**CFSE'1**
**Rennes, 8 juin - 11 juin 1999**
**1ère Conférence Française sur les Systèmes d'Exploitation**

# Capture et restauration du contexte d'exécution d'un *thread* dans l'environnement Java

S. Bouchenak, D. Hagimont, X. Rousset de Pina
Projet SIRAC (IMAG-INRIA)
INRIA, 655 av. de l'Europe, 38330 Montbonnot Saint Martin, France.
Internet : Sara.Bouchenak@inria.fr

## Résumé

La machine virtuelle Java est actuellement portée sur la plupart des systèmes courants et fournit des services facilitant le développement d'applications distribuées tel RMI. Dans cette machine virtuelle, la mobilité est un aspect très important. En effet, Java fournit un mécanisme de sérialisation d'objets permettant de capturer et de restaurer l'état des objets Java et ainsi, de déplacer ces objets entre différentes machines. Java permet aussi le chargement dynamique des classes et leur déplacement vers d'autres nœuds. En revanche, Java ne fournit pas de mécanisme permettant la capture et la restauration de l'état d'un flot d'exécution, la pile d'un flot Java n'étant pas accessible. Un tel mécanisme serait intéressant et pourrait servir pour la migration d'agents mobiles ou comme service de base de capture de points de reprise pour la tolérance aux fautes. Dans cet article, nous présentons notre expérience d'extension de la machine virtuelle Java qui vise à fournir un mécanisme de capture et de restauration de l'état d'un flot d'exécution Java. Nous décrivons la mise en œuvre de ce mécanisme et les résultats des mesures préliminaires de ses coûts.

## Abstract

Today, distributed computing over the Internet is closely linked with Java. The Java virtual machine is ported to most of the current operating systems and provides many services which help developing distributed applications (e.g. RMI). In Java, mobility is a very important aspect. Java provides a serialisation mechanism which allows the capture and restoration of objects states and therefore to migrate objects between machines. It also allows classes to be dynamically loaded and therefore to be moved between nodes. However, Java does not provide a mechanism for capturing and restoring a thread state. The stack of a Java thread is not accessible. Such a mechanism would notably allow a thread to be checkpointed or migrated between different nodes. In this paper, we report on our experience which consisted in extending the Java virtual machine in order to allow the capture and restoration of a thread state. We overview the implementation of this extension and provide preliminary results of its evaluation.

# 1 Introduction

Le développement d'applications réparties, mettant en œuvre un grand nombre de machines interconnectées par des réseaux généraux et notamment l'Internet, est un domaine de recherche en plein essor. Dans ce contexte, le langage à objets Java a connu un rapide succès. Le compilateur Java fournit un code intermédiaire interprété par une machine virtuelle, la *Java Virtual Machine* ou JVM. La JVM est implantée sur la plupart des stations de travail et des ordinateurs personnels, ce qui fait que le couple langage Java-JVM est considéré comme la plate-forme de référence des applications développées pour l'Internet. Un des intérêts majeurs de la très grande diffusion de cette plate-forme est qu'elle permet de considérer l'ensemble des machines virtuelles Java de l'Internet comme une base homogène pour construire des services répartis et notamment des services systèmes.

Les mécanismes auxquels nous nous intéressons dans cet article, appelés mécanismes de capture-restauration de l'état d'un flot de contrôle (ou *thread*), sont ceux qui permettent le déplacement d'un flot d'exécution d'une JVM vers une autre, afin d'y poursuivre son exécution. Ces mécanismes, seuls ou complétés par d'autres, ont des applications dans plusieurs domaines comme : l'équilibrage de charge entre les machines [13], la diminution du trafic réseau par déplacement des clients vers leur serveur [5], l'administration de machines [15], la prise de point de reprise [16], ou enfin la définition de flots de contrôle (ou agents) mobiles [4]. La fourniture de ces services sur les plates-formes Java dépend, en partie, de la disponibilité d'un mécanisme de capture/restauration de l'état d'un flot de contrôle dans l'environnement Java.

L'environnement Java fournit déjà un service de sérialisation d'objets [19], qui permet de capturer l'état d'un objet Java et de le restaurer. Il fournit en outre un service de chargement dynamique des classes qui permet de déplacer le code entre des machines [18]. Ces deux services sont à la base du développement de systèmes dits «à agents mobiles», tels que les Aglets [9] ou Odyssey [7], permettant à des flots de contrôle d'être déplacés entre des machines. Cependant, comme Java ne fournit pas de service permettant d'accéder à l'état d'un flot de contrôle au cours de son exécution, il est difficile de redémarrer l'application au point où elle a été interrompue. Donc ces systèmes ne fournissent qu'une mobilité dite *faible* [1].

Dans cet article, nous présentons une expérience qui a consisté à étendre la machine virtuelle Java afin qu'elle permette de capturer/restaurer l'état d'un flot de contrôle. Un prototype a été réalisé par extension du kit de développement Java (JDK 1.1.3). Le service réalisé permet de capturer l'état d'un flot de contrôle, et d'utiliser l'état capturé pour initialiser un nouveau flot de contrôle. Notre mécanisme ne prend pas en compte les problèmes liés aux objets et aux communications que peut partager le flot de contrôle avec d'autres. Cependant, le mécanisme peut être, ainsi que nous le montrons, facilement adapté aux besoins de l'application qui l'utilise. En effet, il lui permet de spécifier, pour certains types d'objets, des traitements supplémentaires à effectuer lors de la capture du contexte d'un flot d'exécution et de sa restauration. Afin d'évaluer le service fourni d'un point de vue fonctionnel, nous l'avons utilisé pour réaliser un mécanisme de migration d'agents mobiles et comme mécanisme de base pour la capture de points de reprise d'une application. Ces deux expériences démontrent la souplesse du mécanisme que nous présentons. Pour évaluer son efficacité, nous avons effectué un certain nombre de mesures afin d'avoir, d'une part, une première évaluation des coûts élémentaires qu'il induit et, d'autre part, des éléments de

comparaison de notre implantation avec celle, très différente, réalisée par le projet WASP [6].

La suite de cet article est organisée comme suit. La section 2 présente nos motivations, un court état de l'art et les aspects de la JVM utiles à la compréhension de notre travail. Dans la section 3, nous définissons les principes de conception de notre mécanisme de capture/restauration du contexte d'exécution d'un flot de contrôle Java, suivis, dans la section 4, de la présentation de la réalisation de ce mécanisme. La section 5 décrit les expériences d'utilisation effectuées et donne une courte évaluation quantitative de notre mécanisme. Enfin, dans la section 6 nous donnons quelques conclusions et perspectives de notre travail.

# 2 Contexte de travail

Le mécanisme de capture-restauration du contexte d'exécution d'un flot de contrôle a été intégré à la version JDK 1.1.3 de la Machine Virtuelle Java (JVM). Nous présentons, tout d'abord, les raisons de ce choix en les mettant dans la perspective des travaux qui ont été menés par d'autres équipes de recherche. Nous décrivons ensuite les quelques caractéristiques de la JVM qui sont nécessaires à la compréhension de la suite de notre travail.

## 2.1 Motivations

Le service de capture/restauration de l'état d'un flot de contrôle a de nombreuses applications, les principales étant la gestion de points de reprise et le déplacement d'applications d'un nœud vers un autre (ou migration).

Des études des principales techniques développées dans le domaine spécifique de la migration de processus peuvent être trouvées dans [14] et dans [12]. Cependant, si on limite le domaine de mise en œuvre du mécanisme à l'environnement constitué par le langage Java et la JVM, le problème peut être abordé selon deux approches.

La première, que nous appelons *gestion explicite*, consiste à laisser entièrement à la charge du programmeur la gestion de la capture et de la reconstruction de l'état du flot d'exécution. Le programmeur doit donc ajouter du code qui effectue ce travail, en des points fixes de l'application dits points de sauvegarde. En un point de sauvegarde, le code ajouté doit enregistrer l'état de l'exécution : il note la dernière instruction exécutée ainsi que l'état courant de l'application. La reprise de l'exécution se fait à partir du dernier point de sauvegarde enregistré. Cette solution manque de souplesse car la gestion des points de sauvegarde est entièrement à la charge du programmeur. Cette gestion explicite est cependant utilisée dans la plupart des applications utilisant des systèmes à agents mobiles [4] tels que les Aglets [9] ou Mole [1].

La seconde approche est dite *implicite*, car elle consiste à fournir un mécanisme générique qui permet de capturer l'état d'exécution courant d'une application. Il suffit que l'application appelle la primitive qui capture l'état courant de l'exécution, et l'état de chacun des appels de méthodes en cours d'exécution sera capturé. Cette solution donne lieu à deux implantations différentes que nous détaillons ci-dessous.

La première implantation consiste à fournir un pré-processeur qui effectue un traitement supplémentaire sur le code Java de l'application pour y inclure des instructions supplémentaires ; ces instructions copient l'état de l'exécution (essentiellement les variables empilées)

dans une structure de données intermédiaire. La motivation principale de cette méthode est de ne pas modifier la machine virtuelle Java. Ainsi, lorsque l'on désire capturer l'état de l'application, il suffit de prendre la structure intermédiaire dont le contenu à jour par le code injecté dans l'application. Lorsque l'on veut restaurer le contexte d'exécution à partir de cette structure, il suffit d'exécuter un programme qui interprète à nouveau les données de la structure pour initialiser l'état d'un flot de contrôle. Lorsqu'il sera activé, ce flot démarrera au point où l'exécution a été interrompue lors de la capture. L'inconvénient de cette solution est double : elle a un impact non négligeable sur les performances de l'application (dû au code injecté) et l'initialisation du flot de contrôle nécessite une ré-exécution partielle de l'application. Cette solution a notamment été mise en œuvre par le projet WASP [6].

La seconde implantation consiste à étendre la machine virtuelle Java afin de rendre l'état d'un flot de contrôle accessible aux programmes Java. L'extension fournie doit permettre d'extraire l'état pour le stocker dans une structure de données qui pourra être éventuellement soit stockée dans un fichier, soit transférée sur un autre nœud via le réseau. Cette méthode doit également permettre de reconstruire un flot d'exécution dont l'état initial est défini au moyen de la structure de données contenant l'état stocké. Cette approche à été utilisée dans la réalisation de la plate-forme à agents mobiles Sumatra [17] et c'est celle que nous avons choisie pour deux raisons :

- Elle réduit l'impact du mécanisme sur les performances de l'application. En effet, une mise en œuvre qui consiste à étendre la machine virtuelle Java est principalement native (en C), donc beaucoup moins coûteuse que celle qui consiste à utiliser un préprocesseur pour injecter du code Java à l'application. Cette réduction du surcoût est obtenue en minimisant, d'une part, le travail induit par le mécanisme sur l'activité «normale»de l'application et, d'autre part, le travail à effectuer par le mécanisme lors qu'il est utilisé.

- Elle est cohérente avec le fait que les services fournis par ce mécanisme sont génériques ; ils ne sont pas uniquement utilisables par les applications voulant bénéficier de la mobilité mais aussi comme base de mécanismes pour la tolérance aux fautes.

Contrairement à la plate-forme Sumatra, qui fournit un service de mobilité pour ses agents, notre mécanisme est plus générique et peut être utilisé aussi bien pour la mobilité que comme base de capture de points de reprise.

Pour mettre en œuvre ce service de capture/restauration de l'état d'un flot de contrôle, nous avons voulu fournir une interface qui soit la plus proche possible de celle qui est fournie par Java pour capturer ou restaurer l'état d'un objet (*sérialisation*/*dé-sérialisation*) [19]. En effet, notre objectif a été de permettre de *sérialiser*/*dé-sérialiser* un flot de contrôle. La sérialisation d'un flot de contrôle revient à capturer l'état courant de ce flot, sous la forme d'une structure de données Java ; la dé-sérialisation revient à restaurer le flot de contrôle à partir de la structure de données construite précédemment.

## 2.2 Contexte d'exécution d'un flot de contrôle Java

Une Machine Virtuelle Java peut supporter l'exécution de plusieurs flots de contrôle ou *threads* [11]. Cette machine virtuelle manipule trois structures de données principales :

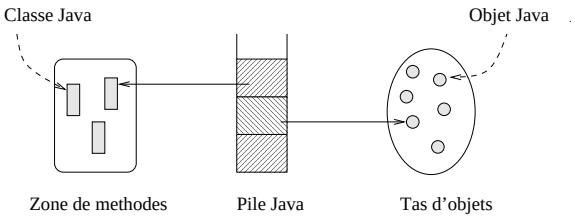


Figure 1: **Contexte d'éxecution d'un thread Java**

- *La pile Java.* Une pile Java est associée à chaque flot de contrôle Java de la machine virtuelle ; cette pile décrit l'état d'avancement de l'exécution du flot. La pile Java contient des régions ou *frames* ; une région est créée lors de l'appel d'une méthode Java puis détruite lors de la terminaison de la méthode associée. La région contient, entre autres, les variables locales à la méthode et les résultats des calculs intermédiaires. De plus, une région contient des registres tels que le registre de sommet de pile ou le registre compteur ordinal indiquant la prochaine instruction de code intermédiaire (*byte code*) à exécuter (si la méthode associée à la région est une méthode Java et non une méthode dite «native»dont le code n'est pas interprété).

- *Le tas d'objets.* Il y a un tas d'objets par machine virtuelle, partagé par tous les flots de contrôle de la machine virtuelle. Il regroupe l'ensemble des objets Java créés et utilisés par les flots de contrôle de la JVM.

- *La zone des méthodes.* La Machine Virtuelle Java possède une zone des méthodes partagée par l'ensemble des flots de contrôle de cette machine. Cette zone regroupe l'ensemble des classes Java utilisées par les flots de la JVM.

Le contexte d'exécution d'un flot de contrôle Java, illustré par la Figure 1, est donc constitué des trois ensembles d'informations suivants : sa pile, son tas d'objets constitués de l'ensemble des objets du tas de la JVM qu'il manipule (dont les références sont dans sa pile), et sa zone de méthodes qui est constituée des classes de la zone de méthodes de la JVM qu'il utilise. Les références des classes utilisées par le flot de contrôle sont également stockées dans sa pile.

# 3 Conception d'un mécanisme de capture/restauration du contexte d'exécution d'un flot de contrôle Java

Nous décrivons, dans cette section, le principe du mécanisme qui permet de capturer puis de reconstruire le contexte d'exécution d'un flot de contrôle Java. Nous décrivons ensuite les différents problèmes que nous avons rencontrés et les choix de conception qui nous ont permis d'y faire face. Dans la suite, l'environnement «JDK 1.1.3»est désigné par le terme Java. De plus, chaque fois que cela ne prête pas à confusion, le terme flot sera utilisé à la place de «flot de contrôle Java».

### 3.1 Principe du mécanisme

Le mécanisme que nous proposons doit fournir deux fonctions : la capture du contexte d'exécution courant d'un flot de contrôle Java puis la restauration de ce contexte. Ce mécanisme permet donc, d'une part, d'interrompre un flot Java en cours d'exécution et d'**extraire** son contexte courant et, d'autre part, d'**intégrer** ce contexte d'exécution à un nouveau flot Java qui, une fois lancé, démarre son exécution au point même où le précédent a été interrompu.

L'extraction du contexte d'exécution d'un flot Java revient à construire une structure de données qui contient toutes les informations permettant de restaurer ce contexte. Cette structure de données doit donc contenir toutes les informations nécessaires à la reconstruction de la pile du flot, de son tas d'objets et de sa zone de méthodes. Pour extraire le contexte d'exécution d'un flot Java, il faut tout d'abord capturer la pile qui est associée à ce flot. Il faut ensuite identifier l'ensemble des objets et des classes Java qui sont utilisés par ce flot ; ceci se fait en parcourant la pile Java du flot pour y récupérer les références vers les objets et les classes Java. Finalement, une structure de données Java est construite, contenant toutes ces informations.

L'intégration d'un contexte d'exécution à un flot Java revient à reconstruire un contexte d'exécution à partir de la structure de données obtenue lors d'une opération d'extraction. Cette opération d'intégration consiste donc à construire un nouveau flot de contrôle Java, ayant une pile, un tas d'objets et une zone de méthodes identiques à ceux du flot ayant subi l'opération d'extraction.

### 3.2 Principaux problèmes

Un des problèmes rencontrés lors de la capture du contexte d'exécution d'un flot Java est le problème du *type des valeurs rangées dans la pile Java* d'un flot. En effet, la pile Java d'un flot contient les valeurs des calculs intermédiaires et les valeurs des variables locales des méthodes en cours d'exécution. Ces valeurs peuvent être de différents types : entier, réel, référence Java etc. Mais la structure de pile Java ne donne aucune information sur le type des valeurs qui y sont contenues. Cela entraîne deux difficultés.

La première concerne l'identification du tas d'objets Java associé à ce flot. En effet, pour identifier l'ensemble des objets Java utilisés par un flot, il faut parcourir la pile Java de ce flot et récupérer l'ensemble des références Java qui y sont utilisées. Mais comment sait-on qu'une valeur sur la pile est une référence Java plutôt qu'un entier ?

La seconde difficulté engendrée par ce problème de types est liée au fait que la pile Java est une structure qui dépend de l'architecture de la machine physique (structure C). Le codage des données contenues dans cette pile diffère donc d'une architecture de machine à une autre. Donc, pour que notre mécanisme de capture/restauration puisse fonctionner entre machines hétérogènes, il faut que la structure de données construite lors de la capture contienne les informations relatives aux types des valeurs contenues dans la pile. On est donc face au même problème que précédemment.

Un autre problème rencontré concerne les adresses absolues contenues dans les registres de la pile Java. En effet, la pile Java d'un flot contient des pseudos registres comme le compteur ordinal (pc) qui pointe vers la prochaine instruction à exécuter ou le registre de sommet de pile. Les valeurs contenues dans ces registres sont des adresses virtuelles absolues qui n'ont plus de sens lors de la restauration de la pile et qui, donc, doivent être

recalculées.

En dehors de ces deux problèmes, se posent les problèmes relatifs à la gestion des données et des communications partagées par les flots de contrôle. En effet, que se passe-t-il lorsqu'un flot se déplace d'un site source vers un site destination alors qu'il accédait à des objets utilisés également par d'autres flots du site source ? Comment sont gérées les communications entre flots de contrôle lorsqu'un des flots mis en œuvre se déplace d'un site à un autre ?

### 3.3 Choix de conception

Nous présentons les solutions que nous avons choisi d'apporter aux problèmes présentés ci-dessus.

En ce qui concerne le problème du type des valeurs contenues dans la pile Java d'un flot, il est nécessaire, lors de la capture du contexte du flot, de récupérer le type de chacune des valeurs se trouvant dans la pile Java. Le seul moment où le type d'une valeur rangée dans la pile peut être connu est losqu'elle est empilée. En effet, les instructions de code intermédiaire sont typées : elles ne s'appliquent qu'à des valeurs d'un même type [11]. Par exemple, les instructions :

- *lstore_* <*n*> : stocke l'entier (long), situé en sommet de pile, dans la *n*-ième variable locale de la région associée à la méthode courante.
- *aload_* <*n*> : charge, au sommet de la pile, la référence Java stockée dans la *n*-ième variable locale de la région associée à la méthode courante.

Pour obtenir le type des valeurs contenues dans la pile Java d'un flot, nous avons étendu l'interprète Java afin de gérer une pile de types, «parallèle»à la pile Java. Cette pile de types donne, pour chaque valeur dans la pile Java, le type de cette valeur. La construction de cette pile de types se fait au fur et à mesure de l'interprétation du code exécuté par le flot ; ainsi, l'instruction interprétée de code intermédiaire nous donne l'information sur le type de la valeur empilée. De cette façon, lors de la capture du contexte d'exécution d'un flot, nous possédons le type des valeurs rangées dans la pile Java du flot. Nous pouvons donc identifier l'ensemble des objets Java utilisés par ce flot ainsi que le type des autres valeurs rangées dans la pile Java. Lors de la restauration du contexte, le tas d'objets du flot pourra être reconstitué et il n'y aura pas de problème dû à l'hétérogénéité des machines.

Quant au problème posé par les adresses absolues contenues dans la pile Java d'un flot, nous l'avons classiquement résolu en transformant les adresses absolues en déplacements. Pour ce qui est des adresses absolues vers la pile, nous calculons des déplacements relatifs au début de pile, et pour les adresses vers le code, relativement au début du code.

Pour ce qui est du problème des objets partagés et des communications ouvertes entre différents flots, nous avons délibérément choisi de ne pas fournir de solution à ces problèmes au niveau de la JVM. Notre objectif, dans un premier temps, est de fournir un nouveau mécanisme pour le JDK, un mécanisme de capture/restauration de base du contexte d'exécution d'un flot Java ; nous ne nous plaçons donc pas dans le cadre d'un système réparti.. Mais, comme nous le montrons en 5.1.4, certains de ces problèmes, comme la communication entre flots, peuvent être résolus en adaptant notre mécanisme aux besoins de l'application qui l'utilise. Par contre, le problème des objets partagés par différents flots n'est pas résolu au niveau de notre mécanisme ; cette gestion reviendrait donc au

programmeur de l'application qui déciderait soit de n'appliquer ce mécanisme que sur un flot isolé soit de fournir un système de désignation globale des objets de son application.

# 4 Réalisation du mécanisme de capture/restauration du contexte d'exécution d'un flot de contrôle Java

Dans cette section, nous décrivons le contexte de réalisation de notre mécanisme, puis l'extension apportée à la JVM grâce à l'ajout de la classe *MobileThread* et nous donnons enfin quelques résultats techniques apportés par cette réalisation.

## 4.1 Plate-forme de réalisation : JVM

Le mécanisme de capture/restauration du contexte d'exécution d'un flot de contrôle que nous avons réalisé a été intégré à la Machine Virtuelle Java et, plus précisément, au JDK 1.1.3 [18]. Le code source du JDK 1.1.3, sur lequel nous nous sommes appuyés, est destiné à plusieurs plates-formes telles que : Solaris sur des architectures Sparc ou de type x86, ou Microsoft Windows NT et Microsoft Windows 95. Ce JDK est constitué d'une arborescence importante, comprenant autour de 470.000 lignes de code Java, 180.000 lignes de code C/C++ et 4.500 lignes de code assembleur.

## 4.2 Notre extension : classe MobileThread

Le mécanisme que nous avons réalisé [2] permet, d'une part, d'interrompre un flot de contrôle Java en cours d'exécution, d'extraire son contexte courant et de construire une structure de données qui synthétise ce contexte et, d'autre part, d'intégrer un contexte d'exécution à un nouveau flot et de lancer son exécution pour qu'il reprenne au point où le précédent a été interrompu.

Pour cela, nous avons construit une classe *MobileThread* [3], qui fait partie du *package java.lang.* du JDK 1.1.3 [18] et qui est une sous-classe de la classe *Thread* de Java. La classe *MobileThread* caractérise les flots Java dont le contexte d'exécution peut être capturé puis restauré.

Par ailleurs, nous avons construit une classe *ExecutionEnvironment* dont les objets sont des structures de données synthétisant le contexte d'exécution d'un flot de la classe *MobileThread.* Pour des raisons de protection, les variables et les méthodes de cette classe sont privées : ces variables ne peuvent pas être manipulées et ces méthodes ne peuvent pas être appelées par des programmes Java utilisateurs. Seul notre mécanisme de capture/restauration du contexte d'exécution d'un flot peut manipuler des objets *ExecutionEnvironment* ; ainsi, nous pouvons être sûrs qu'une structure synthétisant le contexte d'exécution d'un flot est toujours bien construite.

L'interface de la classe *MobileThread* est illustrée par la Figure 2. Un flot de la classe *MobileThread* possède une variable *ExecEnv* qui est un objet *ExecutionEnvironment.* Cette variable n'est initialisée qu'au moment de l'extraction du contexte d'exécution du flot où elle contiendra toutes les informations relatives à ce contexte et nécessaires à sa restauration.

La méthode *extractExecEnv* sert à capturer le contexte d'exécution courant d'un flot. Elle commence, tout d'abord, par interrompre le flot en cours d'exécution et à capturer son contexte courant ; ce contexte est affecté à la variable *ExecEnv* du flot. Une fois la capture

```
abstract class      MobileThread
  extends  Thread   implements      java.io.Serializable {
        public   ExecutionEnvironment ExecEnv;
        public   void extractExecEnv(boolean toStop, String[] args);
        public   abstract void transferExecEnv(ExecutionEnvironment execEnv, String[] args);
        public   static  MobileThread integrateExecEnv(ExecutionEnvironment execEnv);
```

Figure 2: **Interface d'utilisation de la classe MobileThread**

du contexte effectuée, l'exécution de ce flot peut être soit définitivement interrompue, si le paramètre *toStop* est à vrai, soit relancée, si le paramètre *toStop* est à faux. Ainsi, cette primitive peut servir également à effectuer un clonage de flot à distance. Un dernier traitement effectué par la méthode *extractExecEnv* est l'appel de la méthode *transferExecEnv*. Le paramètre *args* de la méthode *extractExecEnv* sera le paramètre passé lors de l'appel de *transferExecEnv*.

La méthode *transferExecEnv* définit le traitement relatif au transfert du contexte d'exécution d'un flot, après sa capture. Cette méthode est abstraite (son interface est définie mais pas son corps) car sa mise en œuvre dépend des besoins de l'application utilisant notre mécanisme. En effet, une application utilisant la migration de flot doit transférer le contexte du flot vers une machine distante tandis qu'une application ayant besoin d'une sauvegarde de l'état du flot pour une reprise ultérieure doit stocker le contexte du flot sur disque. La mise en œuvre de la méthode *transferExecEnv* est donc à la charge du programmeur de l'application utilisant notre mécanisme. Le paramètre *execEnv* de cette méthode est le contexte d'exécution qui va être transféré ou copié. D'autre part, du fait que le traitement effectué par la méthode *transferExecEnv* diffère d'une application à une autre, nous proposons un second paramètre générique qui puisse contenir tous les paramètres dont aurait besoin cette méthode. Ce paramètre est le paramètre *args*.

La méthode *integrateExecEnv* sert à restaurer un contexte d'exécution. Cette méthode crée un nouveau flot et y intègre le contexte d'exécution *execEnv* passé en paramètre. Une fois l'intégration effectuée, l'exécution du nouveau flot est lancée, celle-ci reprend au point où elle a été interrompue.

### 4.3 Résultats techniques

Pour la réalisation de notre mécanisme, nous avons, d'une part, étendu le package java.lang. du JDK 1.1.3 pour l'ajout de nouvelles classes, telles que les classes-utilisateur *MobileThread* et *ExecutionEnvironment* et des classes internes nécessaires à nos traitements.

D'autre part, pour produire une pile de types parallèle à la pile Java d'un flot (voir 3.3), il fallait que l'interprète Java soit étendu. Mais pour ne pas alourdir les performances d'exécution des programmes Java n'utilisant pas notre mécanisme, nous avons proposé un autre interprète Java ; cet interprète proposé effectue les mêmes traitements par rapport à l'interprétation du *byte code*, mais gère en plus la pile de types associée au flot. Cet interprète n'est utilisé que par les flots de contrôle de la classe *MobileThread* ; ainsi, les

performances des autres programmes Java n'en pâtissent pas.

La réalisation de notre mécanisme est modulaire et peut donc être facilement portée vers une autre Machine Virtuelle Java. Cette réalisation a nécessité 700 lignes de code Java et quelques 4.000 lignes de code C.

# 5 Expérimentation et évaluation

Dans cette section, nous décrivons, tout d'abord, les premières expériences effectuées avec notre mécanisme de capture/restauration du contexte d'exécution d'un flot Java. Nous présentons ensuite l'évaluation des différents coûts induits par notre mécanisme.

## 5.1 Expérimentations

Les premières expérimentations effectuées avec notre mécanisme de capture/restauration du contexte d'exécution d'un flot Java servent à illustrer le mode d'utilisation de ce mécanisme par le programmeur d'applications nécessitant la migration, le clonage à distance de flots Java ou la sauvegarde de l'état d'un flot puis sa reprise. Nous illustrons également l'*adaptabilité* de notre mécanisme.

Dans ce qui suit, nous détaillons l'expérimentation de la migration et décrivons, pour les autres expérimentations, la différence entre leur mise en œuvre et celle qui concerne la migration. Le code de ces expérimentations peut être trouvé dans [3].

### 5.1.1 Migration de flot de contrôle Java

Par migration, nous entendons le déplacement de l'exécution d'un flot d'une machine source vers une machine destination. Ce déplacement doit se faire de telle sorte que l'exécution du flot sur la machine destination reprenne au point où elle a été interrompue sur la machine source.

La migration d'un flot Java doit donc interrompre l'exécution du flot sur la machine source et extraire son contexte courant. Ce contexte est ensuite transféré vers la machine destination où il est intégré à un nouveau flot Java. Finalement, l'exécution de ce nouveau flot est lancée sur la machine destination et le flot initial sur la machine source est détruit.

Pour expérimenter la migration d'un flot Java, nous avons tout d'abord écrit une classe Java que nous appelons *MigratingThread*. Cette classe est une sous-classe de la classe *MobileThread* et elle fournit une implantation de la méthode abstraite *transferExecEnv*. La méthode *transferExecEnv* possède un premier paramètre représentant un contexte d'exécution de flot et un second paramètre qui est un tableau contenant l'adresse IP et le numéro de port de la machine destination. Dans le cas de la migration, le rôle de cette méthode est de transférer le contexte d'exécution, passé en paramètre, vers la machine destination dont l'identification est également passée en paramètre. Ce transfert est réalisé grâce à l'opération de sérialisation de Java.

Nous avons ensuite lancé deux Machines Virtuelles Java : une machine source et une machine destination. Sur la machine source, nous avons créé un flot, de la classe *MigratingThread*, qui exécute un programme Java quelconque. Au cours de son exécution, ce flot appelle la méthode *extractExecEnv* avec le paramètre *toStop* à vrai et le paramètre args qui est un tableau contenant l'adresse IP et le numéro de port de la machine destination. Cet appel provoque la capture du contexte d'exécution courant du flot, le transfert de ce contexte vers la machine destination et l'arrêt définitif de ce flot. La machine destination

reçoit ensuite un contexte d'exécution, grâce à l'opération de dé-sérialisation de Java, puis la méthode *integrateExecEnv* est appelée. Cet appel provoque la création d'un nouveau flot de la classe *MigratingThread*, ce nouveau flot poursuit l'exécution du premier au point où elle a été interrompue.

### 5.1.2 Clonage de flot de contrôle Java à distance

Le clonage d'un flot est la création d'un nouveau flot (appelé clone) qui a le même état d'exécution que le premier ; ces deux flots poursuivent ensuite leur exécution séparément. Le clonage d'un flot à distance est la création d'un flot clone mais sur une machine distante. Un des intérêts du clonage à distance est qu'il peut servir à dupliquer l'exécution d'une application sur plusieurs machines pour permettre à cette application de tolérer les fautes [10].

Si, pour un flot qui s'exécute sur une machine source, nous voulons créer un clone sur une machine destination, ceci revient à effectuer une migration du flot de la machine source vers la machine destination avec la seule différence que l'exécution du flot, sur la machine source, est poursuivie au lieu d'être définitivement arrêtée.

La mise en œuvre du clonage de flot à distance en utilisant notre mécanisme est donc exactement la même que celle de la migration de flot, avec la seule différence que la méthode *extractExecEnv* est appelée avec le paramètre *toStop* à faux.

### 5.1.3 Sauvegarde/Reprise d'un flot de contrôle Java

L'expérimentation présentée ici illustre la sauvegarde de l'état d'un flot Java dans un fichier puis sa restauration à partir du contenu du fichier. L'opération de sauvegarde de l'état du flot Java doit donc interrompre l'exécution de ce flot, extraire son contexte courant et le sauver dans un fichier. Le flot peut ensuite poursuivre son exécution. En cas de panne, la restauration de l'état sauvegardé récupère le dernier contexte stocké dans le fichier et l'intègre à un nouveau flot Java dont l'exécution sera lancée ; cette exécution reprendra au point où elle a été interrompue lors de la dernière sauvegarde dans le fichier.

L'expérimentation de la sauvegarde/reprise d'un flot Java et similaire à celle de la migration. La seule différence entre ces deux mises en œuvre réside dans le fait que, pour la sauvegarde/reprise, le contexte capturé du flot est stocké dans un fichier au lieu d'être transféré vers une autre machine à travers le réseau. Pour cela, nous avons écrit une méthode *extractExecEnv* dont le rôle est de sauvegarder un contexte d'exécution dans le fichier dont le nom est passé en paramètre de la méthode. Le transfert du contexte d'exécution vers le fichier est réalisé grâce à l'opération de sérialisation de Java.

### 5.1.4 Adaptation de la migration d'un flot de contrôle Java

Cette dernière expérimentation sert à illustrer la possibilité d'adapter notre mécanisme de capture/restauration du contexte d'exécution d'un flot aux besoins de l'application qui l'utilise. En effet, notre mécanisme permet de capturer le contexte d'exécution global d'un flot Java ; ce contexte est ensuite entièrement transféré (vers une machine distante ou vers un fichier). Par défaut, tous les objets Java utilisés par le flot sont transférés. Mais le transfert de certains objets utilisés par le flot peut être dénué de sens. Ainsi, le transfert d'un objet représentant un canal de communication ou de tout objet fortement dépendant de la machine sur laquelle s'exécute le flot peut conduire à des incohérences.

Une solution à ce problème est d'adapter notre mécanisme aux besoins de l'application qui l'utilise, en lui permettant de spécifier, pour certains types d'objets Java, des traite-

ments supplémentaires à effectuer lors de la capture et de la restauration du contexte d'exécution d'un flot.

Pour illustrer ce principe, nous avons implanté une application de type Client-Serveur, composée de plusieurs sites clients et d'un site serveur connu par tous. Nous avons lancé une Machine Virtuelle Java sur chacun de ces sites. Un site client peut supporter l'exécution de plusieurs flots Java, ces flots étant des objets de la classe *MigratingThread* que nous avons écrite (voir 5.1.1). Un flot, sur le site client, communique avec le serveur via un canal de communication, un canal étant représenté par un objet Java d'une classe nommée *MyConnector*. Que se passe-t-il lorsqu'un flot, sur un site client source, se déplace vers un site client destination ? Et, plus précisément, comment est traité, lors de la migration, l'objet *MyConnector* qui reliait le flot sur le site client source au serveur ?

L'adaptation que nous avons mise en œuvre se fait en deux étapes. La première étape se déroule au moment du transfert du contexte d'exécution du flot où, pour chaque objet *MyConnector* utilisé par le flot, un message de déconnexion est envoyé au serveur et le canal fermé. La seconde étape se fait au moment de la restauration du contexte (sur le site destination) où, pour chaque objet *MyConnector* utilisé par l'ancien flot, un nouvel objet *MyConnector* est créé puis connecté au serveur ; ainsi, la communication entre le serveur et le flot transféré vers le site client destination est rétablie. Ces deux étapes ont été réalisées, respectivement, lors des opérations de sérialisation et de dé-sérialisation de Java. En effet, nous avons surchargé les méthodes Java *writeObject* (sérialisation) et *readObject* (dé-sérialisation) de la classe *MyConnector* pour qu'elles effectuent les traitements correspondant aux deux étapes présentées précédemment.

## 5.2 Évaluation

Nous présentons ici le résultat des mesures préliminaires que nous avons effectuées sur notre prototype. Ces premières mesures visent à évaluer, d'une part, le coût de notre mécanisme et, d'autre part, le surcoût induit par notre mécanisme sur les applications qui ne l'utilisent pas.

### 5.2.1 Coût de notre mécanisme

Nous considérons la migration d'un flot de contrôle entre deux machines virtuelles Java et nous nous proposons de mesurer le coût de la migration proprement dite. Le travail effectué lors de la migration dépend du nombre, de la taille, et de la nature du contenu des régions stockées dans la pile Java du flot. Nous ne nous intéressons dans la suite qu'à l'influence du nombre de régions (*frames*.

Nous voulons donc quantifier la variation du coût de la migration du flot en fonction du nombre de régions stockées dans la pile Java de ce flot. Nous souhaitons aussi comparer les résultats obtenus soit en utilisant notre mécanisme de capture/restauration du contexte d'exécution d'un flot, soit en utilisant le mécanisme fourni par WASP [6].

Pour atteindre ce but, nous avons écrit un programme Java qui calcule récursivement la factorielle d'un entier $n$ et migre quand les $n$ contextes intervenant dans le calcul sont empilés. Nous avons produit deux versions de ce programme, l'une qui utilise notre mécanisme et l'autre qui utilise le mécanisme fourni par WASP. Pour ce qui est du transfert du contexte d'exécution du flot, nous avons utilisé le mécanisme de sérialisation/dé-sérialisation standard de Java, pour les deux versions du programme. Cette expérimentation a été réalisée entre deux machines virtuelles Java (JDK 1.1.3), installées sur des processeurs Sparc

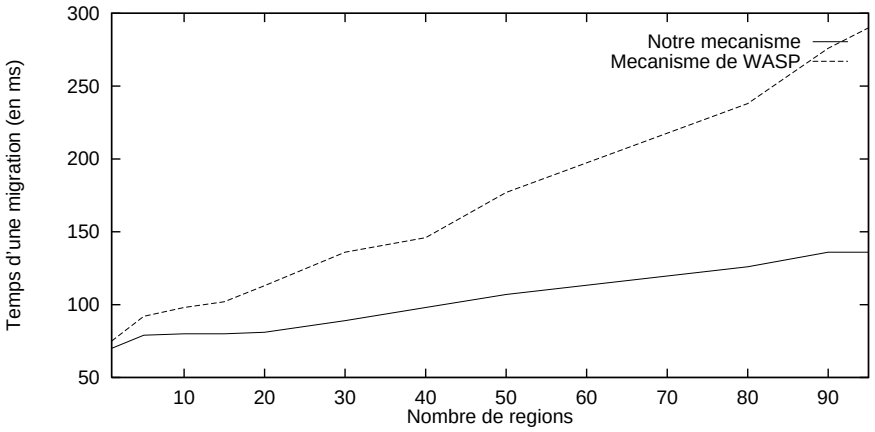


Figure 3: **Coût de la migration**

d'une fréquence de 167 Mhz et sur le système Solaris 2.5 ; le réseau reliant ces machines étant de 10Mb/s.

Les figures 3 et 4 donnent respectivement le temps de migration d'un flot et le temps de transfert du contexte d'exécution d'un flot en fonction du nombre de régions sur la pile Java de ce flot.

La figure 3 donne, en trait plein, le temps de migration en utilisant notre mécanisme et, en pointillés, le temps de migration en utilisant le mécanisme de WASP. Alors que les deux courbes sont très voisines au voisinage de l'origine (faible nombre de zone actives sur la pile), la technique d'implantation de la migration par injection (WASP) est de moins en moins efficace au fur et à mesure que le nombre de zones actives empilées croît.

Pour tenter de localiser où se situe cette perte d'efficacité relative entre la méthode utilisée par WASP et la nôtre, nous avons mesuré, d'une part, la taille des contextes transportés lors de la migration et, d'autre part, le temps de transfert de ces contextes. Le résultat de cette mesure montre que notre mécanisme donne lieu à des contextes dont la taille est toujours un peu plus grande que celle des contextes engendrés par WASP. En revanche, le transfert de nos contextes est moins coûteux que celui des contextes de WASP. Le temps de transfert du contexte (temps de sérialisation augmenté du temps transfert réseau et du temps de dé-sérialisation) est illustré par la figure 4. La différence entre les coûts de transfert des contextes ne peut donc être due qu'à la différence de complexité des structures de données synthétisant ces contextes des deux exécutions.

Par ailleurs, des mesures illustrées par les figures 3 et 4, on peut déduire les coûts induits par les traitements propres à la capture et à la restauration des contextes d'exécution. Le coût d'une migration est égal à la somme des coûts de capture, de transfert et de restauration du contexte. Les coûts moyens de capture/restauration reviendraient donc à 7 ms pour notre mécanisme et à 25 ms pour le mécanisme fourni par WASP. Cette différence est due au fait que notre mécanisme a été intégré à la machine virtuelle Java (implantation principalement native, en C) alors que celui de WASP est mis en œuvre au dessus de la machine virtuelle (injection de code Java dans l'application).

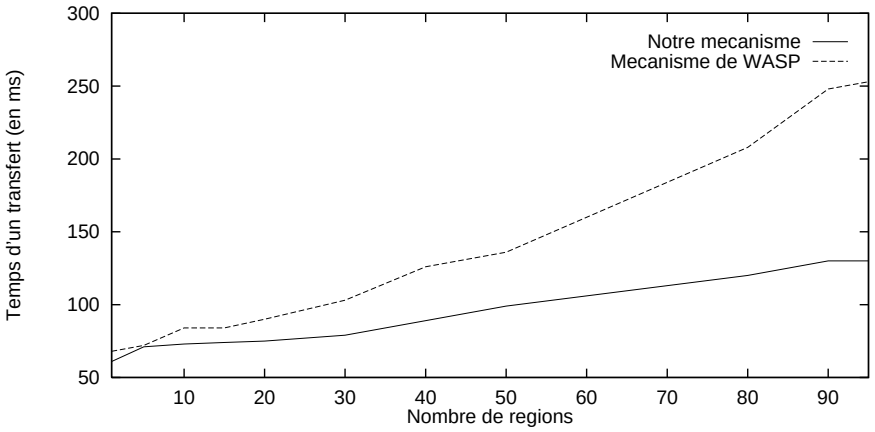


Figure 4: **Coût du transfert du contexte d'exécution**

### 5.2.2 Surcoût induit par notre mécanisme

Nous voulons vérifier ici que notre mécanisme n'induit des coûts supplémentaires que lorsqu'il est utilisé ou susceptible d'être utilisé par une application. Pour le montrer, nous avons mesuré le temps d'exécution d'un même programme exécuté selon différents scénarios :

- le programme est exécuté par un flot de la classe *Thread*, sur une machine virtuelle Java standard (JDK 1.1.3).
- le programme est exécuté par un flot de la classe *Thread*, sur la machine virtuelle Java que nous avons étendue (JDK 1.1.3 étendu).
- le programme est exécuté par un flot de notre classe *MobileThread*, sur la machine virtuelle Java que nous avons étendue (JDK 1.1.3 étendu).

Afin de compléter notre comparaison avec WASP, nous avons défini un dernier scénario :

- le programme, traité par WASP est exécuté sur la machine virtuelle Java standard (JDK 1.1.3)

| **Type de JVM** | **Classe du flot** | **Temps d'exécution (en ms)** |
|---|---|---|
| JVM standard | classe *Thread* | 0,18 |
| JVM étendue | classe *Thread* | 0,18 |
| JVM étendue | classe *MobileThread* | 1,85 |
| JVM standard | classe WASP | 25 |

Figure 5: **Coût d'une exécution de la fonction factorielle(100)**

La figure 5 présente les coûts d'exécution de ces divers scénarios, pour un programme calculant la factorielle de 100. En comparant les résultats des deux premiers scénarios, nous constatons qu'un flot de la classe *Thread*, s'exécutant sur la JVM que nous avons étendue, ne subit aucun surcoût ; le temps de calcul, dans ces deux cas, est de 0,18 ms. En revanche, dans le troisième scénario, le flot de la classe *MobileThread* a un temps d'exécution

de 1,85 ms. Cette exécution, qui peut utiliser notre mécanisme, dure environ 10 fois plus longtemps que l'exécution «normale». Ce surcoût résulte des traitements supplémentaires effectués par notre machine relativement au contexte d'exécution des flots de la classe *MobileThread*. On pourrait considérer comme excessif le coût rajouté par notre mécanisme pour simplement suivre l'évolution du contexte d'exécution du flot. Cependant, ce coût est à comparer avec les 25 ms, mesurées pour le dernier scénario. Ce dernier surcoût, dû au code Java rajouté, multiplie par environ 130 le temps d'exécution de base de la factorielle. Il est clair que notre implantation pourrait être beaucoup plus efficace si le mécanisme avait été réellement intégré à la JVM par ses concepteurs.

## 6 Conclusion et perspectives

Alors que la machine virtuelle Java fournit la plupart des mécanismes de base permettant le déplacement des objets, elle ne permet pas, dans sa version actuelle, le déplacement d'un flot d'exécution (*thread*). En effet, la pile d'un flot d'exécution n'est pas accessibles au programme Java.

L'objectif de notre travail était de remédier à cette déficience en ajoutant un mécanisme permettant de capturer et de restaurer l'état d'un flot de contrôle. Deux approches permettent de construire un tel mécanisme. La première, utilisée par WASP [6], consiste à opérer entièrement au niveau utilisateur en ajoutant, au moyen d'un pré-processeur, du code et des données à l'application exécutée. La seconde consiste à étendre la machine virtuelle Java pour qu'un programme ait accès à la pile du flot de contrôle qui l'exécute. C'est cette dernière voie que nous avons choisie. D'une part, parce qu'elle permet de fournir directement le mécanisme à d'autres applications que la construction d'agents mobiles et, d'autre part, parce qu'elle permet une implantation plus efficace.

Le prototype a été réalisé par extension du JDK 1.1.3. Afin d'évaluer ce prototype d'un point de vue fonctionnel, nous l'avons utilisé à la fois pour le support de la migration forte dans une plate-forme à agents mobiles et pour mettre en œuvre une gestion de points de reprise. Les premières mesures effectuées sur le prototype montrent que le JDK 1.1.3. que nous avons étendu n'induit aucun coût supplémentaire sur les applications qui n'utilisent pas notre mécanisme. Elle montre également que le coût de ce mécanisme est raisonnable, surtout si on le compare au coût de celui fourni par WASP. Les performances du mécanisme proposé pourraient être meilleures si celui-ci était réellement intégré à la machine virtuelle par ses constructeurs.

Ce travail se poursuit actuellement. Des mesures complémentaires sont en cours, le but étant, tout d'abord, d'approfondir et d'élargir à d'autres implantations la comparaison des mises en œuvre de ce mécanisme. Les perspectives à plus long terme concernent des aspects non encore traités du problème. Nous n'avons pas abordé les problèmes liés à la protection des piles que nous rendons accessibles aux programmes Java. Il est important de ne pas introduire de faille dans le système de protection de Java. Enfin, ce mécanisme devrait constituer, avec le mécanisme d'objets répartis fourni par Javanaise [8], un premier pas vers une machine Java répartie.

## Bibliographie

1. J. Baumann, F.Hohl, M. Straber et K. Rothermel. *Mole - Concepts of Mobile Agent*

*System*. WWW Journal, Special issue on Applications and Techniques of Web Agents, 1998.
2. S. Bouchenak-Khelladi. *Mécanismes pour la Migration de Processus - Extension de la Machine Virtuelle Java*. Rapport de Magistère, Université Joseph Fourier, Grenoble, France, 1998.
3. S. Bouchenak-Khelladi. *Extended Java Platform 1.1.3 Core API*. URL : http://sirac.inrialpes.fr/ bouchena/Recherche/DEA/MyAPI/packages.html
4. D. Chess, C. Harrison et A. Kershenbaum. *Mobile Agents: Are They a Good Idea ?* T.J. Watson Research Center, IBM Research Division, mars 1995. URL : http://www.research.ibm.com/iagents
5. F. Douglis et B. Marsh. *The Workstation as a Waysttion : Integrating Mobility into Computing Environment*. The Third Workshop on Workstation Operating System (IEEE), avril 1992.
6. S. Fünfrocken. *Transparent Migration of Java-based Mobile Agents(Capturing and Reestablishing the State of Java Programs)*. Proceedings of Second International Workshop Mobile Agents 98 (MA'98), Stuttgart, Allemagne, septembre 1989.
7. General Magic. *Odyssey Web Site*. URL : http://www.genmagic.com/agents
8. D. Hagimont et D. Louvegnies. *Javanaise : Distributed Shared Objects for Internet Cooperative Applications*. Middleware'98, Angleterre, septembre 1998.
9. IBM Tokyo Research Labs. *Aglets Workbench : Programming Mobile Agents in Java*, 1996. URL : http://www.trl.ibm.co.jp/aglets
10. J. Kim, H. Lee et S. Lee. *Replicated Process Allocation for Load Distributed in Fault-Tolerant Multicomputers*. IEEE Transactions on Computers, pages 499-505, avril 1997.
11. T. Lindholm et F. Yellin. *Java Virtual Machine Specification*. Addison Wesley, 1996.
12. D. S. Milojicic, F. Douglis, Y. Paindaveine, R. Wheeler et S. Zhou. *Process Migration Survey*. The Open Group research Institute, Rapport Technique, volume 5, mars 1997. URL : http://www.camb.opengroup.org/RI/java/moa
13. D. A. Nichols. *Using Idle Workstations in a Shared Computing Environment*. Proceedings of the Eleventh ACM Symposium on Operating Systems Principles, novembre 1987.
14. M. Nuttall. *A Brief Survey of Systems Providing Process or Object Migration Facilities*. Operating Systems Review, octobre 1994.
15. I. Oueichek. *Conception et réalisation d'un noyau d'administration pour un système réparti à objets persistants*. Thèse de Doctorat, Institut National Polythechnique de Grenoble, France, octobre 1996.
16. T. Osman et A. Bargiela. *Process Checkpointing in an Open Distributed Environment*. Proceedings of European Simulation Multiconference, ESM'97, juin 1997.
17. M. Ranganathan, A. Acharya, S. D. Sharma et J. Saltz. *Network-aware Mobile Programs*. Proceedings of the USENIX Annual Technical Conference, Anaheim, Californie, 1997.
18. Sun Microsystems. *JDK 1.1 Documentation*, Sun Microsystems. URL : http://java.sun.com/products/jdk/1.1/docs
19. Sun Microsystems. *Object Serialization*, Sun Microsystems. URL : http://java.sun.com/products/jdk/1.2/docs/guide/serialization