pour l'application et pour le middleware sous-jacent. Autrement dit, le système de tolérance aux fautes doit pouvoir être automatiquement intégré à une application J2EE sans avoir besoin de :
  - modifier manuellement le code de l'application,
  - connaître la sémantique de l'application (ce qui requerrait une aide du programmeur de l'application),
  - modifier la mise en œuvre du middleware sous-jacent (serveurs J2EE).

3.2 Principes de conception

Le traitement d'une faute consisterait ici à *masquer* la faute à l'utilisateur, par des tentatives de re-exécution de la requête fautive sur d'autres serveurs pairs. Un tel traitement de fautes est motivé par le fait que l'occurrence d'une faute peut être due à un état particulier dans lequel se trouve le serveur (par exemple, une surcharge du serveur ou une mauvaise configuration du serveur), et que l'exécution de la même requête sur un autre serveur de la grappe (qui se trouverait dans un autre état) pourrait alors se terminer correctement sans générer de faute. Ce traitement de fautes par masquage revient, en quelque sorte, à *mimer* le comportement d'un utilisateur humain qui, face à une requête qui échoue sur un serveur, tente de re-exécuter cette même requête sur un autre serveur pair.

La Figure 4 décrit le schéma général du système de tolérance aux fautes proposé. Ici, à chaque serveur sont associés deux autres serveurs pairs. Lorsqu'une requête échoue sur un serveur à cause d'une faute, la même requête est rejouée sur un premier serveur pair. Si aucune faute n'est générée sur ce serveur pair, la réponse est renvoyée à l'utilisateur ; sinon,

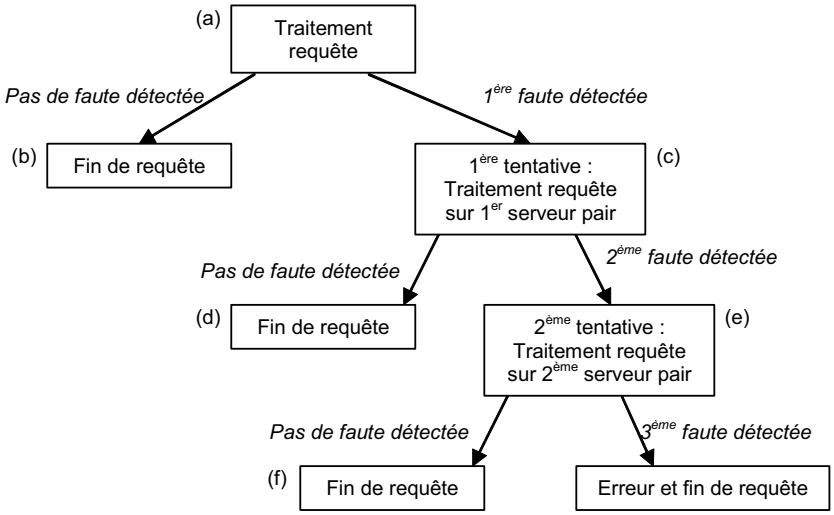


FIG. 4 – Schéma général du système de tolérance aux fautes

la même requête est rejouée sur un second serveur pair et le résultat de celle-ci est renvoyé à l'utilisateur (soit un résultat correct, soit le résultat d'erreur dû à une troisième faute).

Les choix de conception du système de tolérance aux fautes proposé sont principalement guidés par les deux objectifs de scalabilité et de généricité énoncés précédemment :

- **Scalabilité du système de tolérance aux fautes.** Pour construire un système de tolérance aux fautes qui soit scalable, un choix de conception principal était d'éviter toute approche centralisée qui pourrait constituer un goulot d'étranglement lorsque le nombre de requêtes augmente et, à plus forte raison, lorsque le nombre de serveurs constituant la grappe augmente. Nous avons donc opté pour une gestion décentralisée des fautes, autrement dit, une gestion décentralisée de la détection et du traitement des fautes. En effet, à chaque composant logiciel (exemple Servlet Java) qui traite une requête sur un serveur J2EE est associée la gestion de la détection et du traitement de fautes qui peuvent survenir dans ce composant.
- **Généricité du système de tolérance aux fautes.** Pour fournir un système de tolérance aux fautes des serveurs J2EE qui soit générique, c'est-à-dire une conception du système qui soit indépendante de l'application et de l'implantation du middleware sous-jacent, nous avons opté pour l'utilisation de techniques de programmation par aspects.

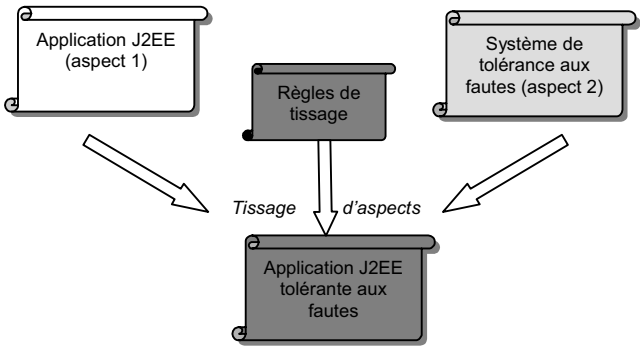


FIG. 5 – Intégration automatique de la tolérance aux fautes

3.3 Programmation par aspects

Nous rappelons ici quelques notions de base de la programmation par aspects (AOP - *Aspect-Oriented Programming*), une approche que nous avons suivie pour la conception du système de tolérance aux fautes proposé [13, 18]. Une des motivations de la programmation par aspects est de pouvoir intégrer, de façon plus ou moins automatique, des propriétés non fonctionnelles ne faisant pas partie de la logique de l'application (par exemple la tolérance aux fautes) à une application. La programmation par aspects est une méthode de programmation qui définit des concepts et des mécanismes pour une conception modulaire des différents *aspects* [2] (fonctionnalités) constituant un système. En effet, avec la programmation par aspects, les différents aspects d'un système sont mis en œuvre séparément dans des modules différents. Pour construire le système final, la programmation pas aspects fournit un moyen de spécifier la relation entre les différents aspects, autrement dit comment les différents modules sont tissés ensemble. La Figure 5 illustre un exemple d'application J2EE (premier aspect) et d'un système de tolérance aux fautes (2ème aspect) qui sont mis en œuvre séparément, puis automatiquement tissés ensemble pour construire une application J2EE tolérante aux fautes.

AspectJ est un environnement de programmation par aspect qui fournit un langage et un ensemble d'outils associés à ce langage ; le langage et les outils AspectJ sont dédiés aux aspects mis en œuvre dans le langage de programmation Java [3].

[2] La logique de l'application est vue comme un *aspect* et les propriétés non fonctionnelles de l'application comme d'autres *aspects*.

```
(a)  aspect ServletExecution {
(b)    // Pointcut definition
(c)    pointcut doGetExecution() :
(d)        execution(
(e)              void HttpServlet+.doGet(
(f)                   HttpServletRequest, HttpServletResponse)) ;
(g)    // Advice definition
(h)    before() : doGetExecution() { ... traitement particulier   ...}
(i)  }
```

FIG. 6 – Exemple de *pointcut* et d'*advice*

Le langage AspectJ définit un ensemble de *join points* qui sont tous les points "capturables" de l'exécution d'un programme Java. De tels *join points* incluent, par exemple, les appels et exécutions de méthodes, les appels et exécutions de constructeurs, les accès aux attributs en lecture ou en écriture, les traitants d'exceptions. Les *pointcuts* permettent, plus particulièrement, de capturer certains *joint points*. Un *advice*, quant à lui, permet de spécifier les actions qui sont effectuées au niveau d'un *pointcut*, comme par exemple effectuer un traitement avant ou après le *pointcut*. La Figure 6 présente un exemple de *pointcut* et d'*advice* dans le langage AspectJ. Les lignes (c) à (f) définissent un *pointcut* nommé *doGetExecution* qui capture l'exécution de la méthode *doGet* de la classe Java standard *HttpServlet* et ses sous-classes [3] et qui prend un premier paramètre de type *HttpServletRequest* et un second paramètre de type *HttpServletResponse*. La ligne (h) déclare un *advice* qui effectue des traitements avant le *pointcut doGetExecution*, donc avant toute exécution de la méthode *doGet* de la classe *HttpServlet*. Autrement dit, la mise ne œuvre décrite ci-dessus s'adresse à toute application J2EE qui a des Servlets comme composants d'entrée (*front-end*) ; ceux-ci peuvent accéder aux données persistantes soit en adressant des requêtes SQL à la base de données, soit via des EJB.

Par ailleurs, les *pointcuts* et *advices* qui définissent les règles de tissage à appliquer sont définis dans un (ou plusieurs) module(s) séparé(s) des modules des aspects eux-mêmes. L'outil d'AspectJ nommé *ajc* effectue le tissage automatique des différents aspects en suivant les règles de tissage spécifiées et construit ainsi l'application finale dans laquelle les différents aspects sont pris en compte.

[3] Dans la Figure 6, le signe + qui suit *HttpServlet* désigne les sous-classes de cette classe.

4 Mise en œuvre du système de tolérance aux fautes

Nous décrivons, dans cette section, les principes de mise en œuvre du système de tolérance aux fautes proposé : JSR (Java Self Recovery). Nous présentons tout d'abord les principes de mise en œuvre d'une solution scalable, puis les principes d'une solution générique, avant de présenter une discussion et l'état courant de la réalisation. Par ailleurs, la mise en œuvre du système de tolérance aux fautes JSR est illustrée, dans la suite, au travers de serveurs de Servlet Java.

4.1 Solution scalable

La scalabilité du système JSR est fournie grâce à une gestion décentralisée de la tolérance aux fautes, où la détection et le traitement des fautes sont faits au niveau de chaque composant logiciel qui est responsable de traiter une requête sur un serveur J2EE. La mise en place de la détection et du traitement de fautes dans le système JSR est décrite ci-dessous.

4.1.1 Détection de fautes

Détecter une faute dans un composant logiciel Java revient à détecter l'occurrence d'une exception Java [4]. Par exemple, dans le cas d'un serveur de Servlets, le traitement d'une requête par une Servlet se fait par la méthode principale *doGet* ou par la méthode *doPost* (ces deux méthodes réalisent respectivement le HTTP GET et le HTTP POST). La détection d'une faute logicielle qui surviendrait dans une Servlet serait mise en œuvre telle qu'illustrée par la Figure 7 (cet exemple illustre la mise en œuvre pour la méthode *doGet* ; mais une mise en œuvre équivalente serait appliquée à la méthode *doPost*). Dans cet exemple, les parties grisées représentent le code propre à la détection de fautes. Ainsi, détecter une faute dans une Servlet reviendrait à entourer le code de sa méthode principale d'un bloc Java *try-catch* qui permet de capturer les exceptions levées et non traitées.

4.1.2 Traitement de fautes par masquage

Pour compléter l'exemple précédemment donné en Figure 7, la Figure 8 illustre le traitement des fautes détectées dans une Servlet. De la même manière, les parties grisées représentent le code relatif au traitement de fautes. Dans cet exemple, le bloc Java *catch* permet de spécifier le traitement des fautes, et en particulier, le traitement des fautes par masquage. Lorsqu'une faute survient, pour la première fois dans une Servlet, le traitement de la faute consiste à effectuer jusqu'à deux tentatives supplémentaires d'exécution de la même requête sur des serveurs pairs (lignes 12 à 16 pour la première tentative, puis lignes 19 à 22 pour la seconde tentative). Les lignes 24 à 27 correspondent au cas où une tentative de re-exécution

[4] Une exception qui est levée et non capturée par un bloc Java *catch*, puisque toute exception capturée représente un cas qui fait partie de la fonctionnalité du composant logiciel de l'application et n'est donc pas considérée comme une faute.

```
// Fault detection
public void doGet(HttpRequest  request, HttpResponse response)
    throws IOException, ServletException {
    try {
        // Normal execution of the software component
        ...
    }
    catch (Exception e) { // A fault occurred
    }
}
```

FIG. 7 – Détection de fautes dans une Servlet Java

de la requête sur un serveur pair échoue ; le serveur d'origine [5] est alors notifié de l'occurrence d'une autre faute pour qu'il puisse, éventuellement, effectuer une autre tentative de re-exécution de la requête sur un second serveur pair. Il faut donc pouvoir différencier une requête s'exécutant pour la première fois sur un serveur d'origine d'une re-exécution de requête sur un serveur pair. Pour ce faire, un paramètre supplémentaire nommé *JSR_RETRY* est ajouté à la requête destinée à un serveur pair (ligne 14 à destination du premier serveur pair, puis ligne 20 à destination du second serveur pair).

Par ailleurs, il faudrait disposer d'un mécanisme de *gestion des serveurs* qui attribuerait des serveurs pairs à chaque serveur de la grappe de serveurs dupliqués. Un tel mécanisme peut, par exemple, se présenter sous la forme d'un service accessible à distance et connu de tous les serveurs d'une grappe ; ce service fournirait une fonction d'attribution de serveurs pairs à un serveur. L'algorithme d'attribution des serveurs pairs peut être aussi simple qu'un tirage aléatoire, ou se baser sur des fonctions plus élaborées telles que l'utilisation de la topologie du réseau pour attribuer des serveurs pairs *voisins* d'un serveur origine. Un serveur connaîtrait ainsi ses serveurs pairs à partir du gestionnaire des serveurs, soit au lancement du serveur, soit lors de l'occurrence de la toute première faute sur ce serveur. Par ailleurs, le gestionnaire des serveurs peut être dupliqué pour tolérer ses propres pannes et fournir ainsi une continuité de service du système de tolérance aux fautes lui-même.

4.2 Solution générique

On souhaiterait, à présent, mettre en œuvre le système scalable JSR présenté en section 4.1 de façon à ce qu'il soit générique, c'est-à-dire sans exiger l'intervention du programmeur de l'application ni la modification du middleware sous-jacent. La mise en place du système JSR, de façon générique, pour les applications J2EE est basée sur ce qui suit :

- *Déterminer de manière générique les points de l'application où des traitements relatifs à la tolérance aux fautes doivent être effectués.* Par exemple, il faudrait avoir la possibi-

[5] Le serveur d'origine est le serveur sur lequel survient l'erreur la première fois.

```
(1)  // Fault management
(2)  public void doGet(HttpRequest  request, HttpResponse response)
(3)       throws IOException, ServletException {
(4)       try {
(5)            // Normal execution of the software component
(6)            ...
(7)       }
(8)       catch (Exception e) {
(9)            // Captured fault is managed
(10)           String retry = request.getParameter("JSR_RETRY");
(11)           If (retry == null) {
(12)                try {
(13)                     // First time the error occurs: First retry
(14)                     … Include JSR_RETRY as a request parameter …
(15)                     … Send request to first peer server …
(16)                     … Return response to client …
(17)                }
(18)                catch (Exception e2) {
(19)                     // Second time the error occurs: Second retry
(20)                     … Include JSR_RETRY as a request parameter …
(21)                     … Send request to second peer server …
(22)                     … Return response to client …
(23)                }
(24)           } else { // Retry request received by a peer server
(25)                 // Not the first time the fault occurs: fault managed by the origin server
(26)                throw e1
(27)           }
(28)      }
(29) }
```

FIG. 8 – Traitement de fautes dans une Servlet Java

```
(1) // Main method in a Java Servlet
(2) pointcut mainMethodExecution(HttpRequest  request, HttpResponse response) :
(3)       (execution (void HttpServlet+.doGet(HttpRequest, HttpResponse)
(4)             throws IOException, ServletException
(5)       || execution (void HttpServlet+.doPost(HttpRequest, HttpResponse)
(6)             throws IOException, ServletException)
(7)       && args(request, response);

(8) // Top-level main method in a Java Servlet
(9) pointcut servletEntryPoint(HttpRequest  request, HttpResponse response) :
(10)      mainMethodExecution(request, response)
(11)            && !cflowbelow(mainMethodExecution(HttpRequest, HttpResponse));
```

FIG. 9 – Capture des points d'entrée d'une Servlet Java

lité de capturer de façon automatique les points d'entrées principaux de l'application, (les méthodes *doGet* et *doPost* d'un composant logiciel Servlet).

- *Automatiquement intégrer à ces points le traitement de la détection et du masquage de fautes.* Il faudrait donc avoir la possibilité d'entourer automatiquement le code principal d'un composant logiciel applicatif d'un bloc Java *try-catch* qui se chargerait de la détection et du traitement de fautes, tels que décrits dans la Figure 8.

La mise en œuvre générique de JSR est basée sur des techniques de programmation par aspects et, plus précisément, sur le langage AspectJ. Ainsi, la capture des points d'entrée d'une Servlet Java avec AspectJ est décrite par la Figure 9. Dans les lignes 1 à 7, un *pointcut* nommé *mainMethodExecution* est défini comme étant l'exécution de toute méthode principale d'une Servlet, c'est-à-dire les méthodes *doGet* ou *doPost*. Par ailleurs, étant donné qu'une méthode *doGet* peut elle-même appeler une méthode *doPost* (et inversement), il faudrait, dans ce cas, ne capturer que la première méthode qui constitue le point d'entrée effectif à la Servlet. Dans les lignes 8 à 11, un *pointcut* nommé *servletEntryPoint* est défini comme étant l'exécution d'une méthode principale de Servlet (*doGet* ou *doPost*) qui ne ferait pas déjà partie du flot d'exécution d'une autre méthode principale (et qui constituerait donc le point d'entrée effectif à la Servlet).

Dans les points d'entrée ainsi capturés, il faudrait automatiquement intégrer le traitement relatif à la détection et au masquage de fautes. La Figure 10 décrit la mise en œuvre avec AspectJ de la transformation automatique de composants applicatifs de type Servlet. Ici, un *advice* de type *around* est défini autour du point d'entrée d'une Servlet (autour du *pointcut servletEntryPoint* précédemment défini). Ceci permet d'entourer l'exécution normale du point d'entrée d'une Servlet d'un bloc Java *try-catch* pour la prise en compte de la tolérance aux fautes. Ainsi, les lignes 5 à 6 de la Figure 10 correspondent à l'exécution normale d'une Servlet, les lignes 8 à 11 de la Figure 10 correspondraient aux lignes 8 à 28 de la Figure 8. a mise en œuvre générique de JSR est basée sur des techniques de programmation par aspects et, plus précisément, sur le langage AspectJ. Ainsi, la capture des points d'entrée

```
(1) around(HttpRequest  request, HttpResponse response)
(2)       throws IOException, ServletException :
(3)            servletEntryPoint (request, response) throws IOException, ServletException {
(4)       try {
(5)            // Normal execution of the software component
(6)            proceed(request, response);
(7)       }
(8)       catch (Exception e1) { // Fault detection
(9)            // Fault management as described in the catch block of Erreur ! Source du
renvoi introuvable.
(10)           …
(11)      }
(12) }
```

FIG. 10 – Mise en oeuvre générique de la tolérance aux fautes

d'une Servlet Java avec AspectJ est décrite par la Figure 9. Dans les lignes 1 à 7, un pointcut nommé mainMethodExecution est défini comme étant l'exécution de toute méthode principale d'une Servlet, c'est-à-dire les méthodes doGet ou doPost. Par ailleurs, étant donné qu'une méthode doGet peut elle-même appeler une méthode doPost (et inversement), il faudrait, dans ce cas, ne capturer que la première méthode qui constitue le point d'entrée effectif à la Servlet. Dans les lignes 8 à 11, un pointcut nommé servletEntryPoint est défini comme étant l'exécution d'une méthode principale de Servlet (doGet ou doPost) qui ne ferait pas déjà partie du flot d'exécution d'une autre méthode principale (et qui constituerait donc le point d'entrée effectif à la Servlet).

Finalement, toute application J2EE utilisant des Servlets Java peut automatiquement bénéficier de la tolérance aux fautes fournie par le système JSR, puisque celui-ci ne fait aucune hypothèse sur l'application elle-même à part le fait qu'elle utilise l'interface standard de Servlets Java.

4.3 Discussion

L'introduction du système JSR à une application J2EE peut soulever plusieurs questions techniques ou conceptuelles parmi lesquelles nous citons les suivantes :

- *Transparence.* Considérons le cas suivant : une requête dédiée à une Servlet est en cours de traitement sur un serveur et commence à émettre le début du résultat à l'utilisateur, puis une faute survient déclenchant le mécanisme de traitement de la faute par re-exécution de la requête sur un autre serveur pair, cette seconde exécution se termine normalement et le résultat est renvoyé à l'utilisateur. Le traitement de la faute n'aura donc pas été transparent pour l'utilisateur puisque celui-ci recevra un début de résultat puis le résultat en entier. Ce problème est plus particulièrement dû au fait que l'envoi de réponse à l'utilisateur est une opération irréversible qui ne peut pas être "défaite" en cas de faute. Pour faire face à ce problème de manque

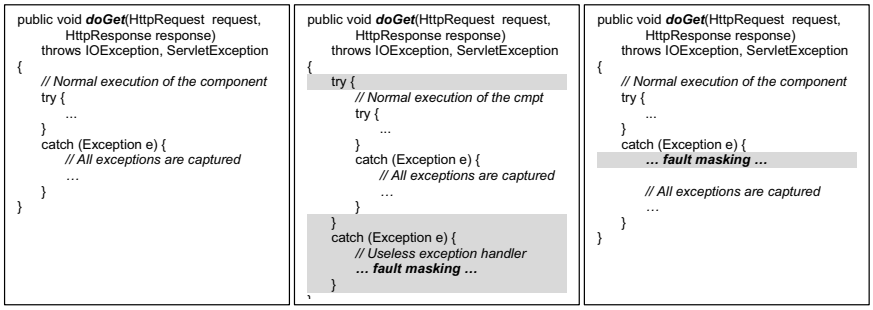
de transparence, nous proposons de temporiser l'envoi du résultat à l'utilisateur : le résultat ne sera retourné à l'utilisateur qu'une fois que l'exécution de la requête est effectivement terminée, soit dans le cas où elle n'a pas généré de faute, soit lorsque toutes les tentatives de re-exécution sur des serveurs pairs sont terminées. Pour la mise en œuvre du retardement de l'envoi du résultat à l'utilisateur, nous utilisons des constructions d'AspectJ qui permettent d'intercepter les opérations Java standards d'envoi de réponse HTTP et de mémoriser le résultat à envoyer jusqu'à la fin de l'exécution de la requête.

- *Cohérence.* Une autre question qui pourrait être posée concernant l'introduction du système JSR à une application J2EE est la suivante : est-ce que le mécanisme de traitement d'une faute par re-exécution de la requête sur un autre serveur pair n'induit pas d'état incohérent du système global ? Autrement dit, est-ce que le fait qu'une requête ait commencé à s'exécuter sur un serveur (puis qu'elle ait été interrompue suite à l'occurrence d'une faute), puis le fait que la même requête soit re-exécutée sur un autre serveur pair n'induit pas d'état incohérent dans le système global, en particulier dans les bases de données sous-jacentes ? Il est important de rappeler ici que le système JSR ne fait que *mimer* le comportement d'un utilisateur humain qui, face à une faute lors de l'exécution d'une requête sur un serveur, pourrait demander *manuellement* de re-exécuter la même requête sur un autre serveur pair. Donc l'intégration du système de tolérance aux fautes JSR à une application J2EE n'induirait pas plus d'incohérence à l'application, mais fournit exactement le même niveau de cohérence que l'application originelle.
- *Pertinence.* Considérons le cas suivant : une application est écrite de telle sorte que toute exception qui est levée est capturée et prise en charge par l'application. Ceci peut être vu comme un bloc Java *try-catch-tout-type-d'exception* qui englobe toute méthode principale de l'application (voir Figure 11.a). Dans ce cas, l'introduction du système JSR à l'application telle qu'illustrée par la Figure 11.b ne capturera aucune exception Java et ne pourra donc pas effectuer de masquage de faute. Est-ce que le système JSR reste pertinent dans ce cas ? Ne serait-il pas plus judicieux ici de capturer le traitant d'exception de l'application [6] et d'y injecter le traitement de fautes par masquage (en utilisant les mécanismes adéquats de programmation par aspects), cf. Figure 11.c ? Nous avons choisi de ne pas suivre cette dernière approche puisque nous considérons que si une application capture elle-même toutes les exceptions, c'est qu'elle prend en charge toutes les fautes et que la tolérance aux fautes fait partie intégrante de l'application. Il ne faudrait, dans ce cas, pas altérer la fonctionnalité de l'application et sa prise en charge des fautes.

4.4 Etat courant et perspectives d'expérimentations

Un prototype du système de tolérance aux fautes JSR est en cours de mise en œuvre. Une évaluation expérimentale de ce système sera effectuée sur différentes applications J2EE, telle

[6] La partie catch du bloc Java *try-catch-tout-type-d'exception*.



(a) Application originelle *(b) Solution avec JSR* *(c) Solution éventuelle*

FIG. 11 – Pertinence de la solution proposée

que l'application de démonstration de référence fournie par Sun Microsystems : l'application Pet Store de vente en ligne [27]. Dans le cadre de cette évaluation, nous prévoyons, dans un premier temps, de provoquer des fautes intentionnelles telles qu'une mauvaise configuration de serveur ou un déploiement incorrect d'une application (cf. section 5) ; ceci permettra de reproduire des erreurs d'opérateur (administrateur). Nous mettrons également en œuvre un mécanisme d'injection de fautes qui aura pour principale fonction d'injecter des exceptions/fautes Java fictives, à des fréquences variables. Ce mécanisme permettra alors d'évaluer la réactivité du système JSR face aux fautes injectées.

Par ailleurs, et pour évaluer plus particulièrement le niveau d'intrusivité du système JSR sur les applications J2EE et mesurer l'éventuel surcoût sur les performances induit par la prise en compte de la tolérance aux fautes, des applications de benchmark pourront être utilisées. Nous pensons par exemple utiliser l'application TPC-W qui met en œuvre une librairie en ligne de type *Amazon.com* [28] ou l'application RUBiS qui modélise un site Web de vente aux enchères tel que *eBay.com* [1].

5 Etudes de cas

Dans cette section, nous présentons des études de cas qui permettent d'illustrer l'utilisation du système de tolérance aux fautes JSR. Nous utilisons ici TPC-W et RUBiS comme exemples d'applications J2EE pour illustrer ces études de cas. Nous décrivons, plus particulièrement, la prise en compte de fautes logicielles ou de fautes de l'opérateur ; ces fautes étant à l'origine d'une grande partie des défaillances dans les systèmes Internet [21, 22].

5.1 Faute de l'opérateur - Mauvais déploiement de l'application

Un exemple de faute dans une application J2EE est un mauvais déploiement de l'application, où l'administrateur de l'application déploie, sur le serveur J2EE, une mauvaise version de librairie utilisée par le code de l'application. Par exemple, dans le cas d'un mauvais déploiement de l'application TPC-W, lors du traitement d'une requête de client sur un serveur J2EE, une exception Java de type *NoSuchMethodException* est levée. Cette exception provoque alors l'interruption du traitement de la requête sur le serveur et une défaillance est ainsi perçue par le client.

5.2 Faute de l'opérateur - Erreur de configuration

Un second exemple de faute de l'opérateur dans une application J2EE est une mauvaise configuration de l'application ; ce type de faute représentant la majorité des fautes de l'opérateur [21]. L'administrateur de l'application aurait, par exemple, spécifié une URL erronée pour faire référence à la base de données sous-jacente. Dans le cas de l'application TPC-W, lors du traitement d'une requête de client sur un serveur J2EE, une exception Java de type *NullPointerException* sera levée à cause de l'utilisation d'une connexion nulle (inexistante) à la base de données. Ceci provoquera l'interruption du traitement de la requête sur le serveur J2EE, résultant en une défaillance perçue par le client.

5.3 Faute temporelle

Un autre exemple de faute dans une application J2EE est une faute temporelle. L'administrateur de l'application J2EE aurait, par exemple, spécifié un délai maximum de traitement de requête qui serait trop court. Dans le cas de l'application RUBiS et lorsque les serveurs sont particulièrement chargés, le traitement d'une requête de client sur un serveur J2EE résultera en une exception Java due à un délai trop court. Cette exception interrompra alors le traitement de la requête sur le serveur J2EE et la défaillance sera perçue par le client.

Le système de tolérance aux fautes JSR peut être introduit aux applications TPC-W et RUBiS de façon automatique, c'est-à-dire sans modification manuelle de l'application, sans intervention du programmeur de l'application et sans modification du middleware. Ceci peut être fait en utilisant le compilateur d'AspectJ qui effectue le tissage automatique de l'aspect tolérance aux fautes de JSR à l'application considérée, tel que décrit par la Figure 5. Ainsi,

les fautes seront automatiquement détectées et traitées par le système JSR qui les masquera par des tentatives de re-exécution des requêtes sur des serveurs pairs.

6 Etat de l'art

La tolérance aux fautes des systèmes informatiques et, en particulier, des systèmes répartis est un domaine de recherche largement étudié [12, 23, 15, 2]. Les travaux dans ce domaine se différencient principalement selon trois critères : le type de fautes prises en compte (fautes de l'opérateur, fautes logicielles ou fautes matérielles), la technique de détection de fautes utilisée et l'approche de traitement de fautes proposée.

En ce qui concerne les travaux sur la tolérance aux fautes de l'opérateur, le traitement des fautes est principalement basé sur l'utilisation d'un journal qui enregistre les opérations effectuées. Lors de la détection d'une faute, des techniques de retour-arrière sont appliquées pour retourner à un état cohérent du système à partir duquel l'exécution peut être reprise ; Brown et. al. ont expérimenté une telle solution [6].

Dans le domaine de la détection de fautes logicielles, une technique récemment étudiée est le *pinpoint* [11]. Cette technique effectue un traçage des requêtes des clients et détermine les composants logiciels applicatifs impliqués dans l'exécution de ces requêtes ; une corrélation entre les fautes et les composants fautifs est alors établie de manière statistique [17]. Le traitement de fautes logicielles peut ensuite se faire grâce au mécanisme de *micro-reboot* qui consiste à arrêter puis redémarrer les composants logiciels fautifs ; l'arrêt puis le redémarrage étant, dans certains cas de fautes logicielles transitoires, l'unique alternative pour revenir à un état cohérent [7].

Quant aux fautes matérielles, elles sont principalement détectées grâce à des adaptations des techniques de *ping* ou de *heartbeat* [4]. Avec la première technique, c'est le gestionnaire de tolérance aux fautes qui teste régulièrement si le composant matériel à surveiller (par exemple un serveur) est encore en vie ; et avec la seconde technique, c'est le composant concerné qui donne régulièrement signe de vie au gestionnaire de tolérance aux fautes. Dans le contexte des applications J2EE s'exécutant sur des grappes de serveurs dupliqués, les solutions existantes de traitement de fautes de type pannes franches (entre autres des fautes matérielles) d'un serveur consistent principalement à masquer la faute : en ignorant le serveur "fautif" et en aiguillant les requêtes vers un autre serveur de la grappe. Le switch L4, le module mod_jk, cmi et c-jdbc sont des solutions qui suivent une telle approche pour traiter les fautes matérielles qui peuvent survenir, respectivement, au niveau de serveurs tels que les serveurs Web Apache, les serveurs de Servlets Tomcat, les serveurs d'EJB Jonas et les serveurs de bases de données [25, 24, 16, 10]. Le traitement de fautes matérielles peut également consister à réparer la faute (au lieu de simplement la masquer) en réallouant de nouvelles ressources et en utilisant des mécanismes de capture et de restauration de l'état du composant "fautif" [5, 8].

D'autres techniques de tolérance aux fautes dites "préventives" ont également été étudiées, telles que des techniques de prévention des fautes de l'opérateur proposées par Nagaraja et. al. [20], et les techniques de redondance et de vote telles que le système TMR (*Triple Modular Redundancy*).

Les solutions existantes de tolérance aux fautes sont ainsi variées et s'adressent à différents types de fautes. Nous ne pensons pas que ces solutions soient exclusives, certaines d'entre elles peuvent même être complémentaires, en particulier, dans le cas des applications

J2EE sur grappes de serveurs. Ainsi, le module mod_jk qui masque les fautes matérielles dans une grappe de serveurs Tomcat peut être complété d'un système JSR qui masque les fautes de l'opérateur et les fautes logicielles dans les serveurs J2EE, et finalement d'un mécanisme de *micro-reboot* qui serait la dernière alternative pour le traitement de fautes logicielles transitoires.

7 Conclusions et perspectives

Des études récentes ont montré que les défaillances dans les applications Internet étaient principalement causées pas des fautes logicielles ou des fautes de l'opérateur [21, 22]. Dans cet article, nous avons proposé un système de tolérance aux fautes logicielles et aux fautes de l'opérateur dédié aux applications J2EE ; la plate-forme J2EE étant devenue un standard pour le développement d'applications Internet réparties dans une architecture multi-tiers. Le système JSR proposé permet ainsi de masquer l'occurrence des fautes aux utilisateurs d'une application J2EE. Son principe est simple mais son originalité vient du fait qu'il soit à la fois *scalable*, *automatique* et *générique*. Nous avons ainsi décrit les principes de conception et les choix de mise en œuvre du système JSR qui font que ce système : (i) supporte la montée en charge des applications, (ii) s'applique automatiquement à toute application J2EE, (iii) ne requiert pas de modification manuelle du code de l'application, (iv) ne requiert pas d'aide du programmeur de l'application, (v) n'exige pas la modification de la mise en œuvre du middleware sous-jacent.

La solution proposée ici constitue ainsi une solution intermédiaire entre l'approche de tolérance aux fautes d'applications J2EE qui consiste à ne traiter que les fautes de type pannes franches et l'approche qui consiste à traiter les fautes logicielles en utilisant des techniques d'arrêt/redémarrage (*reboot*) des composants fautifs. Ces trois approches ne sont en aucun cas exclusives mais nous pensons, au contraire, qu'elles peuvent se compléter.

Nos perspectives, mis à part, à très court terme, la validation expérimentale du système JSR, sont d'appliquer l'approche de masquage de fautes proposée à d'autres niveaux de l'architecture multi-tiers J2EE. En effet, plus tôt une faute est détectée, plus tôt elle peut être réparée et moins elle ne sera propagée et n'induira d'effet de bord. En partant de ce principe, l'application de l'approche de masquage de fautes proposée à des étages plus "bas" dans l'architecture J2EE multi-tiers (comme au niveau du *tier* serveur de base de données) permettra certainement de détecter et de traiter les fautes le tôt possible.

Une perspective à plus long terme serait d'aller au-delà du traitement des fautes par masquage : en proposant des solutions automatiques de réparation des fautes. Nous pensons, par exemple, aux fautes récurrentes de mauvaise configuration des applications J2EE.

Références

[1] C. Amza, E. Cecchet, A. Chanda, A. Cox, S. Elnikety, R. Gil, J. Marguerite, K. Rajamani and W. Zwaenepoel. Specification and Implementation of Dynamic Web Site Benchmarks. *IEEE 5th Annual Workshop on Workload Characterization (WWC-5)*, Austin, TX, USA, nov. 2002. http ://rubis.objectweb.org.

[2] A. Arora, S. Kulkarni. Detectors and Correctors : A Theory of Fault-Tolerance Components. *18th International Conference on Distributed Computing Systems (ICDCS'98)*, Amsterdam, Pays-Bas, mai 1998.

[3] AspectJ 1.1, 2004. http ://www.eclipse.org/aspectj/

[4] A. Bonhote. High-Availability NFS Server with Linux Heartbeat. *Linux Magazine, European edition*, août 2003.

[5] S. Bouchenak, D. Hagimont, S Krakowiak, N. De Palma, F. Boyer. Experiences Implementing Efficient Java Thread Serialization, Mobility and Persistence. *Software - Practice & Experience (SP&E)*, 34(4), pp. 339-393, avr. 2004.

[6] A. B. Brown, D. A. Patterson. Undo for Operators : Building an Undoable E-mail Store. *USENIX Annual Technical Conference*, San Antonio, TX, Etats-Unis, juin 2003.

[7] G. Candea, S. Kawamoto, Y. Fujiki, G. Friedman, A. Fox. A Microrebootable System - Design, Implementation, and Evalua-tion. *6th Symposium on Operating Systems Design and Implementation (OSDI'2004)*, San Francisco, CA, Etats-Unis, déc. 2004.

[8] Y. Cardinale, E. Hernández. Checkpointing Facility on a Metasystem. *European Conference on Parallel Computing (Eu-roPar'2001)*, Manchester, Royaume-Uni, jan. 2001.

[9] R. Cattell, J. Inscore. *J2EE Technology in Practice : Building Business Applications with the Java 2 Platform, Enterprise Edition.* Pearson Education, 2001.

[10] E. Cecchet, J. Marguerite, W. Zwaenepoel. C-JDBC : Flexible Database Clustering Middleware. *FREENIX Technical Sessions, USENIX Annual Technical Conference*, Boston, MA, Etats-Unis, jui. 2004. http ://c-jdbc.objectweb.org/

[11] M. Y. Chen, E. Kiciman, E. Frnaklin, A. Fox, E. Brewer. Pinpoit : Problem Determination in Large, Dynamic Internet Services. *International Conference on Dependable Systems and Networks (DSN 2002)*, Washington, DC, Etats-Unis, juin 2002.

[12] E. W. Dijkstra. Self-Stabilizing Systems in Spite of Distributed Control. *Communications of the ACM*, 17(11), nov. 1974.

[13] R. E. Filman, T. Elrad, S. Clarke, M. Aksit. *Aspect-Oriented Software Development.* Addison-Wesley, oct. 2004.

[14] J. Gray. Why Do Computers Stop and What Can Bed Done About It ? *5th Symposium on Reliability in Distributed Software and Database Systems*, Los Angeles, CA, Etats-Unis, jan. 1986.

[15] P. Jalote. *Fault-Tolerance in Distributed Systems.* Prentice Hall, avr. 1994.

[16] JOnAS Project. *Java Open Application Server (JOnAS) : A J2EE Platform.* http ://jonas.objectweb.org/current/doc/JOnASWP.html

[17] E. Kiciman, A. Fox. Detecting Application-Level Failures in Component-Based Internet Services. *Technical Report, Stanford Computer Science Department*, 2004.

[18] G. Kickzales, J. Lamping, A. Mendhekar, C. Maeda, C. Videira Lopes, J. M. Loingtier, J. Irwin. Aspect-Oriented Programming. *European Conference on Object-Oriented Programming (ECOOP'97)*, Jyväskylä, Finlande, juin 1997.

[19] B. Murphy, N. Davies. Measuring System and Software Availability Drivers for Tru64 Unix. *29th Symposium on Fault-Tolerant Computing*, Madison, WI, Etats-Unis, juin 1999. Tutorial.

[20] K. Nagaraja, F. Oliveira, R. Bianchini, R. P. Martin, T. D. Nguyen. Understanding and Dealing with Operator Mistakes in Internet Services. *6th Sympoisum on Operating Systems Design and Implementation (OSDI'2004)*, San Francisco, CA, Etats-Unis, déc. 2004.

[21] D. Oppenheimer, A. Ganapathi, D. A. Patterson. Why Do Internet Services Fail And What Can Be Done About It ? *4th USENIX Symposium on Internet Technologies and Systems (USITS'03)*, Seattle, WA, Etats-Unis, mar. 2003.

[22] D. Patterson, A. Brown, P. Broadwell, G. Candea, M. Chen, J. Cutler, P. Enriquez, A. Fox, E. Kiciman, M. Merzbacher, D Oppenheimer, N. Sastry, W. Tetzlaff, J. Traupman, N. Treuhaft. Recovery Oriented Computing (ROC) : Motivation, Defini-tion, Techniques, and Case Studies. *Computer Science Technical Report UCB/CSD-02-1175, U.C. Berkeley*, mar. 2002.

[23] F. B. Schneider. Abstractions for Fault-Tolerance in Distributed Systems. *IFIP 10th World Computer Congress*, Dublin, Irlande, sep. 1986.

[24] G. Shachor. *Tomcat Documentation.* The Apache Jakarta Project. http ://jakarta.apache.org/tomcat/tomcat-3.3-doc/

[25] S. Sudarshan, R. Piyush. *Link Level Load Balancing and Fault Tolerance in NetWare 6.* NetWare Cool Solutions Article, mar. 2002. http ://developer.novell.com/research/appnotes/2002/march/03/a020303.pdf

[26] Sun Microsystems. *Java 2 Platform Enterprise Edition (J2EE).* http ://java.sun.com/j2ee/

[27] Sun Microsystems. *Java Pet Store Demo 1.3.2.* http ://java.sun.com/blueprints/code/jps132/docs/

[28] Transation Processing Performance Council. *TPC-W : a transactional web e-Commerce benchmark.* http ://www.tpc.org/tpcw/

Table des matières

Unité de recherche INRIA Rhône-Alpes
655, avenue de l'Europe - 38330 Montbonnot-St-Martin (France)

Unité de recherche INRIA Lorraine : LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex (France)
Unité de recherche INRIA Rennes : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex (France)
Unité de recherche INRIA Rocquencourt : Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex (France)
Unité de recherche INRIA Sophia Antipolis : 2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex (France)

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-6399