INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Laboratoire Logiciels Systèmes Réseaux

Tolérance aux fautes pour des systèmes autonomes

Sylvain Sicard

20 juin 2005

Master 2 Recherche en Informatique- Logiciels et Systèmes

Projet SARDES - INRIA Rhône-Alpes

Jury :

Jean Caelen

Yves Demazeau

Jean-Marc Vincent

Thierry Gautier

Tuteurs :

Sara Bouchenak

Noel De Palma

Remerciements

Je tiens à remercier Sara Bouchenack, Fabienne Boyer, Daniel Hagimont et Noel De Palma pour m'avoir accordé leur attention à chaque fois que j'en ai eu besoin. Je remercie également de façon générale toute l'équipe SARDES pour son accueil chaleureux.

Résumé

Les environnements informatiques d'aujourd'hui sont de plus en plus sophistiqués. Ils intègrent de nombreux logiciels complexes qui coopèrent dans le cadre d'une infrastructure logicielle, potentiellement à grande échelle. Ces logiciels se caractérisent par une grande hétérogénéité, en particulier en ce qui concerne les aspects d'administration qui deviennent des tâches très coûteuses en ressources. Une approche prometteuse pour traiter ces problèmes est de concevoir un logiciel d'administration autonome.

Ce travail s'intéresse à la conception et la validation d'un gestionnaire de défaillances pour les systèmes d'administration autonomes. Ce gestionnaire permet, d'une part la détection des défaillances et, d'autre part, la mise en oeuvre d'actions sur le système administré afin de le réparer automatiquement. Nous nous appuyons sur un modèle à composants logiciels pour mettre en œuvre les fonctions d'observation et de reconfiguration requises pour la gestion autonome des défaillances. La mise en application de ces travaux est faite dans le cadre de l'administration de serveurs J2EE implantés sur une grappe de machine.

Sommaire

Chapitre I.

Introduction

Ce rapport présente le travail que j'ai effectué durant mon stage de deuxième année de Master Recherche. Je commence par présenter le projet Sardes, au sein duquel s'est effectué ce stage, puis la problématique et les motivations de mon projet de recherche. Je décris ensuite la démarche adoptée pour aborder ce problème ainsi que quelques aspects de mise en oeuvre.

1.1. Contexte de travail

Ce travail a été réalisé au sein du projet SARDES[1]. SARDES est un projet INRIA et une équipe de recherche du laboratoire LSR-IMAG (CNRS, INPG, UJF). Le projet SARDES s'inscrit dans la perspective de l'émergence d'un environnement global de traitement de l'information (informatique ubiquitaire) dans lequel la plupart des objets physiques qui nous entourent seront équipés de processeurs de traitement de l'information, et interconnectés par le biais de réseaux divers (de l'internet planétaire au pico-réseau spontané).

Le projet SARDES a pour objectif l'étude de l'architecture et la construction d'infrastructures logicielles réparties pour cet environnement, caractérisé par une grande taille, une très grande hétérogénéité, et par une nature très dynamique. Pour la construction de tels systèmes, sûrs et hautement adaptables, le projet se propose d'exploiter de manière systématique des techniques de réflexion et de construction par composants. Un système réparti peut être défini comme un couple composé :

⇒ D'un ensemble de ressources matérielles physiquement distinctes interconnectées par un système de communication.

[1] *System Architecture for Reflexive Distributed EnvironmentS*

⇒ D'un système logiciel exploitant l'ensemble de ces ressources pour fournir un service.

Le concept de systèmes répartis restant tout de même assez général, et par la même assez vaste, les travaux que nous effectuons à SARDES s'intéressent plus particulièrement aux cas des applications en grappes.

Les grappes appartiennent à un sous ensemble des systèmes répartis où toutes les ressources sont concentrées au sein d'un réseau local à haut débit. Ce type de système présente en outre l'intérêt de borner les variables définissant l'état de l'environnement du système.

De tels systèmes atteignent rapidement une complexité telle qu'il devient impossible d'en garder une complète maîtrise. Ceci, dans le sens où la connaissance des éléments logiciels et matériels mis en jeux devient si vaste qu'elle échappe à la compétence d'un seul homme. En conséquence, l'administration de ces systèmes devient alors un réel défit en soit. Les travaux actuels réalisés dans SARDES sont motivés par ce constat et s'intéressent à automatiser les tâches d'administration à travers des systèmes d'administration autonomes.

1.2. Position du problème

Cette section présente tout d'abord un bref état des lieux concernant les besoins, en terme d'administration, émergeant avec les applications réparties mises en œuvre de nos jours. Suit ensuite une présentation détaillée de ce qu'est un système d'administration autonome.

1.2.1. Motivations et problématiques

Avec l'amélioration des technologies, la réduction des coûts et la miniaturisation des matériels, l'informatique s'insinue partout. Les applications réparties, capables de tirer parti de la coopération de nombreux systèmes issus de ces évolutions, se multiplient et offrent de nouveaux challenges. La structure des logiciels évolue vers une plus grande complexité et une grande hétérogénéité, en particulier en ce qui concerne les modèles de programmation utilisés pour les concevoir, les développer et surtout les administrer. En conséquence, leur administration, actuellement effectuée par l'homme, est une tâche ardue et coûteuse en ressources :

⇒ Humaines, car l'administration nécessite systématiquement une intervention humaine en réaction à des événements (pannes par exemple)

⇒ Matérielles, car la solution généralement adoptée pour prendre en compte des incidents (pannes ou surcharge) est la surréservation de ressources

Face à un tel problème, notre approche consiste à fournir un environnement autonome permettant de gérer automatiquement les tâches d'administration courantes. L'administration autonome est une approche apportant :

⇒ Une meilleure réactivité : concevoir un logiciel d'administration autonome permet d'effectuer des actions d'administration (réglage, optimisation, réparation) dynamiquement, en réponse à des observations et sans intervention humaine.

⇒ Des économies de ressources : un logiciel d'administration autonome permet d'économiser des ressources humaines (car on limite les interventions) et matérielles (on les alloue dynamiquement en fonction des besoins).

⇒ Moins d'erreurs et d'efforts : les logiciels sont administrés à travers une interface programmatique, par opposition à des fichiers de configuration complexes et hétérogènes, ce qui limite les erreurs humaines de configuration et les efforts d'administration.

L'objectif général est donc de concevoir et de valider un logiciel d'administration autonome. La mise en application de ces travaux peut, par exemple, se faire dans le cadre de l'administration de serveurs à haute performance implantés sur une grappe de machines, ou de systèmes embarqués[1].

1.2.2. Définitions de l'administration autonome

L'administration autonome se définit de façon relative à l'administration « traditionnelle ». On attend d'un système autonome qu'il réalise de façon automatisée un certain nombre de tâches similaires ou équivalentes aux actions réalisées « traditionnellement » par un opérateur humain. Dans notre cas, l'administration d'un système réparti couvre un ensemble de tâches variées dont :

⇒ le déploiement, au sens de téléchargement, installation, paramétrage et lancement de l'application.

⇒ la réparation, face aux défaillances diverses dont le système peut potentiellement être victime.

⇒ l'optimisation des performances pour faire face aux variations de la charge de l'application lors de son exécution.

L'**administration autonomme**, ou informatique auto-gérée, désigne un système capable d'administrer lui-même les éléments qui le composent, en l'absence d'intervention humaine, afin d'assurer son bon fonctionnement [Ganek et al., 2003]. A terme, un système autonome devrait fournir des propriétés telles que :

⇒ L'auto-configuration : le système est capable de paramétrer automatiquement ses composants.

⇒ L'auto-réparation : la faculté à détecter et corriger la panne d'un ou plusieurs de ses composants.

[1] *Le terme système embarqué dénote un système à base de logiciel ayant des interactions très fortes avec l'environnement qu'il doit contrôler.*

⇒ L'auto-optimisation : le système entreprend seul des mesures pour remédier aux éventuels problèmes de performances.

⇒ L'auto-protection : la capacité de la plate-forme à se prémunir d'un tiers malveillant.

Par généralisation, un système d'administration autonome est une application comme les autres ; les objectifs cités précédemment s'appliquent donc aussi bien au système administré qu'à la plate-forme d'administration elle même.

1.3. Sujet de recherche : l'auto-réparation pour les systèmes autonomes

Parmi les différents aspects que recouvre l'administration autonome, un élément central et indispensable est l'auto-réparation. L'objectif est de fournir aux systèmes d'administration la capacité de maintenir et réparer le système administré de façon (i) automatisée, (ii) sans avoir recours à une intervention humaine.

Cela signifie que le système d'administration doit pouvoir, d'une part détecter les défaillances et, d'autre part, décider d'une action à entreprendre pour ramener le fonctionnement à un mode normal. Enfin, il doit disposer de moyens d'action sur le système administré afin d'agir sur son état aux vues de son diagnostic.

Un système d'administration autonome doit pouvoir prendre en compte un vaste champ de défaillances, de façon à maintenir le système dans un état de fonctionnement optimum. Dans notre cas, un état de fonctionnement optimum se définit par rapport à son état initial.

Dans l'exemple des systèmes en grappe, le système peut potentiellement être victime de la défaillance d'un nombre arbitraire de machines, ces défaillances étant distribuées dans le temps selon une loi quelconque. Le système autonome administrant une telle architecture doit pouvoir réparer le système quelle que soit la séquence de défaillances survenues. Cela inclut autant de scénarii impliquant des défaillances du système administré, du système d'administration lui-même, ou de défaillances imbriquées. On désigne par le terme de défaillances imbriquées les cas ou une panne survient pendant le processus de réparation d'une précédente panne. Le travail présenté ici s'inscrit dans cet axe de recherche.

Chapitre II.

Etat de l'art sur la tolérance aux fautes

Les systèmes actuels sont soumis à une charge permanente et sont supposés être hautement disponibles. De plus, ils sont de plus en plus complexes de par leur taille et de par l'empilement des couches logicielles mises en œuvre (système d'exploitation, intergiciel, application). La maîtrise globale du système devient donc de plus en plus difficile, ce qui rend la configuration et la maintenance d'autant plus compliquées. De plus, l'usage d'un nombre de plus en plus important de machines dans les grappes augmente la probabilité de voir une défaillance se produire. Dans ces circonstances, on comprend aisément la nécessité de concevoir les systèmes de façon à ce qu'ils soient capables de survivre à l'occurrence d'une panne, tant logicielle que matérielle.

La suite de cette section présente, tout d'abord, quelques rappels et définitions, puis une classification des principaux travaux sur la tolérance aux fautes. Pour finir, nous présentons deux études de cas mettant en oeuvre ces concepts.

2.1. Rappels et définitions

La tolérance aux fautes propose des solutions algorithmiques et/ou matérielles permettant aux applications de fournir un service de *qualité* bien que se trouvant dans un environnement sujet à des *défaillances*. La mesure et la quantification de ces deux notions demandent à être mieux définies. Pour exprimer la qualité d'un service on parlera de sa fiabilité, de sa disponibilité et de sa sécurité

⇒ La **fiabilité** d'un système désigne sa capacité à se trouver dans un état (continu) de rendre le service pour lequel il est conçu. La quantification de la fiabilité est exprimée par la probabilité, en fonction du temps *t*, que le système ne soit pas défaillant entre le temps *0* et le temps *t*. L'espérance de cette fonction donne le temps moyen jusqu'à la prochaine panne (MTTF – *Mean Time To Failure*) et est couramment utilisée pour qualifier la fiabilité des systèmes.

⇒ La **disponibilité** d'un système désigne sa capacité à fournir le service à l'utilisateur (le service doit être disponible en permanence). La disponibilité se mesure sur une période de temps bornée et connue et s'exprime comme étant la fraction du temps où le service était disponible. Pour qu'un système soit disponible, il faut qu'en cas de panne, il soit réparé (remis en état de rendre le service) rapidement. Le temps moyen de remise en service est désigné par le MTTR (*Mean Time To Repair*).

⇒ La **sûreté** désigne la capacité d'un système à gérer les cas de mauvais fonctionnement afin que ces derniers n'aient pas de conséquences graves sur lui-même.

La défaillance (ou panne) d'un système désigne son incapacité à rendre le service pour lequel il est conçu. Une défaillance est la conséquence directe ou non d'une erreur. Le viol d'un invariant ou la rupture d'un lien réseau sont des cas d'erreurs menant à des défaillances.

Les défaillances peuvent être classées de différentes façons, en fonction de leurs origines, de leurs symptômes ou encore de leurs durées. La classification employée ici discrimine le degré de permanence des pannes.

⇒ Les **pannes franches** (ou arrêt sur défaillance). Avant l'occurrence d'une telle panne, le système fonctionne normalement et il délivre un résultat correct. L'occurrence d'une panne franche dans un système provoque l'arrêt définitif du service fourni.

⇒ Les **pannes transitoires**. Le système est sujet à des occurrences de pannes franches mais non définitives. A un instant donné, le système peut fonctionner ou non, sans que cela ne donne aucune indication sur ce que sera son état l'instant suivant (toujours en panne ou a nouveau opérationnel)

⇒ Les **pannes byzantines** (ou arbitraire). Ces pannes sont les plus complexes à détecter et à traiter. Dans ce modèle de panne, le système peut faire n'importe quoi y compris adopter un comportement malveillant. Ces pannes peuvent ne pas être reproductibles et leurs occurrences ne correspondent à aucune « règle ».

L'étude de la tolérance aux fautes est donc l'étude des mécanismes permettant à un système d'être fiable, disponible et sécurisé en dépit de divers types de pannes. La tolérance aux fautes est un vaste domaine, aussi, dans la suite, nous considérerons principalement le cas des pannes franches. La suite de cette section s'organise de la façon suivante : la première partie présente les principes de la tolérance aux fautes, la deuxième partie présente une étude de cas montrant les applications de ces principes.

2.2. Principes de la tolérance aux fautes

Dans cette partie, nous détaillons tout d'abord les mécanismes de détection des fautes et les problématiques rencontrées dans leur mise en oeuvre. Ensuite nous présentons les techniques de réparation et de masquage des défaillances. La réparation et le masquage ne sont pas exclusifs,

l'une et l'autre peuvent parfois se combiner afin de fournir de bonnes propriétés de disponibilité et de fiabilité.

2.2.1. Détection

La détection des fautes dans un système distribué n'est pas une chose triviale et constitue un prérequis indispensable à la mise en oeuvre de solution de tolérance aux fautes. L'étude des détecteurs de fautes est donc très importante. Dans un contexte où le système est sujet à des pannes franches et doté d'un système de communication asynchrone, la détection de fautes devient tout simplement impossible avec un protocole déterministe comme le montrent Fischer, Lynch et Paterson avec le cas du consensus [Fischer et al., 1983]. Le problème provient du fait que dans un système asynchrone, il est impossible de distinguer un processus lent d'un processus en panne. Ce résultat d'impossibilité est la source de ce que Chandra et Toueg ont appelé les « détecteurs de fautes non fiables » [Chandra et al., 1992]. Le principe est que les détecteurs de fautes peuvent commettre des erreurs dans leur diagnostic. Ces erreurs peuvent concerner deux aspects, soit le détecteur peut ne pas voir certaines pannes et on parle alors de complétude, soit le détecteur peut voir des pannes là où il n'y en a pas et on parle alors d'exactitude. L'exactitude forte signifie que le détecteur de panne ne signale jamais de panne alors qu'il n'y en a pas. A l'inverse l'exactitude faible signifie que le détecteur de panne peut détecter une fausse panne. Ces conditions d'exactitude peuvent être encore affaiblies en considérant la dimension temporelle. Ainsi on parlera d'exactitude finalement forte si au bout d'un temps fini, aucun composant correct n'est plus jamais soupçonné. On parlera d'exactitude finalement faible si au bout d'un temps fini, il existe un composant correct qui n'est plus jamais soupçonné. Les propriétés des détecteurs de pannes sont résumées dans le tableau 1.

	Exactitude forte	Exactitude faible	Exactitude finalement forte	Exactitude finalement faible
Complétude forte	P	S	◊P	◊S

Tableau 1 - Classes de détecteurs de pannes

Les détecteurs de pannes des classes P, S, ◊P, ◊S ne sont pas réalisables par des algorithmes déterministes. Dans la pratique on ne peut que chercher à s'en approcher. C'est ce que font les détecteurs *Ping*, *Heartbeat* et *Pinpoint* décrit dans la suite de cette section.

2.2.1.1. Ping

Le détecteur de pannes *Ping* lève l'hypothèse d'asynchronisme en estimant un délai de garde pour la détection de composants en panne. Ce délai de garde estime le temps d'aller et retour d'un message dans le système et le temps de traitement de ce message par un composant

du système. Dans le cas ou le délai est `I` et ou un composant `A` cherche à déterminer si un composant `B` est en panne, `A` émet périodiquement un message à `B` qui doit renvoyer un acquittement. Si `A` n'a pas reçu d'acquittement au bout d'un temps `I`, `A` déclare `B` en panne.

L'estimation du délai `I` est difficile et n'est pas infaillible. Si le délai est trop court le risque de détection de fausses pannes est élevé. A l'inverse, une estimation du délai trop lâche va retarder la détection des pannes.

2.2.1.2. Heartbeat

Il existe deux variantes des détecteurs de pannes Heartbeat. La première variante est similaire à un Ping inversé, son principe est le suivant : les composants émettent, à un intervalle de temps `I` fixé et connu de tous, un message « je suis vivant ». Ce message est reçu par tous. Si après une durée `I`, depuis sa précédente émission un composant n'a pas réémis de message « je suis vivant », tous les autres composants le considèrent comme en panne. Cette technique lève, comme le Ping, l'hypothèse d'asynchronisme en estimant un délai de transmission des messages (compris dans l'intervalle `I`).

Une autre variante du détecteur Heartbeat ne lève pas cette hypothèse [Kawazoe et al., 1999]. Dans ce cas, comme dans la première variante, les composants émettent régulièrement un message de type « je suis vivant ». Ici l'intervalle `I`, séparant deux émissions, ne tient pas compte des délais de transmission (on reste en asynchrone) mais est le même pour tout le monde. Le principe de ce détecteur repose sur un comptage des messages émis par les composants. Chaque composant maintient un vecteur associant pour chaque composant le nombre de messages reçus.

2.2.1.3. Pinpoint

Pinpoint est un détecteur de panne dont le principe repose sur une analyse comportementale du système [Chen et al., 2002]. Les pannes sont détectées par l'observation d'une variation anormale de ce comportement. De plus, le détecteur *Pinpoint* permet de localiser l'origine des défaillances en plus de les détecter.

Dans un système constitué de plusieurs composants logiciels, le comportement du système (ce qu'il *fait*, ce qui s'y *passe*) est caractérisé par les interactions entre composants. On peut donc modéliser le comportement d'un système par l'ensemble des interactions qu'il a entre ses composants et avec le monde extérieur. Les enjeux dans la construction d'un détecteur de panne de type Pinpoint sont donc doubles. D'une part, le détecteur doit être capable de surveiller les interactions entre composants, et d'autre part il doit être capable de caractériser une observation comme étant normale ou non.

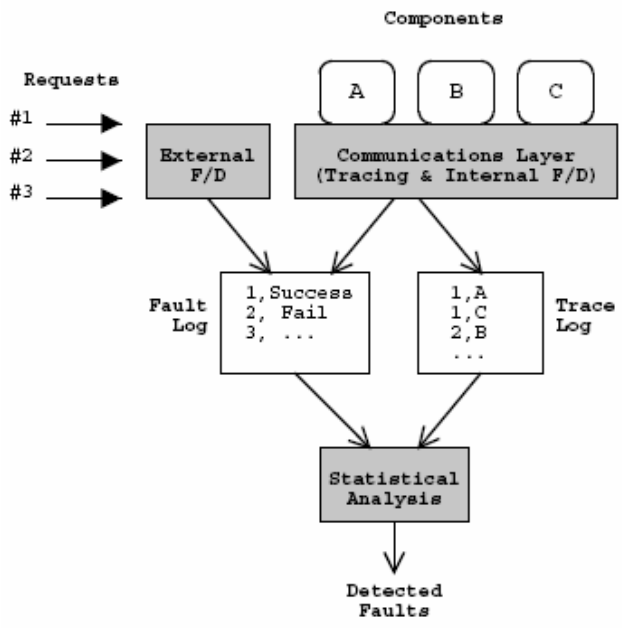

Figure 1 - Principe de fonctionnement de Pinpoint[1]

La Figure 1 montre le principe de fonctionnement du détecteur *Pinpoint*. La surveillance de l'activité inter-composants est assurée à l'aide d'une couche de communication spécifique, jouant le rôle d'un intercepteur transparent vis-à-vis des appels entre composants. De cette façon, tous les appels sont interceptés et pris en compte par le détecteur. Cet intercepteur gère une journalisation (*log*) des interceptions effectuées. Les journaux ainsi générés sont analysés afin d'en extraire une représentation du comportement instantané du système.

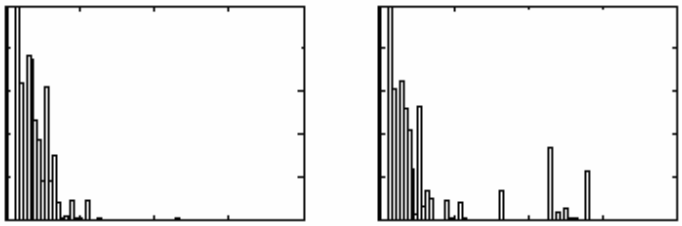

Figure 2 - Représentation du comportement normal / instantané de l'application[1]

[1] *[Chen et al., 2002]*

La Figure 2 (partie gauche) montre le comportement normal de l'application. Le comportement *normal* est obtenu grâce à une phase de tests ou le système est hors ligne. Durant cette phase, une charge artificielle est injectée dans le système en conjonction avec une injection de fautes afin d'en déduire les réactions et la propagation des défaillances dans le système.

La Figure 2 (partie droite) illustre un cas de détection d'erreur où le comportement n'est pas conforme à celui attendu. Cette détection s'appuie sur un test statistique[1] ce qui signifie que la détection ne tiendra pas compte des éventuelles variations ponctuelles mais non moins normales du comportement du système. Pinpoint n'est donc pas un détecteur de pannes exact (ce qui n'est pas surprenant).

2.2.2. Réparation

Lorsqu'une panne intervient dans le système et qu'elle a été détectée, localisée et isolée, il convient ensuite de réparer les éléments[2] en panne. Cela signifie que l'élément doit être remis dans un état cohérent. La suite de cette section décrit les principales approches employées pour la réparation.

2.2.2.1. Checkpoint

La sauvegarde (*checkpoint*) est une méthode classique pour assurer la réparation de fautes dans les systèmes, qu'ils soient distribués ou non [Elnozahy et al., 2002]. Le principe de base est de sauvegarder l'état du système périodiquement sur un support fiable et persistant (par exemple le disque). De cette façon lors du redémarrage après une panne, l'état sauvegardé le plus récent est restauré et l'exécution reprend son cours avant la panne. L'état global d'un système réparti est défini par l'union des états locaux de tous les processus appartenant au système.

Il existe deux types de sauvegardes, la sauvegarde indépendante (*checkpoints* indépendants) et la sauvegarde coordonnée (*checkpoints* coordonnés).

Checkpoints indépendants

Dans ce cas les composants du système procèdent à la sauvegarde de leur état de façon indépendante. Cela peut mener à l'enregistrement d'un état global incohérent. Une phase de synchronisation est alors nécessaire lors de la restauration de l'état après une panne. Cette méthode (optimiste) présente l'avantage de n'impliquer qu'un surcoût minime à l'exécution en l'absence de panne [Wang et al., 1993]. En revanche la reprise est plus complexe dans la mesure où il ne suffit pas de restaurer le point de sauvegarde le plus récent pour chaque composant. En effet, si par exemple un composant tombe en panne juste après avoir envoyé un message et si sa dernière sauvegarde est faite avant cette émission, alors le message deviendra orphelin dans la mesure où il sera pris en compte par son récepteur mais sera « oublié » par son émetteur qui sera revenu à un point de son exécution antérieur à l'émission. L'état ainsi restauré est alors

[1] *Il s'agit du test d'adéquation du χ^2. Ce test permet de tester la similitude de deux séries de données.*

[2] *Selon les cas il peut aussi bien s'agir d'un thread, d'un processus ou d'un nœud entier.*

incohérent. Pour y remédier, on peut alors restaurer un état antérieur à la réception du message chez le composant destinataire. Ceci peut mener à *l'effet domino* illustré dans la Figure 3. Dans cet exemple les composants procèdent à une sauvegarde de leur état avant chaque envoi de message. Dans la Figure 3, on considère que le composant *c2* tombe en panne juste après l'envoi de son troisième message. Dans le cas présent, le seul état cohérent enregistré est l'état initial. Dans tous les autres cas, l'état global fait apparaître un message qui est réceptionné sans jamais avoir été émis.

Ce phénomène d'effet domino peut apparaître parce que les sauvegardes ne sont pas coordonnées. Pour remédier à ce problème, deux extensions de cette méthode ont été proposées. Ces extensions reposent sur une journalisation des messages. Lors de la réparation, les journaux sont analysés afin de détecter et/ou de prévenir la présence de l'effet domino.

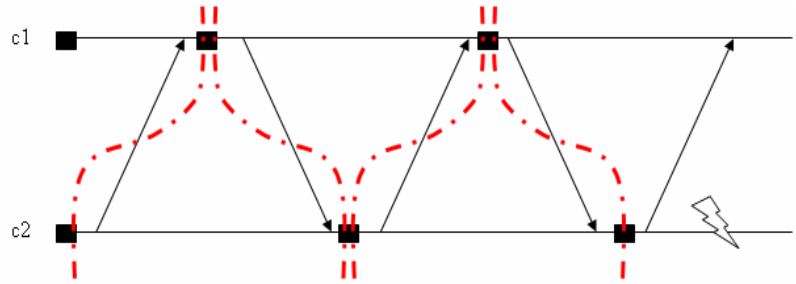

Figure 3 - L'effet domino

Checkpoints coordonnés

Le principe des sauvegardes coordonnées est de synchroniser toutes les sauvegardes locales des composants afin d'enregistrer un état global cohérent ; par opposition à la sauvegarde non coordonnée où la cohérence de l'état est construite lors de la réparation. La sauvegarde coordonnée n'est donc pas affectée par l'effet domino dans la mesure où elle prend en compte les messages en transit au moment de l'enregistrement de l'état [Toueg et al., 1984]. Plusieurs approches sont possibles pour réaliser une telle sauvegarde. Une approche simpliste consiste à bloquer toutes communications lors de l'enregistrement de l'état afin de garantir l'absence de message en transit. Une telle approche peut être réalisée par un protocole de sauvegarde à deux phases : un coordinateur décide de prendre une sauvegarde. Il diffuse alors sa demande à tous les autres composants du système. Ceux-ci stoppent alors leur exécution et vident leurs canaux de communication. Lorsqu'ils ne reçoivent plus de messages et n'en ont plus à émettre, ils enregistrent leur état courant. Ensuite ils envoient un message de confirmation au coordinateur. Lorsque le coordinateur a reçu tous les messages de confirmation, il envoie un message de validation à tout le monde. Lorsque les composants reçoivent un message de validation, ils reprennent le cours normal de leur exécution [Tamir et al., 1984]. Cette technique est parfaitement valide mais sa mise en œuvre engendre un surcoût important à l'exécution. En conséquence, des approches de sauvegardes non bloquantes sont préférées la plupart du temps. Dans ce cas, la difficulté est de détecter les messages de l'application émis au cours de la capture

de l'état et risquant de rendre cette dernière incohérente. L'exemple de sauvegarde non bloquante la plus classique est l'algorithme du « *snapshot distribué* » de Chandy et Lamport [Chandy et al., 1985]. Cet algorithme utilise un marqueur afin de détecter les messages de l'application susceptibles de rendre les sauvegardes incohérentes.

2.2.2.2. Journalisation

A l'inverse de la réparation par sauvegarde, la réparation par journalisation utilise de façon explicite le fait qu'un composant peut être modélisé par un état initial et une suite déterminée d'interactions ou évènements. Un évènement peut être la réception d'un message ou son émission, une interruption système, ou tout autre évènement ayant une conséquence sur le composant. Il y a deux types d'évènements, les évènements déterministes et les évènements indéterministes. Un évènement déterministe, par exemple l'émission d'un message, peut être reproduit avec comme seules informations les causes de son occurrence. A l'inverse, un évènement indéterministe ne peut pas être reproduit, si l'on ne dispose pas des *caractéristiques* exactes de son occurrence [Elnozahy et al., 2002].

La journalisation consiste à enregistrer sur un support fiable les occurrences d'évènements afin d'être en mesure de les rejouer lors du recouvrement d'une panne. La journalisation doit donc comprendre suffisamment d'informations de façon à pouvoir effectuer un rejeu *complet* de l'entité défaillante, c'est-à-dire le rejeu de tous les évènements déterministes et indéterministes ayant influés sur l'état (que l'on cherche à reconstruire) du ou des composants. Le protocole de réparation doit apporter la garantie qu'après une réparation, il n'y a pas de composant dont l'état dépend d'un évènement non déterministe n'ayant pas pu être rejoué.

La journalisation peut être faite de façon optimiste ou pessimiste. Une journalisation optimiste considère que la défaillance d'un composant ne peut survenir que lorsque l'opération de journalisation est achevée (rendue persistante sur un support fiable). La journalisation est effectuée en mémoire volatile et est recopiée périodiquement sur le disque. Cette approche permet de limiter le surcoût du processus de journalisation. En revanche si une défaillance survient alors que l'opération de journalisation n'a pas encore été rendue persistante, la restauration de l'état est rendue plus complexe du fait de la perte d'une partie du journal.

Les protocoles de journalisation pessimistes sont conçus dans l'optique où la défaillance peut survenir après chaque occurrence d'évènement. Dans ce type d'approche, l'opération de journalisation est systématiquement rendue persistante après chaque occurrence d'évènement. La journalisation est effectuée avant que l'évènement ne soit pris en compte par l'application. De cette façon si un évènement n'est pas dans le journal, aucun composant ne peut en dépendre ce qui permet d'éviter les orphelins après un recouvrement.

2.2.3. Masquage

Pour construire un système tolérant aux fautes, une autre technique consiste à dupliquer ce dernier sur différentes machines afin de réduire la probabilité d'indisponibilité du système global ainsi formé (le système n'est indisponible que si toutes les machines sont en défaut simultanément). Nous détaillons dans la suite de la section les deux classes de méthode pour la duplication. Ensuite nous faisons un bref rappel sur les protocoles de communication de groupe, outil fondamental pour la duplication.

2.2.3.1. Schéma de duplication Active

La première méthode consiste à dupliquer le système en entités idempotentes jouant toutes un rôle symétrique [Guerraoui et al., 1996].

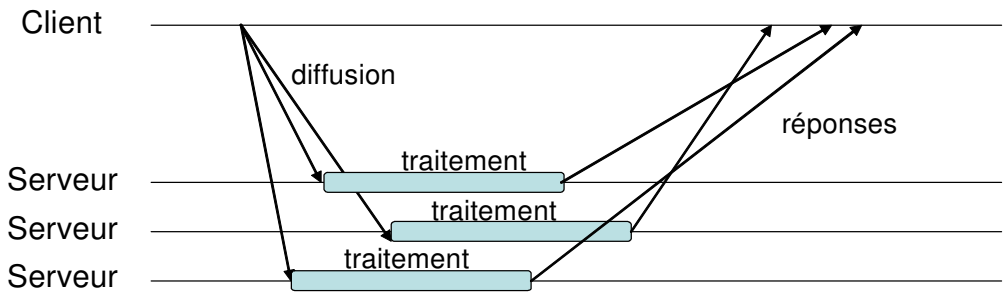

Figure 4 - Duplication active

Le principe de fonctionnement est assez simple, le client diffuse de façon atomique et ordonnée une requête à tous les serveurs (voir la Figure 4). Tous les serveurs traitent la requête en parallèle et renvoient la réponse au client. Pour le client deux solutions sont possibles, en fonction du type de panne que l'on souhaite gérer. Si l'on considère le cas des pannes franches ou transitoires, le client attend la première réception de réponse. En effet, toutes les réponses seront identiques (donc il n'y a aucun intérêt à les attendre toutes avant de poursuivre) et de plus le client ignore le nombre de réponses qu'il recevra effectivement (nombre de serveurs en pannes inconnu du client).

Cette technique présente l'inconvénient de *gaspiller* des ressources en raison de la redondance d'exécution mais, en revanche, elle masque totalement au client les n-1 premières défaillances de serveur. D'autre part la mise en place de ce dispositif n'est pas transparente pour le client qui doit d'une façon ou d'une autre prendre en compte la redondance des serveurs.

2.2.3.2. Schéma de duplication Primaire – Secondaire

Une seconde méthode pour la duplication consiste à seconder le serveur existant par des *copies* de secours dont l'état sera maintenu à jour au fil du temps (voir la Figure 5) [Guerraoui et al., 1996]. Dans ce cas, la différenciation entre primaire et secours implique un traitement adapté à la défaillance de l'un ou de l'autre. La défaillance de l'un des serveurs primaires est transparente et ne nécessite pas d'action particulière. A l'inverse, la panne du serveur primaire nécessite une phase de remplacement de ce dernier par l'un des serveurs de secours.

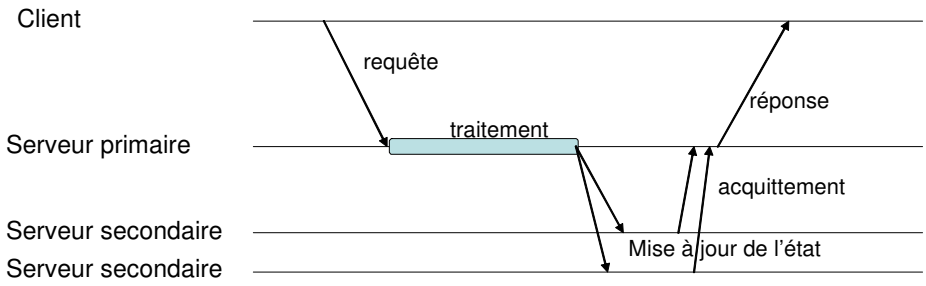

Figure 5 - serveur primaire - serveur secondaire

Le système fonctionne de la façon suivante : le client adresse ses requêtes au serveur principal ; ce dernier traite les requêtes reçues et met à jour les serveurs de secours par diffusion fiable de son nouvel état. Lorsque tous les serveurs de secours ont acquitté, le serveur principal envoie la réponse au client. Si le serveur primaire défaille, la panne est détectée par les serveurs de secours et le client. Un nouveau serveur primaire est choisi parmi les serveurs de secours (l'algorithme de choix doit être commun au client et aux serveurs de secours, par exemple le premier serveur vivant d'une liste ordonnée). Le client réémet sa requête au nouveau serveur primaire qui la traite.

Dans certains cas de figure une optimisation de ce schéma de duplication peut être utilisée. Cette optimisation consiste à utiliser chaque serveur à la fois comme un primaire et un secondaire en même temps. Dans un tel modèle, pour un ensemble de `n` serveurs, chaque serveur est un primaire avec `n-1` secondaires.

2.2.3.3. Protocoles de diffusion

Les protocoles de diffusion font partie des outils de base permettant d'implémenter de la duplication de serveur dans un système distribué. De tels protocoles permettent à un émetteur de communiquer avec un groupe de N destinataires. La notion de groupe (g_N) est l'abstraction la plus couramment utilisée pour désigner un ensemble de N processus communiquant.

La gestion du groupe

Deux approches peuvent êtres employées pour gérer la constitution du groupe. Le groupe peut être constitué statiquement ou dynamiquement. Une constitution statique signifie que l'appartenance au groupe ne change pas durant toute la durée de l'exécution du système. Dans ce cas les processus sont supposés ne pas tomber en panne lors de cette exécution. Si tel était le cas, la constitution du groupe ne serait pas modifiée afin de prendre en compte la défaillance de l'un des membres. Les groupes statiques sont donc adaptés aux cas ne nécessitant pas de traitement spécifique lors de la panne de l'un des membres.

Dans le cas d'une constitution dynamique du groupe, les processus peuvent entrer et sortir à tout moment du groupe. La constitution du groupe varie donc au fil des pannes et des « réparations » impliquant des entrées et des sorties du groupe de la part des processus correspondants. A un instant *t* le groupe comprend donc un nombre n de membres compris entre

0 et N, lorsque N est le nombre total de processus impliqués dans le système. La notion de *vue* (V) modélise l'appartenance au groupe des n processus au fil du temps. La vue $V_0(g_N)$ désigne la composition du groupe à l'instant 0.

Dans un système asynchrone sujet à des pannes franches, il est impossible de fournir, à tout instant, à chaque membre du groupe, une vue exacte de la position courante du groupe [Chandra et al., 1996].

Propriétés des protocoles de diffusion

Dans un contexte de tolérance aux fautes, les protocoles de diffusion peuvent fournir différents niveaux de « fiabilité » et garantir, ou non, un certain nombre de propriétés parmi lesquelles on distinguera :

La **diffusion fiable**. Tout message diffusé est délivré exactement une fois à tous les processus corrects.

La **diffusion atomique**. La diffusion est fiable et les messages sont délivrés dans le même ordre à tous les destinataires.

La **diffusion FIFO**. Si un processus diffuse un message m1 avant un message m2, alors tous les processus doivent délivrer m1 avant m2.

La **diffusion causale**. Si la diffusion d'un message m1 précède causalement la diffusion d'un message m2, tous les processus corrects doivent délivrer m1 avant m2.

L'uniformité. L'uniformité d'une propriété indique que la dite propriété s'applique aussi bien aux processus corrects que fautifs.

2.3. Etude de cas

Nous allons présenter dans cette partie deux systèmes mettant en œuvre les mécanismes présentés dans les sections précédentes. Les études de cas porteront sur les plates-formes J2EE et .NET. Ces cas d'études ont été choisis pour plusieurs raisons. D'une part, à elles deux ces plates-formes illustrent les approches décrites précédemment et représentent les solutions majoritairement adoptées dans les autres systèmes. De plus le contexte J2EE est le cas d'étude choisi par Sardes afin de mettre en application les approches mises en œuvre.

2.3.1. Plate-forme J2EE

Cette section s'organise de la façon suivante : nous commencerons par rappeler les principes et l'architecture des systèmes J2EE puis nous détaillerons les mécanismes de tolérance aux fautes mis en œuvre dans ces systèmes.

2.3.1.1. Rappel sur les architectures J2EE

La spécification J2EE[1] précise qu'un environnement conforme doit fournir tous les outils nécessaires à un programmeur pour qu'il soit à même de ne se concentrer que sur le code métier de son application [Sun, 2005]. Dans le monde J2EE, les objets Java encapsulant le code métier ainsi que les données de l'application sont appelées des EJB[2]. Ces derniers nécessitent d'être hébergés par un serveur d'EJB afin d'être utilisable.

Une architecture typique de serveur J2EE (voir Figure 6) intègre les étages suivants :

⇒ Un serveur HTTP

⇒ Un serveur de Servlet

⇒ Un serveur à EJB

⇒ Un système de gestion de base de données

Lorsque une requête est reçue par le serveur HTTP, deux cas de figure peuvent se présenter. Soit la page demandée est statique, soit elle est dynamique. Dans le cas où elle est statique, le serveur HTTP la lit directement depuis son système de fichier et la renvoie au client. Dans le cas contraire, l'url demandée fait référence à une Servlet. La page HTML retournée au client est le résultat de l'exécution de la Servlet en question.

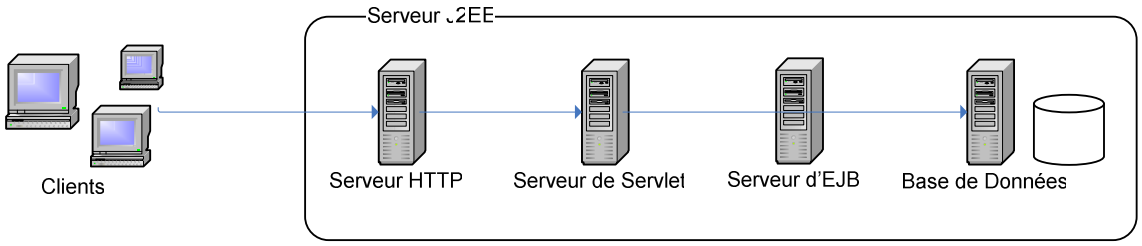

Figure 6 - Architecture d'une plate-forme J2EE

Cette dernière, lors de son exécution, peut faire appel à un ou des EJBs, dans ce cas, la Servlet utilise un annuaire pour localiser l'EJB dont elle a besoin et ensuite invoque à distance une méthode dessus. Le serveur d'EJB utilise ensuite une base de données pour stocker les données persistantes de l'application.

---

[1] *Java 2 Platform, Enterprise Edition*

[2] *Entreprise Java Bean*

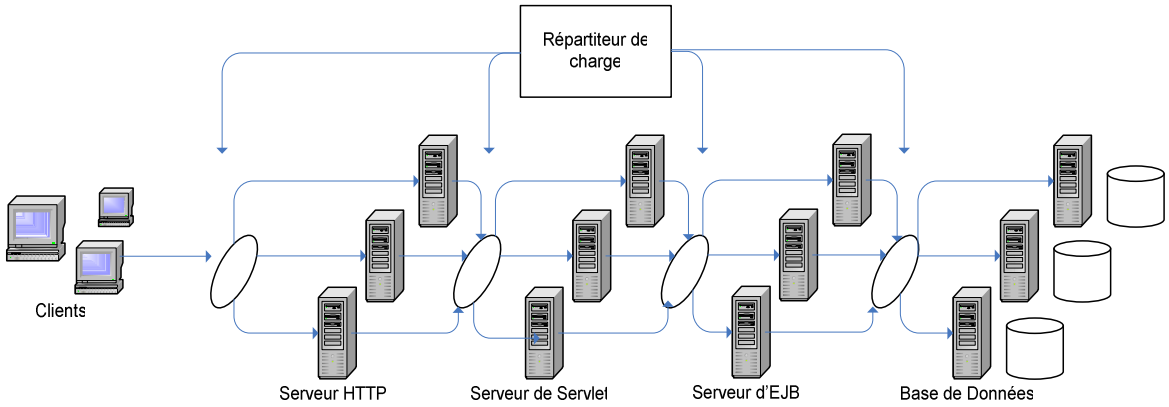

Figure 7 - Architecture J2EE en grappe

Afin de supporter les charges importantes auxquelles sont soumises les services Internet, les architectures multi-tiers autorisent la duplication de chacun des tiers comme illustré sur la Figure 7. Dans ce cas l'aiguillage des requêtes dans le système est géré par des répartiteurs de charges positionnés en amont de chaque étage dupliqué. Ces répartiteurs assurent que la charge a laquelle est soumise chaque machine est homogène.

2.3.1.2. Tolérance aux fautes dans les architectures J2EE

Le problème de la tolérance aux fautes et de la reprise après panne se pose de façon d'autant plus aiguë que l'intergiciel et l'application entretiennent un état et des données de session[1], ce qui est classiquement le cas dans un serveur d'application et une application J2EE. Afin de gérer correctement la tolérance aux fautes, il convient donc de localiser ces données afin de leur appliquer un traitement adéquat dans le but de les rendre pérennes. Le serveur Web Apache est sans état, son fonctionnement ne met pas en jeu d'état significatif pour l'application (son rôle se bornant au service de contenu statique). Le serveur de Servlet Tomcat en revanche possède un état. Les Servlets ont la possibilité de maintenir des données de session persistantes entre deux requêtes d'un client. Ces données doivent donc être prises en compte. Le conteneur des Servlets ne maintient aucune donnée non *reconstructible* lors de son éventuel recouvrement. Le serveur d'EJB héberge entre autre des Beans Session avec état. Ce type d'EJB maintient des données relatives à une session d'un client. Ces EJBs n'existent qu'en mémoire volatile sur le serveur. Ils sont donc sensibles à la défaillance de leur conteneur ou encore du serveur. La base de données est normalement manipulée dans un contexte transactionnel et garantie, par

[1] *Les données de session sont des données relatives à un ensemble d'interaction entre un client et le serveur (par exemple le contenu du caddy d'un client sur un site web de e-commerce).*

définition, le maintien de la cohérence et la persistance des données. Il n'y a donc pas de risque à ce niveau là.

Serveur Web - Apache

Le serveur Web Apache ne gère que le service de contenu statique, il n'a donc pas besoin de maintenir un état qui nécessiterait d'être restauré après une défaillance. La gestion des fautes est donc extrêmement simple dans la mesure où la réinsertion ne demande aucune procédure particulière, autre que la prise en compte au niveau du répartiteur de charge en amont.

Conteneur de Servlet - Tomcat

Le tiers présentation d'une architecture J2EE est le conteneur à Servlet. Le plus répandu est Tomcat [Shachor, 2005]. Nous étudierons ici les mécanismes mis en œuvre dans la version 5 de Tomcat pour la gestion de la tolérance aux fautes. Tomcat fournit une solution intégrée pour la mise en grappe et la duplication de session. Nous allons voir dans cette section en quoi ces solutions consistent.

Les communications entre les entités Tomcat sont assurées via des canaux IP de deux natures. Des canaux IP[1] pour la communication de groupe (*multicast*) sont employés pour la détection des pannes (diffusion de signaux HeartBeat). Les communications entre serveurs sont assurées via des canaux IP point à point classiques (par exemple pour le transfert de données de session, …).

Le serveur Tomcat implémente plusieurs algorithmes pour la duplication de session. La duplication sur le système de fichier, la duplication en base de données ou la duplication en mémoire. Seule la duplication en mémoire offre des performances optimales aussi nous ne détaillerons que celle-ci par la suite [Gama et al., 2004].

Le principe mis en œuvre pour la duplication de session en mémoire est une duplication du serveur de type Primaire-Sauvegarde où chaque répliquât joue un rôle symétrique (d'une façon similaire aux paires de processus dans le système Tandem [Bartlett, 1981]). Chaque serveur Tomcat joue deux rôles à la fois : le rôle de primaire en considérant tous les autres comme ses sauvegardes et le rôle de sauvegarde pour chacun des autres. Chaque serveur réplique donc les données des sessions sur tous les autres serveurs de la grappe.

[1] *Internet Protocol*

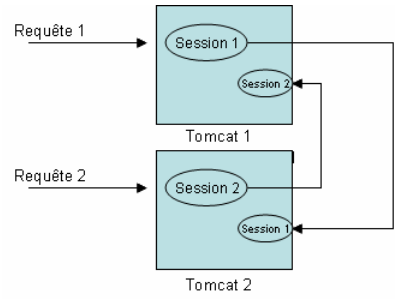

Figure 8 - Duplication des sessions sur Tomcat

Pour cela les Tomcats doivent se connaître entre eux. A cette fin, chaque serveur émet périodiquement des signaux HeartBeat en utilisant un protocole de communication de groupe (heartbeat avec délais de garde). De cette façon chaque serveur connaît la liste des machines « vivantes ». Ensuite lorsqu'une session est modifiée par une requête, le serveur répercute cette modification sur chacun de ses pairs vivants avant de renvoyer la page au client.

Conteneur EJB - JonAS / JBoss

Au niveau du tiers EJB, deux approches intéressantes de part leur complémentarité sont à noter : celle de CMI[1] qui tolère des pannes franches de machines et celle de JAGR[2] qui permet de tolérer les pannes transitoires d'EJB.

- **CMI – Cluster Method Invocation**

CMI est une évolution de RMI[3] conçu pour permettre de masquer les pannes franches de serveur JonAS. Le principe de fonctionnement est illustré sur la Figure 9. Chaque serveur JonAS déploie tous les Beans de l'application et son propre serveur JNDI[4] (l'application et le serveur sont donc en quelque sorte clonés). Les serveurs JNDI se connaissent entre eux et se synchronisent via des communications de groupe (Multicast) afin de tous enregistrer un talon client CMI.

[1] *Remote Method Invocation*

[2] *JBoss with Application-Generic Recovery*

[3] *Remote Method Invocation. C'est une couche de communication permettant à des objets distribués de communiquer à travers un réseau de façon transparente.*

[4] *Java Naming Directory Interface. C'est un service de nommage permettant à des objets distribués de se localiser les un les autres.*

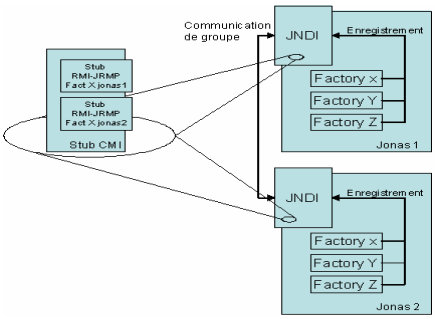

Figure 9 - Principe de fonctionnement de CMI

Les clients, dans notre cas les Servlets sur Tomcat, localisent un EJB en interrogeant l'un des serveurs JNDI disponible et récupèrent un talon client CMI. La servlet élit ensuite l'un des talons client RMI contenue dans celui de CMI et l'utilise ensuite comme pour une invocation RMI conventionnelle. Si lors de cet appel RMI, la connexion TCP est refusée, alors on considère que le serveur est défaillant et un autre talon client RMI est élu. Les serveurs étant tous idempotents cela n'a aucun impact sur le déroulement de la requête lui-même.

Cette solution présente le problème de ne pas autoriser l'application à maintenir des données de session ou conserver un quelconque cache local. Et ceci afin de garantir l'idempotence de tous les répliquâts de serveur.

- **JAGR**

Le serveur d'EJB JAGR (JBoss with Application-Generic Recovery) est une extension de JBoss qui permet de prendre en compte les défaillances transitoires au niveau application [Candea et al., 2003]. La Figure 10 montre l'architecture de ce serveur. Les éléments en rouge sont les ajouts effectués sur le serveur JBoss. Ces ajouts sont principalement des moniteurs, des gestionnaires, un composant d'interception et un injecteur de fautes. Les moniteurs servent à surveiller le système et à détecter les composants défaillants. Les gestionnaires assurent la réparation des composants et le composant d'interception sert à réguler le flot des requêtes durant les réparations.

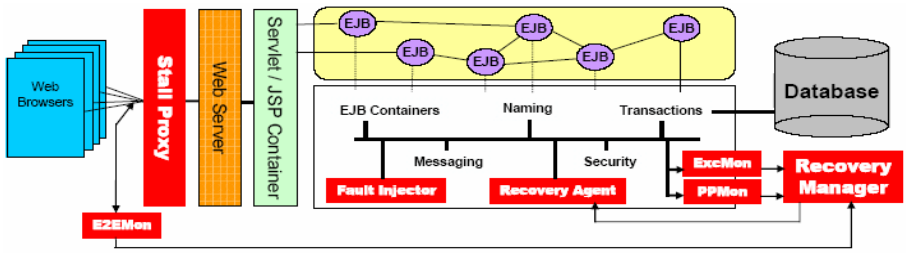

Figure 10 - Architecture de JAGR

JAGR exploite la notion de micro-reboot comme moyen de réparation pour les composants applicatifs (EJB) sujets à des pannes transitoires. Un micro-reboot est le redémarrage isolé d'un composant de l'application sans arrêter celle-ci pour autant. Dans ce cas un traitement n'accédant pas au composant en cours de micro-reboot n'est pas affecté par cette opération de recouvrement.

Les défaillances sont détectées à l'aide de trois moniteurs, un détecteur de type Pinpoint qui détecte les comportements anormaux de l'application, un détecteur d'exception (exception Java) et un moniteur End2End qui détecte les pages d'erreurs renvoyées par le serveur aux clients. Ces trois moniteurs sont légèrement redondants dans leur détection mais permettent de couvrir un large spectre de manifestations de défaillance (voir même détecter des défaillances là où il n'y en a pas ; détecteur complet mais pas exact).

L'implantation des micro-reboot dans JAGR ne tient pas compte de l'état du composant, s'il en possède un il est perdu lors du recouvrement. Afin de ne pas rencontrer de problème de cohérence, l'usage de Bean session à état est donc prohibé. De cette façon un micro-reboot peut survenir à tout moment sans pour autant générer de perte d'information.

Le micro-reboot étant très peu coûteux en temps, les défaillances sont masquées au client à l'aide d'un proxy (Stall Proxy) en entrée du système dont la fonction est le retardement des requêtes accédant à un composant défaillant jusqu'à la fin de son recouvrement.

Système de gestion de base de données - MySQL

Il est également possible de dupliquer la base de données afin d'en augmenter la disponibilité et la fiabilité. Cette duplication est mise en œuvre par l'outil c-jdbc[1] [Cecchet et al., 2004]. C-jdbc est un outil permettant de mettre en grappe une base de données de façon entièrement transparente pour l'application. La seule contrainte est que les applications doivent accéder aux données via le protocole standard JDBC.

[1] *Clustered – Java DataBase Connectivity*

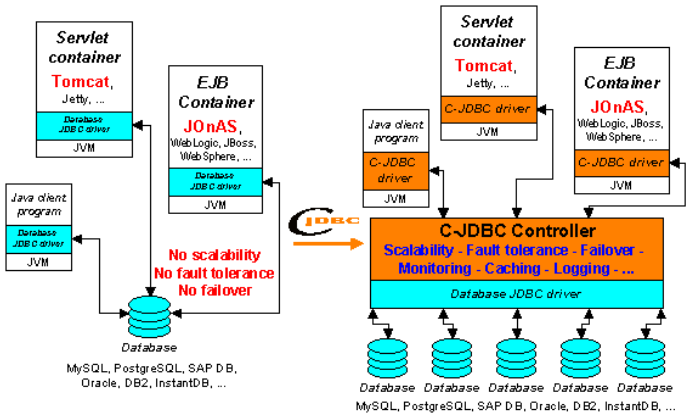

Figure 11 - Principe de c-jdbc – architecture sans/avec c-jdbc

Le principe de fonctionnement de c-jdbc est inspiré des mécanismes de RAID mis en œuvre sur les disques durs. L'ensemble Base de données et SGBD est répliqué sur plusieurs machines avec un protocole de type « réplication active ». L'ensemble des BD indépendantes ainsi formé est accédé exclusivement depuis un contrôleur (le contrôleur c-jdbc) qui maintient la cohérence entre les différentes bases. Le contrôleur se comporte comme s'il était un SGBD centralisé. Les applications envoient leurs requêtes SQL au contrôleur qui les transmet aux bases. Si une base vient à défaillir, les autres prennent le relais de façon entièrement transparente.

La plate-forme J2EE (avec ses variantes au niveau du tiers métier) permet de traiter les défaillances de type panne franche. L'accent est rarement mis sur le masquage des défaillances pour les requêtes en cours d'exécution au moment de la panne (JAGR fait figure d'exception).

La maîtrise de la cohérence de l'état persistant de l'application est en revanche excellent moyennant quelques contraintes sur la conception de l'application. Cela tient en très grande partie au fait que cette tâche est déléguée au système de gestion de base de données qui gère cette tâche de façon intrinsèque.

2.3.2. Plate-forme .NET

Phoenix/APP est une extension du runtime .NET qui permet d'ajouter de la tolérance aux fautes de façon transparente à une application .NET [Barga et al., 1999]. Le principe de Phoenix/App est de rendre tous les composants de l'application persistants en les dotant d'un mécanisme de réparation automatique avec une restauration de leur état précédant la défaillance.

Le principe exploité dans Phoenix consiste à journaliser (sur un support fiable) l'intégralité des interactions de chacun des composants. En cas de défaillance les journaux permettront par rejeu de restaurer l'état du composant tel qu'il était au moment de la défaillance. Le runtime de .NET permet de façon native une interception des requêtes entrant ou sortant d'un composant. Phoenix exploite cette possibilité de façon à implanter les mécanismes de réparation de manière entièrement générique par rapport à l'application (voir Figure 12).

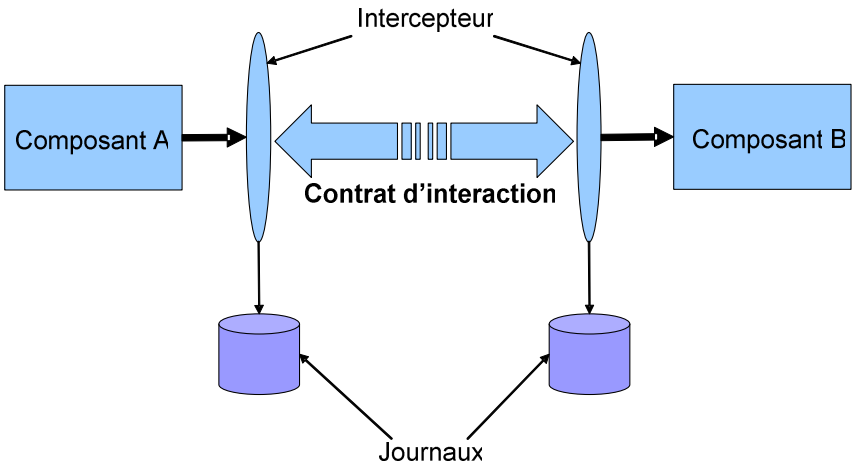

Figure 12 - principe de fonctionnement des Contrats d'interactions dans Phoenix/App

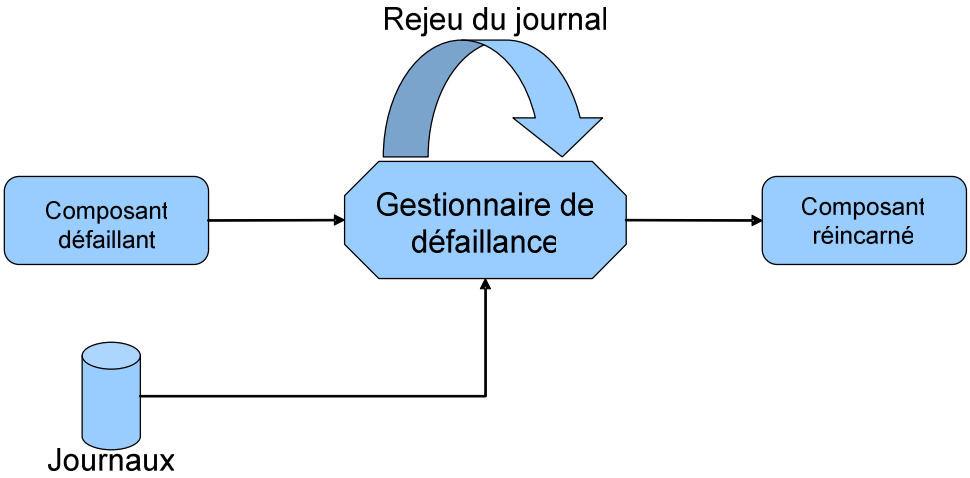

Figure 13 - principe de réparation des composants dans Phoenix/App

Phoenix définit un contrat d'interaction entre toute paire de composants communiquant entre eux. Ce contrat spécifie quel doit être le comportement de chacun des composants lors de l'occurrence d'une défaillance au cours d'une interaction [Barga et al., 2002]. Le respect de ce contrat d'interaction assure que dans tous les cas de pannes possibles, l'interaction sera exécutée une, et une seule fois. Au besoin, les composants défaillants sont réparés et leur état est restauré via un rejeu des interactions antérieures à la défaillance (voir Figure 13). Par opposition une approche classique à base de transaction n'aurait pu offrir qu'un niveau de garantie de type « tout ou rien ».

Cette approche permet de masquer l'occurrence de défaillance au client dans la mesure où l'exécution de sa requête est garantie par le respect des contrats d'interactions à l'intérieur du système.

2.4. Synthèse

Le domaine de la tolérance aux fautes est doté d'un travail théorique riche, aussi les algorithmes présentés au début de cette section sont les plus répandus et les plus fondamentaux du domaine. Il existe de nombreuses autres études sur la détection, le masquage et la réparation de défaillances aussi bien dans des contextes répartis que centralisés. Ces études abordent bien souvent ces problématiques de façon parfaitement décorrélées les unes des autres, or la gestion autonome des défaillances aborde l'ensemble de ces problèmes de façon simultanée et coordonnée. La gestion autonome implique l'intégration de ces diverses préoccupations dans une seule approche.

De façon générale, l'état de l'art concernant la tolérance aux fautes montre que :

⇒ **Les approches de tolérance aux fautes par masquage sont souvent privilégiées**.

Les travaux menés dans le contexte J2EE ne ciblent clairement pas du tout les systèmes autonomes et sont conçus dans l'optique d'augmenter la disponibilité des services Web. Dans ce contexte, l'approche consistant à rendre redondant les serveurs par duplication est largement préférée. Ces approches fournissent le masquage des défaillances de nœud dans les serveurs J2EE en grappe. En revanche très peu de travaux s'intéressent à la réparation des défaillances. On entend par là, la restauration de l'architecture du serveur après la défaillance d'un nœud. Prenons l'exemple d'un serveur J2EE tel que illustré dans la Figure 7 dans lequel chaque tiers est dupliqué sur trois nœuds. Dans ce cas le serveur peut tolérer la défaillance de deux machines tout en garantissant que le service restera disponible. Au delà, toutes les machines d'un tiers peuvent être en panne simultanément, et par conséquent, le service sera indisponible. D'une façon générale, on dira qu'un système de ce type est tolérant à n défaillances si chacun des tiers est dupliqué $n+1$ fois. Avec les mécanismes mis en œuvre actuellement dans les plates-formes J2EE, après l'occurrence d'une $i^{ème}$ défaillance, le système devient tolérant à $n-i$ défaillances. Rien ne permet actuellement de ramener le système à son état de tolérance à n défaillances. Pour cela, il faudrait réparer la défaillance en plus de la masquer.

⇒ **Les systèmes autonomes de réparation induisent souvent un surcoût à l'exécution conséquent**.

Il existe assez peu de validation expérimentale complète des travaux sur la tolérance aux fautes orientés vers des systèmes autonomes. Les quelques expérimentations dans ce sens montrent que les contre-performances des approches conventionnelles peuvent être problématiques, comme le montrent les évaluations de la plate-forme Phoenix/App. Cette plate-forme fournit le masquage des défaillances ainsi que la réparation des composants

défaillants. Pour y parvenir Phoenix/APP capture l'état du système via une journalisation des interactions entre les composants. Cette approche est très coûteuse et dégrade très largement les performances du système. Les expérimentations de cette plate-forme montrent une très sérieuse baisse de performance (de l'ordre d'un facteur 10) qui est essentiellement causée par l'implantation des contrats d'interactions. Ce type d'approche n'est donc vraiment adapté que dans certains cas très particuliers où la criticité de l'application justifie le coût du procédé. De façon générale, dans un contexte Web, le coût des algorithmes de capture de l'état, tels que décrits précédemment, est trop élevé pour être satisfaisant. L'intrusivité des modèles actuels semble donc être un problème majeur pour une intégration généralisée de ce type d'approche dans les applications courantes.

Chapitre III.

Conception du système d'auto-réparation

Ce chapitre décrit la conception du système d'auto-réparation. Ce système ayant vocation à être intégré dans une plate-forme d'administration autonome plus générale et plus complète, nous commençons d'abord par présenter cette dernière. Ensuite nous présentons les objectifs du système d'auto-réparation ainsi que les principes de conception.

3.1. Contexte

Le laboratoire Sardes développe un système d'administration autonome appelé Jade. Le projet Jade a pour objectif de fédérer et d'intégrer un ensemble de sous-projets focalisant chacun un aspect de l'administration autonome tel que décrit dans la section 1.2.2. Jade est conçu à l'aide d'un modèle de programmation par composants appelé Fractal [Bruneton et al., 2004]. La section 3.1.1 décrit brièvement les principes de ce modèle. La section 3.1.2 présente Jade dans un contexte général et la section 3.1.3 présente la personnalité J2EE de Jade. Cette personnalité de Jade est celle que nous utiliserons pour nos expérimentations (voir Chapitre V.).

3.1.1. Le modèle à composant Fractal

Le modèle Fractal est dédié à la construction et la configuration dynamique de systèmes. Fractal est un modèle général qui n'est pas dédié à un langage ou à un environnement d'exécution particulier. Une implantation du modèle Fractal se présente sous la forme d'une librairie qui permet la création et la manipulation de composants et d'architectures à base de composants. Le modèle Fractal distingue deux types de composants : les composants primitifs et les composants composites qui encapsulent un groupe de composants primitifs et/ou composites. Une caractéristique originale du modèle est qu'un composant peut être encapsulé simultanément dans plusieurs composites.

Un composant est constitué de deux parties : la partie de contrôle — qui expose les interfaces du composant et comporte des objets contrôleurs et intercepteurs —, et la partie

fonctionnelle — qui peut être soit une classe Java (pour les composants primitifs), soit des sous-composants (pour les composants composites).

Les interfaces d'un composant correspondent à ses points d'accès. On distingue deux types d'interfaces : les interfaces serveurs sont des points d'accès acceptant des appels de méthodes entrants. Elles correspondent donc aux services fournis par le composant. Les interfaces clients sont des points d'accès permettant des appels de méthodes sortants. Elles correspondent aux services requis par le composant.

La communication entre composants Fractal est uniquement possible si leurs interfaces sont liées. Fractal supporte deux types de liaisons : primitives et composites. Une liaison primitive est une liaison entre une interface client et une interface serveur appartenant au même espace d'adressage. Une liaison composite est un chemin de communication entre un nombre arbitraire d'interfaces de composants. Ces liaisons sont construites à l'aide d'un ensemble de liaisons primitives et de composants de liaisons.

3.1.2. Principes et architecture de Jade

Le choix d'une conception à base de composant est motivé par des besoins (i) de reconfiguration dynamique de la structure de l'application (tant d'administration que administrée), (ii) d'instrumentation du système administré et (iii) la construction d'une représentation du système causalement connecté à lui-même. Fractal permet de répondre à ces besoins, soit de façon immédiate (i) et (ii), soit en facilitant leurs mises en œuvre (iii). Jade utilise Julia, une implantation Java du modèle Fractal.

Jade est conçu pour administrer des applications quelconques pouvant ou non reposer sur un modèle de programmation homogène à Jade (composant Fractal versus code patrimonial). Pour gérer cette hétérogénéité, les applications sont encapsulées dans des composants Fractal. Ceci fournit la même abstraction pour n'importe quelle application patrimoniale. Ces composants ont à leur charge de réifier les actions de reconfiguration qu'ils subissent sur l'application encapsulée. Ainsi, on se ramène à un environnement homogène et uniforme constitué de composants Fractal.

La construction d'un système à base de composants Fractal fournit des capacités de reconfiguration dynamique avec le composant comme brique de base pour la reconfiguration. Nous exploitons cela dans Jade afin de mener à bien des opérations d'administration dont l'effet implique une reconfiguration de la structure du système (par exemple un ajout ou une suppression d'une machine dans un serveur en grappe).

La Figure 15 montre l'architecture de Jade d'un point de vue conceptuel. Trois grandes composantes sont identifiables :

Le système administré. Abstrait sous la forme de composants, il se présente comme un sous ensemble parmi les composants du système.

La représentation du système (*System Representation* – SR dans la suite du document). Le SR réifie l'architecture de Jade dans son ensemble (gestionnaires et système administré). Cela signifie que le SR est une vue[1] du système causalement connecté à lui-même. Ainsi chaque modification sur l'architecture du système est répercutée dans le SR et vice-versa (voir la Figure 14). En cas de défaillance de celui-ci, le SR est utilisé afin de retrouver des informations sur la configuration du système avant la défaillance (architecture du système, configuration de ses composants, etc.).

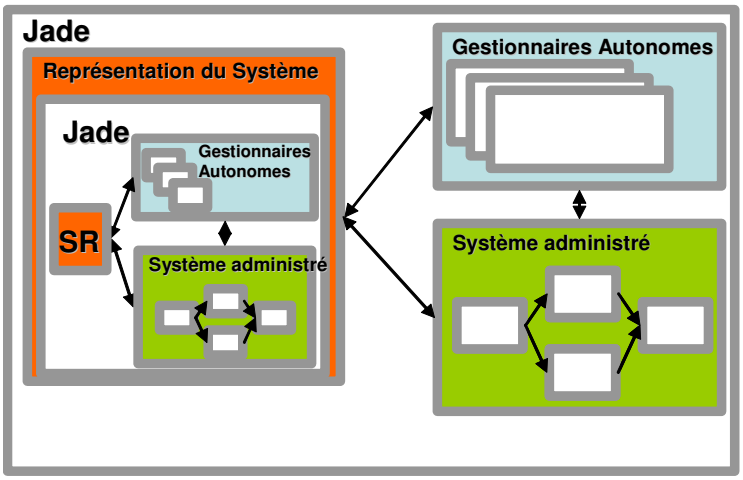

Figure 14 – Principe de la Représentation du Système

Les gestionnaires autonomes. Un gestionnaire autonome correspond à un aspect d'administration. Il fournit un comportement autonome au niveau de la ressource[2] dont il a la charge. Son usage est généralement optionnel et dépend du besoin particulier des applications et de leurs contextes d'utilisations. L'approche à composants permet d'obtenir une extrême souplesse dans les possibilités d'adaptation et d'extension de Jade. Chaque gestionnaire peut être ajouté ou enlevé de façon indépendante ou bien reconfiguré afin d'implanter une politique particulière de gestion de sa ressource adaptée au contexte. Jade fournit les gestionnaires suivants :

[1] *Dire que le SR est une vue signifie que le composant SR est utilisé comme une structure de données. La relation de composition dans ce cas prend une sémantique particulière. Les sous composants du SR ne servent à pas à déléguer une fonctionnalité mais constituent les donnés contenue par le SR. Dans ce cas précis ces donnés sont relatives à l'état de l'architecture du système.*

[2] *Ici le terme ressource peut désigner aussi bien une entité logiciel que matériel.*

⇒ **Le Gestionnaire de Grappes** (*Cluster Manager*). Le rôle du gestionnaire de grappe est d'implanter une politique d'allocation de nœud dans une grappe et de gérer le placement des composants du système sur les nœuds alloués. Chaque nœud est représenté par un composant qui fournit un mécanisme de création de composant à distance.

⇒ **Le Gestionnaire d'Application** (*Application Manager*). Le gestionnaire d'application gère l'architecture de l'application. Dans le cas présent, le gestionnaire permet de déployer et configurer un intergiciel J2EE ainsi qu'une application J2EE.

⇒ **Le Gestionnaire de Défaillance** (*Failure Manager*). Le gestionnaire de défaillance implémente une politique de réparation. Il est constitué d'une boucle de contrôle afin de superviser et réparer le système en cas de défaillance de nœud.

Le dépôt de ressources logicielles *(Software repository)*. Le dépôt permet de stoker et de manipuler les intergiciels et les applications constituant le système administré.

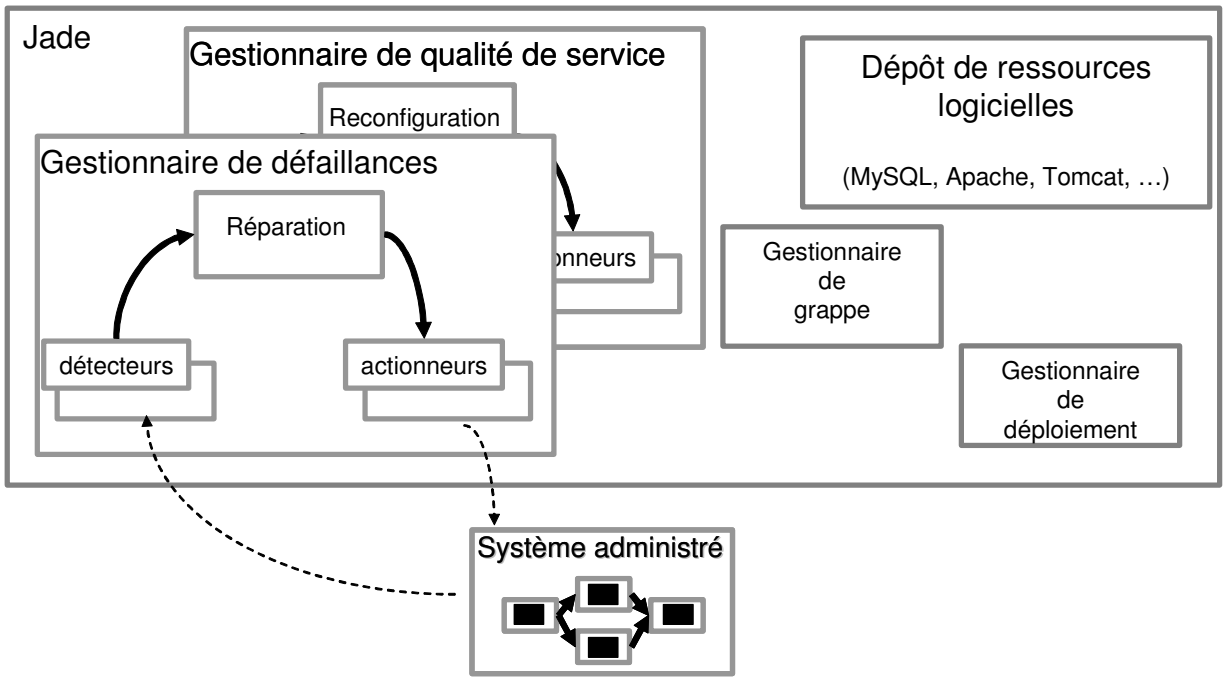

Figure 15 - Architecture générale de JADE

3.1.3. Etude de cas : Jade pour les systèmes J2EE en grappes

Chaque type de système à administrer présente des points communs mais également des spécificités propres. L'administration, pour être suffisamment *fine*, nécessite la prise en compte de ces spécificités. Dans cette optique, Jade propose des spécialisations pour certains types de

systèmes. Nous décrivons ici une personnalité particulière de Jade destinée à l'administration de systèmes J2EE. La suite de cette section décrit un certain nombre de rappels sur les systèmes J2EE et ensuite les particularités de Jade pour J2EE. Les systèmes J2EE sont d'une extrême complexité à installer, configurer et maintenir pour un administrateur humain. Ce type de système présente donc un réel challenge pour les systèmes d'administration autonome.

Comme nous l'avons déjà vu précédemment les systèmes d'administration autonome ne peuvent être conçus de façon parfaitement générique. Pour cela nous avons dérivé de Jade une personnalité J2EE. Elle fournit en plus des gestionnaires décrits précédemment un ensemble de composants adaptés à la supervision de ce type de plate-forme.

⇒ **L'ensemble des composants encapsulant Apache, Tomcat, JonAS et MySQL**. Ils sont spécifiques à l'application qu'ils encapsulent. Ils sont donc dépendants plus que tout du contexte applicatif. Les informations de configuration de ces serveurs étant dans des fichiers de configuration de format divers et varié, les composants d'encapsulation génèrent ces fichiers de façon à produire une configuration adéquate.

3.2. Objectifs

L'objectif est de construire un gestionnaire autonome de défaillances pour Jade, qui bien que adapté au contexte J2EE, reste le plus générique possible. Le gestionnaire devra assurer :

⇒ **Le traitement de pannes franches.** Le système devra assurer la détection et le traitement de pannes franches de machines.

⇒ **Traitement de pannes imbriquées.** Rappelons que `MTTF` est le temps moyen entre deux défaillances, et `MTTR` la durée moyenne de détection et de réparation. En fonction des valeurs de ces deux grandeurs, la probabilité d'avoir plusieurs machines défaillantes simultanément peut être potentiellement grande [Menascé et al., 2000]. Le gestionnaire de défaillance doit donc pouvoir réparer les pannes même en présence de nouvelles pannes au cours de la réparation.

⇒ **Auto-réparation du système d'administration**. Ce cas est particulier dans la mesure où une panne du processus de réparation lui-même implique un traitement particulier pour la détection (détecter la panne d'un détecteur) et la réparation (réparer le code de réparation).

Le contexte application utilisé pour l'expérimentation et la validation de notre approche sera les serveurs J2EE en grappe. Nous devrons donc assurer une réparation automatique des pannes franches de serveurs dans ce type d'architecture.

Une action de réparation se définit par une politique particulière, nous souhaitons pouvoir configurer (voir même reconfigurer dynamiquement) la politique de réparation mise en œuvre par le gestionnaire de défaillance. Le terme réparation est utilisé par opposition au terme masquage,

ce système s'inscrit donc dans une complémentarité des systèmes de masquage de défaillance dans les architectures J2EE[1] tel que décrit dans la section précédente.

3.3. Principes de conception

Dans cette section nous présentons les principes de conception du gestionnaire de défaillance. Nous commençons par détailler l'architecture générale du gestionnaire, puis nous présentons l'algorithme de réparation. Ensuite nous abordons le traitement pour les cas de défaillance imbriquée ainsi que celle du système d'administration lui-même.

3.3.1. Architecture générale

L'abstraction fournie par Jade permet de considérer l'ensemble formé par Jade et le système administré comme une composition quelconque de composants. L'ensemble peut potentiellement être distribué d'une façon arbitraire sur un ensemble quelconque de machines. Lorsque l'on considère dans ce contexte la défaillance d'une machine, le problème de sa réparation se ramène au problème de la réparation d'une structure à composants. La Figure 16 montre un exemple d'architecture de composants distribuée sur un ensemble de trois machines. La défaillance de l'une d'elle implique la perte d'un ensemble **quelconque** des sous composants de l'application.

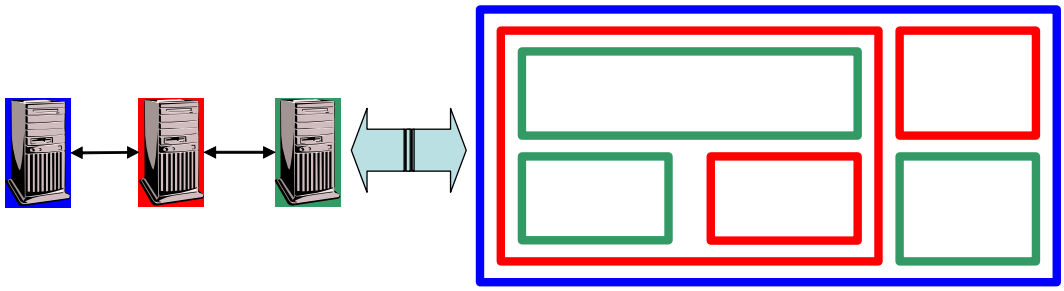

Figure 16 – Déploiement distribué sur trois nœuds, d'un composant hiérarchique.

La réparation d'une structure à composant signifie :

⇒ La reconstruction de toutes les relations de composition. Chaque composant a la connaissance de ses sous-composants et de ses parents. Cette connaissance doit être

[1] *Le module JK, CMI et C-JDBC*

maintenue à jour lors de la réinsertion d'un composant dans une architecture pré-existante.

⇒ La reconstruction de tous les liens entre composants (*binding*). Les liens entres les composants se matérialisent par des références nécessitant d'être mises à jour lors de la réparation d'un composant.

⇒ La configuration de chaque composant. Les composants peuvent maintenir un état qui doit être restauré lors de leur réparation.

Cette abstraction nous garantit que la restauration dans l'architecture d'un composant encapsulant une application patrimoniale implique une reconfiguration adéquate de ladite application. En d'autres termes nous assurons que la réparation de la structure à composants implique bien la réparation du système sous-jacent.

La Figure 17 présente l'architecture du gestionnaire de défaillance. La boucle de contrôle ainsi explicité se compose des trois éléments détecteurs-décision-actionneurs mais également du composant SR (Représentation du Système).

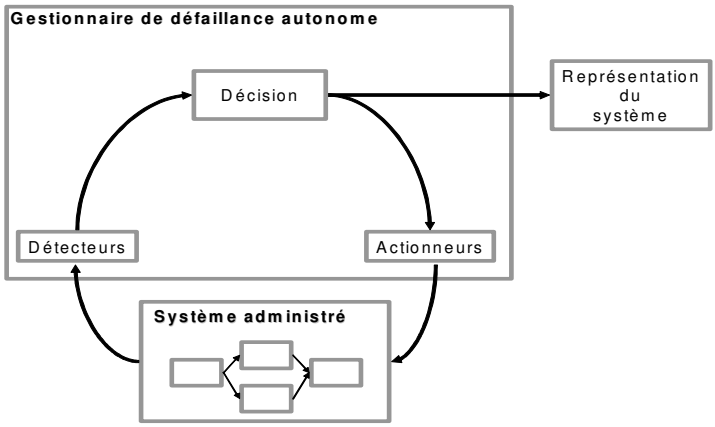

Figure 17 – Architecture du gestionnaire de défaillance

Nous allons à présent détailler chacun des composants mis en œuvre dans le gestionnaire.

3.3.1.1. Les détecteurs

Les hypothèses sur les pannes que nous souhaitons traiter guident le choix des détecteurs à mettre en œuvre. Dans le cas présent, nous traitons les défaillances franches de machine. Dans ces circonstances, de simples détecteurs de type *Ping* ou *Heartbeat* sont suffisants. Nous avons choisi d'utiliser des détecteurs *Ping* pour des raisons de simplicité de mise en œuvre.

3.3.1.2. La décision

L'étage de décision de la boucle de contrôle implante la politique de réparation mise en œuvre. Ce composant est notifié par les détecteurs de l'occurrence de défaillances et il déclenche ensuite, en fonction de la politique qu'il implante, des actions de réparation. Un changement de politique se matérialise par une reconfiguration de ce composant.

3.3.1.3. Le SR

Bien que les structures à composants Fractal soient introspectables[1] nous ne pouvons exploiter cette seule fonctionnalité du modèle afin de reconstruire une hiérarchie de composants après une défaillance, et ce pour la raison évidente qu'il n'est pas possible d'introspecter un composant défaillant (car il n'existe plus). Le SR nous permet de disposer d'une duplication de la structure des composants ainsi que de leurs propriétés. Ainsi lors du processus de réparation nous pouvons nous appuyer sur cette structure afin de retrouver toutes les informations pertinentes à la recréation et la restauration de la sous partie défaillante du système.

3.3.1.4. Les actionneurs

Les actionneurs ont été conçus pour les besoins du gestionnaire de défaillances. Néanmoins ils en sont conceptuellement séparés et n'en dépendent pas du tout. Le principe de base est que tout composant utilisé dans le système doit fournir un actionneur permettant d'agir sur lui-même. Ainsi le concepteur d'un composant doit fournir un actionneur associé.

3.3.2. Algorithme général de réparation

Il est possible d'exprimer un algorithme général de réparation d'une structure quelconque de composants. Cet algorithme s'appuie uniquement sur l'abstraction fournie par les composants et ne voit pas le fait que certains d'entre eux peuvent encapsuler des applications patrimoniales (par exemple un serveur d'EJB).

De façon générale, le traitement consiste à identifier les composants défaillants et à les restaurer à l'identique. Si l'on dénote par $e(t)$ l'état (d'un point de vu architectural) du système à un instant t, alors $e(t-\varepsilon)$ désigne l'état de l'architecture du système à l'instant précédant t. On souhaite que pour une défaillance survenant à un instant t, le processus de réparation garantisse que : $e(t-\varepsilon)=e(t+MTTR)$. En d'autres termes le traitement mis en œuvre doit garantir que le système sera restauré dans son état précédent la défaillance. Notre cadre de travail n'excluant pas qu'une défaillance puisse survenir dans l'intervalle de temps $[t , t+MTTR]$, cela signifie que le traitement doit être capable de réévaluer le travail déjà accompli au moment de la nouvelle défaillance et éventuellement de le modifier (reconfiguration) lorsqu'il ne s'agit pas de le recommencer.

[1] *Les composants Fractal fournissent un certain nombre de contrôleur qui permettent d'inspecter et manipuler leur configuration et leur contenu. A partir d'un composant nous pouvons donc connaître ses parents (super composants) et ses sous composants s'il en a.*

Les principales étapes de l'algorithme de réparation sont les suivantes :

⇒ **Isolation de la défaillance.** Cette étape consiste à calculer l'ensemble des composants impactés par la défaillance : soit directement car le composant était déployé sur le nœud en panne, soit indirectement car le composant était lié[1] à un composant défaillant. La connaissance d'un nœud défaillant n'implique pas de façon immédiate la connaissance des composants défaillants, aussi cette information est retrouvée à l'aide du SR.

⇒ **Allocation d'un nouveau nœud.** Nos hypothèses de défaillances concernent les pannes franches de nœud. La réparation doit donc nécessairement avoir lieu sur une machine autre que celle défaillante. Pour cela nous devons allouer une machine de remplacement. Diverses politiques d'allocation sont envisageables à ce niveau.

⇒ **Recréation des composants défaillants.** A partir des informations contenues dans le SR et l'isolation de la défaillance, nous réinstancions de nouveaux composants sur la machine précédemment allouée.

⇒ **Reconfiguration des composant recréés.** Chaque composant précédemment recréé est reconfiguré et réinséré dans l'architecture des composants non impactés par la défaillance.

⇒ **Redémarrage des composants.** Les composants nouvellement créés doivent être démarrés ainsi que ceux qui les utilisaient.

⇒ **Mise à jour des éléments d'administration.** La disparition de la machine défaillante doit être répercutée sur un certain nombre de composants d'administration. Par exemple les détecteurs doivent cesser de surveiller la machine et le SR doit mettre à jour sa structure afin de refléter cette reconfiguration du système à l'exécution.

3.3.3. Pannes imbriquées

Rappelons que le terme de panne imbriquée désigne le cas où une défaillance survient lors du processus de réparation d'une précédente défaillance. La première façon venant à l'esprit pour gérer ce cas consiste à sérialiser les traitements liés à chaque réparation. La Figure 18 illustre un tel scénario mettant en œuvre l'imbrication de deux défaillances. La figure indique en trait épais les traitements liés à la réparation d'une défaillance i en fonction du temps.

[1] *Pour un composant il y a deux façons d'être lié à un autre : soit par un lien de composition, au quel cas l'un est un sous composant de l'autre, soit par une liaison (binding) entre une interface cliente de l'un et une interface serveur de l'autre.*

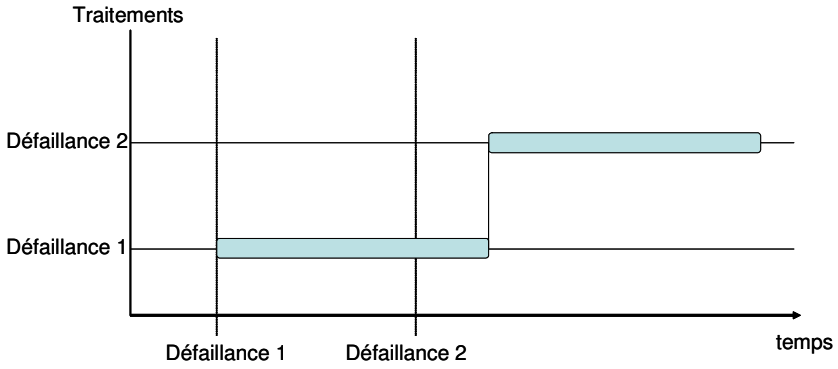

Figure 18 – Traitement sérialisé des défaillances

Dans cet exemple l'occurrence de la défaillance 2 est ignorée jusqu'à la fin du traitement de la défaillance 1. Cette approche, optimiste, fonctionne très bien tant que les ensembles de composants impactés par les deux défaillances sont disjoints.

En revanche, lorsque ces ensembles ne sont pas disjoints, certaines des étapes de l'algorithme de la section précédente seront impactées par l'occurrence de la seconde défaillance. Ce cas se produit lorsque l'une des étapes de l'algorithme fait appel à des composants devenus indisponibles en raison de la seconde défaillance. Par exemple l'étape consistant à recréer les composants en panne est impactée par la défaillance du nœud élu pour la réparation.

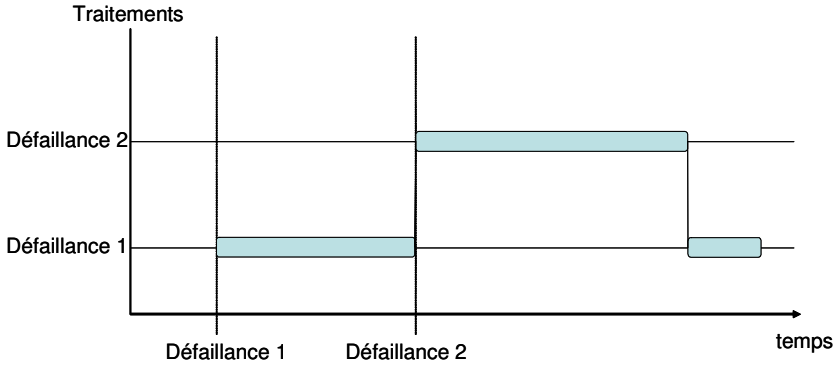

Figure 19 – Traitement imbriqué des défaillances

Une solution permettant de résoudre ce problème est de rendre le traitement de la défaillance imbriquée prioritaire sur le traitement en cours. De cette façon, les composants manipulés dans le premier traitement seront restaurés par le second traitement. La Figure 19 illustre un tel scénario. Le traitement de la défaillance 1 est arrêté lors de l'occurrence de la

seconde défaillance. Cette dernière est traitée puis le traitement dérouté de la première défaillance est repris. Cette approche, pessimiste présente l'inconvénient de systématiquement dérouter un traitement en cas de nouvelle défaillance, même si les deux traitements sont entièrement disjoints.

Il existe une approche intermédiaire consistant à ne dérouter un traitement que lorsque celui-ci est impacté par une autre défaillance. Ainsi, deux défaillances imbriquées impliquant chacune deux ensembles disjoints de composant seront traitées séquentiellement. En revanche deux défaillances liées seront traitées de façons imbriquées.

3.3.4. Défaillances du système d'administration

La section 3.3.2 décrivait le traitement des défaillances dans le cas général, toutefois il existe deux cas de panne particuliers nécessitant un traitement spécifique. L'objet de cette section est de détailler les cas où une défaillance survient dans le gestionnaire de défaillance lui-même et dans le SR.

3.3.4.1. Pannes du gestionnaire de défaillance

La façon la plus simple de traiter ce cas de panne est d'utiliser une approche similaire à ce qui est fait dans Tandem [Bartlett, 1981]. Dans notre cas le principe consiste à déployer un second gestionnaire de défaillance. Les deux gestionnaires fonctionnent ensemble selon un modèle de duplication de type primaire-secondaire. Le second gestionnaire a deux rôles :

⇒ Il assure son rôle de gestionnaire secondaire vis-à-vis du gestionnaire primaire. Cela signifie qu'en cas de défaillance du primaire, il pourra achever les opérations de réparation en cours au moment de la défaillance. Cela signifie que le traitement de réparation doit être interruptible et doit pouvoir être repris. Le mécanisme assurant ceci est le même que celui utilisé pour le traitement des défaillances imbriquées.

⇒ Le gestionnaire secondaire assure la réparation du primaire. Cette réparation ne pose pas de problèmes spécifiques. Notons que le primaire est également capable de réparer le secondaire. Cette réparation n'est pas différente de n'importe quelle autre réparation.

3.3.4.2. Pannes de la Représentation du Système

Le SR est la structure de donnée représentant le système, le cas de sa défaillance doit donc être traité de façon spécifique afin de ne pas mettre le système dans un état de vulnérabilité. En effet, nous pourrions imaginer des solutions visant à reconstruire l'état du SR à partir des données du système à l'exécution. L'inconvénient de telles approches est que le système se retrouve vulnérable face à une seconde défaillance.

En conséquence, le SR doit être rendu persistant d'une façon ou d'une autre. L'approche la plus simple consiste à sérialiser son état sur un support persistant (par exemple un montage NFS). Une approche plus optimum serait de dupliquer le SR selon l'un modèle de primaire-secondaire.

Chapitre IV.

Mise en œuvre du système d'auto-réparation

Nous présentons dans ce chapitre les détails d'implantation et de mise en œuvre de chacun des éléments de la boucle de contrôle mis en évidences dans le chapitre précédent (détecteurs – décision – actionneurs / voir Figure 20).

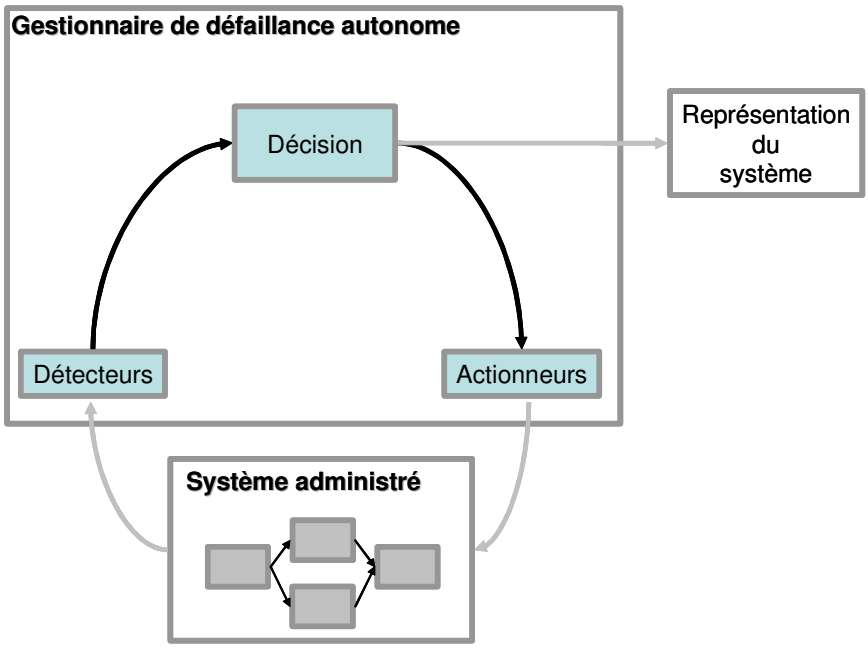

Figure 20 – boucle de contrôle du gestionnaire de défaillances

Nous commencerons par la présentation des détecteurs de défaillance, ensuite nous présenterons l'étage de décision et enfin les actionneurs.

4.1. Détecteurs

Les détecteurs déployés sont de type Ping. Ils assurent une surveillance du système et détectent la panne franche de nœud physique. Ils se présentent sous la forme de composants Fractal dont la fonction consiste à envoyer une requête Ping sur une liste de machines. La liste de machines à surveiller est construite dynamiquement lors du déploiement de l'architecture sur une grappe.

Lorsque l'un des détecteurs détecte la défaillance d'un nœud du système, il envoie une notification au composant décisionnel de la boucle de contrôle. Les détecteurs sont conçus pour supporter une reconfiguration dynamique afin prendre en compte l'entrée et la sortie de nœuds dans le système. Ainsi lors d'une panne de machine, l'une des étapes du processus de réparation sera de reconfigurer les sondes afin de ne plus surveiller la machine défaillante.

Notons que nos hypothèses de fautes concernent les pannes franches de machine, aussi nos détecteurs Ping fonctionnent à un grain machine. Néanmoins la même mécanique continue à être valide lorsque l'on considère le cas des pannes franche de composants. Dans ce cas de figure, les éléments à *Pinger* seront les composant plutôt que les machines.

4.2. Décision : reconfiguration pour la réparation de pannes

Dans le chapitre précédent, nous avons spécifié le comportement et les fonctionnalités que doit apporter le composant de décision. Rappelons qu'il doit implanter une politique de réparation et mettre en œuvre l'algorithme décrit précédemment.

L'algorithme de réparation est décomposable en étapes clairement identifiées, chacune ayant une action précise sur le système. Rappelons que les actions sur le système sont effectuées par le biais des actionneurs fournis par le système lui-même. Dans ce contexte l'algorithme de réparation s'exprime par un assemblage d'actionneurs. La mise en œuvre de l'algorithme par le composant de décision se résume donc à la gestion d'un ordonnancement particulier des traitements des différents actionneurs.

Cette décomposition de l'algorithme sous la forme d'un automate à état rend son exécution d'autant plus facile à interrompre et à reprendre. Si l'on mémorise l'état courant de l'automate, il est possible de stopper une action de réparation et de la reprendre plus tard. Ceci permet de prendre en compte aussi bien le cas d'un traitement de défaillance imbriquée que la reprise d'un traitement interrompu. L'état courant du traitement est conservé dans un composant spécifique appelé le Plan de Réparation (*RepairPlan*). La Figure 21 montre l'architecture complète de la boucle de contrôle d'un point de vue d'implantation.

L'étage de décision se présente sous la forme d'un composant Fractal disposant de trois interfaces. La première permet aux détecteurs de lui notifier la présence d'une défaillance et lui transmettre les informations relatives à cet évènement (en l'occurrence l'identité du nœud défaillant). La seconde interface relie le composant de décision à chacun des actionneurs du système. Ces liens servent à lancer le traitement des actionneurs. La troisième permet au composant décisionnel d'écrire dans le plan de réparation l'état courant du processus de

réparation ainsi que les informations relatives à ce processus (par exemple l'identité du nœud à réparer).

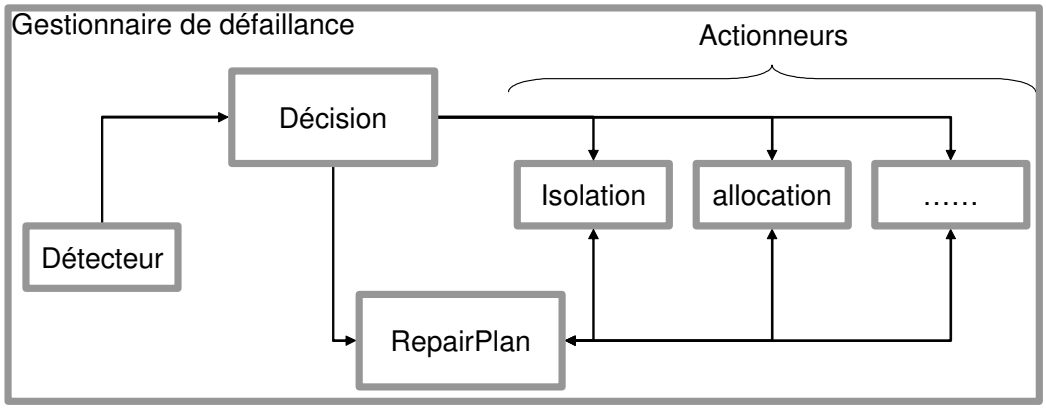

Figure 21 – Architecture générale de la boucle de contrôle

Le RepairPlan est un composant du système comme un autre, il est donc naturellement représenté dans le SR. Toutes les informations qu'il contient sont donc sauvegardées *gratuitement* par ce mécanisme. Si une panne du gestionnaire de défaillance survient au cours d'une réparation alors toutes les informations permettant une reprise du processus de réparation pourront être restaurées d'après le SR. Ce mécanisme de reprise n'est pour l'instant pas encore implémenté, en revanche toutes les briques de bases pour un tel traitement sont en place. Toutes les informations du RepairPlan étant modélisées sous la forme de composants, leurs restaurations sont accomplies strictement de la même façon que n'importe quel autre composant.

Si une deuxième notification de défaillance intervient lors d'un processus de réparation, l'étage de décision stoppe l'exécution de la réparation en cours et produit un nouveau plan de réparation correspondant à la nouvelle défaillance. Ensuite la nouvelle défaillance est traitée de façon classique. Une fois terminé, le traitement précédent est repris à partir du point de son exécution auquel il avait été stoppé.

4.3. Actionneurs

Les actionneurs sont conçus pour être manipulés de façon totalement générique. Pour cela ils utilisent tous une interface unique (voir la Figure 23).

Un actionneur fonctionne nécessairement conjointement avec un plan de réparation (*RepairPlan*). Le plan de réparation comporte toutes les informations pertinentes pour l'action d'un actionneur. Une information dans le plan de réparation peut être vue comme une ressource du point de vue d'un actionneur. La sémantique de la méthode `actuate()` d'un actionneur

comprend deux choses : (i) l'actionneur s'autoconfigure grâce au plan de réparation, cette configuration peut se voir comme la consommation de l'information du plan de réparation. (ii) Il effectue l'action pour laquelle il est conçu (voir Figure 24). Notons que l'action d'un actionneur peut produire une information dans le plan de réparation. Par exemple, un actionneur servant à allouer un nouveau nœud dans une grappe va certes, effectuer l'allocation, mais aussi exporter l'information concernant le nœud alloué vers le plan de réparation. Cette information pourra ensuite être utilisée pour la configuration d'un autre actionneur (par exemple un Recréateur de nœud). La Figure 22 illustre le mode de fonctionnement des actionneurs entres eux. L'information nouveau nœud est produite par l'électeur puis consommée par le Recréateur afin de paramétrer le choix de la machine où les composants seront recréés.

Ce mode de fonctionnement permet un assemblage arbitraire d'actionneurs afin d'implanter une politique particulière. Cet assemblage doit néanmoins être correct. Un assemblage est dit correct s'il est défini par une combinaison d'actionneurs telle que toute information demandée par un actionneur sera produite par un autre au bout d'un temps fini. Ces derniers se synchronisent ensuite sur les informations qu'ils produisent/accèdent dans le plan de réparation.

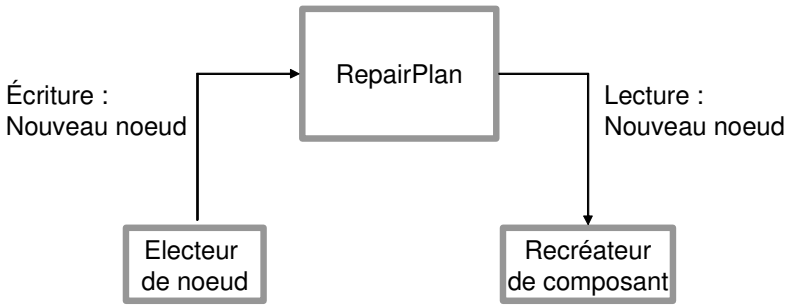

Figure 22 - Mode de fonctionnement des actionneurs

L'assemblage d'actionneurs est décrit à l'aide d'un ADL[1] Fractal dans lequel les actionneurs mis en œuvre dans le système sont spécifiés. Il serait, par ailleurs, intéressant de fournir un support permettant d'effectuer une vérification statique de la validité de l'assemblage.

A titre d'exemple, voici quelques-uns des actionneurs dont nous disposons dans notre système actuel : un électeur de nouveau nœud, un recréateur de composant ou encore un actionneur mettant à jour un détecteur. Lorsque cela a un sens, ces composants sont spécialisés de plusieurs façons afin de fournir des politiques particulières de réparation. Ainsi par reconfiguration du composant électeur de nœud nous pouvons modifier son comportement.

[1] *Architecture Description Language. Il s'agit d'un langage permettant de décrire une architecture logicielle à base de composants. Pour cela un ADL permet la description explicite des composants, de leurs interconnexions et de leurs configurations.*

L'algorithme de choix du nœud sur lequel l'application sera réparée peut forcer l'allocation d'un nouveau nœud ou au contraire réparer l'application sur un nœud qu'elle occupait déjà.

```
Public interface Actuator{

    public void actuate() throws Exception ;

}
```

Figure 23 - Interface serveur d'un actionneur

```
public abstract class AbstractActuator implements Actuator {
       public void actuate() {
              try{
                     configure();
                     doActuate();
              }catch(Exception e){
                     throw new ActuatorException() ;
              }
       }
       protected abstract void configure() throws Exception;
       protected abstract void doActuate() throws Exception;
}
```

Figure 24 – Comportement générique d'un actionneur

4.4. Réalisations

Voici quelques données sur la dimension et la complexité du gestionnaire de défaillance ainsi que notre plate-forme d'administration autonome, Jade.

	Gestionnaire de défaillance	Tout Jade
Nombre de classes Java	30	130
Nombre de fichiers ADL	18	51
Nombre de lignes de code	5525	21506
Nombre de détecteurs	1	
Nombre d'actionneurs	12	
Nombre de composants de décision	2	

Tableau 3 – Données sur le prototype

Chapitre V.

Validation expérimentale

Afin d'évaluer la pertinence et la faisabilité de ces travaux, une série d'expériences préliminaires ont été menées au préalable à l'aide d'un prototype minimal de Jade [Bouchenak et al., 2005]. Cette validation expérimentale s'appuie sur une étude comparative des performances d'un système administré avec Jade versus une administration humaine. Deux scénarii seront développés dans la suite de ce chapitre.

5.1. Environnement expérimental

5.1.1. Application d'expérimentation

Notre application d'expérimentation est RUBiS en version Servlets seules [Amza et al., 2002], [RUBiS, 2005]. RUBiS est une application J2EE modélisée d'après le site d'enchères électronique eBay. RUBiS implémente toutes les fonctions de bases de ce type de site avec notamment la possibilité pour un utilisateur de vendre, enchérir, rechercher et naviguer entre les objets mis en vente. RUBiS comporte également un injecteur de charge. Cet injecteur émule des clients se connectant sur le site à l'aide de leur navigateur web.

5.1.2. Environnement logiciel

L'architecture J2EE que nous utilisons met en œuvre : un serveur web Apache pour le service des pages statiques, quatre conteneurs de Servlets Tomcat pour le service des pages à contenu dynamique et une base de données MySQL (voir la Figure 25).

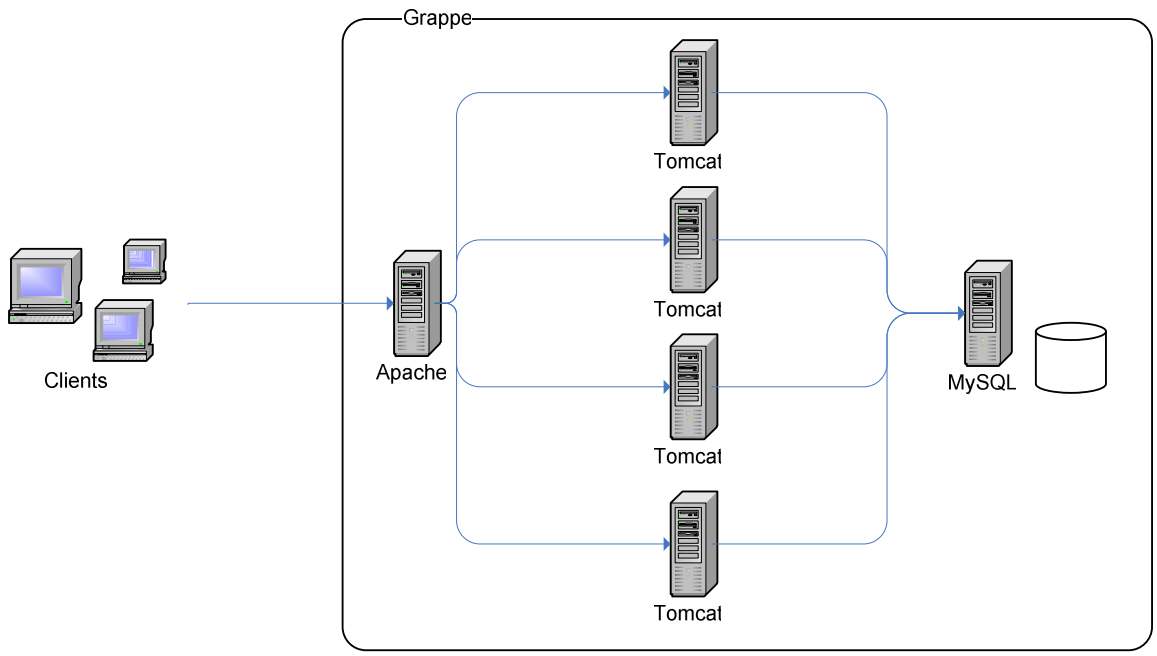

Figure 25 – Plate-forme expérimentale

5.1.3. Environnement matériel

La plate-forme matérielle d'expérimentation se compose de six machines dédiées interconnectées à travers un réseau Ethernet gigabit. Chaque machine est un biprocesseur Itanium à 900 MHz avec 3GB de mémoire vive ainsi qu'un disque SCSI à 10000 tours par minute. Les machines utilisent un noyau Linux 2.4 SMP ia64.

5.2. Auto-réparation du système administré

Le scénario présenté ici a pour but d'évaluer le gain amené par le gestionnaire autonome de défaillance. Dans cette expérience, nous injectons successivement trois défaillances sur les serveurs de Servlet Tomcat. La Figure 26 montre les évolutions du taux d'utilisation du processeur attendues sur les serveurs Tomcat. Dans le cas où le système est administré sans Jade, lors de la défaillance de l'un des serveurs, la panne est masquée et la charge qu'il supportait est redirigée vers les serveurs restants. Dans les cas où le système est administré automatiquement, le palier observé lors de la défaillance est le même, mais le système est réparé et la charge revient à son état initial.

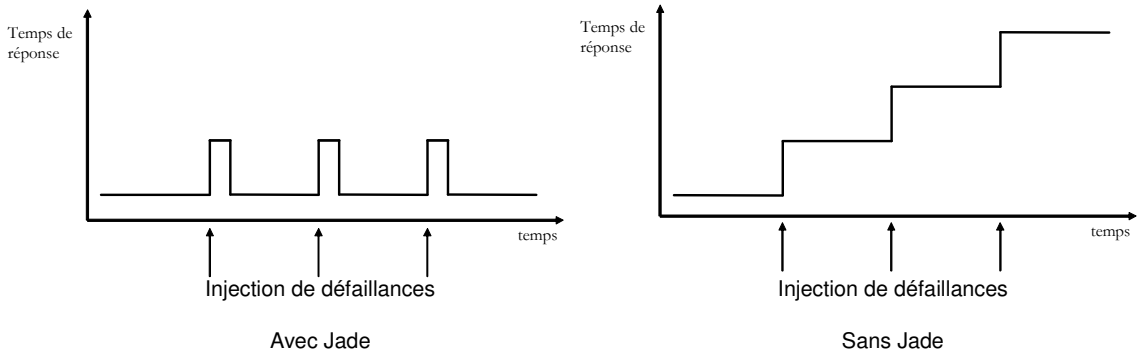

Figure 26 – Temps de réponse de l'application

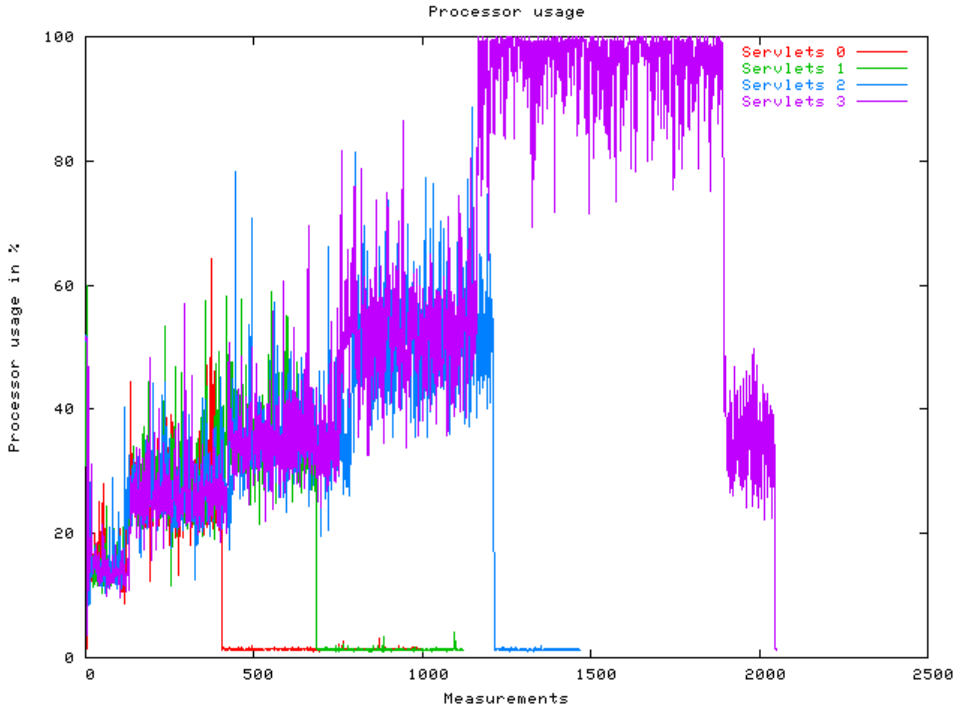

Figure 27 – Performances de l'application sans Jade[1]

[1] *[Bouchenak et al., 2005]*

La Figure 27 montre l'évolution du taux d'utilisation du processeur sur chacun des quatre serveurs Tomcat. Comme l'équilibrage de charge fonctionne bien les courbes de chaque serveur se superposent. Nous observons que à chaque défaillance de serveur, la charge se re-répartit sur les machines restantes, ce qui conduit à une augmentation du niveau de ressource utilisée. Le système continue à fonctionner car le masquage de la défaillance est assuré mais le système fonctionne en mode dégradé jusqu'à arriver à la saturation complète après la troisième défaillance.

Avec Jade nous montrons que la réparation automatique des défaillances permet de maintenir le système dans sont état optimal.

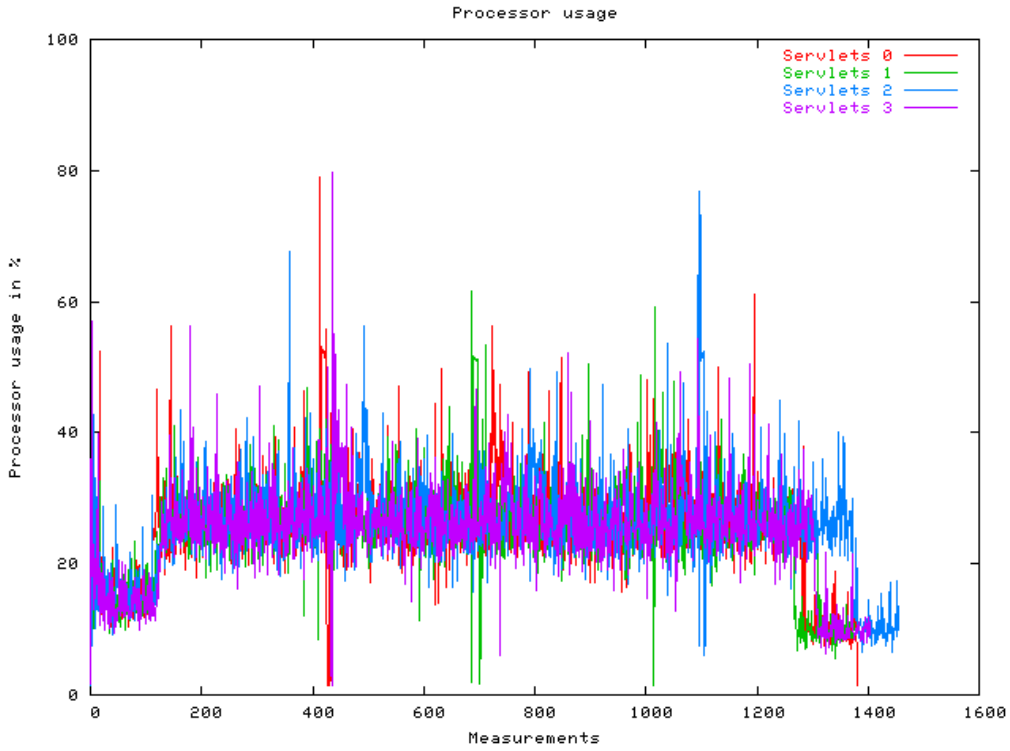

Figure 28 - Performances de l'application avec Jade [1]

La Figure 28 montre l'évolution de l'utilisation du processeur sur les serveurs Tomcat lors d'un scénario identique au précédant à la différence que le système est administré par Jade. Nous observons que la quantité de ressources utilisées reste constante tout au long du banc d'essai

[1] *[Bouchenak et al., 2005]*

excepté lors du traitement des défaillances ou le système est légèrement plus chargé. Les pics observés correspondent bien aux prévisions de la Figure 26.

5.3. Evaluation de l'intrusivité de la boucle de contrôle

Le but de cette expérience est d'évaluer l'impact du code d'administration de Jade sur les performances du système administré. Les types de scénarii envisagés ici ne font pas intervenir de défaillance. Le but est simplement de mettre en évidence la très faible intrusivité de notre approche.

	Système administré sans Jade	Système administré par Jade
Débit (req/sec)	82,75	82,75
Temps de réponse moyen (ms)	194,5	214,75

Tableau 4 - Evaluation du surcoût à l'exécution de l'application RUBiS avec Jade[1].

Le tableau 4 indique les débits et temps de latence obtenus pour l'application RUBiS avec et sans Jade. Nous constatons que le débit est rigoureusement identique dans les deux cas de figure. En revanche le temps de latence est légèrement supérieur lorsque le système fonctionne sous l'administration de Jade. Le seul surcoût impliqué par Jade est celui induit par les détecteurs du gestionnaire de défaillances.

[1] *[Bouchenak et al., 2005]*

Chapitre VI.

Conclusion et perspectives

Les systèmes informatiques sont d'une complexité croissante, et de même, l'administration de ces systèmes se complexifie elle aussi. En conséquence, la gestion au jour le jour de gros système, peut se révéler coûteuse tant en ressources humaines qu'en ressources matérielles. Devant ce constat, les systèmes permettant d'automatiser l'administration de ces applications sont appelés à devenir des outils indispensables. Parmi l'ensemble des aspects que couvre l'administration autonome, nous avons vu que l'auto-réparation est l'une des préoccupations centrales du domaine.

Notre travail s'inscrit dans la conception et le prototypage d'un service d'auto-réparation pour les outils d'administration autonome. Le but est de fournir un mécanisme de réparation des défaillances, complémentaire à ceux, plus classiques, de masquages souvent intégrés dans les applications elles-mêmes.

Le prototype décrit dans ce rapport permet de réparer, de façon entièrement automatisée, une application répartie quelconque. Les algorithmes mis en œuvre ne font aucune supposition sur la nature de l'application gérée par l'outil d'administration et, par conséquent, restent les plus génériques possibles.

Les perspectives à court terme sont d'achever la mise en œuvre et la validation des cas de défaillances imbriquées. Cela demande plusieurs choses : la production d'un banc d'essai approprié et la fin de la conception et de l'implantation de la préemption sur le processus de réparation.

Pour l'instant nous ne nous sommes intéressé qu'à un modèle de défaillances de type pannes franches de machines. Ce type de défaillance reste malgré tout simpliste, aussi, a plus long terme les objectifs sont de permettre les traitements de types de pannes différentes. Nous pensons principalement au cas des défaillances logicielles franches et transitoires. De plus afin de valider la généricité de notre approche, il est indispensable d'appliquer nos méthodes sur d'autres cas d'utilisation que J2EE afin de mettre en évidence la portabilité de notre outil.

Chapitre VII.

Références bibliographiques

[Amza et al., 2002], C. Amza, A. Chanda, A.L. Cox, S. Elnikety, R. Gil, K. Rajamani, W. Zwaenepoel, E. Cecchet et J. Marguerite, "Specification and Implementation of Dynamic Web Site Benchmarks", *WWC-5: IEEE 5th Annual Workshop on Workload Characterization,* 25 novembre 2002.

[Barga et al., 2002], R. Barga, D. Lomet et G. Weikum, "Recovery Guarantees for General Multi-Tier Applications.", *Proceedings of the 18th International Conference on Data Engineering,* San Jose*,* 26 février - 1 mars 2002.

[Barga et al., 1999], R. Barga et D. Lomet, "Phoenix: Making Applications Robust.", *Proceedings ACM SIGMOD International Conference on Management of Data,* Philadelphie, Pennsylvanie, Etats-Unis*.,* 1-3 juin 1999.

[Bartlett, 1981], J.F. Bartlett, "A NonStop Kernel.", *Proceedings of the eighth ACM symposium on Operating systems principles,* 1981.

[Bouchenak et al., 2005], S. Bouchenak, N. De Palma et D. Hagimont,Autonomic Administration of Clustered J2EE Applications, *IFIP/IEEE International Workshop on Self-Managed Systems & Services (SelfMan'05)*, 2005

[Bruneton et al., 2004], E. Bruneton, T. Coupaye, M. Leclercq, V. Quéma et J.B. Stefani, "An Open Component Model and its Support in Java", *Proceedings of the International Symposium on Component-based Software Engineering,* 2004.

[Candea et al., 2003], G. Candea, E. Kiciman, S. Zhang, P. Keyani et A. Fox, "JAGR: An Autonomous Self-Recovering Application Server.", *5th Annual International Workshop on Active Middleware Services (AMS 2003) 2003 Autonomic Computing Workshop*, Seattle*,* Etats-Unis, 25 juin 2003.

[Cecchet et al., 2004], E. Cecchet, J. Marguerite et W. Zwaenepoel, "C-JDBC: Flexible Database Clustering Middleware", *USENIX Annual Technical Conference, Freenix track*, 2004.

[Chandra et al., 1996], T.D. Chandra, V. Hadzilacos et S. Toueg, "On the Impossibility of Group Membership", *Symposium on Principles of Distributed Computing*, 1996.

[Chandra et al., 1992], T.D. Chandra, V. Hadzilacos et S. Toueg, "The Weakest Failure Detector for Solving Consensus.", *Proceedings of the 11th Annual {ACM} Symposium on Principles of Distributed Computing,* 1992.

[Chandy et al., 1985], K.M. Chandy et L. Lamport, "Distributed Snapshots: Determining Global States of Distributed Systems", *ACM Transactions on Computer Systems*, Volume 3, Pages 63 – 75, février 1985.

[Chen et al., 2002], M.Y. Chen, E. Fratkin, E. Kiciman, A. Fox et E. Brewer, "Pinpoint: Problem Determination in Large, Dynamic Internet Services.", *International Conference on Dependable Systems and Networks (DSN 2002),* Bethesda, Etats-Unis, 23-26 juin 2002.

[Elnozahy et al., 2002], E.N. Elnozahy, L. Alvisi, Y.M. Wang et D.B. Johnson, "A survey of rollback-recovery protocols in message-passing systems.", *ACM Computing Surveys*, Volume 34, Pages 375 – 408, septembre 2002.

[Fischer et al., 1983], M.J. Fischer, N. Lynch. et Paterson, "Impossibility of Distributed Consensus with One Faulty Process.", *Second ACM SIGACT-SIGMOD Symposium on Principles of Database Systems,* Atlanta, Georgie*,* 21-23 mars 1983.

[Gama et al., 2004], G. Gama, K. Nagaraja, R. Bianchini, R.R. Martin, W. Meira et T.D. Nguyen, "State Maintenance and its Impact on the Performability of Multi-tiered Internet Services", *23rd Symposium on Reliable Distributed Systems*, 2004.

[Ganek et al., 2003], A.G. Ganek, T. Corbi., "The dawning of the autonomic computing area", *IBM SYSTEMS JOURNAL*, Volume 42, 2003.

[Guerraoui et al., 1996], R. Guerraoui et A. Schiper, "Fault-Tolerance by Replication in Distributed Systems.", *Ada-Europe International Conference on Reliable Software Technologies,* Montreux, Suisse*,* 1996.

[Kawazoe et al., 1999], M. Kawazoe Aguilera, W. Chen et S. Toueg, "Using the Heartbeat Failure Detector for Quiescent Reliable Communication and Consensus in Partitionable Networks.", *Theoretical Computer Science*, Volume 22à, Pages 3 – 30, 1999.

[Menascé et al., 2000], Menascé et Almeida, "*Capacity Planning for Web Services*", 2000.

[RUBiS, 2005], RUBiS, "Projet ObjectWeb - Rice University Bidding System", *http://rubis.objectweb.org,* 2005.

[Shachor, 2005], G. Shachor, "Tomcat documentation", http://jakarta.apache.org/tomcat/tomcat-3.3-doc/, 2005.

[Sun, 2005], Sun Microsystems, "Java Platform, 2 Entreprise Edition (J2EE)", *http://java.sun.com/j2ee,* 2005.

[Tamir et al., 1984], Y. Tamir et C.H. Séquin, "Error recovery in multicomputers using global checkpoints", *International Conference on Parallel Processing,* 1984.

[Toueg et al., 1984], S. Toueg et O. Babaoglu, "On the Optimum Checkpoint Selection Problem.", *SIAM Journal on Computing*, Volume 13 , Pages 630 – 649, août 1984.

[Wang et al., 1993], Y.M. Wang et W.K. Fuchs, "Lazy Checkpointing Coordination for Bounding Rollback Propagation.", *Symposium on Reliable Distributed Systems,* 1993.

Table des figures