JAMUS : une plate-forme d'accueil sécurisée pour le code mobile

Nicolas Le Sommer, Frédéric Guidec

JAMUS

Une plate-forme d'accueil sécurisée pour le code mobile

Nicolas Le Sommer — Frédéric Guidec

Laboratoire VALORIA, Université de Bretagne Sud
Nicolas.LeSommer@univ-ubs.fr – Frederic.Guidec@univ-ubs.fr

RÉSUMÉ. *La plate-forme d'accueil JAMUS (*Java Accommodation of Mobile Untrusted Software*) est dédiée à l'hébergement de programmes d'application capables d'exprimer leurs besoins propres vis à vis des ressources offertes par le système qui les accueille. Elle repose sur le principe de la contractualisation de l'accès aux ressources afin d'offrir une certaine qualité de service aux programmes hébergés, tout en leur assurant une relative sécurité au cours de leur exécution.*

ABSTRACT. *JAMUS (*Java Accommodation of Mobile Untrusted Software*) is a Java platform dedicated to the accommodation of untrusted programs that can specify their requirements regarding the resources available on the system. It relies on the principle of resource contracting in order to offer a certain level of quality of service to hosted programs, while ensuring a safe execution of these programs at runtime.*

MOTS-CLÉS : *code mobile, sécurité, qualité de service*

KEYWORDS: *mobile code, security, quality of service*

L'objet – 8/2002. LMO'2002, pages 203 à 215

1. Introduction

La plate-forme d'accueil JAMUS décrite dans cet article est développée dans le cadre du projet RASC[1](*Resource-Aware Software Components*), dont l'objectif est de favoriser l'essor d'un véritable marché du composant logiciel « sur étagères » déployé grâce à un mécanisme de code mobile. Dans ce projet nous nous focalisons sur la prise en compte de certaines caractéristiques non-fonctionnelles de tels composants mobiles, à savoir celles portant sur les ressources qui leurs sont nécessaires en cours d'exécution.

La plate-forme JAMUS (*Java Accommodation of Mobile Untrusted Software*) est dédiée à l'hébergement de programmes d'application capables d'exprimer leurs besoins propres vis à vis des ressources offertes par le système qui les accueille. Chaque programme candidat à l'hébergement par la plate-forme doit décrire ses besoins sous forme qualitative (e.g., droits d'accès à tout ou partie du système de fichiers) et quantitative (e.g., quotas d'accès requis en lecture et en écriture). La plate-forme est capable d'analyser ces besoins en les confrontant à un ensemble de contraintes posées sur l'utilisation des ressources dont elle dispose. Elle met en œuvre un mécanisme de contrôle d'admission reposant sur le principe de la réservation de ressources afin de décider si elle peut ou non accepter d'héberger un programme. Dans l'affirmative, le programme accueilli est soumis à une supervision constante au cours de son exécution, la plate-forme empêchant toute utilisation de ressources non conforme aux besoins exprimés initialement par ce programme.

Dans la section suivante, la plate-forme JAMUS est présentée dans les grandes lignes. Son architecture est ensuite détaillée dans les sections 2 à 5. Dans la section 3 est introduit le concept de « profil d'utilisation de ressources », qui permet à la fois de spécifier les conditions de fonctionnement de la plate-forme JAMUS, et d'exprimer les besoins d'un programme d'application. Le mécanisme de contrôle d'admission grâce auquel la plate-forme décide d'accepter ou de refuser un programme est présenté dans la section 4. Les mécanismes permettant la supervision d'un programme au cours de son exécution sont abordés dans la section 5. Quelques travaux dont la démarche ou la finalité se rapprochent de la nôtre sont évoqués dans la section 6. En guise de conclusion, la section 7 dresse l'état d'avancement du développement de la plate-forme JAMUS, et énumère quelques unes des perspectives offertes par ce travail.

2. Vue d'ensemble de la plate-forme JAMUS

La plate-forme JAMUS s'appuie sur les services offerts par l'environnement d'exécution RAJE (*Resource-Aware Java Environment*), services qui autorisent une gestion précise des ressources utilisées par les programmes Java. RAJE est également conçu dans le cadre du projet RASC. Il est basé sur la JVM (*Java Virtual Machine*) Kaffe-1.0.6 distribuée par *TransVirtual Technology*, et intègre notamment des classes réifiant

[1] `.http ://www.univ-ubs.fr/valoria/Orcade/RASC`

certaines ressources système (CPU, interfaces réseau, etc.). En outre les classes de l'API standard Java qui modélisent des ressources conceptuelles (*Socket*, *File*, etc.) ont été modifiées de manière à autoriser un contrôle précis de leur utilisation. Enfin l'environnement RAJE intègre divers mécanismes facilitant la supervision des ressources utilisées par des programmes Java, cette supervision pouvant être assurée de manière synchrone ou asynchrone.

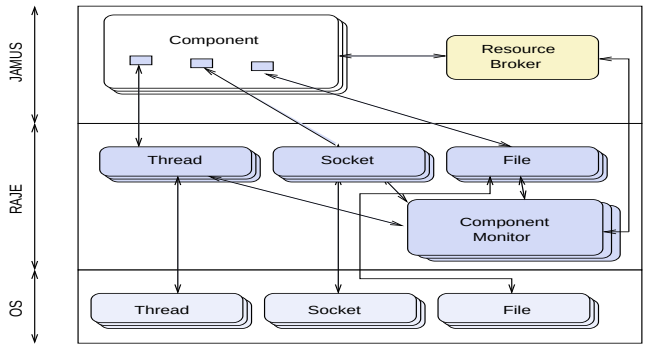

Figure 1. *Architecture de la plate-forme JAMUS*

La plate-forme JAMUS (dont l'architecture générale est présentée dans la figure 1) permet une forme élémentaire de contractualisation de l'accès aux ressources nécessaires aux programmes qu'elle héberge. Tout programme d'application candidat à l'hébergement par la plate-forme doit être capable d'exprimer ses besoins en décrivant les ressources qui vont lui être utiles au cours de son exécution, et les conditions dans lesquelles il doit pouvoir accéder à ces ressources. En exprimant ses besoins, un programme demande à bénéficier d'un certain service de la part de la plate-forme, et s'engage dans le même temps à ne pas utiliser d'autres ressources que celles dont il fait explicitement la demande. De son côté, la plate-forme acceptant d'accueillir un programme s'engage à lui fournir les ressources demandées, et se réserve en même temps le droit de sanctionner le programme hébergé si le comportement de celui-ci n'est pas conforme aux engagements pris lors du contrôle d'admission.

L'originalité de la plate-forme JAMUS réside probablement dans la réciprocité des engagements pris entre la plate-forme d'accueil et les programmes qu'elle héberge. Les besoins exprimés par un programme sont à la fois interprétés comme tels, et comme un engagement à ne pas chercher à accéder à d'autres ressources que celles dont il est fait explicitement mention. Grâce à cet engagement, JAMUS peut offrir une certaine qualité de service aux programmes hébergés, tout en leur assurant une relative sécurité au cours de leur exécution (sécurité résultant essentiellement de la prévention du déni de service). Ces caractéristiques résultent de l'application des deux principes suivants au sein de la plate-forme JAMUS.

Principe du contrôle d'admission. Tout programme candidat à l'hébergement doit passer avec succès une épreuve de contrôle d'admission. Ce contrôle est assuré par un courtier de ressources, capable d'interroger le programme afin de connaître ses besoins propres, et d'utiliser cette information pour décider du sort de ce programme. Les besoins exprimés par le programme le sont sous la forme de profils d'utilisation de ressources, dont la structure est détaillée dans la section 3. Le fonctionnement du courtier de ressources est quant à lui décrit dans la section 4.

Principe de la supervision des programmes hébergés. Le fait qu'un programme ait passé avec succès l'épreuve du contrôle d'admission ne signifie pas nécessairement qu'en cours d'exécution il se contentera d'utiliser les ressources qui lui ont été accordées à l'issue de ce contrôle. En effet un programme conçu de façon maladroite ou dans une optique malveillante pourrait tenter d'accéder à des ressources qui ne lui sont *a priori* pas destinées, ou bien encore chercher à consommer plus de ressources qu'il ne lui en a été accordé. Un programme accueilli par la plate-forme JAMUS est donc considéré comme étant *a priori* non digne de confiance. En conséquence son exécution doit être supervisée afin de vérifier que l'utilisation qu'il fait des ressources demeure conforme aux modalités convenues lors du contrôle d'admission. Les mécanismes assurant la supervision d'un programme en cours d'exécution sont présentés dans la section 5.

3. Profils d'utilisation de ressources

Lors de son lancement la plate-forme d'accueil JAMUS reçoit en paramètres une description des ressources qu'elle va pouvoir mettre à la disposition des programmes d'applications hébergés. Cette description est une énumération des –ou d'une partie des– ressources disponibles dans le système, ainsi que les modalités d'utilisation de ces ressources (permissions et quotas).

Les informations décrivant les ressources et leurs modalités d'utilisation sont fournies sous la forme de profils d'utilisation de ressources (instances de la classe *ResourceUsageProfile* définie dans l'API de l'environnement RAJE). Cette classe permet de décrire à la fois les contraintes posées sur les ressources offertes par la plate-forme JAMUS, et les exigences des programmes d'application vis à vis de ces mêmes ressources. Un profil d'utilisation de ressources est décrit sous la forme d'un objet possédant trois attributs *pattern*, *permission*, et *quota*, référençant respectivement des objets implémentant les interfaces *ResourcePattern*, *ResourcePermission*, et *ResourceQuota* (voir figure 2). Il existe des implémentations spécialisées de ces interfaces pour chaque type de ressource considéré à l'heure actuelle dans l'environnement RAJE. En outre de nouvelles classes peuvent aisément être développées afin de décrire de nouvelles modalités d'utilisation pour des ressources déjà prises en compte, ou bien afin d'intégrer de nouveaux types de ressources dans le système.

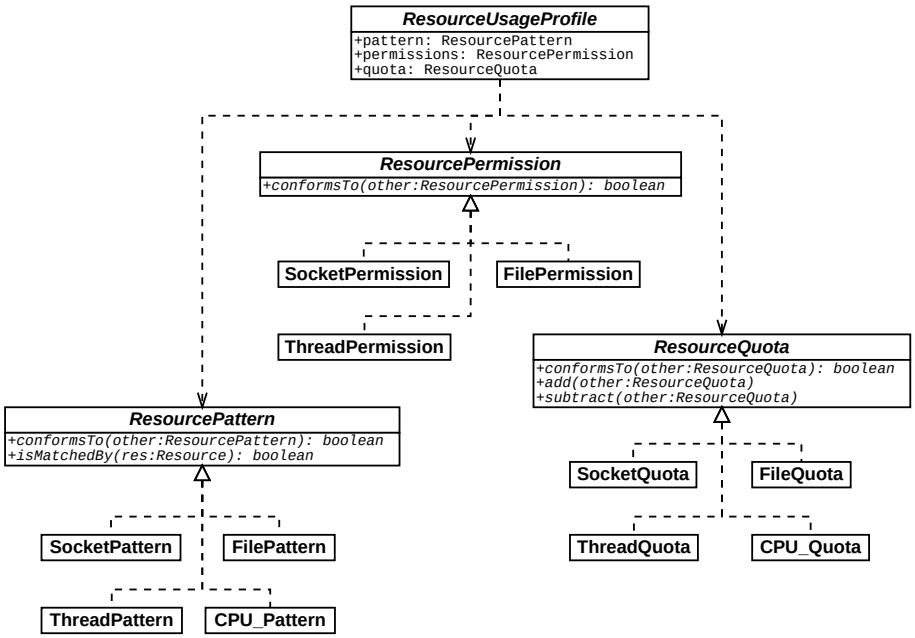

Figure 2. *Modélisation des profils d'utilisation de ressources sous forme d'objets.*

L'attribut *pattern* d'un profil d'utilisation sert à identifier précisément sur quelle catégorie de ressources portent les besoins ou contraintes exprimées dans ce profil. Il permettra de sélectionner en cours d'exécution parmi les divers objets modélisant des ressources dans le système ceux qui seront concernés par ce profil d'utilisation. L'interface *ResourcePattern* définit une fonction *isMatchedBy()*, qui prend en paramètre un objet ressource et retourne une valeur booléenne indiquant si cet objet satisfait ou non le critère de sélection considéré. Dans le cas le plus simple une classe implémentant *ResourcePattern* peut appuyer la sélection sur le type des objets modélisant les ressources considérées. Toutefois on peut également développer des mécanismes de sélection plus élaborés afin de sélectionner par exemple les ressources en fonction des valeurs portées par certains attributs dans les objets qui les modélisent.

L'attribut *permission* d'un profil d'utilisation précise quelles sont les opérations permises (par le système) ou requises (par un composant) sur le type de ressource considéré. Il définit donc les modalités d'utilisation des ressources sous une forme qualitative. L'attribut *quota* complète cette information en précisant pour sa part les modalités d'utilisation des ressources sous une forme quantitative.

Les trois interfaces *ResourcePattern*, *ResourcePermission*, et *ResourceQuota* définissent chacune une fonction booléenne *conformsTo()* permettant de procéder à un test de conformité entre deux objets implémentant la même interface. Ces fonctions permettent de vérifier par exemple si le critère de sélection défini dans un objet de

type *ResourcePattern* englobe le critère de sélection défini dans un autre objet de type *ResourcePattern* (auquel cas toutes les ressources satisfaisant le premier critère satisferont également le second critère). On pourra de même vérifier si les droits d'accès demandés par un programme pour un certain type de ressources sont conformes aux droits d'accès accordés par la plate-forme JAMUS pour ce même type de ressources, ou si les quotas demandés par un programme sont compatibles avec les quotas autorisés par la plate-forme.

```
public class MyProgram {
 public static Set getResourceRequirements(String[] args) {
  Set requirements = new HashSet() ;
  int Ko = 1024 ; int Mo = 1024*1024 ;

  // Profil d'utilisation global s'appliquant à toutes les communications par sockets :
  // 12 sockets ouverts max., quota de 2 Mo en émission, 8 Mo en réception.
  requirements.add(new ResourceUsageProfile(new SocketPattern(),
                                    new SocketPermission(SocketPermission.all),
                                    new SocketQuota(12, 2*Mo, 8*Mo))) ;

  // Profil d'utilisation sélectif ne concernant que les connexions établies
  // vers des hôtes du domaine 195.83.160/24 :
  // 4 sockets ouverts max., quota de 512 Koctets en émission, 128 Koctets en réception.
  requirements.add(new ResourceUsageProfile(new SocketPattern("195.83.160/24"),
                                    new SocketPermission(SocketPermission.all),
                                    new SocketQuota(4, 512*Ko, 128*Ko))) ;

  // Profil d'utilisation global s'appliquant à tous les accès au système de fichiers :
  // 8 fichiers ouverts max., quota de 1 Mo en écriture, 2 Mo en lecture.
  requirements.add(new ResourceUsageProfile(new FilePattern("/tmp"),
                                    new FilePermission(FilePermission.all),
                                    new FileQuota(8, 1*Mo, 2*Mo))) ;

  // Profil d'utilisation sélectif ne concernant que le sous répertoire /tmp/myStuff :
  // 4 fichiers ouverts max., quota de 5 Ko en lecture, écriture interdite.
  requirements.add(new ResourceUsageProfile(new FilePattern("/tmp/myStuff"),
                                    new FilePermission(FilePermission.readOnly),
                                    new FileQuota(4, 0, 5*Ko))) ;

  return requirements ;
 }

 public static void main(String[] args) { . . . } :
}
```

Figure 3. *Exemple de programme d'application capable d'exprimer ses besoins propres à travers la méthode* getResourceRequirements(), *qui retourne un ensemble d'objets de type* ResourceUsageProfile.

La figure 3 illustre la manière selon laquelle un programme d'application susceptible d'être accueilli par la plate-forme JAMUS peut exprimer ses besoins vis à vis des ressources offertes par cette plate-forme. Dans cet exemple le programme définit la méthode statique *getResourceRequirements()*, à laquelle le lanceur de la plate-forme JAMUS fera systématiquement appel pour se renseigner sur les besoins des programmes candidats à l'hébergement. Dans le cas présent, les besoins du programme de la figure 3 se déclinent en quatre profils d'utilisation distincts. Les deux premiers profils concernent l'utilisation de ressources de type *Socket*. Le premier profil spécifie que le programme requiert de pouvoir établir jusqu'à 12 connexions simultanées, avec une capacité de transmission de 2 Mo en émission et 8 Mo en réception. Le second profil pose des contraintes supplémentaires sur les échanges de données réalisés vers un domaine Internet précis (domaine décrit dans cet exemple sous la forme d'une adresse CIDR). Les deux derniers profils portent quant à eux sur l'utilisation du système de fichiers. Le premier de ces profils indique que le programme requiert de pouvoir accéder (en lecture et en écriture) au répertoire */tmp* dans les conditions indiquées : au plus 8 fichiers ouverts simultanément, quota de 1 Mo en écriture, 2 Mo en lecture. Le second pose des contraintes supplémentaires sur l'accès au sous-répertoire */tmp/myStuff :* au plus 2 fichiers ouverts simultanément, quota de 5 Ko en lecture, et écriture interdite.

Le programme d'exemple de la figure 3 sera de nouveau évoqué dans les sections suivantes, qui précisent comment les besoins exprimés par un programme sont pris en compte par la plate-forme JAMUS.

4. Courtier de ressources

Le courtier de ressources intégré à la plate-forme JAMUS met en œuvre un mécanisme de réservation de ressources (inspiré du mécanisme décrit dans [KIM 00]) afin de garantir aux programmes hébergés la disponibilité des ressources dont ils ont fait la demande. Dans sa version actuelle, le contrôle d'admission mis en œuvre dans la plate-forme JAMUS s'applique uniquement aux ressources CPU, réseau et système de fichier.

Contrôle d'admission. Lors du lancement de la plate-forme le courtier de ressources reçoit en paramètre un ensemble de profils d'utilisation décrivant les contraintes posées sur les ressources disponibles sur la plate-forme et dont il va devoir se préoccuper. Lorsqu'un programme candidat à l'hébergement est soumis au contrôle d'admission, le courtier de ressources analyse les différentes demandes formulées par ce programme, demandes qui sont également exprimées sous la forme de profils d'utilisation de ressources. Pour chaque demande formulée par le programme d'application, le courtier décide de la validité de celle-ci au vu des contraintes définies dans la plate-forme. Un programme ne peut être admis que si toutes les demandes formulées par ce dernier sont valides.

Réservation et libération des ressources. La politique d'allocation des ressources mise en œuvre dans la version actuelle de la plate-forme JAMUS est basée sur le modèle de la réservation. Lors de l'analyse d'une demande formulée par un programme d'application, le courtier de ressource réalisant cette analyse identifie les contraintes dont le profil coïncide avec celui de la demande. On dit que le profil d'une contrainte coïncide avec celui d'une demande lorsque le critère de sélection défini par l'attribut *pattern* de la contrainte est vérifié par celui de la demande (on rappelle que l'interface *ResourcePattern* définit une fonction booléenne *conformsTo()*). Une fois identifiées les contraintes dont le profil coïncident avec celui de la demande, le courtier de ressource doit s'assurer que les permissions d'accès demandées sont conformes aux permissions accordées dans ces contraintes. Là encore, un test de conformité réalisé à l'aide de la fonction booléenne *conformsTo()* définie dans l'interface *ResourcePermission* permet de s'assurer que la demande ne va pas à l'encontre des contraintes posées sur le système. Enfin, un dernier test de conformité permet de vérifier que les quotas d'accès spécifiés dans la demande peuvent être satisfaits compte tenu des quotas accordés.

Si les tests de conformité réalisés sur les permissions et sur les quotas se terminent avec succès, cela signifie que la demande considérée peut *a priori* être satisfaite sans violer les contraintes posées sur les ressources disponibles. Cependant le programme ayant formulé cette demande ne pourra être déclaré admissible que si toutes les autres demandes formulées par le programme peuvent également être satisfaites.

Dans l'affirmative, une deuxième passe est réalisée sur les demandes du programme afin que le courtier de ressources lui réserve effectivement les ressources demandées. Pour chaque demande ainsi examinée une seconde fois, la réservation s'effectue en prélevant sur les quotas associés aux contraintes concernées les valeurs de quotas associés à cette demande. L'interface *ResourceQuota* définit deux méthodes *add()* et *subtract()* qui supportent une telle « arithmétique » de quotas (voir figure 2).

Lorsqu'un programme hébergé par la plate-forme JAMUS arrive au terme de son exécution, le courtier de ressources déclenche une série d'opérations inverses à celles décrites ci-dessus afin de recréditer les ressources qui avaient été réservées pour ce programme.

5. Supervision en cours d'exécution

Fonction et principe du moniteur de composant.

Chaque programme d'application hébergé par la plate-forme JAMUS s'exécute sous la supervision d'un moniteur de composant, instance de la classe *ComponentMonitor*. Ce moniteur utilise les profils d'utilisation fournis par le programme pour instancier des moniteurs de ressources (objets implémentant l'interface *ResourceMonitor*) chargés de vérifier le respect de ces profils. Un moniteur de ressource admet en paramètre lors de sa création un profil d'utilisation de ressources. Son rôle est de superviser l'utilisation qui est faite d'une certaine catégorie de ressources (celles qui

satisfont le critère de sélection exprimé par l'attribut *pattern* du profil d'utilisation associé), et de s'assurer que cette utilisation est conforme aux contraintes exprimées par les attributs *permission* et *quota* de ce même profil.

L'environnement RAJE fournit une catégorie de moniteurs de ressources spécifique pour chaque type de ressource considéré. Ainsi la classe *SocketMonitor* définit un type de moniteur capable de superviser l'utilisation des ressources de type *Socket*. Un *SocketMonitor* admet en paramètre lors de sa création un profil d'utilisation dont les attributs référencent des objets de type *SocketPattern*, *SocketPermission*, et *SocketQuota*. Il a donc pour fonction de vérifier pendant l'exécution d'un programme d'application que les *sockets* créés par celui-ci et qui satisfont le critère de sélection du *pattern* spécifié dans le profil sont utilisés conformément aux modalités précisées par les attributs *permission* et *quota* de ce profil.

Par exemple, lors du lancement du programme d'application reproduit dans la figure 3, un moniteur de ressources spécifique va être lancé pour chacun des quatre profils d'utilisation définis par ce programme pour exprimer ses besoins. Deux de ces moniteurs superviseront l'utilisation des ressources de type *Socket*, le premier s'intéressant à tous les *sockets* créés en cours d'exécution, et le second ne s'intéressant qu'aux *sockets* servant à établir des connexions avec des machines du domaine 195.83.160/24. Deux autres moniteurs superviseront l'utilisation des ressources de type *File*, le premier limitant les accès au seul répertoire */tmp* et appliquant un système de quotas à ces accès, le second imposant des contraintes encore plus strictes sur les accès au sous-répertoire */tmp/myStuff*.

Fonction et principe du registre de ressources. Comme de nouveaux objets modélisant des ressources (e.g., *Socket*, *File*, *Thread*) peuvent être créés et détruits dynamiquement tout au long de l'exécution d'un programme, le *ComponentMonitor* chargé de superviser ce programme doit être capable de mettre en relation tout objet modélisant une ressource au sein du programme avec les moniteurs de ressources censés en superviser l'utilisation. Pour ce faire, le *ComponentMonitor* s'appuie sur un registre de ressources (instance de la classe *ResourceRegister*) qui permet d'identifier et d'assurer le suivi des diverses ressources utilisées par le composant. Dans la mise en œuvre actuelle un registre de ressources distinct est créé pour chaque nouveau programme lancé sur la plate-forme JAMUS. Un *classloader* distinct est utilisé pour le lancement de chaque programme, garantissant ainsi que les objets ressources manipulés par des programmes distincts seront enregistrés dans des registres distincts.

Fonction et principe des moniteurs de ressources. Les mécanismes grâce auxquels un moniteur de ressources peut observer et, le cas échéant, contrôler l'utilisation de certaines ressources ne sont pas décrits en détail dans cet article, faute de place suffisante pour ce faire. Nous nous contentons de dresser ici un bref aperçu de ces mécanismes.

La supervision de l'utilisation des ressources peut être réalisée de manière synchrone ou asynchrone, selon le type de ressources considéré. Les moniteurs de ressources mis en œuvre dans la plate-forme JAMUS sont capables d'adopter indifféremment l'un ou l'autre mode de fonctionnement. Le moniteur de composant se charge, lors du lancement d'un programme d'application, de sélectionner le mode de supervision approprié en fonction du type de ressources considéré.

Principe de la supervision synchrone. On parle de supervision synchrone lorsque toute tentative d'accès à une certaine ressource peut être interceptée et faire l'objet d'une prise en compte immédiate par les moniteurs chargés de superviser l'utilisation de ce type de ressources. Cette approche se prête bien à la supervision des ressources qui sont mises en œuvre sous la forme d'objets accesseurs, c'est-à-dire lorsque tout accès élémentaire à une ressource implique l'exécution d'une méthode de l'objet modélisant cette ressource. En Java les classes *Socket*, *DatagramSocket*, *File*, etc. modélisent ainsi des ressources conceptuelles, et peuvent être soumises à une supervision synchrone. Ces classes ont donc été modifiées dans l'environnement RAJE afin de mettre en œuvre un mécanisme de « rappel » *(callback)* : un objet moniteur peut s'inscrire auprès d'un objet ressource, et être ensuite tenu informé par celui-ci de toute opération réalisée sur la ressource qu'il modélise. En outre un moniteur peut s'opposer à la réalisation d'une certaine opération sur un objet ressource en retournant un signal d'exception (objet de type *ResourceAccessException)* vers cet objet. Ce dernier renonce alors à l'opération qu'il s'apprêtait à réaliser et retourne à son tour le signal d'exception approprié vers le programme d'application.

Principe de la supervision asynchrone. Il existe des ressources dont la supervision ne peut être réalisée selon le modèle synchrone présenté ci-dessus. C'est par exemple le cas de la ressource CPU, dont l'utilisation par un programme d'application ou par un *thread* Java n'est pas conditionnée par l'appel de méthodes sur un objet modélisant cette ressource. L'accès à la ressource CPU demeure sous le contrôle exclusif de l'ordonnanceur de la machine virtuelle Java (JVM) ou du système d'exploitation sous-jacent.

Pour de telles ressources l'environnement RAJE met en œuvre des mécanismes d'observation et de verrouillage asynchrones. L'observation asynchrone d'une ressource consiste à en consulter ponctuellement l'état, sans que l'instant auquel s'effectue cette consultation coïncide nécessairement avec une tentative d'utilisation de cette ressource. Pour être observable de manière asynchrone, un objet modélisant une ressource doit être capable de produire à la demande un rapport d'observation. À chaque type de ressource correspond un type de rapport d'observation spécifique dans l'environnement RAJE. Ainsi, un objet de type *Thread* pourra produire à la demande un rapport d'observation renseignant l'appelant (un moniteur de ressource dans le cas de la plate-forme JAMUS) sur le niveau de priorité de ce *thread*, et sur le taux d'utilisation du CPU par ce *thread* au cours de la dernière période d'observation. Si l'analyse d'un tel rapport par un moniteur révèle que l'utilisation de la ressource considérée

n'est pas conforme aux modalités définies dans le profil d'utilisation associé à ce moniteur, alors celui-ci peut verrouiller la ressource. Le fait de verrouiller une ressource se traduira par la levée d'une exception lors de toute tentative ultérieure d'utilisation de cette ressource par le programme d'application.

6. Travaux apparentés

L'environnement d'exécution de Java (JRE *: Java Runtime Environment*) met en œuvre un modèle de sécurité connu sous le nom de *SandBox*. Dans les premières versions de la JRE, ce modèle donnait au code local – considéré comme sûr – la possibilité d'accéder à toutes les ressources du système. En revanche un code distant (téléchargé sous la forme d'une *applet)* était considéré comme suspect, et se voyait interdire l'accès à la plupart des ressources sensibles du système [GON 97]. Avec la plate-forme Java 2 ce modèle du « tout ou rien » fut abandonné au profit d'un nouveau modèle reposant sur la notion de domaine de protection [GON 97, GON 98, VEN 98]. Un domaine de protection constitue un environnement d'exécution dont la politique de sécurité peut être spécifiée sous la forme de permissions. Un contrôleur d'accès attaché à chaque domaine de protection supervise les accès aux ressources et applique la politique de sécurité définie par les permissions accordées au domaine.

J-Kernel étend cette notion de domaine de protection en offrant un mécanisme de communication et de partage de données entre domaines [HAW 98]. La communication est cependant limitée aux appels de méthodes sur des objets nommés *capabilities*.

Le modèle de sécurité mis en œuvre dans J-Kernel et dans le JRE repose sur des mécanismes « sans état ». On ne peut conditionner l'accès à une certaine ressource en fonction des accès réalisés précédemment à cette même ressource. On ne peut donc poser des contraintes d'ordre quantitatif (quantité de CPU, quotas d'entrée-sortie, etc.) sur les ressources manipulées au sein d'un domaine de protection. Les mécanismes de J-Kernel et du JRE ne permettent donc pas de prévenir les dysfonctionnements résultant de l'utilisation abusive d'une certaine ressource (attaques de type déni de service, etc.).

Des environnements tels que JRes [BAG 97, CZA 98], GVM [BAC 00b], et KaffeOS [BAC 00a] apportent un début de réponse au problème du contrôle quantitatif des ressources posé par les environnements Java traditionnels. Ils offrent en effet des mécanismes permettant de comptabiliser – et, éventuellement, de limiter – l'utilisation de chaque type de ressource par une entité active (un *thread* dans le cas de JRes, un processus dans le cas de GVM et de KaffeOS). Cependant dans ces environnements les ressources sur lesquelles portent la comptabilisation et le contrôle sont des ressources globales. On peut ainsi comptabiliser les accès au réseau réalisés par un *thread* (ou processus), mais on ne peut distinguer les transmissions réalisées vers une certaine machine, ou vers un numéro de port précis. On peut de même comptabiliser les accès au système de fichier, mais on ne peut poser de contrainte particulière sur les accès réalisés vers des répertoires ou fichiers spécifiques.

Les projets Naccio [EVA 00, EVA 99] et Ariel [PAN 99], orientés vers la sécurité, proposent chacun un langage et des mécanismes permettant de définir de manière très précise la politique de sécurité qui doit être appliquée à un programme d'application lors de son exécution. L'application d'une politique de sécurité est réalisée statiquement, par réécriture du *bytecode* du programme d'application, mais aussi des classes de l'API de la plate-forme Java. Les auteurs de Naccio précisent fort justement que cette approche permet de réduire le surcoût engendré par la supervision d'un programme au cours de son exécution. En effet les segments de code assurant la supervision des accès à un certain type de ressource ne sont insérés dans les classes de l'API que lorsque la politique choisie le justifie. En revanche la génération d'une nouvelle API vérifiant une politique de sécurité donnée est une opération extrêmement coûteuse. Cette approche se prête donc bien à la génération anticipée d'un ensemble d'API pré-définies garantissant chacune le respect d'une politique de sécurité générique. Par contre elle ne permettrait pas, comme le fait la plate-forme JAMUS, d'assurer la supervision d'un programme d'application en fonction d'une politique de sécurité définie à partir des besoins exprimés par ce même programme lors de son démarrage.

[CHA 00] présente une approche « *sandbox* » permettant d'imposer des restrictions qualitatives et quantitatives sur les ressources système utilisées par les programmes d'applications. Cette approche mise en œuvre au niveau système offre la possibilité de contrôler la consommation processeur, mémoire et réseau afin de prévenir l'utilisation abusive de ces ressources. Elle vise également à offrir une relative équité vis à vis du partage des ressources entre les différents programmes d'applications qui s'exécutent au sein du système. Contrairement à JAMUS cette approche *« sandbox »*, mise en œuvre à la fois sous Windows NT et sous Linux, ne permet pas de prendre en compte les besoins – ou exigences – des applications vis à vis des ressources.

7. Conclusion

Dans cet article nous avons présenté la plate-forme d'accueil JAMUS, qui autorise une forme élémentaire de contractualisation de l'accès aux ressources nécessaires aux programmes qu'elle héberge.

Certains types de ressources – dont la ressource mémoire – ne sont pas encore pris en compte au sein de l'environnement RAJE, et ne peuvent donc faire l'objet d'une supervision sous JAMUS. D'autre part la plate-forme ne peut dans son état actuel héberger que des programmes d'application Java. Ayant pour vocation principale l'accueil de code mobile, elle devrait à terme être en mesure de recevoir et d'héberger des *applets*, voire des *aglets* Java.

Le développement de la plate-forme n'étant pas achevé, il n'est pas encore possible de procéder à des mesures de performances sur des cas réalistes. JAMUS étant bâtie selon une approche « tout dynamique » (à la différence de Naccio et d'Ariel), il est à

prévoir que le surcoût engendré par les mécanismes assurant la supervision des programmes en cours d'exécution soit relativement élevé. Nous espérons cependant que ce surcoût demeurera acceptable, compte tenu des avantages apportés par le contrôle fin des ressources. Une perspective possible étant de faire évoluer la plate-forme JAMUS afin qu'elle permette la négociation – voire la renégociation – dynamiques de nouveaux contrats portant sur les ressources au cours de l'exécution d'un programme, l'approche « tout dynamique » pourrait constituer un atout majeur dans cette optique.

8. Bibliographie

[BAC 00a] BACK G., HSIEH W. C., LEPREAU J., « Processes in KaffeOS : Isolation, Resource Management, and Sharing in Java », *4th Symposium on Operating Systems Design and Implementation*, octobre 2000.

[BAC 00b] BACK G., TULLMANN P., STOLLER L., HSIEH W. C., LEPREAU J., « Techniques for the Design of Java Operating Systems », *USENIX Annual Technical Conference*, juin 2000.

[BAG 97] BAGARATNAN N., BYRNE S. B., « Resource Access Control for an Internet UserAgent », *Third USENIX Conference on Object-Oriented Technologie and Systems*, 1997.

[CHA 00] CHANG F., ITZKOVITZ A., KARAMACHETI V., « User-level Resource-constrained Sandboxing », *4th USENIX Windows Systems Symposium*, , 2000.

[CZA 98] CZAJKOWSKI G., VON EICKEN T., « JRes : a Resource Accounting Interface for Java », *ACM OOPSLA Conference*, 1998.

[EVA 99] EVANS D., TWYMAN A., « Flexible Policy-Directed Code Safety », *IEEE Security and Privacy*, mai 1999.

[EVA 00] EVANS D., « Policy-Directed Code Safety », PhD thesis, Massachussets Institute of Technology, février 2000.

[GON 97] GONG L., « Java Security : Present and Near Future », *IEEE Micro*, vol. -, 1997, p. 14-19.

[GON 98] GONG L., SCHEMERS R., « Implementing Protection Domains in the Java Development Kit 1.2 », *Internet Society Symposium on Network and Distributed System Scurity*, mars 1998.

[HAW 98] HAWBLITZEL C., CHANG C.-C., CZAJKOWSKI G., HU D., VON EICKEN T., « Implementing Multiple Protection Domains in Java », *USENIX Annual Technical Conference*, juin 1998.

[KIM 00] KIM K., NAHRSTEDT K., « A Resource Broker Model with Integrated Reservation Scheme », *IEEE International Conference on Multimedia and Expo (II)*, 2000, p. 859-862.

[PAN 99] PANDEY R., HASHII B., « Providing Fine-Grained Access Control for Java Programs », *The 13th Conference on Object-Oriented Programming, ECOOP'99*, Springer-Verlag, juin 1999.

[VEN 98] VENERS B., *Inside the Java 2 Virtual Machine,* Mac Graw-Hill, 1998.