RenPar'17 / SympA'2006 / CFSE'5 / JC'2006
Canet en Roussillon, 4 au 6 octobre 2006

Une approche architecturale pour l'auto-protection de systèmes répartis

Benoit Claudel[1], Noël De Palma[1], Renaud Lachaize[2], Sara Bouchenak[2], Daniel Hagimont[3]

[1] Institut National Polytechnique de Grenoble
[2] Université Joseph Fourier, Grenoble
[3] Institut National Polytechnique de Toulouse

INRIA Rhône-Alpes, 655 avenue de l'Europe, Montbonnot 38334 St Ismier Cedex, France

{Benoit.Claude,Noel.Depalma,Renaud.Lachaize,Sara.Bouchenak}@inrialpes.fr
Daniel.Hagimont@enseeiht.fr

Résumé
La complexité des systèmes répartis d'aujourd'hui est telle que la présence de bogues et de failles de sécurité est statistiquement inévitable. Une approche très prometteuse pour répondre à ce problème est la construction d'un système auto protégé, qui fonctionnerait de façon analogue à un système immunitaire biologique, capable de détecter l'intrusion d'éléments suspects au sein du système et de réagir à une attaque en cours.
Nous avons conçu et implémenté un système autonome appelé JADE qui repose sur une architecture à composants logiciels pour reconfigurer des applications en réaction à des évènements observés. La connaissance de l'architecture de l'application peut être utilisée pour détecter des activités étrangères et pour déclencher des contre-mesures. Nous décrivons comment cette approche peut être appliquée pour ajouter la capacité d'auto protection à une grappe d'application J2EE.

Mots-clés : Informatique Autonome, Auto Protection, Applications J2EE.

---

1. Introduction

Assurer la sécurité d'un système informatique nécessite des capacités clés. Tout d'abord, en mesure préventive, il est important de définir une politique sévère de contrôle d'accès pour que les pirates puissent difficilement s'introduire dans le système et camoufler leurs traces. Ensuite, il faut être capable de distinguer les activités suspectes parmi les opérations « normales » du système. Finalement, une fois détectés, les processus malveillants doivent être arrêtés de manière efficace et radicale[1]. De plus, il est souhaitable de journaliser les activités du système avec un bon niveau de détail (et de protection/redondance pour prévenir une destruction des journaux par les assaillants), ainsi la séquence complète des actions malveillantes et leur magnitude peuvent être déterminées a posteriori pour lancer les procédures de restauration et prendre des nouvelles mesures contre de futures attaques.
Malheureusement, ces objectifs sont très difficiles à atteindre en pratique pour plusieurs raisons :
– Il est notoirement complexe de spécifier et maintenir des politiques de contrôle d'accès qui soient efficaces, globalement cohérentes (au travers des programmes et d'ordinateurs différents) et pas trop restrictives pour les utilisateurs.
– La complexité des logiciels actuels (et de leurs interactions) est telle que la présence de bogues et de failles de sécurité est statistiquement inévitable. Cela laisse l'opportunité à des assaillants de développer de nouvelles techniques de prise de contrôle (« exploits ») à un rythme très élevé. Maintenir

[1] La « procédure d'éradication » doit être rapide et éviter, autant que possible, des solutions drastiques (par exemple une réinstallation complète des logiciels) qui conduiraient à une durée d'indisponibilité non négligeable du système.

le système à niveau avec les mises à jour de sécurité appropriées demande une vigilance de chaque instant.
– Détecter des activités malveillantes à l'intérieur du système est, en général, loin d'être trivial et repose presque exclusivement sur une expertise humaine. Pour cette raison, la plupart des intrusions ne sont remarquées qu'après que des dégâts importants aient été commis.
– Une journalisation détaillée des activités du système dans des circonstances normales conduit souvent à des performances inacceptables et des besoins importants en termes de capacité de stockage. Extraire des indices cruciaux de journaux verbeux n'est pas du tout évident.

Plus généralement les difficultés évoquées ci-dessus sont liées au fait que les administrateurs (humains) ont du mal à assumer la quantité de travail requise pour sécuriser correctement une infrastructure informatique à l'époque de l'Internet. Nous proposons de résoudre ces problèmes grâce à la conception et l'implémentation d'un système d'auto protection. Nos deux objectifs principaux sont : (i) simplifier la configuration (et la reconfiguration) des composants de sécurité en tenant compte de la structure et des opérations du système et (ii) faciliter le développement de contre-mesures automatisées à différents types d'attaques.

Nous avons conçu et implémenté un prototype de système d'administration autonome (appelé JADE) qui a été utilisé avec succès pour donner des capacités d'auto réparation [3] et d'auto optimisation [13] à une architecture J2EE en grappe. Nous étudions actuellement l'utilisation de JADE pour apporter des capacités d'auto protection à la même classe d'applications.

La suite de ce document est structurée de la façon suivante. La section 2 présente la démarche d'auto protection. Nous décrivons le contexte applicatif choisi dans la section 3. La section 4 donne une présentation d'ensemble de JADE, le système d'administration autonome que nous avons implémenté, et de son action sur les grappes d'applications J2EE. La section 5 décrit un scénario qui illustre l'utilisation de JADE pour implémenter l'auto protection sur une application J2EE. Nous donnons une vue d'ensemble des travaux connexes aux nôtres dans la section 6 avant de conclure ce document (section 7).

2. Auto protection

Notre démarche est centrée sur le fait que la connaissance de la structure d'une application répartie peut simplifier la configuration d'outils de sécurité, la détection d'activités illicites ainsi que la mise en œuvre de réponses adaptées. Nous commençons ici par dresser un rapide panorama des mécanismes de sécurité employés aujourd'hui puis nous expliquons le rôle complémentaire de notre proposition.

2.1. Outils classiques de sécurité

Cette section passe brièvement en revue les principaux outils et techniques actuellement utilisés par les experts en sécurité pour combattre les intrusions. Nous faisons la distinction entre différentes fonctionnalités (filtres de protection, détecteurs d'intrusion, outils de journalisation) bien que de nombreuses solutions disponibles intègrent plusieurs d'entre elles.

Les filtres de protection sont utilisés pour restreindre les interactions entre machines (ou, plus généralement, entre des processus/ressources répartis) à un nombre limité et bien défini de motifs. Par exemple, un pare-feu agit comme un filtre réseau qui vérifie si chaque paquet peut être transmis en fonction de son protocole, de ses adresses source/destination et de ses ports. Un autre exemple est celui de la spécification explicite des droits pour l'accès aux données ou l'exécution de programmes, par l'intermédiaire de l'utilisation de capacités ou de listes de contrôle d'accès (ACL[2]).

Les détecteurs (ou *scanners*) examinent les éléments de l'activité système (contenus/attributs de niveau applicatif ou requêtes de niveau réseau, fichiers, etc.) et les comparent à une bibliothèque de motifs typiques d'attaques connues. Si une intrusion est détectée, une alerte est transmise à l'administrateur. De plus, les scanners peuvent aussi réagir indépendamment contre l'intrusion, mais leur action est généralement d'ampleur limitée (bloquer la requête/paquet en violation, mettre en quarantaine la ressource suspecte) et de portée restreinte (pas de coordination entre les différents serveurs). Ainsi une (rapide) intervention humaine est de toute façon généralement requise pour une étude plus approfondie et pour neutraliser le problème, en particulier si l'activité suspecte a été détectée dans le réseau interne (suggérant ainsi une brèche dans la « frontière » principale avec le monde extérieur).

[2] Access Control Lists

Les journaux enregistrent des données détaillées (et éventuellement protégées/redondantes) sur l'activité du système. Ainsi, une fois qu'une tentative d'intrusion a été détectée, il est possible de déterminer la séquence d'évènements qui conduisent à l'intrusion et l'étendue des dégâts potentiels (par exemple vols/pertes de données). Des outils d'analyse de journaux peuvent aider à automatiser des parties de cette procédure mais une expertise humaine est toujours requise pour obtenir une compréhension précise de l'attaque.

En résumé, les filtres de protection imposent une série de mesures préventives de contrôle d'accès. Les scanners essaient de remarquer et de bloquer dynamiquement les activités suspectes. Les outils de journalisation permettent une analyse de la vie récente du système pour déclencher une procédure de nettoyage appropriée et prendre des mesures nouvelles contre de futures attaques du même type.
Ces outils sont inestimables pour les administrateurs système. Cependant, ils ne sont pas suffisamment puissants pour garantir un bon niveau de sécurité. Tout d'abord, la plupart des détecteurs peuvent uniquement protéger le système contre des attaques connues. Par conséquent les pirates possèdent toujours une longueur d'avance grâce au recours à l'exploitation de nouvelles failles de sécurité, ce qui permet d'éviter les filtres et les scanners. De plus, les administrateurs humains sont très sollicités par les alertes produites par les scanners. En particulier, ils sont généralement chargés d'initier de nombreuses actions, à la fois pour coordonner la défense à l'échelle de la grappe (par exemple grâce à la reconfiguration des filtres et des scanners) et l'audit des dégâts (par exemple avec les outils d'analyse). Par ailleurs, les infrastructures et les services déployés par les administrateurs sont toujours plus complexes et hétérogènes. En conséquence, il est très difficile pour les administrateurs de définir des politiques de sécurité globalement cohérentes pour protéger convenablement ces services.
Pour toutes ces raisons, les ressources humaines représentent le goulot d'étranglement principal des infrastructures de sécurité, ce qui tend à accroître la fenêtre de vulnérabilité d'un système exposé à un nouveau type d'attaque. L'objectif de notre travail n'est pas de remplacer les outils existants mais plutôt d'apporter une approche systématique permettant un couplage plus fort entre ces divers éléments. Ainsi, coordonnée à l'échelle de la grappe, la réaction contre une attaque peut être automatisée et donc, plus efficace.

2.2. Démarche d'auto protection

La recherche sur les techniques d'auto protection est une initiative récente. Elle se place dans le contexte de l'« Informatique Autonome » (AC[3]), qui renferme aussi un intérêt pour d'autres aspects telles que (auto) configuration, optimisation et réparation (après une panne). Cette approche est notamment inspirée par l'opération du corps humain et a mené au concept de *système immunitaire informatique* (CIS[4]) au milieu des années 1990.
L'objectif principal des systèmes immunitaires biologiques est de protéger un être vivant d'agents pathogènes. Cette mission dépend d'une capacité essentielle, la « perception du soi » (SoS[5]), c'est-à-dire la capacité de détecter l'intrusion d'éléments étrangers à l'intérieur du « système » (dans ce cas, le corps) par la distinction entre le soi et le non soi. Une fois qu'un intrus est détecté, des mesures peuvent être prises pour le détruire (ou au minimum limiter sa progression et les dégâts qu'il cause). Dans le contexte d'un système informatique, le non soi peut correspondre à l'activité d'un programme malveillant ou d'un utilisateur non autorisé. En se basant sur cette analogie, Forrest *et al.* [7] ont déterminé les principes primordiaux de conception requis pour construire des systèmes immunitaires informatiques, qui sont résumés ci-dessous.
– Autonomie : le système immunitaire ne requiert pas (trop) d'administration externe ou de maintenance. Il classifie et élimine de façon autonome les attaques, c'est à dire il est capable de reconnaître des attaques rencontrées auparavant de même que de nouveaux types d'intrusions.
– Répartition : il n'y a pas de coordination centrale et, par conséquent, pas de point de faute unique dans le système immunitaire. Cela implique qu'aucun composant n'est essentiel et que le comportement incorrect ou la mort d'un certain nombre de composants de sécurité peut être compensé par la réparation ou la création de nouveaux composants.

[3] Autonomic computing
[4] Computer Immune System
[5] Sense of self

– Multicouches : des couches multiples avec des mécanismes différents sont combinées pour procurer des dispositifs robustes et flexibles pour la sécurité.

Inspirés par ces principes, nous proposons des schémas architecturaux pour améliorer la coordination entre les éléments multiples qui composent une infrastructure de sécurité. Nos efforts principaux ne portent pas sur le développement de nouvelles techniques spécifiques pour le contrôle d'accès ou la détection d'intrusion mais plutôt sur les mécanismes qui permettent une intégration efficace et flexible de ces différents outils dans un processus de contrôle automatisé et global.

De nombreuses études ont montré que l'intégrité d'un système informatique est inversement proportionnelle à l'intervalle de temps dont dispose un intrus avant d'être détecté et chassé. Cependant, comme expliqué précédemment, cet intervalle est souvent relativement long car la majeure partie du travail de détection et de contre-attaque incombe à des administrateurs hautement qualifiés et souvent débordés. Les détecteurs d'attaques les plus génériques (c'est-à-dire capables de déceler des techniques d'intrusion inédites) reposent sur des méthodes statistiques et génèrent un nombre important de fausses alertes. En conséquence, il est difficilement envisageable de se baser sur ce type de dispositif pour mettre en œuvre une procédure automatisée de réponse à une attaque. Par opposition, nous cherchons à utiliser la connaissance de l'architecture d'une application répartie afin de distinguer les interactions normales et illicites entre les différentes entités du système. Cette méthode de détection présente l'avantage important de ne pas produire de fausses alarmes. Elle peut donc fournir un socle viable pour la mise en œuvre d'une réponse automatisée, qui peut elle-même être spécialisée en fonction de la structure de l'application considérée.

3. Contexte applicatif

Notre but est de construire et d'évaluer une plate-forme expérimental pour l'auto protection de systèmes distribués. Nous avons choisi les grappes J2EE (*Java 2 Enterprise Edition*) [12] comme premier domaine d'application. Les plate-formes J2EE permettent la construction d'applications web (sites de vente en ligne, portails Internet, etc.). De telles applications sont généralement organisées en 3 ou 4 étages : un étage web, un étage de présentation, un étage de logique métier (optionnel) et un étage de logique de données. Cette organisation (figure 1) peut être résumé de la manière suivante :

1. *La logique web*. Il s'agit d'un serveur HTTP (par exemple un serveur Apache). Sa fonction est de recevoir et traiter les requêtes en provenance des clients. Si la réponse à une requête est une page statique, elle est délivrée directement par cet étage ; si elle nécessite la construction d'un contenu dynamique, la requête est envoyé à l'étage suivant.
2. *La logique de présentation*. La fonction de cet étage (par exemple un serveur Tomcat) est de contrôler l'exécution de *Servlets*, qui dirigent eux-mêmes l'exécution de l'application et synthétise ses résultats sous forme de pages dynamiques.
3. *La logique métier* (optionnel). La fonction de cet étage (par exemple des EJBs *Enterprise Java Beans*) est d'implémenter l'application (accès et traitement des données) si cela n'est pas fait par l'étage précédent.
4. *La logique de données*. La fonction de cet étage (par exemple une base de donnée MySQL) est de stocker les données persistantes.

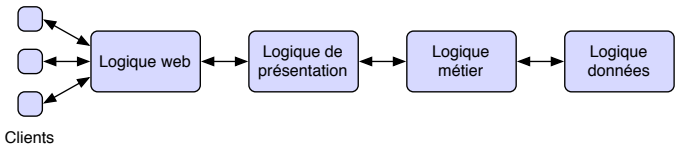

FIG. 1 – Architecture d'une plate-forme J2EE

4. Présentation du système d'auto administration JADE

4.1. Conception de JADE

Nous avons adopté l'organisation générale proposée pour l'informatique autonome [2] (figure 2). Un Gestionnaire Autonome implémente une boucle de contrôle qui régule une partie du système, appelée *Ressource Administrée*. Pour permettre un contrôle hiérarchique, un Gestionnaire Autonome peut lui-même jouer le rôle d'une Ressource Administrée.

De façon à être contrôlable, une Ressource Administrée doit être munie d'une interface d'administration qui procure des points d'entrée (capteurs et actionneurs) pour un Gestionnaire Autonome. Cette interface doit permettre au gestionnaire d'observer et de changer l'état de la ressource.

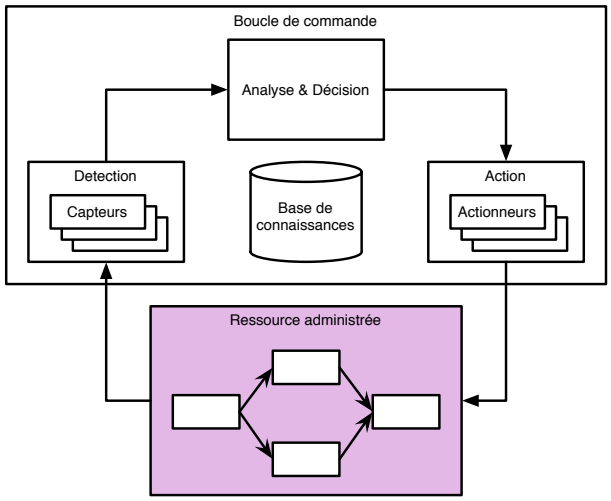

FIG. 2 – Structure d'un système autonome

L'interface d'administration doit permettre les fonctionnalités suivantes.

- Inspecter le contenu d'une ressource administrée, c'est à dire consulter n'importe quel paramètre en lecture ; lire les valeurs de n'importe quelle sonde attachée à la ressource ; si la ressource administrée est composite (faite d'un assemblage de parties), récupérer les informations sur la structure de cet assemblage.
- Déployer et (re)configurer la ressource administrée ; si, à nouveau, la ressource est composite, modifier la structure de l'assemblage, par exemple insérer, refaire les liens, ou enlever dynamiquement certaines de ses parties (par exemple insérer une sonde, ajouter un nœud à une grappe, etc.).

Nous proposons d'utiliser un modèle à composants comme base pour l'implémentation des Ressources Administrées. Le modèle à composants que nous utilisons est Fractal [5], qui possède les avantages suivants :

- Il fournit une interface de contrôle uniforme et adaptable qui permet l'introspection (l'observation des propriétés des composants) et les liaisons dynamiques (reconfiguration d'un assemblage de composants).
- Il définit un modèle de composition hiérarchique pour des composants, permettant à un sous composant d'être partagé entre les composants qui l'englobent, à n'importe quel niveau de granularité.

Nous utilisons Fractal pour encapsuler chaque ressource administrée dans un composant, fournissant ainsi une interface d'administration uniforme pour toutes les ressources. Cela procure un moyen pour :

- Administrer des entités patrimoniales en utilisant un modèle uniforme, au lieu de dépendre de fichiers de configuration gérés à la main et spécifiques aux ressources.
- Administrer des environnements complexes avec des points de vue différents. Par exemple, en utili-

sant des composants d'encapsulation appropriés, il est possible de représenter la topologie du réseau, la configuration de l'intergiciel J2EE ou la configuration d'une application sur l'intergiciel J2EE.
– Ajouter un comportement de contrôle sur les entités patrimoniales encapsulées (par exemple surveillance, interception et reconfiguration).

4.2. Auto administration des applications J2EE

L'approche ci-dessus est illustrée dans le cas d'une architecture J2EE sur grappe. Dans ce cadre, un répartiteur de charge (switch L5) équilibre les requêtes entre deux serveurs web (Apache) répliqués. Les serveurs Apache sont connectés à deux serveurs de servlets (Tomcat) répliqués. Les serveurs Tomcat sont tous les deux connectés à la même base de données (MySQL).

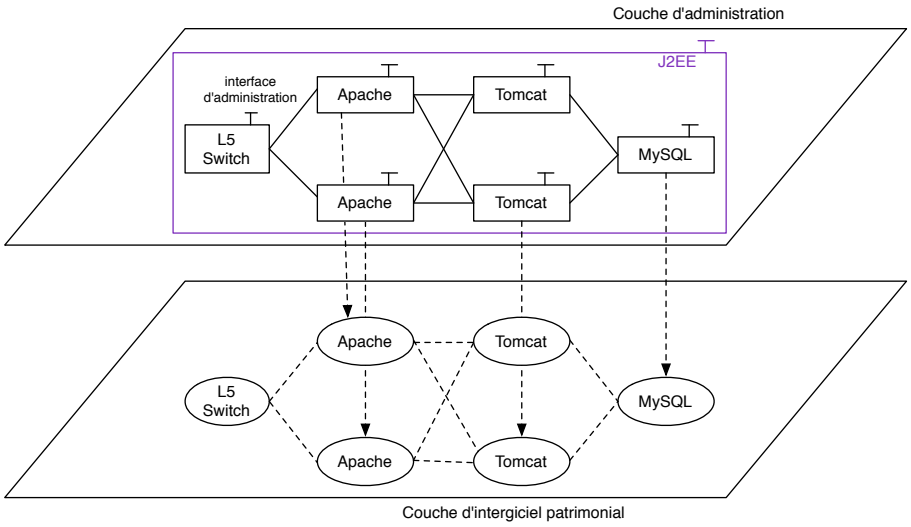

FIG. 3 – Administration à base de composants d'applications patrimoniales avec JADE

Dans la figure 3, les flèches pointillées verticales représentent les relations d'administration entre les composants et les entités patrimoniales encapsulées. Dans la couche patrimoniale, les lignes horizontales pointillées représentent les relations (ou liens) entre les entités patrimoniales. Ces liens sont représentés dans la couche d'administration par des liens entre composants (lignes pleines dans la figure).

Encapsulation des ressources administrées

Dans la couche d'administration, tous les composants fournissent la même interface d'administration (uniforme) pour les ressources encapsulées, et l'implémentation correspondante est spécifique à chaque ressource (par exemple dans le cas de J2EE, Apache, Tomcat, MySQL, etc.). L'interface permet l'administration des attributs de la ressource, de ses liens et de son cycle de vie.

Grâce à cette couche, des programmes patrimoniaux peuvent être administrés sans avoir à gérer les mécanismes de configuration propriétaires qui sont cachés par les composants d'encapsulation. La couche d'administration fournit tous les éléments requis pour implémenter de tels programmes d'administration :

– **Introspection** : L'interface d'introspection permet l'observation des ressources administrées (RA). Par exemple, un programme d'administration peut inspecter une RA Apache (qui encapsule le serveur Apache) pour découvrir que le serveur s'exécute sur *nœud1 :port80* et possède un lien avec un serveur Tomcat qui s'exécute sur *nœud2 :port66*. Il peut aussi inspecter l'infrastructure J2EE dans son ensemble, considérée comme une seule RA, pour découvrir qu'elle est composée de deux serveurs Apache interconnectés avec deux serveurs Tomcat connectés au même serveur MySQL.

– **Reconfiguration** : L'interface de reconfiguration qui permet le contrôle de l'architecture de composants. En particulier, cette interface de contrôle autorise la modification des attributs d'un composant ou des liens entre composants. Ces changements de configuration sont reflétés sur la couche patrimoniale. Par exemple, un programme d'administration peut ajouter ou enlever un serveur Apache répliqué dans l'infrastructure J2EE pour s'adapter à la charge effective.

L'implémentation de la couche d'administration repose sur les composants Fractal qui encapsulent les logiciels patrimoniaux administrés. De manière plus précise, le modèle à composant Fractal permet la gestion de différents aspects :

– **Attributs** : un attribut est une propriété configurable d'un composant. L'interface du composant expose des méthodes *getter* et *setter* pour les attributs. Une modification de l'attribut du composant est reflétée sur la configuration du logiciel patrimonial. Par exemple, la valeur de l'attribut port du composant apache est reflétée sur le fichier de configuration (*http.conf*) du serveur Apache patrimonial.
– **Liens** : l'interface d'un composant expose des méthodes permettant de contrôler les liens entre composants. La configuration des liens dans la couche d'administration est répercutée sur la couche de logiciel patrimonial. Par exemple, en configurant un lien entre un composant apache et un composant tomcat, un serveur Apache *http* peut être connecté à un serveur Tomcat à qui il délègue les requêtes dynamiques. L'implémentation de ce lien configure le fichier *worker.properties* dans le logiciel patrimonial.
– **Cycle de vie** : l'interface d'un composant expose des méthodes pour contrôler l'exécution du composant. Les opérations élémentaires concernant le cycle de vie d'un système patrimonial peuvent être effectuées grâce à cette interface de cycle de vie (par exemple démarrer ou stopper l'exécution d'un composant). Dans le cas d'un composant Apache, elles sont implémentées en appelant les commandes du serveur Apache permettant de démarrer ou d'arrêter le serveur.

Auto optimisation et auto réparation

L'auto optimisation est un objectif important de l'administration autonome que nous traitons dans JADE. JADE vise à augmenter/diminuer automatiquement le nombre de ressources répliquées utilisées par l'application quand la charge augmente/diminue. Ceci permet de maximiser efficacement l'utilisation des ressources (c'est à dire pas de surréservation de ressource).

Dans ce but, un gestionnaire de Qualité de Service (QoS[6]) utilise des détecteurs pour mesurer la charge du système. Ces détecteurs sondent l'utilisation des CPU ou le temps de réponse des requêtes de niveau applicatif. Le gestionnaire QoS utilise aussi des actionneurs pour reconfigurer le système. Grâce à la conception non spécifique de JADE, les actionneurs utilisés par le gestionnaire QoS sont génériques. En effet, augmenter/diminuer le nombre de ressources d'une application est implémenté comme un ajout/suppression de composants dans la structure de l'application.

En plus des détecteurs et des actionneurs, le gestionnaire QoS se sert également d'un composant d'analyse/décision qui est responsable de l'implémentation de l'algorithme d'auto optimisation. Ce composant reçoit les notifications des détecteurs et, si la reconfiguration est nécessaire, il augmente le nombre de ressources en allouant de nouveaux nœuds disponibles. Il déploie alors ces ressources logicielles sur les nouveaux nœuds et les ajoute dans la structure existante de l'application simplement en créant les composants associés sur ces nœuds. De façon symétrique, si les ressources allouées à une application sont en sous-utilisation, le gestionnaire QoS effectue une reconfiguration pour enlever certains éléments répliqués et rendre disponibles leurs ressources.

L'auto réparation est un autre comportement de l'administration autonome que nous traitons dans JADE. Dans un système basé sur la réplication, quand une ressource répliquée tombe en panne, le service reste disponible grâce à la réplication. Cependant, nous visons à réparer automatiquement le système administré en remplaçant la ressource répliquée en panne par une nouvelle. Notre objectif actuel est de traiter les pannes franches (ou arrêts sur défaillance). La politique de réparation implémentée rétablit le système administré défaillant dans l'état où il se trouvait avant la survenue de la panne. Dans ce but, le gestionnaire de panne utilise des détecteurs qui surveillent la santé des ressources utilisées grâce à des sondes installées sur les nœuds hébergeant le système administré. Ces sondes sont implémentées en utilisant des techniques de notification périodique (*heartbeat*). Le gestionnaire de panne utilise aussi

[6] Quality of Service

un composant spécifique appelé la Représentation du Système (RS). Le composant RS conserve une représentation de la structure architecturale actuelle du système administré. Il est utilisé pour récupérer de pannes. Certains pourraient objecter que le modèle à composants sous-jacent peut être utilisé pour introspecter dynamiquement l'architecture du système administré et ainsi utiliser l'information sur la structure pour récupérer de pannes. Mais si un nœud hébergeant une ressource répliquée meurt, le composant encapsulant cette ressource répliquée est perdu. C'est pourquoi un mécanisme de représentation du système est nécessaire. Cette représentation reflète, à l'instant considéré, la structure architecturale du système (qui peut évoluer). Elle est fiable dans le sens où elle est elle-même répliquée pour permettre une tolérance aux pannes. La représentation du système est implémentée comme un instantané de l'intégralité de l'architecture à composants.
En plus de la représentation du système, des détecteurs et des actionneurs, le gestionnaire de pannes utilise un composant d'analyse/décision qui implémente le comportement de réparation autonome. Il reçoit des notifications des détecteurs *heartbeat* et, lorsqu'un nœud tombe en panne, se sert de la représentation du système pour récupérer les informations nécessaires concernant le nœud en panne (c'est à dire les ressources logicielles qui s'exécutaient sur ce nœud avant la panne et leurs liens avec d'autres ressources). Il alloue alors un nouveau nœud disponible et redéploie ces ressources logicielles sur le nouveau nœud. La représentation du système est alors mise à jour pour correspondre à la nouvelle configuration.
Une description et une évaluation plus détaillées de l'auto optimisation et de l'auto réparation dans JADE sont disponibles dans d'autres documents [3, 13].

5. Auto protection avec JADE pour les applications J2EE

5.1. Perception du soi basée sur l'architecture

Comme il est mentionné dans la section 2.2, la « perception du soi » (SoS) est la capacité de détecter l'intrusion d'éléments étrangers à l'intérieur du système administré grâce à la distinction entre *soi* et *non soi*. Nous avons vu dans la présentation du système JADE que notre objectif est de fournir une représentation architecturale du système administré basée sur des composants de façon à permettre des observations et des reconfigurations. L'architecture de l'application apporte une notion de perception de soi car elle définit les composants qui sont supposés s'exécuter sur les machines ainsi que les canaux de communication qui peuvent être utilisés par ces composants. Toute exécution qui ne trouve pas sa place parmi ces composants ou ces canaux de communication est considérée comme étrangère au système. La conception de JADE suit aussi les principes de conception des systèmes immunitaires (présentés en section 2.2) :

- **Autonomie** : la politique d'auto protection qui peut être définie avec JADE n'a pas besoin d'une grande connaissance sur les attaques qu'elle peut affronter. Elle peut détecter des comportements anormaux, c'est-à-dire ceux qui sont étrangers au système (ceux qui ne correspondent pas à l'architecture de l'application). Dans notre scénario, nous détectons les intrusions comme des comportements n'appartenant pas au système. Par exemple, si une application essaie d'utiliser un canal de communication (lien) non déclaré, une alerte sera émise et déclenchera une contre-mesure.
- **Répartition** : il n'y a pas de point central de défaillance pour la gestion de la sécurité. Chaque nœud peut détecter une exécution anormale, sur la machine locale ou comme une requête provenant d'un autre nœud. Dans notre scénario, chaque nœud est responsable de la détection de communications provenant du non soi.
- **Multicouches** : JADE autorise la combinaison de nombreuses techniques de sécurisation. De nombreux mécanismes de détection et de contre-mesure peuvent être ajoutés en encapsulant, déployant et configurant les outils existants. Dans notre scénario, nous encapsulons un pare-feu pour détecter les communications suspectes.

5.2. Scénario

Nous décrivons dans cette section un scénario simple qui vise à illustrer l'implémentation d'un comportement d'auto protection se rajoutant au système JADE. Ce scénario est actuellement en cours de développement.
Nous considérons un défaut de sécurité qui permet à un assaillant d'exécuter du code arbitraire sur

un étage d'une architecture J2EE déployée. Un exemple de ce type de défaut est le *Chunked Encoding Overflow* du serveur Apache défini de la manière suivante [1] :
« Le serveur web Apache contient un défaut qui permet à un assaillant distant d'exécuter du code arbitraire. Le problème est causé par le mécanisme calculant la taille du bloc d'encodage qui n'interprète pas correctement la taille dans le cache de la donnée transférée. En émettant bloc de données spécialement fabriqué, un assaillant peut éventuellement exécuter du code arbitraire ou arrêter le serveur. »
Ainsi, en exploitant ce défaut de sécurité (ou un autre de même type), un assaillant peut gagner le contrôle d'une machine qui exécute ce serveur Apache et subséquemment d'attaquer d'autres machines. Ce défaut peut être exploité même si le serveur Apache se trouve derrière un pare-feu. Nous assumons que des assaillants trouveront toujours un moyen d'éviter les barrières de protection statiques.
Pour pouvoir détecter une exécution illégale (non soi), nous déployons un pare-feu sur chaque machine impliquée dans l'architecture J2EE, comme illustré dans la Figure 4.

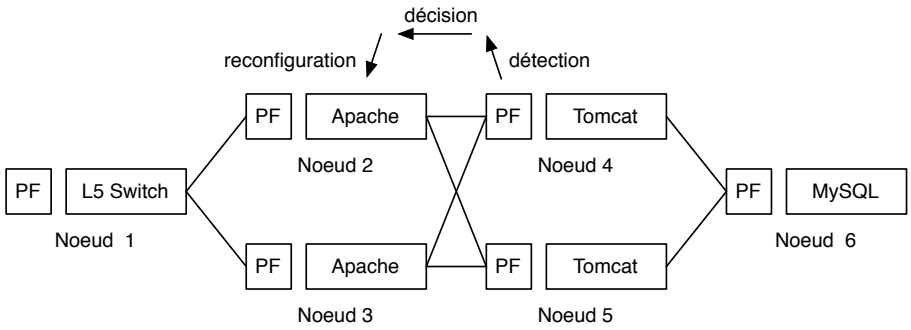

FIG. 4 – Scénario d'auto protection

Le logiciel de pare-feu est encapsulé dans un composant, de façon à ce qu'il puisse être déployé et configuré par JADE (comme les autres ressources logicielles). Chaque pare-feu est configuré automatiquement pour qu'il accepte uniquement les requêtes émises par des machines qui ont un lien déclaré pointant sur lui. Par exemple, le pare-feu sur le *Nœud4* n'acceptera que les requêtes provenant de *Nœud2* et *Nœud3*. Les pare-feu sont configurés automatiquement en concordance avec l'architecture J2EE déployée.
De plus, chaque pare-feu est configuré pour fonctionner comme un détecteur d'intrusion. Quand l'architecture J2EE est déployée, les ports de communication utilisés pour connecter les différents étages de l'achitecture J2EE sont choisis aléatoirement, ce qui rend l'exploitation des liens légaux plus difficile pour un assaillant. Si une tentative d'utiliser/sonder un port non lié est détectée par le pare-feu, ce dernier émet un événement d'alerte.
Ainsi la configuration des pare-feu autorise uniquement les requêtes qui suivent les liens légaux et les requêtes illégales déclenche une alerte. Le déploiement et la configuration des pare-feu permet donc de détecter (en partie) les comportements illégaux.
Chaque fois qu'une alerte est émise, différentes contre-mesures peuvent être exécutées :
– La machine (dans la grappe) qui a émis la requête peut être isolée de l'architecture J2EE et remplacée par une autre (comme dans l'algorithme de réparation). La reconfiguration de l'architecture met aussi à jour l'ensemble des pare-feu de façon cohérente.
– Un message peut être envoyé à un administrateur humain qui pourra prendre des mesures complémentaires.
– Une analyse des journaux sur les différents étages de l'infrastructure peut éventuellement permettre de déterminer la machine distante qui a émis la requête (sur le serveur web global J2EE) et ainsi entraîner une reconfiguration du pare-feu placé devant l'architecture J2EE (sur le Nœud1) pour interdire l'accès depuis cette machine distante.

6. Travaux connexes

Définir un système immunitaire informatique est une tendance majeure pour les systèmes auto protégés. Cette approche a été décrite par Forrest [7] et Kephart [11]. L'idée de base derrière le système immunitaire est de distinguer le comportement légal (le soi) du comportement illégal (le non soi) (typiquement les virus, vers, injections SQL, etc.).

Forrest décrit une forme de perception du soi dans le cas de processus unix en identifiant des séquences d'appels systèmes qui procurent une signature compacte du soi, distinguant ainsi le soi du comportement illégal (non soi). Ce système nécessite l'établissement d'une base de données décrivant le comportement normal de chaque programme suivi.

Vigilante [6] est un système anti-virus dans lequel les détecteurs sont basés sur une analogie avec le système immunitaire et sont capables de détecter des virus inconnus. De plus, quand un nouveau virus est trouvé, sa signature est propagée à travers le réseau à tous les ordinateurs protégés. Notre travail est de la même veine, mais nous proposons d'exploiter la connaissance de l'architecture de l'application pour détecter des activités illégales.

Les systèmes auto nettoyants (*self-cleansing systems*) [10] sont une autre solution pour construire des logiciels auto protégés. Cette approche pessimiste fait la supposition que toutes les intrusions ne peuvent pas être détectées et bloquées. Le système est considéré comme étant compromis après un certain délai. La boucle de contrôle utilisée par ce système réinstalle périodiquement une partie du système depuis une zone de stockage sécurisée. Ce genre de technique est bien évidemment limité au cas de services sans état.

Une propriété importante pour les systèmes auto protégés est la faculté de dissimuler les connaissances importantes comme la structure du système, les versions des logiciels, les fichiers de données des utilisateurs, etc. SDS [8] est un système qui sécurise les données en les cryptant et en les dispersant sur de multiples ordinateurs. Un fichier est découpé en plusieurs morceaux de données cryptées. Ainsi, si un ordinateur est compromis, le pirate ne peut obtenir que la partie de donnée incomplète se trouvant sur cet ordinateur. Dans notre scénario, les ports de connexion qui sont utilisés pour implémenter les liens entre composants sont choisis aléatoirement, rendant ainsi l'exploitation des liens légaux plus difficile pour un assaillant. Avec notre approche, cacher (autant que possible) l'architecture de l'application est aussi un point crucial.

Quand un système est compromis, une autre fonctionnalité importante est la capacité de restaurer le système dans un état sécurisé. Des mécanismes de sauvegardes périodiques et globalement cohérentes peuvent bien sûr être utiles à cet égard. Cependant cette solution annule aussi les modifications légales effectuées sur les données par les utilisateurs. Le système Taser [9] fourni au système de fichiers une capacité d'auto restauration sélective. Taser journalise tous les accès au système de fichiers pour chaque processus. Si un processus est compromis, Taser détermine les accès illégaux pour chaque fichier et peut annuler les modifications illégales. Cependant si une dépendance est trouvée entre un accès légal et un accès illégal (e.g. une opération de lecture légale après une opération d'écriture compromise), Taser requiert une intervention humaine. Une approche similaire pourrait être suivie pour restaurer la base de données d'une application J2EE à chaque fois qu'une activité illégale sur celle-ci est détectée *a posteriori*.

7. Conclusion et perspectives

Actuellement les environnements répartis en informatique sont de plus en plus complexes et difficiles à administrer. Cette complexité est telle que la présence de bogues et de failles de sécurité est statistiquement inévitable. Ainsi les politiques de contrôle d'accès deviennent très difficiles à spécifier et à faire respecter.

Une approche très prometteuse pour traiter ce problème est de concevoir, en suivant la vision de l'informatique autonome, un système auto protégé capable de distinguer les comportements légaux (soi) des comportements illégaux (non soi). La détection d'un comportement illégal déclenche une contremesure pour limiter l'exploitation de l'intrusion et prévenir toute occurrence future de celle-ci. Dans cette veine, nous avons conçu et implémenté un système appelé JADE qui permet la définition de programmes d'administration autonome. JADE repose sur un modèle à composants pour encapsuler les ressources administrées et fournir une infrastructure pour la définition de gestionnaires autonomes qui

captent les évènements significatifs et déclenchent les réponses appropriées. JADE a été utilisé avec succès pour implémenter les propriétés autonomiques d'auto optimisation et d'auto réparation sur des applications J2EE en grappe.
Dans ce document, nous étudions l'application des caractéristiques de JADE pour implémenter l'auto protection sur des applications J2EE. Nous avons montré comment la connaissance de l'architecture (à base de composants) de l'application administrée peut être exploitée pour implémenter une perception du soi. Dans notre scénario, des pare-feu sont déployés automatiquement sur chaque nœud et configurés en fonction de l'architecture J2EE déployée. Ces pare-feux permettent la détection (d'une partie) des comportements illégaux (non soi) et déclenchent des contre-mesures adéquates.
Ce travail est encore au stade préliminaire, mais il ouvre de nombreuses perspectives. Nous sommes actuellement en train d'achever l'implantation d'un premier prototype du système d'auto-protection. Au-delà de la validation fonctionnelle de notre approche, ce prototype permettra d'évaluer sa viabilité en termes de performances. Il est notamment nécessaire de déterminer le coût associé au déploiement d'un pare-feu sur chaque noeud d'une grappe.
Par ailleurs, nous cherchons à étendre les mécanismes de détection des comportements illégaux intégrés à notre infrastructure. En complément de l'approche architecturale (qui permet de mettre en évidence des attaques qui transgressent les canaux de communication autorisés pour une application), il est également nécessaire d'encapsuler des outils patrimoniaux au sein de notre système pour lutter contre les attaques qui ne peuvent pas être décelées via des critères d'interaction. Par exemple, nous prévoyons d'étudier l'intégration de détecteurs d'attaques par injections SQL [4] basés sur l'analyse des journaux générés par les composants du logiciel administré.
A plus long terme, deux problèmes significatifs doivent être considérés. D'une part, il est important de pouvoir garantir la sécurité de l'infrastructure de protection elle-même afin d'empêcher son éventuel détournement par des intrus. D'autre part, la généralisation de notre approche à des systèmes répartis plus complexes tels que les grilles (très grande échelle, multiples domaines d'administration) nécessitera vraisemblablement le développement d'une infrastructure de protection complètement décentralisée.

Bibliographie

1. Apache chunked encoding overflow. http ://www.osvdb.org/838.
2. An architectural blueprint for autonomic computing. *IBM and Autonomic Computing*, April 2003. http ://www-306.ibm.com/autonomic/pdfs/ACwpFinal.pdf.
3. S. Bouchenak, F. Boyer, D. Hagimont, S. Krakowiak, A. Mos, N. De Palma, V. Quéma, and J-B. Stefani. Architecture-Based Autonomous Repair Management : An Application to J2EE Clusters. In *24th IEEE Symposium on Reliable Distributed Systems (SRDS)*, October 2005.
4. S. Boyd and A. Keromytis. Sqlrand : Preventing sql injection attacks. *Proceedings of the 2nd Applied Cryptography and Network Security (ACNS) Conference*, 3089 :292–304, 2004.
5. E. Bruneton, T. Coupaye, and J.B. Stefani. Recursive and dynamic software composition with sharing. In *Proceedings of the 7th ECOOP International Workshop on Component-Oriented Programming (WCOP'02)*, June 2002.
6. M. Costa, J Crowsoft, M. Castro, A. Rowstron, L. Zhou, L. Zhang, and P. Barham. Vigilante : end-to-end containment of Internet worms. In *SOSP '05 : Proceedings of the Twentieth ACM Symposium on Operating Systems Principles*, pages 133–147, New York, NY, USA, 2005. ACM Press.
7. S. Forrest, S. A. Hofmeyr, and A. Somayaji. Computer immunology. *Communication of the ACM*, 40(10) :88–96, 1997.
8. Juan A. Garay, Rosario Gennaro, Charanjit Jutla, and Tal Rabin. Secure distributed storage and retrieval. *Theoretical Computer Science*, 243(1-2) :363–389, 2000.
9. A. Goel, K. Po, K. Farhadi, Z. Li, and E. de Lara. The Taser intrusion recovery system. In *SOSP '05 : Proceedings of the Twentieth ACM Symposium on Operating Systems Principles*, pages 163–176, New York, NY, USA, 2005. ACM Press.
10. Y. Huang and Sood A. Self-cleansing systems for intrusion containment. In *Workshop on Self-Healing, Adaptive and self-MANaged Systems (SHAMAN)*, 2002.
11. Jeffrey O. Kephart. A biologically inspired immune system for computers. In *Artificial Life IV : Proceedings of the Fourth International Workshop on the Synthesis and Simulation of Living Systems*, pages

130–139, Cambridge, MA, USA, 1994. MIT Press.
12. Sun Microsystems. Java 2 platform enterprise edition (J2EE). http ://java.sun.com/j2ee/.
13. C. Taton, S. Bouchenak, F. Boyer, N. De Palma, D. Hagimont, and A. Mos. Self-manageable replicated servers. In *VLDB Workshop on Design, Implementation, and Deployment of Database Replication*, Trondheim, Norway, August 2005.