Mobilité et persistance des applications dans l'environnement Java

Sara Bouchenak

No attribué par la bibliothèque

THÈSE

Présentée par
Sara BOUCHENAK

Pour obtenir le titre de
DOCTEUR DE L'INSTITUT NATIONAL POLYTECHNIQUE DE GRENOBLE

Discipline : **Informatique**

Mobilité et Persistance des Applications dans l'Environnement Java

Directeur de thèse : Daniel Hagimont

Date de soutenance : 19 Octobre 2001

Jury :

Président :	Jacques Mossière
Rapporteurs :	Bertil Folliot Rachid Guerraoui
Examinateurs :	Roland Balter Andrzej Duda

A mes parents

Remerciements

Je remercie, tout d'abord, Daniel Hagimont, directeur de cette thèse, pour m'avoir donné l'opportunité de mener à bien ces travaux de recherche. J'ai particulièrement apprécié, chez Daniel, sa disponibilité, son ouverture d'esprit et les nombreuses discussions qui ont animé ces trois années de thèse.

Je tiens également à remercier les membres du jury de thèse :

Jacques Mossière, Professeur à l'Institut National Polytechnique de Grenoble, de m'avoir fait l'honneur de présider le jury.

Bertil Folliot, Professeur à l'Université Paris VI, et Rachid Guerraoui, Professeur à l'Ecole Polytechnique Fédérale de Lausanne, d'avoir accepté de juger ce travail.

Roland Balter, Directeur du projet Sirac et Professeur à l'Université Joseph Fourier, de m'avoir accueillie au sein de son projet de recherche et Andrzej Duda, Professeur à l'Institut National Polytechnique de Grenoble, d'avoir accepté de faire partie du jury.

Je remercie également Timothy Lindholm, un des pères fondateurs de la machine virtuelle Java, pour ses nombreux éclaircissements quant au fonctionnement de la machine virtuelle.

Je souhaiterais remercier les lecteurs des premières versions de ce mémoire ainsi que les lecteurs des diverses publications de ces travaux de thèse dans des conférences scientifiques, je pense aux Professeurs Xavier Rousset de Pina, Sacha Krakowiak et Jacques Mossière. Xavier a été un des initiateurs de ce sujet de thèse. J'ai eu, par la suite, la chance de côtoyer Sacha et Jacques auprès desquels j'ai beaucoup appris quant à la rigueur scientifique et à la générosité.

Je souhaiterais remercier mes collègues du projet Sirac pour leurs encouragements et leur aide : Vanouchka, Nono, Manu, Sébichouk, Eric, Fabienne, Valérie et les autres. Je remercie également les personnes que j'ai connues à l'INRIA pour la bonne humeur qui a régné dans cet agréable cadre de travail.

Je remercie finalement les miens, parents et amis, pour leur soutien, leurs encouragements et leur présence même de loin.

Et puis je remercie tous ceux que je n'aurais bien sûr pas dû oublier …

TABLE DES MATIERES

CHAPITRE 1 - MOBILITÉ ET PERSISTANCE DES APPLICATIONS.....9

CHAPITRE 2 - JAVA : ENVIRONNEMENT ET MACHINE VIRTUELLE.....37

CHAPITRE 3 - MISE EN ŒUVRE DE NOS SERVICES DE MOBILITÉ ET DE PERSISTANCE 67

CHAPITRE 4 - EXPÉRIMENTATIONS 113

INTRODUCTION

1. Introduction

Les dernières années ont été marquées par une forte évolution des équipements utilisés dans les environnements répartis. Nous sommes successivement passés de réseaux locaux de stations de travail à des réseaux de machines à grande échelle (de type Internet), puis à des réseaux sans fil interconnectant des équipements mobiles comme des téléphones portables ou des assistants numériques personnels (PDA). Cette évolution des environnements répartis vers une informatique globale et omniprésente a eu trois conséquences majeures :

- Démocratisation des applications réparties. Rares sont les ordinateurs qui ne sont pas connectés d'une façon ou d'une autre à un réseau, voire à l'Internet. La plupart des applications sont aujourd'hui réparties, même si la répartition se résume parfois à de simples échanges de fichiers.
- Développement du nomadisme. Les usagers qui se déplacent doivent pouvoir utiliser les mêmes outils informatiques, soit à partir d'un équipement mobile (portable, téléphone, PDA), soit en accédant à ces outils depuis des postes d'accueil différents mis à disposition lors des déplacements. Les applications qu'ils utilisent doivent donc être accessibles quel que soit le terminal ou le réseau utilisé pour accéder à l'application.
- Hétérogénéité des équipements. Les équipements utilisés par les applications réparties, que ce soient des terminaux d'accès aux applications ou des infrastructures de communication utilisées par ces terminaux, se sont diversifiés. Les terminaux peuvent être aussi bien des stations de travail que des machines portables ou des PDA. De même, les réseaux utilisés peuvent être des réseaux sans fils de proximité (Bluetooth), des réseaux téléphoniques sans fil (UMTS) ou des réseaux filaires locaux ou à grande échelle.

Chacune de ces évolutions se traduit par des besoins pressants au niveau des systèmes d'exploitation :

- Besoins issus des applications réparties. Leslie Lamport définit un système réparti comme suit : « *A distributed system is one in which the failure of a computer you didn't even know existed can render your own computer unusable* » [82]. Du fait qu'elles dépendent de plusieurs nœuds d'un réseau, les applications réparties sont plus sensibles aux pannes. Une prévention des pannes nécessite alors des outils particuliers, tels que des services de sauvegarde d'une image persistante des applications, image qui peut ensuite servir à la reprise de l'application.
- Besoins issus du nomadisme. Le nomadisme des usagers et des équipements a fait apparaître dans les systèmes d'exploitation des mécanismes de mobilité des

applications, soit pour reporter la charge de calcul des machines mobiles à faibles capacités vers des serveurs plus performants, soit pour mettre à disposition une application sur un terminal d'accès à partir d'un serveur.

- Besoins issus de l'hétérogénéité. La forte hétérogénéité des environnements matériels a amené un regain d'intérêt pour les techniques à base de machines virtuelles. Des exemples sont les machines virtuelles Tcl, Python ou Java. Ces machines virtuelles offrent l'abstraction d'un monde homogène dans un environnement hétérogène, ce qui permet à une même application de s'exécuter sur des plates-formes de natures variées. En particulier, la machine virtuelle Java s'est imposée grâce à sa large diffusion, des implantations existent sur la plupart des architectures de machines et systèmes d'exploitation [85].

Ces besoins au niveau des systèmes d'exploitation se sont traduits par des recherches intensives autour des techniques de persistance et de mobilité dans des environnements répartis. Bon nombre de ces recherches ont été effectuées sous la forme d'intergiciels (middleware), souvent sur la machine virtuelle Java, car elle résout les problèmes d'hétérogénéité en définissant un format de code et de données portables. De plus, Java répond en partie aux besoins de mobilité et de persistance en fournissant des services de mobilité du code Java (chargement dynamique de classe, applet) [132] et des services de persistance des données (sérialisation des objets) [136].

Cependant Java n'apporte pas de réponse complète aux besoins de mobilité et de persistance des applications : une application Java – avec son code, ses données et l'état de son exécution – ne peut pas être rendue mobile ni persistante. La prise en compte du « contexte d'exécution »/« état d'exécution » des applications Java pour des besoins de mobilité et de persistance fait l'objet de cette thèse.

2. Objectifs

L'étude de la mobilité et de la persistance des applications n'est pas un problème nouveau [100] [93] [12] [39] [44]. Mais l'application de ce problème à l'environnement Java est nouvelle. Les objectifs de nos travaux sont de proposer des services système Java qui permettent de migrer une application, de cloner une application localement ou à distance, de sauvegarder une image persistante de l'application et de reprendre l'exécution d'une application à partir de son image persistante. Ceci se traduit concrètement par des services de mobilité et de persistance des threads Java, basés sur des mécanismes de plus bas niveau, des mécanismes de capture et de restauration de l'état d'exécution des threads Java.

Nos travaux ont été menés en suivant deux axes principaux : respecter l'abstraction d'homogénéité fournie par Java et respecter les performances des applications Java :

- Java garantit la portabilité du code et des données. Nos services doivent, de plus, garantir la portabilité de l'exécution des applications. Ainsi, une application qui commence son exécution sur un site source peut se déplacer vers un site destination et y poursuivre son exécution, indépendamment de la plate-forme sous-jacente au site destination. De la même façon, une application qui est sauvegardée sur une plate-forme peut, par la suite, être reprise sur un autre type de plate-forme.
- De plus, les services proposés doivent présenter des performances raisonnables. En effet, l'utilisation limitée de la mobilité et de la persistance des applications est, en général, due au coût élevé de ces opérations. Ajouté à cela, l'environnement Java s'est longtemps vu reprocher la faiblesse de ses performances. A cet effet, de nombreux efforts ont été faits par les concepteurs de Java pour améliorer le temps d'exécution d'un code Java et le rapprocher du temps nécessaire à une exécution native ; ces efforts se traduisent, par exemple, par des compilateurs Java à la volée (JIT). La proposition de nouveaux services système Java ne doit donc pas annuler les efforts faits en matière de performances à l'exécution.

Les principes relatifs à la portabilité et au respect des performances des applications ont été suivis comme ceci :

- La principale difficulté concernant la portabilité de l'état d'exécution des application Java est que la représentation de la pile d'exécution des threads dans la machine virtuelle Java est non portable. Nous proposons à cet effet deux techniques de transformation de la pile :
  - une technique basée sur un interprète java étendu [22] [23]
  - et une technique basé sur une analyse dynamique de flot [24] [25].
- Concernant le respect des performances des applications Java, nous proposons des services de capture d'état qui sont utilisables même en présence de compilation Java à la volée. Ceci n'est possible qu'avec l'utilisation de techniques de dés-optimisation à la volée du code Java compilé. Ce sont de telles techniques que nous utilisons dans nos services.

L'étude, la conception et les mises en œuvre effectuées durant nos travaux ont abouti à la réalisation de services complets et opérationnels pour la mobilité et la persistance des applications Java. Le résultat de notre contribution se présente sous la forme d'extensions de la machine virtuelle Java, à travers de nouveaux sous-systèmes et de nouvelles structures de données pour la machine virtuelle.

Ces services ont été validés via des applications de démonstration et via la construction d'un système de sauvegarde/reprise de calculs parallèles sur la plate-forme de metacomputing SUMA.

L'évaluation comparative de nos services avec d'autres systèmes Java existants montre que nous proposons l'unique approche qui est à la fois complète et qui n'induit aucun surcoût sur les performances des applications Java utilisant les services de mobilité ou de persistance.

3. Cadre de travail

Ces travaux de recherche se sont effectués au sein du projet Sirac (Systèmes Informatiques Répartis pour Applications Coopératives), un projet commun à l'INRIA, l'INPG et l'UJF. Un des domaines de recherche menés dans Sirac est la construction d'applications réparties adaptables. L'activité principale de ce domaine porte sur la construction d'applications à base de composants logiciels, avec des capacités d'adaptation et de reconfiguration dynamiques pour répondre à l'évolution des besoins des applications et des conditions d'utilisation. Les recherches actuelles, menées dans ce domaine, sont organisées selon les axes suivants :

- *Extensibilité*. L'étude de l'extensibilité et de l'adaptabilité des applications est faite à travers une plate-forme de composants logiciels *JavaPod*.
- *Reconfiguration*. Des algorithmes et des outils de reconfiguration dynamique d'applications distribuées sont proposés sur un bus logiciel *AAA*.
- *Duplication*. Des techniques de duplication des données et de gestion de la cohérence sont étudiées dans le système *Javanaise*.
- *Mobilité*. L'étude et l'évaluation du paradigme de code mobile à travers la plate-forme à agents mobiles *Mobilet*.

Nos travaux ont permis d'introduire une nouvelle thématique : la mobilité et la persistance des applications. Ces travaux sont potentiellement réutilisables dans toutes les plates-formes citées précédemment.

4. Plan du document

Ce document est organisé en trois parties principales : la première partie est un positionnement du problème de la mobilité et de la persistance en général, elle inclut les chapitres 1 et 2. La seconde partie est une présentation de nos services de mobilité et de persistance des applications Java, elle correspond au chapitre 3. La troisième partie présente la validation et l'évaluation de nos services, elle est constituée des chapitres 4 et 5.

Chapitre 1

Ce chapitre présente les motivations de la mobilité et de la persistance des applications avant de décrire les travaux effectués dans ces domaines. Pour la

présentation de ces travaux, nous proposons huit critères de classification, qui sont le degré de mobilité et de persistance, le niveau de mise en œuvre, l'hétérogénéité, le mode d'initiation, la granularité, l'adaptabilité, les performances et la transparence.

Ces critères de classification sont ensuite utilisés pour présenter les hypothèses de base de nos services de mobilité et de persistance et les choix de conception faits pour vérifier ces hypothèses.

Chapitre 2

Un des choix de conception est le choix de Java comme environnement de nos travaux. Ce chapitre est constitué de deux grandes parties.

La première partie est un rappel des principaux concepts de l'environnement Java : le langage de programmation orienté-objet, l'abstraction d'un environnement homogène et l'environnement de programmation qui intègre divers outils et services pour la programmation des applications. Cette partie est recommandé au lecteur novice de l'environnement Java mais le lecteur familier de Java peut directement se référer à la partie suivante.

La seconde partie est une présentation des principales caractéristiques de la machine virtuelle Java, caractéristiques définies par la spécification de la machine virtuelle. Cette partie est nécessaire à la compréhension des concepts et détails de mise en œuvre présentés dans les chapitres suivants.

Chapitre 3

Pour la présentation des détails de conception et de mise en œuvre de nos services, nous avons choisi de procéder de façon ascendante : en partant des services de bas niveau (capture, restauration) vers des services de plus haut niveau (mobilité, persistance). Ce chapitre commence ainsi par décrire les détails de conception et de mise en œuvre de nos mécanismes de capture et de restauration de l'état d'exécution des applications Java. Il présente ensuite une description de la construction de nos services de mobilité et de persistance au-dessus des mécanismes de capture et de restauration.

Chapitre 4

Nos services étant opérationnels, ce chapitre décrit leur utilisation à travers des exemples d'utilisation et des expérimentations. Il présente, tout d'abord, des exemples de programmes mettant en œuvre la mobilité tels que la migration d'application ou le clonage d'application à distance et des exemples mettant en œuvre la persistance tels que la sauvegarde puis la reprise d'application.

Ce chapitre donne ensuite des exemples d'expérimentations. Une première expérimentation a été élaborée à des fins de démonstration de nos services de mobilité.

Elle met en œuvre le calcul et la visualisation d'une courbe fractale qui se déplace vers d'autres sites pour y poursuivre son calcul et sa visualisation. La seconde expérimentation est basée sur nos services de persistance des applications. Elle met en œuvre un système de sauvegarde/reprise de calculs parallèles, sur une plate-forme de metacomputing appelée SUMA.

Après avoir rappelé les concepts de Java dans le chapitre précédent, nous pouvons introduire nos services de mobilité et de persistance des applications Java. Ce chapitre décrit l'interface de nos services et illustre l'utilisation des services de mobilité à travers des exemples de migration d'application ou de clonage d'application à distance et décrit l'utilisation des services de persistance à travers des exemples de sauvegarde puis de reprise d'application.

Chapitre 5

Ce chapitre présente une évaluation des services réalisés. Notre évaluation est de deux natures : une évaluation qualitative et une évaluation quantitative. L'évaluation qualitative se présente sous la forme d'une comparaison des fonctionnalités de nos services avec d'autres systèmes existants et récemment proposés. L'évaluation quantitative présente, quant à elle, les résultats des mesures de performances de nos services et une comparaison de ces performances avec celles proposées par les services de systèmes similaires.

Conclusion

La conclusion résume l'apport essentiel de ces travaux. Elle ouvre également de nouveaux éléments de réflexion et quelques perspectives de recherche.

CHAPITRE 1 - MOBILITE ET PERSISTANCE DES APPLICATIONS

Ce chapitre donne quelques définitions élémentaires de la mobilité et de la persistance des applications, avant de motiver leur utilisation et de présenter les travaux effectués dans ces domaines.

Ce chapitre ne se veut pas une étude de tous les travaux, marquants ou plus discrets, effectués dans le domaine de la mobilité et de la persistance mais une sélection de quelques travaux représentatifs des critères de classification proposés. Ces critères sont par la suite utilisés pour définir nos hypothèses et choix de conception.

1. Définitions

Les termes « mobilité » et « persistance » sont employés dans différents domaines et peuvent ainsi avoir plusieurs significations. Cette section présente notre définition de la mobilité et de la persistance des applications telle qu'elle sera employée dans la suite de ce document.

1.1. Mobilité des applications

L'*informatique mobile* peut indifféremment désigner la mobilité matérielle ou la mobilité logicielle. La mobilité matérielle est le déplacement d'un terminal physique, tel qu'un ordinateur portable, un téléphone ou un assistant digital personnel (PDA). Alors que la mobilité logicielle est le déplacement d'un programme logiciel entre deux terminaux physiques. L'entité logicielle mobile est alors appelée composant, agent, calcul ou application mobile.

Dans la suite, nous nous intéressons plus particulièrement à la mobilité logicielle que nous désignons par mobilité des applications. Plus de détails sur la mobilité matérielle peuvent être trouvés dans [62] [15] [54].

1.2. Persistance des applications

La persistance d'une application est la survie d'une image de cette application après la terminaison – normale ou anormale – de l'application. Cette image de l'application peut ensuite servir à reprendre l'exécution de l'application. La persistance d'une application repose ainsi sur deux opérations : la sauvegarde de l'application et la reprise de l'application.

La *sauvegarde d'une application* est l'opération de stockage d'une image de cette application sur un support non volatile tel que le disque. Et la *reprise de l'application* est la restitution, à partir d'une image persistante, de l'exécution de l'application. L'application reprise commence son exécution au point où elle a été interrompue au moment de la sauvegarde.

La persistance des applications repose, d'une part, sur les mécanismes de construction d'une image de l'application et, d'autre part, sur les mécanismes de stockage de cette image sur un support persistant, tels que les fichiers ou les bases de données. Dans la suite nous nous intéressons plus particulièrement au premier point et invitons le lecteur intéressé par les mécanismes de stockage (bases de données, transactions, sécurité et accès concurrent) à se référer à [117] [9].

1.3. Pourquoi parler de mobilité ET de persistance ?

Certains travaux de recherche ont noté que la mobilité et la persistance étaient conceptuellement similaires [37]. En effet, la persistance d'une application n'est qu'une mobilité dans laquelle la destination de l'application est représentée par un support persistant. Et inversement, la mobilité d'une application n'est autre qu'une persistance dans laquelle le support persistant est remplacé par un nœud du réseau. Ce chapitre abordera donc conjointement les deux thèmes de mobilité et de persistance des applications.

D'autres travaux de recherche se sont intéressés à la combinaison de la mobilité et de la persistance. Le problème ici consiste à rendre une application mobile dans un contexte persistant ou à fournir la persistance à une application mobile. Nous n'aborderons pas ce problème de combinaison de la mobilité et de la persistance, dont les détails sont décrits par Mira da Silva et Atkinson dans [94].

2. Motivations de la mobilité et de la persistance des applications

La mobilité des applications a plusieurs domaines d'utilisation, parmi lesquels nous citons la répartition dynamique de charge, la diminution du trafic réseau, la reconfiguration dynamique d'applications distribuées, l'administration du système ou le nomadisme de l'utilisateur. Et la persistance des applications peut avoir plusieurs utilisations, telles que la tolérance aux pannes ou le débogage des applications.

- ***Répartition dynamique de charge***

Des applications qui s'exécutent dans un système distribué peuvent se déplacer vers de nouveaux sites et profiter ainsi des ressources physiques des sites d'accueil, telles que la mémoire ou le processeur. Un déplacement judicieux des applications permet d'équilibrer dynamiquement la charge à travers le système distribué [99].

- ***Diminution du trafic réseau***

Le déplacement d'une application cliente vers le serveur qui héberge les informations (données ou services) auxquelles l'application accède fréquemment permet de profiter de la localité des informations et de diminuer le trafic réseau [43]. C'est sur ce principe que sont basées la plupart des plates-formes à agents mobiles [34].

- ***Reconfiguration dynamique d'applications***

La reconfiguration dynamique d'une application distribuée peut se présenter sous différents aspects : la modification de l'architecture de l'application par ajout ou suppression de certains de ses composants logiciels, la modification de la mise en œuvre

des composants ou le changement de la distribution géographique de l'application [40] [64]. La modification de la distribution géographique de l'application consiste à déplacer des composants logiciels de l'application vers de nouveaux sites.

- ***Administration du système***

L'administration de systèmes distribués nécessite parfois la ré-initialisation de machines sans pour autant vouloir interrompre certains services (applications) en cours d'exécution sur ces machines. Ces services peuvent alors être transférés vers d'autres machines [103].

- ***Nomadisme de l'utilisateur***

Un utilisateur nomade est un utilisateur qui se déplace géographiquement en transportant avec lui son dispositif matériel, tel qu'un téléphone portable ou un assistant personnel. Un utilisateur nomade est également un utilisateur qui se déplace d'un dispositif matériel à un autre, dans quel cas, l'utilisateur peut vouloir transporter avec lui les applications qui constituent son environnement personnel [16].

- ***Tolérance aux pannes***

La tolérance aux pannes est la motivation majeure de la persistance des applications. Une panne qui survient sur la machine sur laquelle s'exécute une application provoque la terminaison prématurée de l'application. Mais si des sauvegardes régulières de l'application ont été effectuées et stockées sur disque, l'application peut être reprise à partir de sa dernière sauvegarde [68].

- ***Débogage des applications***

La persistance des applications peut également servir au débogage des applications [104]. Certains *debuggers* permettent de suivre, pas à pas, l'exécution de l'application, du début jusqu'à l'occurrence de l'erreur. D'autres debuggers permettent d'examiner l'état dans lequel se trouve l'application lorsque celle-ci s'arrête prématurément ; cet examen se fait à travers l'analyse d'un fichier *core* généré automatiquement lors de l'arrêt de l'application.

Si une application est sauvegardée régulièrement et qu'une erreur provoque l'arrêt soudain de l'application, les différentes sauvegardes effectuées peuvent être utilisées pour relancer l'application à divers points de départ. Le système IGOR suit cette approche pour la mise en œuvre de son mécanisme de débogage des applications [47].

3. Mobilité et Persistance : Politique ou mécanisme ?

Le problème de la mobilité et de la persistance des applications peut être abordé selon deux aspects : la *politique* ou le *mécanisme* de mobilité et de persistance.

La *politique* de la mobilité et de la persistance s'intéresse aux questions relatives aux choix :

- de l'entité mobile,
- du moment de la mobilité,
- de la destination de l'entité mobile,
- de la fréquence des sauvegardes de l'entité persistante,
- du système de gestion des informations persistantes (SGF, SGBD)
- et du site de reprise de l'entité persistante.

Les concepteurs des systèmes MOSIX [19] et GatoStar [48] se sont intéressés à la politique de migration de processus.

Quant au *mécanisme* de mobilité/persistance, il aborde les techniques de mise en œuvre de la mobilité et de la persistance, autrement dit :

- les informations rendues mobiles/persistantes,
- les contraintes faites sur la plate-forme sur laquelle se déplace une application mobile ou sur laquelle est sauvegardée puis reprise une application persistante,
- les coûts induits par le déplacement d'une application mobile ou par la sauvegarde/reprise d'une application persistante,
- l'adaptabilité de la mobilité et de la persistance aux besoins des applications qui les utilisent
- et la transparence de la mobilité/persistance.

Nos travaux ont porté sur l'étude, la conception, la mise en œuvre et l'expérimentation de mécanismes de mobilité et de mécanismes de persistance des applications. Nous présentons, dans la suite, les principaux mécanismes réalisés dans ces domaines. Nous n'aborderons pas les questions relatives à la politique de mobilité/persistance, questions qui sont étudiées par les concepteurs des systèmes MOSIX [19] et GatoStar [48].

4. Travaux relatifs à la mobilité et à la persistance des applications

Différentes recherches ont été menées dans le domaine de la mobilité et de la persistance des applications et différents systèmes expérimentaux ont été réalisés. Des synthèses décrivant les principales techniques développées dans le domaine de la mobilité sont proposées par Nuttall [100] et par Milojicic et al. [92] et des synthèses des travaux relatifs à la persistance sont proposées par Atkinson et al. [10], par Dearle et al. [39] et par Elnozahy et al. [44]. Les travaux menés dans ces domaine peuvent être classés selon plusieurs critères. Nous avons choisi de les présenter selon les critères suivants :

- *Degré de mobilité ou de persistance*. La mobilité ou la persistance des applications sont-elles des mobilité/persistance du code, des mobilité/persistance faibles ou fortes ?
- *Niveau de mise en œuvre.* Les services de mobilité/persistance sont-ils intégrés à un système existant ou sont-ils proposés avec un nouveau système ?
- *Hétérogénéité*. Une application mobile peut-elle se déplacer entre des plates-formes de natures différentes ou est-elle assignée à un type de plate-forme ? Une application persistante, sauvegardée sur une plate-forme, peut-elle être reprise sur un autre type de plate-forme ?
- *Mode d'initiation*. L'opération de mobilité ou de persistance est-elle initiée par l'entité mobile/persistante ou peut-elle être initiée par une entité externe ?
- *Granularité*. L'opération de mobilité ou de persistance est-elle applicable à grain fin (sur une ressource : donnée, composant, processus, thread) ou à gros grain (sur l'ensemble des ressources) ?
- *Adaptabilité*. Est-il possible d'adapter les services de mobilité/persistance aux besoins des utilisations qui en sont faites ?
- *Performances*. Les coût induits par la latence de la mobilité/persistance et les surcoûts engendrés par ces services sont-ils acceptables par les applications qui les utilisent ?
- *Transparence*. La prise en compte de la mobilité/persistance affecte-t-elle la programmation de l'application ou le comportement de l'application ?

Dans la suite, nous définissons chacun de ces critères et présentons, au fur et à mesure, les principaux travaux qui y adhèrent.

4.1. Degré de mobilité ou de persistance

Il existe plusieurs degrés de mobilité des applications, tels qu'illustrés par la Figure 1-1 : la *mobilité du code*, la *mobilité faible* et la *mobilité forte*. Ces degrés s'appliquent également à la persistance des applications.

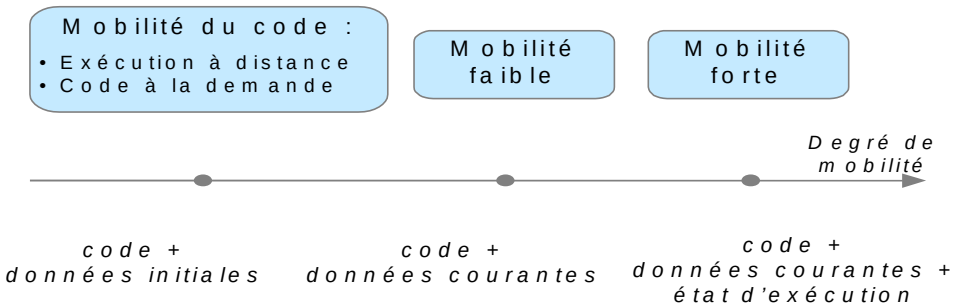

Figure 1-1. Degré de mobilité des applications

4.1.1. Mobilité/Persistance du code

La *mobilité du code* se traduit par le transfert du code d'une application d'un site source vers un site destination puis le lancement de l'exécution de ce code sur le site destination. Le code transféré est parfois accompagné de données initiales nécessaires au lancement de l'application, telles que le profil du client dans le cas d'une application de commerce électronique. D'autre part, et selon le sens qu'on lui donne, la mobilité du code se présente sous la forme :

- *d'exécution à distance*, si c'est le site source qui demande le transfert d'un code vers le site destination pour que le code y soit exécuté
- ou de *code à la demande*, si c'est le site destination qui demande le transfert d'un code se trouvant sur le site source pour l'exécuter localement.

L'*exécution à distance* est similaire au mécanisme d'appel de procédure à distance (RPC) [21], où un client fait appel à un service proposé par un serveur distant. Mais contrairement au RPC, où l'interface des services disponibles sur un serveur est fixée d'avance, l'exécution à distance permet d'envoyer un code à exécuter sur le serveur et d'étendre ainsi dynamiquement les fonctionnalités du serveur. L'exécution à distance est fournie par plusieurs systèmes tels que le modèle d'évaluation à distance proposé par Stamos [125].

Contrairement à l'exécution à distance, avec le *code à la demande*, c'est le site destination qui initie le transfert du code, en amenant un code distant pour l'exécuter localement. Parmi les technologies de code à la demande les plus connues figurent ActiveX de Microsoft [1] ou les Applets Java de Sun Microsystems [132].

Quant à la *persistance du code*, elle est triviale ; c'est le résultat des compilateurs de programmes.

4.1.2. Mobilité/Persistance faible

Contrairement à la mobilité du code qui s'effectue avant le lancement de l'exécution de l'application, la *mobilité faible* intervient au cours de l'exécution d'une application. Elle consiste à transférer l'exécution de l'application d'une machine source vers une machine destination, en passant par :

- Tout d'abord, l'interruption de l'exécution de l'application sur le site source.
- Puis le transfert du code et de l'état courant des données utilisées par l'application du site source vers le site destination.
- Et enfin la reprise de l'exécution de l'application sur le site destination. Arrivée sur le site destination, l'application mobile reprend son exécution depuis le début, tout en possédant les valeurs mises à jour de ses données.

Dans la suite, nous présenterons, tout d'abord, les principaux travaux sur la mobilité faible, travaux que nous classerons en deux catégories :

- les travaux effectués au niveau d'environnement orientés-objet
- et les travaux effectués au niveau de plates-formes à agents mobiles.

Nous présenterons ensuite les travaux relatifs à la persistance faible et distinguerons les travaux effectués :

- au niveau de langages de programmation (extension du langage, extension de la machine virtuelle associée au langage ou extension du compilateur du langage)
- ou au niveau de modèles à composants et de systèmes orientés-objet distribués.

La mobilité faible, dans un environnement orienté-objet[1], se présente sous la forme de mobilité des objets. Plusieurs systèmes fournissant la mobilité d'objets ont été réalisés, parmi lesquels le langage orienté-objet Ellie qui propose un mécanisme de distribution de charge basé sur la mobilité de ses objets [5], le langage orienté-objet et le système distribué Emerald [74], la couche orientée-objet COOL sur le micro-noyau Chorus [4] ou le mécanisme de sérialisation d'objets proposé par le langage et la machine virtuelle Java [114] [136].

De ce fait, la plupart des plates-formes à agents mobiles qui se basent sur des systèmes orientés-objet fournissent une mobilité faible à leurs agents. Les Aglets [69], Ajanta [146], Concordia [150] ou Mole [20] sont de telles plates-formes sur Java. Il est important, par ailleurs, de noter qu'avec des plates-formes à agents "faiblement" mobiles, il revient au programmeur de l'application de spécifier les données qui feront partie de l'état mobile de l'agent : les données dont les valeurs seront transférées lors du déplacement de l'agent. Ceci aura un impact sur la transparence de la mobilité à la programmation de l'application mobile (voir la section 4.8 de ce chapitre).

Quant à la *persistance faible*, elle a été largement étudiée au niveau de langages de programmation [28]. Les mises en œuvre au niveau du langage reposent sur :

- une extension du langage,
- une extension de la machine virtuelle associée au langage
- ou une extension du compilateur du langage.

PS-algol, une extension de S-algol, est un des premiers langages fournissant la persistance des données [8]. Alltalk est une extension de Smalltalk, pour la prise en compte de la persistance des objets [127]. Le langage E, qui est une extension de C++, propose une persistance des objets C++ ; cette persistance est construite sur le système de gestion de bases de données Exodus [112]. Shapiro et al. proposent également une persistance et une mobilité des objets C++ [119]. Plusieurs mises en œuvre de la persistance faible pour le langage Smalltalk ont été proposées [127].

[1] Pour plus de détails sur l'approche orientée-objet, consulter la section 1.1.1 du Chapitre 2

D'autres mises en œuvre de la persistance faible reposent sur une extension de la machine virtuelle associée au langage de programmation, telles que PJama qui est une extension de la machine virtuelle Java [11].

Il est également possible de mettre en œuvre la persistance faible en modifiant le compilateur du langage de programmation. Cette approche est suivie par SPIN (Support for Persistence, Interoperability and Naming), un mécanisme de persistance des objets CLOS et Java [76]. Dans JSPIN, une version de SPIN dédiée à Java, le compilateur Java modifié transforme les classes d'objets Java en classes d'objets persistants [113]. Ainsi, une application dont le programme a été compilé avec ce compilateur modifié voit ses objets devenus persistants.

En plus de l'approche qui consiste à mettre en œuvre la persistance faible au niveau des langages de programmation, une autre approche consiste à fournir la persistance au niveau de modèles à composants logiciels ou de systèmes orientés-objet distribués. Les modèles à composants EJB (Entreprise Java Beans) de Sun Microsystems [131] ou San Francisco d'IBM [95] proposent une gestion de la persistance faible de leurs composants logiciels Java. Amadeus [27] et Guide [59] sont des exemples de systèmes orientés-objet distribués, qui gèrent la persistance faible de leurs objets.

4.1.3. Mobilité/Persistance forte

En plus des informations prises en compte par la mobilité/persistance faible (code + état courant des données), la *mobilité/persistance forte* prend en compte l'état courant de l'exécution de l'application. Ainsi, une application fortement mobile, qui se déplace au cours de son exécution d'un site source à un site destination, commence son exécution sur le site destination au point où elle a été interrompue sur le site de départ. Et la reprise d'une application fortement persistante se poursuit à la suite du point d'interruption dans la précédente sauvegarde de l'application.

La *mobilité forte* d'une application se traduit par alors :

- L'interruption de l'exécution de l'application sur le site source.
- Puis le transfert du code, de l'état courant des données et de l'état courant de l'exécution de l'application du site source vers le site destination.
- Et enfin la reprise de l'exécution de l'application sur le site destination. Arrivée sur le site destination, l'application mobile ne reprend pas son exécution depuis le début mais la poursuit au point même où elle a été interrompue sur le site de départ.

La plupart des mises en œuvre de mobilité forte se présentent sous la forme de migration de processus (flot d'exécution de l'application). Des synthèses des principaux systèmes fournissant la migration de processus sont proposées par Smith en 1988 [121], Nuttall en 1994 [100], Eskicioglu en 1990 [46] ou Milojicic et al. en 1997 et en 1999 [92] [93].

Parmi les services de migration de processus, nous pouvons citer le mécanisme de migration de processus Unix proposé par Freedman [50], la migration de processus au-dessus du micro-noyau Mach [51] ou la migration de processus dans les systèmes distribués Charlotte [7], Sprite [42] et Emerald [74].

Quant à la persistance forte, elle est également proposée sous la forme de persistance de processus [39]. Les librairies *libckp* proposée par Wang et al. [149], *libckpt* proposée par Plank et al. [107] ou la librairie proposée par Litzkow et al. pour le système Condor [87] fournissent des mécanismes de sauvegarde/reprise des processus Unix.

Dans la suite, nous nous intéressons plus particulièrement aux travaux réalisés autour de la mobilité et de la persistance forte des applications : mobilité et persistance qui prennent en compte l'état courant de l'exécution des applications.

4.2. Niveau de mise en œuvre

Les concepteurs de mécanismes de mobilité et de persistance fortes suivent une des deux approches suivantes :

- Ils construisent un nouveau système qui propose, entre autres fonctionnalités, la mobilité et la persistance fortes. Cette approche a l'avantage de fournir une liberté de mise en œuvre.
- Ou ils construisent leurs mécanismes sur la base d'un système existant et largement diffusé, soit au-dessus du système, soit par extension du système. Cette approche présente l'avantage de favoriser la diffusion des mécanismes de mobilité et de persistance eux-mêmes.

Nous employons le terme « système » pour désigner, aussi bien, un micro-noyau, un système d'exploitation, un middleware ou un langage/environnement de programmation.

4.2.1. Nouveau système

Les nouveaux systèmes construits, qui proposent des mécanismes de mobilité et de persistance, se déclinent sous la forme de nouveaux langages de programmation ou de nouveau middleware et systèmes d'exploitation.

Facile est un langage conçu pour la programmation d'agents mobiles [78] et Tycoon est un langage qui fournit les fonctionnalités nécessaires à la construction des applications mobiles ou persistantes [90] [89].

Accent [151], Charlotte [7], Emerald [126], NOW MOSIX [18] et Sprite [42] sont des systèmes distribués qui fournissent des mécanismes de migration de processus.

Grasshopper est un système d'exploitation qui gère la persistance de ses processus [86]. Dans Grasshopper, un *container* est l'abstraction d'un support persistant et les *loci* sont l'abstraction de processus s'exécutant dans des containers. Les loci

bénéficient ainsi de la persistance des containers sous-jacents et peuvent migrer d'un container à un autre. Dans [38], une proposition d'utilisation de Grasshopper pour fournir la propriété de persistance des threads Java est décrite.

4.2.2. Système existant

Contrairement à la première approche, qui construit un nouveau système, la seconde approche se base sur un système existant et largement diffusé. Cette seconde approche est mise en œuvre :

- soit au-dessus du système considéré, sans le modifier
- soit en étendant le système avec les fonctionnalités nécessaires à la mobilité et à la persistance.

Dans la suite, nous citons, tout d'abord, des exemples de mécanismes de mobilité/persistance réalisés au-dessus de systèmes existants (système Unix, langages de programmation C/C++, Java). Nous présentons, ensuite, des mécanismes réalisés par extension de systèmes existants (langages de programmation Java, Scheme, Tcl, C/C++ ou systèmes Unix, UnixWare, Mach, Chorus).

Parmi les mécanismes de mobilité et de persistance réalisés au-dessus d'un système existant, nous pouvons citer le package Condor qui propose des outils de sauvegarde/reprise de processus Unix, réalisés au-dessus du système d'exploitation Unix [87].

Nous pouvons également citer les travaux effectués au-dessus de langages de programmation, tels que C/C++ ou Java. Ces travaux se basent sur un pré-processeur du programme de l'application, pré-processeur qui augmente le code de l'application de traitements qui rendent l'application mobile ou persistante. En se basant sur un tel pré-processeur, le langage de programmation reste inchangé. Le service Arachne de migration de threads dans les langages C/C++ a été mis en œuvre avec un pré-processeur des programmes C/C++ [41]. Cette technique est également utilisée pour la mise en œuvre de la plate-forme à agents mobiles proposée par le projet Wasp [53] ou la plate-forme à agents mobiles JavaGo [118], ces deux plates-formes étant basées sur un pré-processeur du code source Java. Un pré-processeur de bytecode est utilisé pour la construction d'un système de migration de threads Java [116] ou de capture/restauration de l'état des threads Java dans le système Brakes [147].

Quant à l'approche qui consiste à étendre un système existant avec de nouvelles fonctionnalités, elle est utilisée au niveau de langages de programmation ou au niveau de systèmes d'exploitation.

Cette approche est effectivement adoptée par des langages interprétés, où l'interprète du langage est étendu pour proposer les fonctionnalités nécessaires à la mobilité/persistance. Les plates-formes à agents mobiles Sumatra (basée sur un

interprète Java) [2], MAP (interprète Scheme) [106], ARA[1] (interprètes Tcl et C/C++) [105] et D'Agents (interprètes Tcl et Java) [56] suivent une telle approche pour mettre en œuvre la mobilité de leurs agents.

Quant aux systèmes d'exploitation étendus pour la prise en compte de la mobilité et de la persistance, il en existe plusieurs tels que UnixWare, Mach ou Chorus. Le système UnixWare a été étendu avec le mécanisme Kcpt de sauvegarde/reprise de processus [66] et les micro-noyaux Mach et Chorus ont été respectivement augmentés de mécanismes de migration de tâches [51] et de migration d'acteurs [102].

4.3. Hétérogénéité

Les premiers travaux concernant la mobilité forte des applications ont été réalisés sur des plates-formes homogènes. Ainsi, une application s'exécutant sur un site ne peut se déplacer que vers un site de même architecture physique et de même système d'exploitation. Ceci n'était peut-être pas une réelle contrainte puisque la plupart des systèmes distribués étaient homogènes. DEMOS/MP [109], un des premiers systèmes fournissant la migration de processus, est destiné à des plates-formes homogènes. De même que le mécanisme de migration dans le système V, proposé par Theimer et al. [143].

Les recherches menées dans le domaine se sont ensuite intéressées à la réalisation de la mobilité des applications sur des systèmes hétérogènes, de plus en plus courants. La complexité d'une telle réalisation dépend, en particulier, du fait que la représentation de l'état d'exécution d'une application soit dépendante ou indépendante du système sous-jacent.

Lorsque la représentation de l'état d'exécution d'une application ne dépend pas de la plate-forme sous-jacente, la mise en œuvre de la mobilité de l'application dans un système hétérogène est équivalente à sa mise en œuvre dans un système homogène. Une telle approche est discutée dans [13] où les auteurs définissent l'abstraction d'une machine universelle et donc, d'un monde homogène.

Mais si la représentation de l'état d'exécution d'une application est fortement dépendant de la plate-forme sous-jacente et varie d'une plate-forme à une autre, l'état d'exécution ne peut être directement transféré et utilisé par des plates-formes de natures variées. Pour faire face à ce problème, il faut procéder à des traductions entre les différents formats ; deux solutions sont envisageables.

La première solution consiste à traduire la représentation de l'état d'exécution de l'application mobile du format de la plate-forme source vers le format de la plate-forme

[1] Il est prévu de prendre en compte le langage Java dans la plate-forme Ara.

destination. Dans ce cas, il faut autant de traducteurs qu'il existe de couples de plates-formes différentes. De plus, lorsqu'une nouvelle plate-forme doit être prise en compte par le mécanisme de mobilité, il faut construire *2N* nouveaux traducteurs, *N* étant le nombre de plates-formes déjà prises en compte par le mécanisme de mobilité. Cette approche a été suivie pour la mise en œuvre d'un mécanisme de migration de processus proposé par Shub [120].

Une alternative à la traduction directe des formats de deux plates-formes est de se baser sur un format intermédiaire pour représenter l'état d'exécution d'une application mobile. Ici, la représentation de l'état d'exécution de l'application mobile est traduit du format de la plate-forme source vers le format intermédiaire puis du format intermédiaire vers le format de la plate-forme destination. Cette solution nécessite alors deux traductions à chaque déplacement de l'application. Mais lors de la prise en compte d'un nouveau type de plate-forme, elle présente l'avantage de n'exiger la construction que de deux traducteurs : un traducteur du format de la nouvelle plate-forme vers le format intermédiaire et un traducteur inverse. Les systèmes distribués Emerald [126] et Stardust [26] et le système de migration Tui [123] fournissent un mécanisme de migration de processus qui suit une telle approche.

Quant aux travaux sur la persistance forte des applications, quelques-uns se présentent sous la forme de mécanismes dédiés à une plate-forme particulière, tels le mécanisme proposé par Srouji et al. pour la sauvegarde de processus Windows NT [124] ou les diverses librairies de persistance des processus Unix [107] [149]. Alors que d'autres travaux sont utilisables sur des plates-formes hétérogènes, tels que l'approche basée sur un pré-processeur des programmes Java [147] [53].

4.4. Mode d'initiation

Une autre caractéristique de la mobilité des applications est le mode d'initiation de la mobilité. Le mode d'initiation est identifié en répondant aux questions :

- Est-ce l'entité mobile qui initie sa propre mobilité ?
- Ou est-ce une entité externe qui initie la mobilité d'une entité ?

Nous définissons ainsi deux modes d'initiation à la mobilité : une mobilité décidée et une mobilité forcée.

Si la mobilité d'une entité est initiée par l'entité elle-même, la mobilité est dite *décidée*. Ceci se traduit par l'appel à une primitive de mobilité par le programme de l'entité mobile elle-même. La mobilité décidée est parfois l'unique mode d'initiation proposé par un système. C'est le cas de plates-formes à agents mobiles, où les agents sont autonomes et prennent l'initiative de leurs propres déplacements. Sumatra [2] et MAP [106] sont de telles plates-formes. Mais c'est également le cas de certains

mécanismes de migration de threads, tels que ceux proposés par Sakamoto et al. [116] ou par Sekiguchi et al. [118].

Dans le cas où la mobilité d'une entité n'est pas décidée par l'entité mobile elle-même mais par une entité externe, la mobilité est dite *forcée*. Ceci se traduit par l'appel à une primitive de mobilité par l'entité externe et l'application de cette primitive à l'entité mobile. La mobilité forcée peut s'avérer nécessaire à l'automatisation de la répartition dynamique de charge dans un système distribué ou à la reconfiguration dynamique d'applications distribuées. Le système distribué NOW MOSIX propose un mécanisme de mobilité forcée en autorisant un nœud du système distribué à ordonner la migration d'un processus de ce nœud [18]. Le système Nomads fournit une mobilité décidée et une mobilité forcée aux threads Java [142]. Et le service Kcpt de persistance de processus fournit une primitive de sauvegarde décidée (*ckp*) et une primitive de sauvegarde forcée par un processus externe (*tckp*) [66].

4.5. Granularité

La mobilité d'une application peut être, selon le nombre d'entités rendues conjointement mobiles, une mobilité *à gros grain* ou une mobilité *à grain fin*. Avec une mobilité *à gros grain*, l'entité mobile ne peut être déplacée qu'accompagnée de l'ensemble des entités résidant sur le même site qu'elle. Dans le cas, par exemple, d'une mobilité faible dans un système orienté-objet distribué, un objet ne peut être déplacé d'un site source vers un site destination qu'avec l'ensemble des objets résidant sur le site source. Cette approche est suivie par PJama, où une extension de la machine virtuelle Java gère la persistance des objets Java et rend l'ensemble des objets de la JVM potentiellement persistants [12]. Le système Merpati propose un service de mobilité forte dans Java (mobilité des threads), par extension de la machine virtuelle (JVM) [129]. La mobilité d'une application Java, avec Merpati, se traduit par le déplacement de l'ensemble des threads Java résidant sur la JVM, d'un site source vers un site destination. Dans Merpati, à une instance de JVM est associée une seule application Java. Cette approche est également suivie par la version actuelle de la machine virtuelle sur laquelle est construite la plate-forme à agents mobiles Java Nomads [142]. Quant à la mise en œuvre de la persistance des applications Java proposée par Howell, elle repose sur une couche logicielle entre la JVM et le système d'exploitation. Cette couche logicielle sauvegarde l'ensemble des threads Java s'exécutant dans la JVM [67].

Une mobilité est dite *à grain fin* lorsqu'une entité individuelle peut se déplacer, indépendamment des autres entités résidant sur le même site qu'elle. C'est le cas, par exemple, de la migration d'une page mémoire dans une mémoire répartie partagée, de la mobilité d'un objet dans un système orienté-objet distribué ou de la mobilité d'un flot

de contrôle (processus ou thread) dans un système distribué. Le système orienté-objet distribué Emerald [74] et le mécanisme de sérialisation Java [136] proposent une mobilité à grain fin pour leurs objets.

4.6. Adaptabilité

Le terme « adaptabilité » peut avoir plusieurs significations, selon le contexte dans lequel il est utilisé. Nous l'utilisons ici pour caractériser :

- un mécanisme qui peut être utilisé aussi bien pour la mobilité que pour la persistance des applications
- et un mécanisme de mobilité et de persistance qui peut être spécialisé aux besoins des utilisations qui en sont faites.

La plupart des mécanismes réalisés sont soit des mécanismes dédiés à la mobilité, soit des mécanismes dédiés à la persistance. Mais ces mécanismes sont conceptuellement similaires et peuvent donc être conjointement mis en œuvre. C'est ce qui est proposé par la plate-forme Merpati, qui fournit des mécanismes de mobilité et de persistance des threads Java [129].

Pour ce qui est de la spécialisation des mécanismes de mobilité et de persistance, ceux-ci peuvent être des mécanismes monolithiques, dédiés à une utilisation particulière de la mobilité ou de la persistance. C'est le cas, par exemple, de la plate-forme à agents mobiles Sumatra, qui fournit la primitive *engine.go()* pour la migration d'un agent vers une localisation identifiée par *engine* [111]. Ici, la primitive de migration ne peut pas être modifiée pour des besoins spécifiques à une utilisation de la mobilité, comme le cryptage de l'agent lors de son déplacement sur le réseau.

Il est également possible de proposer des mobilité et persistance qui peuvent être adaptées à l'utilisation qui est faite de la mobilité/persistance. Considérons la sérialisation d'objets Java, qui permet de transférer des objets Java vers un nouveau site [136]. Ce mécanisme propose un comportement par défaut qui consiste à transférer un objet et tous les objets qui peuvent être atteints à partir de cet objet. Mais en plus de ce comportement par défaut, il est possible de définir une politique de sérialisation propre à chaque type d'objet (classe d'objet). Une autre spécialisation possible est celle proposée par Freedman, dans son mécanisme de migration de processus Unix [50]. Ce mécanisme fournit, en effet, des points d'entrée qui permettent au programmeur d'une application de spécifier des traitements à effectuer avant et après une opération de migration.

4.7. Performances

Les performances d'un mécanisme de mobilité sont évaluées selon trois coûts :

- la latence de la mobilité,

- le surcoût sur l'application mobile
- et le surcoût sur les applications non mobiles.

Dans le cas de la persistance d'une application, les coûts à prendre en compte sont la latence de la sauvegarde, la latence de la reprise et les surcoûts sur l'application persistante et sur les applications non persistantes. Ces coûts peuvent également être accompagnés d'une évaluation de la taille de l'état d'exécution exporté de l'application mobile/persistante. Dans la suite de cette section, nous décrivons les coûts relatifs à la mobilité mais ce discours reste valable pour la persistance.

La latence de la mobilité d'une application est le temps nécessaire pour déplacer l'application d'un site source vers un site destination. Ce temps est borné par l'instant d'appel d'une primitive de mobilité sur l'application et l'instant d'arrivée de l'application mobile sur son site destination, avant le lancement de son exécution. La latence de la mobilité dépend, d'une part, du temps d'accès et de manipulation des structures internes de l'application (état des données, état de l'exécution) et, d'autre part, de la taille de l'état de l'application, état qui sera transféré sur le réseau. Un paramètre qui peut avoir un impact sur la latence de la mobilité est le niveau de mise en œuvre du mécanisme de mobilité. Un mécanisme de mobilité peut, en effet, être réalisé au niveau de l'application, au niveau du middleware ou au niveau du système d'exploitation. Généralement, lorsque le niveau de mise en œuvre de la mobilité est bas (niveau du système d'exploitation), les informations sur l'état d'exécution de l'application sont directement accessibles et utilisables. C'est à ce niveau que l'accès et la manipulation de l'état d'exécution devraient être les moins coûteux. Mais en même temps, c'est au niveau de l'application qu'il y a la connaissance de la sémantique de l'application. C'est à ce niveau qu'il est possible de spécifier plus finement ce qui doit faire partie de l'état mobile de l'application et de minimiser ainsi la taille de l'état transféré [81].

Par ailleurs, rendre une application mobile peut éventuellement induire un surcoût sur les performances de l'application elle-même. Autrement dit, les traitements qui sont propres à une application - traitements qui ne concernent pas l'opération de mobilité - peuvent voir leurs performances se dégrader en raison de la prise en compte de la mobilité. Ce surcoût est évalué en comparant le temps nécessaire à l'exécution d'une application mobile, en dehors de l'opération de mobilité, et le temps nécessaire à l'exécution de cette même application lorsqu'elle n'est pas mobile. Un tel surcoût est, par exemple, induit par le mécanisme de mobilité proposé par le système JavaGo [118] puisque ce système est basé sur une augmentation du code source de l'application mobile. Un surcoût sur les performances de l'application mobile est également induit par le mécanisme Java proposé par Illmann et al., car la mise en œuvre de ce mécanisme

est basée sur le debugger Java, ce qui induit un surcoût considérable sur l'application [70].

Quant au troisième coût, il représente l'éventuel surcoût induit par la mobilité sur les performances des applications non mobiles. Ce coût décrit la dégradation des performances des applications non mobiles, même en l'absence d'applications mobiles s'exécutant sur le même système qu'elles. Ceci peut, par exemple, se produire à la suite de l'intégration d'un service de mobilité à un système existant, intégration qui a exigé la modification d'autres services du système. Mais une mise en œuvre modulaire permet généralement de séparer le service de mobilité des autres services et de réduire ainsi l'impact que peut avoir la mobilité sur les performances des applications non mobiles.

4.8. Transparence

Dire que la mobilité/persistance d'une application s'effectue de façon transparente pour l'application mobile/persistante peut avoir plusieurs significations. Nous distinguons deux catégories de transparence : transparence de la mobilité/persistance *à la programmation* de l'application et transparence de la mobilité/persistance *à l'exécution* de l'application.

4.8.1. Transparence à la programmation de l'application

La transparence de la mobilité/persistance à la programmation de l'application mobile/persistante est relative au besoin de modifier les traitements effectués par l'application (son programme) pour la prise en compte de la mobilité/persistance.

Rendre une application mobile ou persistante peut n'exiger aucun changement ni aucun ajout au programme de l'application. Dans ce cas, la mobilité ou la persistance sont totalement transparentes à la programmation de l'application. Certaines techniques de persistance qui sont basées sur des compilateurs proposent une persistance totalement transparente à la programmation de l'application. Ces compilateurs augmentent le code de l'application de traitements relatif à la persistance et décident du placement des sauvegardes dans le code [72].

La mobilité peut toutefois nécessiter un ajout ou un changement mineur au programme de l'application, tel que l'ajout d'une nouvelle propriété à l'application. Dans ce cas, la mobilité est dite partiellement transparente à la programmation de l'application. Par exemple, pour qu'un objet Java puisse être sérialisé, il faut ajouter une propriété de « sérialisabilité » à la classe de l'objet (sa classe doit implanter l'interface *java.io.Serializable*) [136].

La transparence de la mobilité/persistance à la programmation de l'application confère une plus grande souplesse à l'évolution de l'application puisque la mise en œuvre de la mobilité/persistance est extérieure à l'application.

Inversement, une mobilité ou une persistance qui nécessitent une restructuration complète de l'application sont non transparentes à la programmation de l'application.

4.8.2. Transparence à l'exécution de l'application

La transparence de la mobilité à l'exécution d'une application requiert que ni l'application mobile ni les autres applications du système ne perçoivent le déplacement de l'application, excepté par une éventuelle différence de performances. Et la transparence de la persistance à l'exécution d'une application requiert que ni l'application persistante ni les autres applications du système ne perçoivent la sauvegarde de l'application. Ainsi, les communications de l'application mobile peuvent être mises en attente durant le déplacement de l'application mais les messages en transit ne doivent pas être perdus. Arrivée sur le site destination, l'application mobile doit pouvoir accéder aux fichiers précédemment ouverts sur le site source. De façon générale, la mobilité est transparente à l'exécution de l'application si, après le déplacement de l'application, celle-ci peut accéder aux ressources (logicielles ou matérielles) auxquelles elle accédait avant son déplacement.

La mise en place de la transparence de la mobilité et de la persistance à l'exécution de l'application mobile/persistante aborde alors des questions relatives à l'identification de l'application mobile/persistante, aux ressources partagées par l'application mobile/persistante et par d'autres applications, à la cohérence du code utilisé par l'application mobile/persistante et à la gestion des entrées/sorties de l'application. Dans la suite, nous abordons, une à une, chacune de ces questions et présentons les réponses qui leur ont été apportées. Ces questions sont abordées dans le cas de la mobilité mais restent valables pour la persistance.

- ***Identification de l'entité mobile***

Dans cette section, nous nous intéressons à l'identification de l'application mobile par les autres applications du système.

Dans certains systèmes distribués, l'identification d'une application est dépendante de la localisation courante de l'application. Ceci facilite la recherche de la localisation de l'application mobile à partir de son nom. L'identification de l'application peut, par exemple, contenir une adresse IP et un numéro de port. Les systèmes Aglets [69], Tacoma [71] et D'Agents [56] utilisent une telle identification. Dans ce cas, après le déplacement d'une application vers une nouvelle localisation, le précédent identificateur de l'application n'est plus valide et est remplacé par le nouvel identificateur. Or il est peut-être utilisé par d'autres applications du système. Dans ce cas, deux solutions se présentent. Soit les autres applications sont informées de la nouvelle identité de l'application mobile, juste après le déplacement de l'application. Soit la nouvelle

identité de l'application est obtenue par les autres applications lorsque celles-ci s'aperçoivent de l'invalidité de l'ancienne identité.

Une autre approche pour l'identification de l'entité mobile est l'utilisation d'une identification qui est indépendante de la localisation courante de l'entité mobile. Une entité mobile est ainsi identifiée de la même manière, avant et après son déplacement vers un nouveau site. Cette approche peut être mise en œuvre de deux manières :

- Utilisant d'un représentant local (proxy) qui donne la localisation courante de l'entité mobile distante. Cette solution est adoptée par la plate-forme à agents mobiles Voyager [101]. Le système orienté-objet distribué Emerald utilise cette solution pour l'identification de ses objets globaux et se base sur une identification dépendante de la localisation pour ses objets locaux [74].
- Utilisation d'un système de nommage global et indépendant de la localisation courante de l'entité mobile. Cette solution est utilisée dans le système Ajanta [146].

- ***Ressources partagées***

Une application mobile qui manipule une ressource peut soit ne pas emporter cette ressource, si la ressource n'est plus utilisée par l'application par la suite, soit déplacer cette ressource avec elle, si la ressource peut être déplacée et si elle n'est pas utilisée par d'autres applications du site source, soit ne pas déplacer la ressource et utiliser une ressource équivalente sur le site d'arrivée [52]. Mais que se passe-t-il si une application se déplace d'un site source vers un site destination, alors qu'elle utilisait une ressource partagée avec d'autres applications sur le site source ?

Une première solution utilise l'accès distant par liens de poursuite (référence à distance). Ici, la ressource partagée n'est pas transportée avec l'application mobile. Mais arrivée sur le site destination, l'application mobile accède à la ressource à distance. Cet accès à distance se fait via des liens de poursuite qui lient la nouvelle localisation de l'application à son ancienne localisation. Il est également possible d'opter pour la solution symétrique : la ressource partagée est déplacée avec l'application mobile et ce sont les autres applications qui accèdent à cette ressource à distance. Il est important de noter que le choix des ressources déplacées conjointement a un impact sur la disponibilité de l'application en cas de panne d'un des sites et sur la localité des accès à une ressource, en cas d'interactions fréquentes entre les ressources. Ce choix est déterminé par la politique de mobilité sous-jacente.

Une autre solution au problème de partage de ressources est basée sur l'accès distant par duplication. Une ressource partagée entre l'application mobile et les autres applications du site source est dupliquée et sa copie est transférée avec l'application mobile. Il faut alors disposer d'un mécanisme de gestion de la cohérence entre les

différentes copies d'une même ressource. Il faut également gérer les accès concurrents synchronisés à une ressource partagée. Le problème de duplication et de cohérence des ressources a été largement abordé dans les systèmes [96] et a été étudié plus récemment dans les systèmes IceCube [77] et Javanaise [60].

Dans la suite, nous nous intéressons à deux types de ressources particulières : le code et les entrées/sorties des applications mobiles.

- ***Cohérence du code***

Une application mobile, qui utilise un code sur un site source, peut arriver sur un site destination sur lequel se trouve un code homonyme. Un code homonyme à un code d'origine désigne un code qui porte le même nom mais qui peut être sémantiquement différent du code d'origine. Il faut alors choisir le code que l'application doit utiliser sur le site destination.

Une première solution est de transporter le code de l'application lors des déplacements de l'application mobile et d'utiliser ce code sur les sites visités par l'application.

Une autre solution consiste à utiliser, sur le site d'arrivée, un code équivalent au code de l'application. Ceci peut s'avérer intéressant pour répondre à des besoins d'adaptabilité de la mobilité. En effet, considérons le cas d'une application qui se déplace d'un ordinateur personnel (PC) vers un assistant numérique personnel (PDA) aux capacités physiques réduites. Un exemple d'adaptation de la mobilité est de remplacer le code de l'interface-utilisateur évoluée, utilisée sur l'ordinateur personnel, par le code d'une interface-utilisateur minimale, sur l'assistant numérique.

- ***Entrées/Sorties de l'entité mobile***

Les questions relatives aux entrées/sorties d'une application mobile concernent la gestion des communications en cours, des fichiers et de l'interface-utilisateur de l'application mobile. Dans la suite, nous abordons un à un ces trois points.

Considérons, tout d'abord, le problème des communications en cours entre plusieurs applications, lorsque l'une des applications mises en œuvre se déplace vers un nouveau site. Une première solution consiste à recréer, sur le site d'arrivée, de nouveaux canaux de communication qui connectent l'application mobile aux autres applications. Ceci revient à fermer les canaux de communications entrants et sortants de l'application mobile, sur le site source, et à recréer ces canaux sur le site destination. Cette recréation des canaux passe par une phase d'interruption momentanée des communications.

Une autre solution au problème de communications en cours est basée sur la redirection de ces communications. Après le déplacement de l'application vers le site destination, les messages qui lui sont destinés arrivent toujours sur le site source mais

sont redirigés vers le site destination. Cette redirection est mise en place via des liens de poursuite qui lient le site destination de l'application au site source. Dans le système Accent, prédécesseur du micro-noyau Mach, l'abstraction de *port* désigne un canal de communication entre les processus. Un message qui est envoyé sur le port d'un processus migrant est redirigé vers la nouvelle localisation du processus [151]. Avec une solution basée sur l'utilisation de liens de poursuite entre les différents sites visités par une application mobile, l'enchaînement de ces liens peut être plus ou moins long, en fonction du nombre de déplacements effectués par l'application mobile. Cet enchaînement multiple peut être coûteux en temps d'accès et doit alors être raccourci. D'autre part, avec une solution basée sur des liens de poursuite, un problème de disponibilité se pose dans le cas d'une panne survenue sur un des sites visités par l'application mobile. Lu et al. proposent d'éliminer les dépendances résiduelles sur les différents sites visités par les processus migrants dans leur système [88].

Un autre type d'entrées/sorties des applications mobiles est le fichier. En effet, que se passe-t-il lorsqu'une application se déplace vers un site destination alors qu'elle accédait à un fichier sur le site source ? L'application peut, par exemple, continuer à accéder à ce même fichier, à distance, tel que proposé par le système Condor [87]. Le fichier peut, sinon, être transféré avec l'application [58]. Dans le service de persistance Kcpt, la sauvegarde d'un processus prend en compte les identificateurs des fichiers utilisés par le processus et la position dans chacun de ces fichiers [66]. Ainsi, à la reprise du processus, l'accès aux fichiers utilisés se fera à la bonne position. Mais il faut également considérer les problèmes de partage de fichier, de concurrence d'accès par plusieurs applications et de cohérence.

Un troisième type d'entrées/sorties des applications mobiles est l'interface-utilisateur de l'application. Nous considérons ici le cas d'une application qui se déplace vers un site destination avec son interface-utilisateur. Même si le déplacement de l'application a transféré toutes les informations relatives à l'application et à son interface-utilisateur, cette interface ne sera pas pour autant ré-affichée sur l'écran du site destination. Dans ce cas, il faut donc que le déplacement de l'application dotée d'une interface-utilisateur procède au rafraîchissement de cette interface.

5. Synthèse des travaux

Plusieurs travaux dans les domaines de la mobilité et de la persistance des applications ont été réalisés. Ce chapitre présente certains de ces travaux, en se basant sur les critères définis précédemment et qui sont :

- Le degré de mobilité/persistance.
- Le niveau de mise en œuvre.

- L'hétérogénéité.
- Le mode d'initiation de la mobilité/persistance.
- La granularité.
- L'adaptabilité.
- Les performances.
- La transparence à la programmation et à l'exécution des applications.

Parmi les travaux traitant de la mobilité et de la persistance des applications, beaucoup se sont intéressés à la mobilité et persistance faibles, tels que les projets Mole [20] ou PJama [12]. D'autres se sont penchés sur l'intérêt de la mobilité/persistance fortes par rapport aux autres formes de mobilité/persistance, ceci est discuté par Cardelli [30] ou par Harrisson [61]. Mais même si la mobilité et persistance faibles restent un degré acceptable pour certains types d'applications, ce degré n'est pas suffisant pour d'autres applications, telles que les applications aux calculs longs.

Plusieurs travaux ont également été menés dans les domaines de la mobilité et de la persistance fortes, tels que le système de mobilité forte Emerald [74] ou la librairie de persistance forte libckpt [107]. Emerald est un système distribué complet, proposant un mécanisme fonctionnel de mobilité forte. Mais le fait qu'Emerald implante un nouvel environnement d'exécution, avec son propre modèle à objet, a quelque peu limité sa diffusion. D'autres projets ont choisi d'adresser le problème de la mobilité et de la persistance en se basant sur des systèmes déjà largement diffusés. Libckpt est une librairie qui propose un mécanisme de persistance forte par sauvegarde/reprise des processus Unix. Ce mécanisme est aisément utilisable et son fonctionnement peut être adapté en fixant certaines options qui lui sont associées.

Alors que des solutions comme libckpt sont dédiées à des plates-formes homogènes (plate-forme Unix), d'autres projets ont visé le support d'environnement hétérogènes. La plate-forme à agents mobiles Sumatra fournit un mécanisme de mobilité forte, utilisable dans un système hétérogène (environnement Java) [111]. Les agents fortement mobiles sont des threads Java. Mais Sumatra ne propose qu'un seul mode d'initiation de la mobilité : la mobilité décidée, initiée par l'agent mobile lui-même. La plate-forme Nomads [142], également basée sur l'environnement Java, propose une mobilité décidée par le thread mobile et une mobilité forcée par un thread externe. Mais dans l'état actuel de Nomads, un thread ne peut être rendu mobile individuellement : il est obligatoirement accompagné de l'ensemble des threads qui s'exécutent sur la même machine virtuelle que lui. Le système Moba, quant à lui, propose une mobilité à grain plus fin, une mobilité dont l'unité est le thread Java [80]. Mais la mise en œuvre de Moba est, d'une part, dédiée à la mobilité et, d'autre part, non adaptable à des besoins particuliers d'une application mobile.

Par ailleurs, pour que les mécanismes de mobilité et de persistance soient effectivement utilisables, ils doivent présenter des performances acceptables par les applications qui font appel à ces mécanismes. Ces performances oscillent entre la latence due aux opérations de mobilité/persistance et le surcoût induit sur le temps d'exécution des applications. Généralement, le temps de latence est inversement proportionnel au surcoût. La nature de l'application et sa fréquence d'appel aux services de mobilité/persistance permet de déterminer s'il est plus intéressant de sacrifier les performances habituelles de l'application au profit d'une faible latence ou s'il est, au contraire, plus intéressant d'éliminer le surcoût à l'exécution en reportant tous les coûts sur la latence.

Finalement, pour que les mécanismes de mobilité et de persistance soient aisément utilisables, ils doivent proposer une certaine transparence à la programmation de des applications. Ils doivent, de plus, permettre de mettre en place diverses politiques de gestion de la transparence de la mobilité/persistance à l'exécution des applications, politiques relatives aux problèmes :

- d'identification de l'entité mobile/persistante,
- des entrées/sorties de l'entité mobile/persistante
- et de partage de ressources.

6. Hypothèses et choix de conception

Après avoir présenté les principaux travaux dans le domaine de la mobilité et de la persistance des applications, nous décrivons, dans cette section, les hypothèses sur lesquelles se basent nos services de mobilité et de persistance et les choix de conception de ces services. Nous utilisons les précédents critères de classification pour présenter nos hypothèses et choix de conception.

- ***Mobilité et persistance fortes***

La mobilité et la persistance fournies doivent être une mobilité et une persistance fortes. A l'arrivée d'une application mobile sur un site destination, l'application a directement accès à ses données mises à jour et elle poursuit son exécution au point où elle l'a interrompue sur le site de départ. De façon similaire, la reprise d'une application persistante donne directement accès aux données actualisées de l'application et commence l'exécution au point même où celle-ci a été interrompue au moment de la sauvegarde.

Pour que la mobilité et la persistance fournies soient fortes, l'entité mobile/persistante considérée doit prendre en compte le code, l'état des données et l'état de l'exécution de l'application. L'entité mobile ou persistante représente donc un flot d'exécution (processus ou thread).

- ***Hétérogénéité des plates-formes***

La mobilité et la persistance proposées doivent être utilisables sur des plates-formes de natures variées. Ainsi, une application mobile peut se déplacer d'un site à un autre, indépendamment des plates-formes sous-jacentes. Et une application persistante, sauvegardée sur une plate-forme, peut ensuite être reprise sur un autre type de plate-forme.

Pour que la mobilité et la persistance proposées puissent être utilisées sur des plates-formes hétérogènes, leur mise en œuvre ne doit pas être rattachée à une architecture de machine ni à un quelconque système d'exploitation. Pour ce faire, nous avons basé la conception de la mobilité et de la persistance sur l'abstraction d'un environnement homogène : l'environnement Java[1].

- ***Niveau de mise en œuvre***

Pour faciliter la diffusion des mécanismes de mobilité et de persistance des applications, ceux-ci doivent être mis en œuvre dans un système existant.

La mise en œuvre de la mobilité et de la persistance se fera dans l'environnement Java, un environnement largement répandu, qui facilitera la diffusion de nos services de mobilité/persistance.

- ***Mobilité et persistance décidées ou forcées***

La mobilité fournie doit proposer les deux modes d'initiation de la mobilité : la mobilité décidée, initiée par l'entité mobile elle-même, et la mobilité forcée, initiée par une autre entité que l'entité mobile. Proposer ces deux modes d'initiation de la mobilité permet d'adresser un large champ d'applications. De façon similaire, la persistance des applications fournie doit proposer une persistance décidée et une persistance forcée.

La mobilité et la persistance proposées doivent fournir une interface de programmation (API) qui propose deux modes d'initiation de la mobilité et de la persistance : une initiation décidée et une initiation forcée.

- ***Granularité***

La mobilité et la persistance proposées doivent être à grain fin, tout en permettant de rendre un ensemble/groupe d'entités conjointement mobiles ou conjointement persistantes.

Pour que la mobilité et la persistance proposées soient à grain fin, l'unité de mobilité et de persistance choisie est le thread Java. Mais il sera également possible de rendre conjointement mobiles ou conjointement persistants un groupe de threads.

[1] L'environnement Java est décrit dans le Chapitre 2

- ***Adaptabilité***

Les mécanismes proposés doivent pouvoir être utilisés aussi bien pour la mobilité que pour la persistance des applications. La conception de ces mécanismes doit donc être une conception générique.

De plus, ces mécanismes doivent pouvoir être spécialisés aux besoins des applications qui les utilisent. Considérons, par exemple, le cas d'une application mobile, qui se déplace d'un ordinateur personnel (PC) vers un assistant numérique personnel (PDA) aux capacités physiques réduites. Un exemple d'adaptation de cette application est de remplacer l'interface-utilisateur graphique, utilisée sur l'ordinateur personnel, par une interface-utilisateur minimale, adaptée aux caractéristiques et aux capacité réduites du nouveau site hôte. Quant à l'adaptabilité de la persistance, elle peut, par exemple, permettre au programmeur de l'application persistante de spécifier des traitements supplémentaires à effectuer lors de la sauvegarde de son application et lors de la reprise de son application, tels que le cryptage/décryptage de l'image sauvegardée de l'application.

- ***Performances***

Les performances des mécanismes de mobilité et de persistance des applications représentent un facteur important et déterminant de l'utilisabilité de ces mécanismes. La mobilité et la persistance proposées doivent donc présenter des performances raisonnables en termes de latence et de surcoût. Mais minimiser un de ces coûts se fait généralement au détriment de l'autre. Pour cela, nous avons suivi un des conseils donnés par Butler W. Lampson dans [79] : « *Handle normal and worst cases separately*: *The normal case must be fast. The worst case must make some progress*. ».

Ainsi, en considérant que les opérations de mobilité et de persistance sont peu fréquentes dans la durée de vie d'une application, nous avons choisi de minimiser voire annuler le surcoût induit par la mobilité et la persistance sur le comportement normal des applications. Ceci doit se faire en préservant des temps de latence raisonnables de la mobilité, la sauvegarde ou la reprise.

- ***Transparence***

  - *Transparence à la programmation de l'application*

La mobilité et la persistance proposées doivent être transparentes à la programmation des applications, cette transparence pouvant être totale ou partielle.

La transparence à la programmation d'une application est obtenue par une automatisation de la mobilité et de la persistance, via des services dédiés. Séparer les traitements relatifs à la mobilité et à la persistance des traitements qui sont propres à l'application revient tout simplement à placer les traitements de mobilité/persistance à

un autre niveau : au niveau de services dédiés à la mobilité et à la persistance des applications.

- *Transparence à l'exécution de l'application*

Il existe plusieurs politiques de gestion de la transparence de la mobilité et de la persistance à l'exécution de l'application mobile/persistante. Considérons, par exemple, le problème de partage de ressources entre l'application mobile/persistante et d'autres applications. Ce problème peut être géré par un accès distant aux ressources partagées via des références distantes ou par un accès local aux ressources en se basant sur la duplication. Les autres problèmes de transparence à l'exécution (identification de l'entité mobile/persistante, cohérence du code, entrées/sorties) ont également plusieurs solutions envisageables.

La mobilité et la persistance proposées ne fournissent pas une politique de gestion de la transparence à l'exécution mais doivent permettre d'intégrer la politique la plus appropriée à l'application considérée. Cette transparence à l'exécution peut, en partie, être construite par l'adaptation de nos services de mobilité et de persistance. Ceci est discuté dans le Chapitre 4.

7. Conclusion

Les travaux dans les domaines de la mobilité et de la persistance des applications se différencient par :

- le degré de mobilité ou de persistance proposé,
- le niveau de mise en œuvre,
- l'hétérogénéité de l'environnement adressé,
- le mode d'initiation de la mobilité/persistance,
- la granularité,
- l'adaptabilité,
- les performances
- et la transparence à la programmation et à l'exécution des applications.

Ces critères sont utilisés pour la définition des hypothèses et choix de conception de nos services de mobilité et de persistance, tels que résumés par le Tableau 1-1.

Nous avons donc choisi de fournir des services de mobilité et de persistance fortes des applications à travers les threads Java. Une application mobile ou persistante est donc un thread Java mobile ou persistant. Nous avons également choisi de fournir des mobilité et persistance qui peuvent être décidées ou forcées, qui peuvent être gérées de façon transparente et qui peuvent être spécialisées aux besoins des applications qui les utilisent.

Les hypothèses faites sur nos services de mobilité et de persistance vont être démontrées tout au long des chapitres suivants :

- Dans le Chapitre 4, nous illustrons la force de la mobilité et de la persistance fournies par nos services, leurs deux modes d'initiation (décidé ou forcé), leur transparence à la programmation et l'hétérogénéité des plates-formes sur lesquelles ils sont utilisés.
- Dans le Chapitre 3, nous décrivons l'adaptabilité et le niveau de mise en œuvre de nos services.
- Dans le Chapitre 4, nous présentons la mise en place de la transparence de la mobilité et de la persistance à l'exécution des applications utilisant nos services.
- Dans le Chapitre 5, nous décrivons les performances fournies par ces services.

Hypothèses de conception	**Choix de conception**
Mobilité et persistance fortes	Mobilité et persistance des threads
Hétérogénéité	Environnement Java
Niveau de mise en œuvre	Environnement Java
Mobilité et persistance décidées ou forcées	Interface de mobilité/persistance avec deux modes d'initiation
Adaptabilité	Services génériques
Performances	◆ Pas de modification de l'exécution des applications non mobiles et non persistantes ◆ Pas de modification de l'exécution normale des applications mobiles ou persistantes ◆ Prise en compte des informations/traitements minimaux pour la mobilité/persistance
Transparence à la programmation	Automatisation de la mobilité/persistance à travers des services dédiés
Transparence à l'exécution	Adaptation des services de mobilité et de persistance

Tableau 1-1. Hypothèses et choix de conception de la mobilité et de la persistance des applications

CHAPITRE 2 - JAVA : ENVIRONNEMENT ET MACHINE VIRTUELLE

Java est à la fois un environnement de programmation et une machine virtuelle. Ce chapitre, qui est une présentation de Java, est composé de deux grandes parties.

La première partie présente Java en tant qu'environnement de programmation. Elle rappelle les concepts généraux du langage de programmation et décrit la richesse de ses API. Cette première partie est recommandée aux novices de l'environnement de programmation Java. Mais le lecteur familier de Java peut directement se référer à la partie suivante.

La seconde partie décrit Java en tant que machine virtuelle. Cette partie présente les concepts nécessaires à la compréhension de la conception et de la mise en œuvre de nos services de mobilité et de persistance.

1. Environnement Java

Java est à la fois un langage de programmation, l'abstraction d'un environnement homogène porté sur des plates-formes variées et des API riches et utiles pour la programmation :

- Java est un langage qui intègre plusieurs concepts : l'approche objet, le typage fort, la sécurité, les threads et les exceptions.
- Java est un environnement d'exécution qui s'appuie sur les progrès faits en matière de machine virtuelle depuis vingt-cinq ans pour concilier la portabilité du code avec le souci de performance.
- Java est un langage qui intègre des API utiles pour programmer les systèmes d'information (bases de données, texte, son, image, réseau, architectures distribuées).

Cette section est composée de trois parties. La première partie présente Java en tant que langage de programmation. La seconde partie décrit Java comme l'abstraction d'un environnement homogène. La troisième partie présente Java en tant qu'environnement intégrant des API utiles à la programmation de systèmes d'information et plus particulièrement, de systèmes pour la mobilité et la persistance des applications.

1.1. Langage de programmation Java

Java est un langage de programmation qui a été défini à partir de 1990, dans le cadre d'un projet de recherche, par Sun Microsystems [6]. Il s'agissait de concevoir un langage bien adapté aux environnements de travail en réseau et capable de gérer des informations de natures variées telles que le texte, les données numériques ou les informations sonores et graphiques. Les concepteurs de Java ont décidé de s'appuyer sur les expériences les plus récentes en matière de technologie de programmation, ils ont choisi :

- de spécifier un langage objet,
- de s'orienter, pour la philosophie du langage, vers la simplicité des concepts introduits par Smalltalk [83] : Java est donc un langage de haut niveau qui évite au programmeur les difficultés d'un langage comme C++ [128],
- de s'appuyer, pour la forme, sur un fait industriel largement accepté, la syntaxe C/C++ : c'est sans doute un moyen de favoriser une diffusion rapide de Java.

1.1.1. Langage orienté-objet

Tous les langages de programmation fournissent des abstractions. La complexité des problèmes résolus est directement liée au type d'abstraction fournie par un langage. Prenons, par exemple, le cas d'un problème simplifié de gestion des employés d'une

entreprise. Le modèle de ce problème est décrit par un cahier des charges qui spécifie les informations relatives à chaque employé et les opérations applicables sur un employé :

- Les informations sur l'employé comprennent, par exemple, le nom, l'adresse et le salaire de l'employé.
- Les opérations à appliquer sur un employé peuvent être le changement d'adresse ou l'augmentation du salaire de l'employé.

La résolution du problème de gestion des employés par un programme informatique sera plus ou moins complexe en fonction de l'abstraction fournie par le langage de programmation utilisé.

L'évolution des abstractions fournies par les langages de programmation est illustrée par la Figure 2-1. L'abstraction fournie par un langage assembleur est le modèle de la machine physique sous-jacente. Programmer en assembleur revient à réfléchir en termes de registres, d'instructions. Il revient donc au programmeur d'établir une correspondance entre le modèle du problème abordé (le cahier des charges) et le modèle de la machine qui lui est proposé par le langage de programmation. Ceci rend parfois les programmes difficiles à écrire et chers à maintenir.

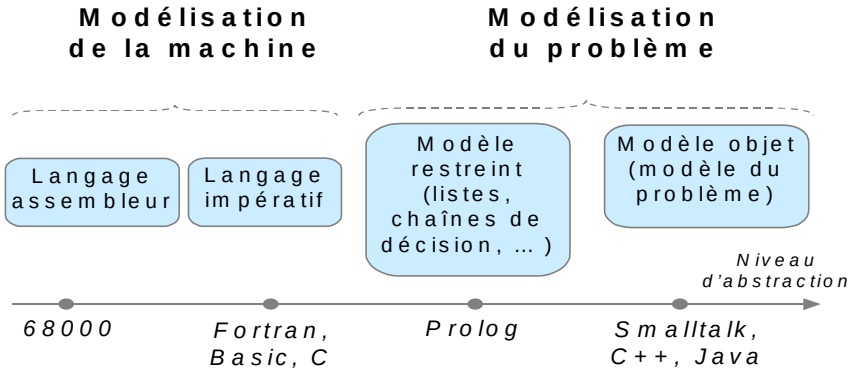

Figure 2-1. Vers les langages orientés-objet

Les langages dits impératifs, tels que Fortran, Basic ou C, représentent une grande amélioration du langage assembleur. Mais l'abstraction proposée par ces langages reste le modèle de la machine : le programmeur doit réfléchir en termes de structures de la machine (mémoire, processus) plutôt qu'en termes de modèle du problème à résoudre.

Une alternative à la modélisation de la machine est la modélisation du problème. Des langages de programmation fournissent une représentation des éléments du problème. Prolog, par exemple, considère les problèmes comme des chaînes de décision. Chacun de ces langages est une bonne solution pour une catégorie particulière de problèmes.

Mais sortis de leurs catégories de problèmes, ces langages deviennent peu commodes à utiliser.

Une autre approche consiste à fournir au programmeur des outils qui lui permettent de définir ses propres abstractions. Avec cette approche, le programmeur peut définir une abstraction pour chaque catégorie de problèmes abordés et n'est plus contraint à une catégorie particulière de problèmes. Ainsi, l'abstraction définie par le programmeur sera le modèle du problème lui-même. Les entités dans le modèle du problème (et donc, dans l'abstraction définie par le programmeur) sont appelées *objets* et cette approche de programmation est dite *orientée-objet* [36] [108] [57]. Smalltalk [83], C++ [128] et Java [6] sont des langages de programmation orientés-objet.

1.1.2. Quelques concepts du langage Java

Nous supposons ici que le lecteur possède les concepts de base du langage de programmation Java, concepts qui sont largement décrits dans [6] [73] [29] [35]. Cette section ne se veut pas une introduction au langage Java mais un rappel de certains concepts nécessaires à la compréhension de la suite de ce document. Cette section présente des concepts relatifs à l'approche objet dans Java et au modèle d'exécution Java.

L'approche objet dans Java est basée sur des concepts de *types de données*, de *classes*, d'*interfaces* et de *tableaux*, qui permettent au programmeur d'une application de définir ses propres abstractions. Java est également basé sur la notion d'*objet* qui est le représentant ou l'instance d'une abstraction définie par le programmeur. Java définit les *variables*, les *blocs de code* qui sont des suites d'instructions ou de traitements d'un programme et les *méthodes* qui sont des blocs de code qui peuvent être appelés par d'autres blocs de code.

Quant aux concepts relatifs au modèle d'exécution Java, ils concernent les *threads* et la *synchronisation*.

- ***Types de données***

Le type d'une donnée détermine la valeur que peut avoir la donnée et les opérations qui peuvent être effectuées sur cette donnée. Le langage Java utilise plusieurs types de données, tels qu'illustrés par la Figure 2-2. Il existe deux catégories principales de types de données Java : les *types primitifs* et les *types références*.

Les types primitifs comprennent les types entiers signés *byte*, *short*, *int* et *long*, les types flottants *float* et *double*, le type caractère *char* et le type booléen *boolean*.

Les types références représentent des classes, des interfaces ou des tableaux Java, qui sont présentés dans les sections suivantes. Contrairement à une donnée de type primitif

qui représente une valeur, une donnée de type référence est une référence à une valeur (instance de classe) ou un ensemble de valeurs (tableau).

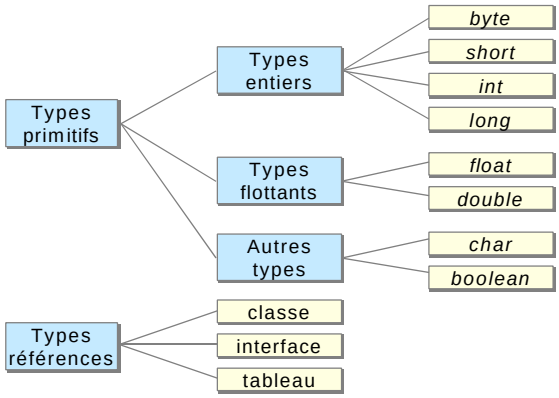

Figure 2-2. Types de données du langage Java

- ***Interface***

Une interface est un type défini par le programmeur. Une interface définit donc la valeur que peut avoir une donnée et les opérations applicables sur la donnée. Dans une interface, les méthodes sont spécifiées par leurs signatures mais ne sont pas mises en œuvre par un code.

- ***Classe***

Une classe est un type défini par le programmeur. Une classe détermine la valeur que peut avoir une *donnée* de la classe, les opérations qui peuvent être effectuées sur une *donnée* de la classe et, éventuellement, la mise en œuvre de ces opérations. La donnée d'une classe est appelée *objet* ou *instance de classe.*

Une classe est dite *abstraite* si certaines de ses opérations (méthodes) sont spécifiées mais non mises en œuvre. Dans ce cas, cette classe ne peut être instanciée directement mais à travers une de ses sous-classes.

- ***Objet***

Un objet, appelé également *instance de classe*, est une entité caractérisée par un *état* et par un *comportement*. L'état de l'objet représente l'ensemble des caractéristiques qui sont propres à l'objet et le comportement de l'objet représente l'ensemble des opérations applicables à l'objet. Ainsi, dans l'exemple de gestion des employés d'une entreprise, présenté dans la section 1.1.1 de ce chapitre :

- l'état d'un objet *employé* est constitué du nom, de l'adresse et du salaire de l'employé,
- et le comportement d'un objet *employé* est représenté par les opérations de changement d'adresse et d'augmentation de salaire.

L'état d'un objet est décrit par ses *variables* et son comportement est représenté par des *méthodes*.

- ***Variable***

Les caractéristiques qui décrivent l'état d'un objet sont appelées variables d'instance ou variables d'objet, tels que les *nom*, *adresse* et *salaire* d'un objet *employé*.

Il existe également des variables de classe qui sont des caractéristiques qui décrivent l'état d'une classe. Ces variables sont généralement utilisées pour représenter une caractéristique commune à toutes les instances de la classe. Une variable de classe de la classe des employés pourrait être le *salaire minimum* de tout objet *employé* de cette classe.

Une variable, d'instance ou de classe, est décrite par un type et un identificateur.

- ***Méthode***

Les opérations qui décrivent le comportement d'un objet sont appelées méthodes d'instance ou méthodes d'objet, telles que les opérations de *changement d'adresse* ou d'*augmentation de salaire* d'un objet *employé*.

Il existe également des méthodes de classe qui sont des opérations qui décrivent un comportement propre à la classe. Une méthode de classe est identifiée par le mot clé *static*.

Une méthode est décrite par sa signature (un nom et une liste d'arguments), le type du résultat retourné par la méthode si celle-ci retourne un résultat et le code de la méthode. Le code d'une méthode peut être soit un code Java soit un autre format de code. Si le code de la méthode est du code Java, la méthode est appelée *méthode Java*, sinon la méthode est dite *native*. Les méthodes natives sont généralement utilisées pour l'optimisation de certains traitements pour des architectures ciblées, tels que les traitements relatifs au réseau ou à l'interface graphique.

Une méthode, d'instance ou de classe, peut également être marquée synchronisée.

- ***Bloc de code***

Un bloc de code est une suite d'instructions délimitée par une accolade ouvrante et une accolade fermante. Un bloc de code peut être soit inclus dans une méthode Java soit défini, dans une classe, en dehors de toute méthode.

Un bloc de code de classe est un bloc qui est inclus dans une méthode de classe ou un bloc qui est défini en dehors de toute méthode mais précédé du mot clé *static*. Autrement, le bloc de code est un bloc de code d'instance.

- ***Héritage***

L'héritage de classe est un mécanisme de sous-typage qui permet de construire de nouvelles classes par spécialisation de classes existantes. Une *classe fille* qui hérite d'une *classe mère* est dite sous-classe de la classe mère. Une classe fille est une classe qui :

- d'une part, reprend et éventuellement redéfinit les caractéristiques (variables) et les opérations (méthodes) de la classe mère
- et d'autre part, peut spécifier de nouvelles caractéristiques et de nouvelles opérations qui lui sont propres.

La classe *Object* est la classe mère de toutes les classes Java : toute classe Java est implicitement sous-classe de la classe *Object*.

Par ailleurs, Java ne permet d'effectuer qu'un héritage simple entre les classes : une classe fille ne peut hériter que d'une seule classe mère. Pour permettre une forme d'héritage multiple, il est possible d'utiliser les interfaces. Une classe peut, en effet, hériter de plusieurs interfaces. On dit, dans ce cas, que la classe *implante* les interfaces. Une interface peut elle-même hériter de plusieurs interfaces. Une classe non abstraite qui implante une interface doit mettre en œuvre toutes les méthodes spécifiées par l'interface.

- ***Tableau***

Un tableau est une suite d'éléments de même type. Un tableau est également un objet d'une classe particulière qui hérite de la classe *Object*. Par exemple, un tableau d'entiers est une instance de la classe qui représente les « tableaux d'entiers » et un tableau de *Object* est une instance de la classe qui représente les « tableaux de *Object* ».

- ***Package***

Un *package* est une librairie qui regroupe des classes, des interfaces et d'autres packages (appelés sous-packages). Le regroupement de classes ou d'interfaces en packages permet de faciliter et de contrôler l'utilisation de ces classes et interfaces.

La machine Java de Sun Microsystems, le JDK, fournit plusieurs niveaux de packages, dont la racine commune est le package *java*. Le package *java.lang*, par exemple, rassemble les classes et interfaces les plus usuelles dont les classes *Object*, *Thread* et l'interface *Runnable*. Pour qu'une classe puisse utiliser les classes et interfaces fournies par un package, elle doit importer ce package. L'importation d'un package est faite par l'expression *import nom_du_package;*

Le nom d'un package apparaît dans le nom des classes et interfaces appartenant à ce package. Une classe ou interface *C* qui appartient a un package *P* a pour nom *P.C*. Ainsi, la classe *Object* est identifiée par *java.lang.Object*. Mais pour plus de simplicité

et lorsque cela ne prête pas à confusion, un nom de classe *P.C* peut tout simplement être raccourci en *C* lorsque le package *P* est importé.

- ***Thread***

Après avoir présenté les concepts relatifs à l'approche objet, intéressons-nous à présent à des concepts relatifs au modèle d'exécution de Java et qui sont le *thread* et la *synchronisation*.

Un thread est un flot d'exécution qui peut s'exécuter de façon concurrente avec d'autres flots d'exécution. Les threads qui s'exécutent dans un même environnement Java[1] partagent entre eux code et données. Un thread Java est une instance de la classe *Thread* ou l'instance d'une classe qui implante l'interface *Runnable*.

- ***Synchronisation***

Pour permettre à plusieurs threads d'accéder de façon cohérente à une ressource partagée, il est indispensable de synchroniser les opérations d'accès à cette ressource. Dans Java, la gestion de la synchronisation des accès concurrents aux ressources partagées se fait par des verrous. Le thread qui possède le verrou d'une ressource est le seul thread (dans le groupe des threads synchronisés sur la ressource) qui accède à la ressource. Java gère un verrou par objet et un verrou par classe.

Un thread prend le verrou d'un objet lorsqu'il exécute sur l'objet une méthode d'instance synchronisée ou un bloc de code d'instance synchronisé. Un thread rend le verrou d'un objet lorsqu'il quitte une méthode d'instance synchronisée ou un bloc de code d'instance synchronisé de l'objet.

Un thread prend le verrou d'une classe lorsqu'il exécute une méthode de classe synchronisée ou un bloc de code de classe synchronisé. Un thread rend le verrou d'une classe lorsqu'il quitte une méthode de classe synchronisée ou un bloc de code de classe synchronisé.

1.2. Abstraction d'un environnement homogène

L'environnement Java fournit l'abstraction d'un environnement homogène grâce à :

- la spécification d'une machine unique, la *machine virtuelle Java (JVM)*
- et un format unique de fichier de code exécutable Java, le format de fichier *.class*.

La spécification d'une machine virtuelle Java unique et son implantation sur diverses plates-formes permettent de voir les diverses plates-formes comme une seule et même plate-forme. Ainsi, quels que soient les systèmes d'exploitation et les architectures de

[1] Un environnement Java signifie ici une JVM, voir la section 2 de ce chapitre.

machines sous-jacentes, toutes les plates-formes auront l'apparence d'une même plate-forme[1].

D'autre part, le format de fichier de code exécutable Java, le fichier *.class*, est un format unique, portable et exécutable sur toute machine virtuelle Java. Ainsi, un code Java s'exécutant sur une plate-forme n'a pas besoin d'être porté ou recompilé pour être exécuté sur une autre plate-forme. Un même code Java est donc directement portable et exécutable sur diverses plates-formes, si celles-ci sont dotées d'une machine virtuelle Java.

1.3. API de l'environnement Java

Un des aspects importants de l'environnement Java est la richesse de ses librairies de classes Java, accessibles via l'interface de programmation d'applications (API: Application Programming Interface) [137] [33]. Cette API, illustrée par la Figure 2-3, propose divers outils pour faciliter la programmation de systèmes d'information variés et manipuler des bases de données, du texte, du son, des images, le réseau ou les architectures distribuées.

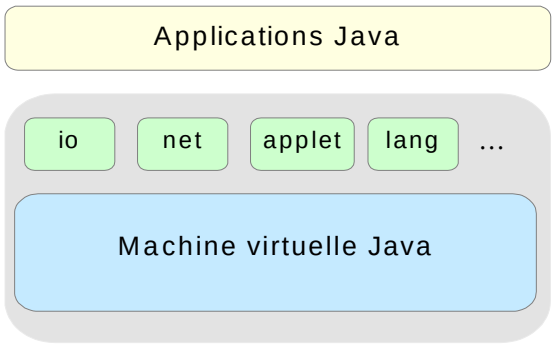

Figure 2-3. API de l'environnement Java

Les librairies de classes Java sont organisées en plusieurs packages, dont le package *java.net* qui contient des outils concernant le réseau, le package *java.io* qui propose des outils de manipulation d'entrées/sorties, le package *java.applet* qui fournit une interface d'accès aux navigateurs web et le package *java.lang* qui contient les classes et interfaces les plus usuelles. Le package *java.lang* contient, par exemple :

- la classe *Object* qui est la classe-mère de toutes les autres classes et interfaces,
- la classe *Class* qui représente les classes Java,
- la classe *String* qui représente les chaînes de caractères
- et la classe *Thread* ou l'interface *Runnable* qui représentent les flots d'exécution (threads) Java.

[1] La machine virtuelle Java est abordée plus en détail dans la section 2 de ce chapitre.

Un autre exemple de package est le package *java.io* qui contient, par exemple :

- la classe *InputStream* qui représente les flux de données entrantes, sous forme d'octets
- et la classe *OutputStream* qui représente les flux de données sortantes, sous forme d'octets.

Les flux de données peuvent représenter des flux de données réseau, si les flux sont connectés à une liaison réseau, ou des flux de données disque, si les flux sont reliés à un fichier.

Dans la suite, nous nous intéresserons plus particulièrement aux outils proposés par l'API Java qui sont dédiés à la mobilité et à la persistance des applications.

1.3.1. Java comme support pour la mobilité

Java a été conçu initialement pour les environnements et applications mobiles, qui se déplacent d'un site à un autre, à travers le réseau. C'est pour cette raison que l'API de Java intègre divers outils pour la programmation d'applications mobiles : des mécanismes pour la mobilité du code Java et des mécanismes pour la mobilité des données Java. Les expressions « code Java » et « classe Java » seront utilisées indifféremment, de même que les expressions « donnée Java » et « objet Java ».

- ***Mécanismes pour la mobilité du code Java***

L'environnement Java fournit un mécanisme de *chargement dynamique de classes* Java, décrit dans le chapitre 12 de la spécification du langage Java [73]. Le chargement d'une classe est l'opération qui consiste à construire, à partir du code exécutable d'une classe Java, les structures d'exécution qui décrivent cette classe dans la JVM. Le chargement de classes est dit dynamique car il peut être fait au cours de l'exécution des applications, lors de la première utilisation d'une classe.

Ce mécanisme permet au programmeur d'une application de définir ses propres politiques de chargement de classes Java, en spécialisant :

- le format de la classe à charger (fichier *.class*, fichier ZIP, fichier compressé),
- la localisation source de la classe (système de fichiers, réseau),
- le moyen de localiser la classe à charger dans la localisation source (le répertoire ou le fichier JAR dans lequel la classe doit être recherchée dans le cas d'une localisation dans un système de fichiers, la machine à contacter dans le cas d'une localisation via le réseau).

Le mécanisme de chargement de classe est représenté par la classe *java.lang.ClassLoader*. Il peut servir à transférer du code Java à travers le réseau, tel qu'illustré par la Figure 2-4. La classe *java.net.URLClassLoader* est une sous-classe de

la classe *java.lang.ClassLoader*, elle permet de charger des classes Java à partir de localisations identifiées par des URL.

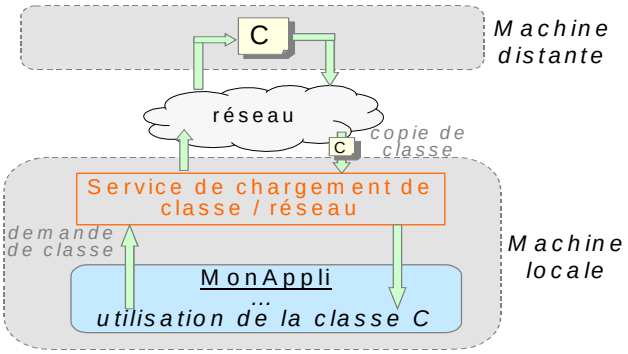

Figure 2-4. Chargement de classe à travers le réseau

Un autre mécanisme de mobilité des applications Java est l'*applet*, représentée par la classe *java.applet.Applet* [132]. Une applet est un code Java auquel fait référence une page HTML d'un serveur web. La localisation du code de l'applet peut être le serveur web lui-même ou un autre site. Lorsqu'un client web accède à une page HTML contenant une applet, le code de l'applet est chargé à partir de la localisation de l'applet, pour être exécuté sur le site du client.

- ***Mécanismes pour la mobilité des données Java***

L'environnement Java fournit un mécanisme de *sérialisation/dé-sérialisation* des objets Java [136]. La *sérialisation* d'un objet Java consiste à convertir la structure de données décrivant un objet en un flot d'octets. La Figure 2-5 illustre une sérialisation d'objet Java. La *dé-sérialisation* est le processus de conversion de la forme sérialisée d'un objet (flot d'octets) en une copie de l'objet. Un objet Java est sérialisable si sa classe ou une de ses super-classes implante l'interface *java.io.Serializable*.

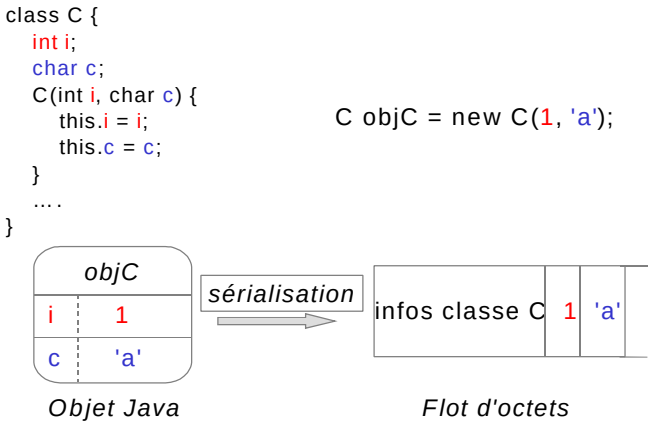

Figure 2-5. Sérialisation d'un objet Java

L'*externalisation* est une forme de sérialisation qui permet de contrôler plus finement les données écrites dans le flot d'octets. L'utilisation de l'externalisation au lieu de la sérialisation classique peut réduire le temps de sérialisation des objets Java jusqu'à 40% [133]. Un objet Java est externalisable si sa classe ou une de ses super-classes implante l'interface *java.io.Externalizable*.

Le mécanisme de sérialisation/dé-sérialisation est spécialisable et utilisable dans divers contextes. La sérialisation/dé-sérialisation peut en particulier être utilisée pour le transfert d'objets Java à travers le réseau. Ainsi, les données sérialisées sont envoyées d'une machine source vers une machine destination, à travers un flux réseau reliant ces deux machines.

1.3.2. Java comme support pour la persistance

En plus de la mobilité, l'environnement Java fournit des outils pour la programmation d'applications persistantes. Une application est dite persistante si ses composants (code et/ou données) subsistent après la fin de l'application. La mise en œuvre de la persistance d'une entité se traduit, d'une part, par la sauvegarde de cette entité sur un support persistant (disque) et, d'autre part, par la possibilité de reprendre cette entité à partir du support persistant pour sa réutilisation.

- ***Mécanismes pour la persistance du code Java***

Rendre un code Java persistant consiste, d'une part, à sauvegarder le code Java dans un fichier sur disque et, d'autre part, à reprendre et charger en mémoire le code Java à partir d'un fichier.

La sauvegarde d'un code Java peut se faire via la compilation Java. En effet, la compilation d'un programme source Java produit un code Java qui est directement stocké dans un fichier *.class*. Mais dans le cas où le code Java n'est pas obtenu par compilation mais par génération dynamique, la sauvegarde du code Java peut être faite en spécialisant le service de chargement de classes pour qu'il sauvegarde sur fichier toute classe chargée en mémoire.

De façon symétrique, la reprise d'un code Java à partir d'une copie persistante est tout simplement rendue possible en spécialisant le mécanisme de chargement de classes pour que les classes soient lues à partir du disque : c'est ce que fait le système de chargement de classes par défaut de la JVM.

- ***Mécanismes pour la persistance des données Java***

Le mécanisme de sérialisation/dé-sérialisation, qui est spécialisable pour divers besoins, peut également être utilisé pour la mise en œuvre de la persistance des données Java. La *sérialisation* Java peut servir à stocker des objets Java sur un fichier disque et la *dé-sérialisation* peut être utilisée pour restaurer les données préalablement

sauvegardées sur disque. Pour ce faire, les opérations de sérialisation/dé-sérialisation doivent tout simplement être spécialisées pour lire/écrire les données sérialisées dans un flux disque.

1.4. Conclusion

Java fournit l'abstraction d'un environnement homogène universel. Cette abstraction masque l'hétérogénéité des plates-formes sur lesquelles est implanté Java et les diverses plates-formes sont, de ce fait, vues comme une seule et même plate-forme. Ceci confère la caractéristique de portabilité au code et aux données Java.

Java est également une riche collection de librairies de classes. Les classes Java fournissent plusieurs outils pour la mobilité des applications et divers moyens de rendre une application Java persistante. Parmi ces outils, nous pouvons citer le service de chargement dynamique de classes qui peut être utilisé pour mettre en œuvre la mobilité ou la persistance d'un code Java. Il y a également le mécanisme de sérialisation/dé-sérialisation des objets, utilisé pour la mise en œuvre de la mobilité ou de la persistance des données Java.

Cependant, ces outils Java, qui sont nécessaires à la mise en œuvre de la mobilité et de la persistance des applications, ne suffisent pas pour rendre l'*exécution d'une application* mobile ou persistante. Une application qui s'exécute est constituée de code, de données et de flots d'exécution or, dans l'état actuel de Java et de la machine virtuelle, il n'y a pas de service pour la mobilité et la persistance des flots d'exécution.

2. Machine virtuelle Java

Cette présentation de la machine virtuelle Java commence, tout d'abord, par la définition des concepts introduits par la spécification de la machine virtuelle. Ce sont ces concepts qui seront, par la suite, utilisés pour la description de la construction de nos services de mobilité et de persistance.

2.1. Qu'est-ce que la machine virtuelle Java ?

La machine virtuelle Java (JVM : *Java Virtual Machine*) est dite virtuelle car elle représente une machine abstraite définie par une spécification. Pour exécuter un programme Java, il est nécessaire d'avoir une mise en œuvre concrète de cette spécification abstraite. La machine virtuelle Java désigne ainsi une des trois entités suivantes :

- une spécification abstraite,
- une mise en œuvre concrète
- et une instance d'exécution.

Cette section décrit la spécification abstraite de la machine virtuelle Java, illustre cette définition abstraite par des mises en œuvre concrètes puis définit une instance d'exécution de la machine virtuelle Java. Un exemple est donné en fin de section pour illustrer les différents concepts présentés.

2.2. Spécification de la machine virtuelle Java

La spécification abstraite de la machine virtuelle Java est décrite en détail par Tim Lindholm et Frank Yellin dans [85]. Cette spécification définit l'abstraction d'une machine de calcul dotée d'un ensemble d'instructions et permettant d'effectuer des calculs numériques, de contrôler l'accès à la mémoire, de gérer le contrôle des flots d'exécution et de fournir aux programmes un moyen d'accéder au matériel attaché au système.

La spécification de la machine virtuelle Java décrit le comportement de la machine virtuelle en termes de *types de données*, de *sous-systèmes* et de *structures de données d'exécution*. Ces composants décrivent l'architecture interne abstraite de la machine virtuelle Java. L'intérêt de ces composants est de définir un comportement externe commun aux différentes mises en œuvre de la machine virtuelle Java.

2.2.1. Types de données

La JVM manipule deux catégories de types : les *types primitifs* et les *types références*, présentés par le Tableau 2-1. Dans d'autres langages, le format et la taille des *types primitifs* dépendent de la plate-forme sur laquelle s'exécute le programme. Pour éviter cette dépendance au système sous-jacent, le couple langage Java/JVM spécifie la taille et le format de ses types primitifs, tels que décrits par le Tableau 2-1.

Ainsi, les types primitifs entiers signés *byte*, *short*, *int* et *long* sont représentés en complément à 2, sur respectivement 8 bits, 16 bits, 32 bits et 64 bits. Les types primitifs flottants *float* et *double* sont codés dans le format IEEE 754, sur respectivement 32 bits et 64 bits. Le type primitif *char* est un entier non signé sur 16 bits, représentant un caractère Unicode et le type primitif *boolean* est représenté par deux valeurs entières, l'entier *1* pour représenter la valeur *vrai* et l'entier *0* pour représenter la valeur *faux*. Enfin, le type primitif *returnAddress*, manipulé par la machine virtuelle et non présent dans le langage Java, représente l'adresse d'une instruction du programme exécuté par la JVM. Ce type est utilisé pour la mise en œuvre de la clause *finally* dans les programmes Java et, plus précisément, par les instructions de bytecode *jsr* et *jsr_w*[1]. Pour plus de détails sur la clause *finally*, consulter le chapitre 11 de *La spécification du langage Java* [73].

[1] Pour plus de détails sur la clause *finally*, consulter l'Annexe II. 9

Une seconde catégorie de types de la JVM est le *type référence* qui est soit un type classe, soit un type tableau, soit un type interface, représentant respectivement une référence vers une instance de classe, un tableau ou une instance de classe implantant une interface. Une valeur de ce type peut également être la valeur spéciale *null*, ne représentant ainsi une référence vers aucun objet.

	Type	Taille	Description
Types primitifs entiers	byte	8 bits	entiers signés en complément à 2
	short	16 bits	
	int	32 bits	
	long	64 bits	
Types primitifs flottants	float	32 bits	réels au format IEEE 754
	double	64 bits	
Autres types primitifs	char	16 bits	entier non signé représentant un caractère Unicode
	boolean	non spécifié	entier 1 pour *vrai* et entier 0 pour *faux*
	returnAddress	non spécifié	pointeur vers instruction
Types références	reference	non spécifié	référence vers une instance de classe ou un tableau

Tableau 2-1. Types de données dans la JVM

2.2.2. Sous-systèmes

En plus des types de données, la spécification de la machine virtuelle Java définit des sous-systèmes. Les principaux sous-systèmes de la JVM sont illustrés par la Figure 2-6. Chaque JVM a un *sous-système de chargement de classes*, qui est un mécanisme qui charge les classes et interfaces à partir d'un nom. Chaque JVM possède également un *sous-système d'exécution*, qui est un mécanisme responsable d'exécuter le code des méthodes des classes chargées.

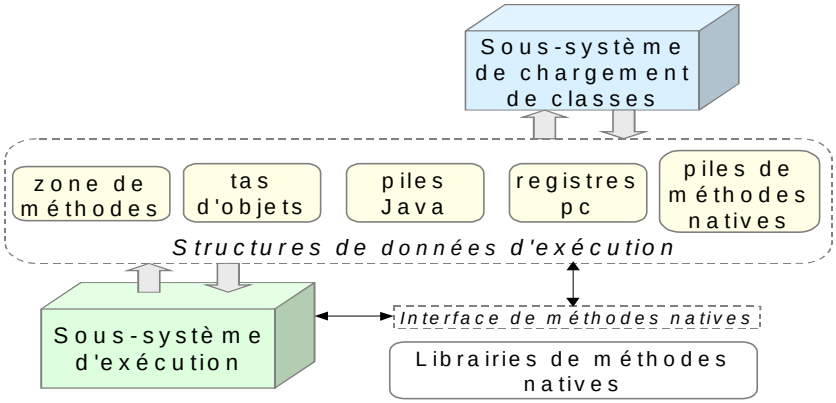

Figure 2-6. Architecture de la machine virtuelle Java

- ***Sous-système de chargement de classe***

Chaque machine virtuelle Java possède un *sous-système de chargement de classes* ou *chargeur de classes*. Ce sous-système a pour objectif de charger en mémoire, dans la zone de méthodes de la JVM (voir section 2.2.3 de ce chapitre), les classes et interfaces utilisées et décrites par des noms complets. Plusieurs chargeurs de classes peuvent cohabiter au sein d'une même JVM. Ainsi, le programmeur d'une application peut définir et utiliser ses propres chargeurs de classes.

- ***Sous-système d'exécution***

Chaque machine virtuelle Java possède un *sous-système d'exécution*, qui est le mécanisme responsable de l'exécution du bytecode contenu dans les méthodes des classes chargées. De même que la machine virtuelle Java, le sous-système d'exécution peut prendre une des trois formes suivantes : une spécification abstraite, une mise en œuvre concrète ou une instance d'exécution. La spécification abstraite décrit le comportement d'un sous-système d'exécution en termes d'instructions de la JVM. La mise en œuvre concrète peut être une mise en œuvre logicielle, matérielle ou une combinaison des deux. Une mise en œuvre concrète d'un sous-système d'exécution peut, par exemple, être basée sur un interprète de bytecode, un compilateur à la volée, une exécution native ou une combinaison de ces différentes techniques. Enfin, une instance d'exécution d'un sous-système d'exécution est un thread Java.

- **Spécification abstraite : Ensemble d'instructions**

Un compilateur Java traduit un programme écrit dans le langage source Java vers un langage cible appelé *bytecode*. Mais contrairement à la plupart des compilateurs dont le langage cible est le langage d'un processeur physique, le compilateur Java a pour cible une machine virtuelle. Ainsi, le bytecode est "compris" par toute machine de même "architecture" : par toute machine virtuelle Java.

Le bytecode d'une méthode Java est une suite d'instructions de la machine virtuelle Java. Chaque instruction est constituée d'un *opcode* seul ou suivi d'un ou plusieurs *opérandes*. Un opcode est codé sur un octet (un *byte*). Un opcode spécifie l'opération à effectuer et indique lui-même s'il est suivi ou non d'opérandes et de quels types d'opérandes il s'agit. Par exemple, les opcodes d'addition *iadd*, *ladd*, *fadd* et *dadd* sont respectivement suivis de deux opérandes de type *int*, *long*, *float* et *double.*

Pour des raisons d'indépendance de la plate-forme sous-jacente, le choix et la spécification des instructions de la machine virtuelle Java ont porté sur une approche basée sur une structure de pile, au lieu d'une approche basée sur des registres. Ce choix a été motivé par la facilité de la mise en œuvre d'une approche orientée-pile sur des architectures variées. La Figure 2-7 illustre le fonctionnement de l'instruction d'addition de deux entiers dans la JVM. Avant l'exécution de l'instruction *iadd*, le sommet de la pile contient deux valeurs *val1* et *val2* de type *int*. Pour exécuter l'instruction *iadd*, le sous-système d'exécution de la JVM dépile les valeurs *val1* et *val2*, les additionne et empile le résultat *res* qui est de type *int*[1].

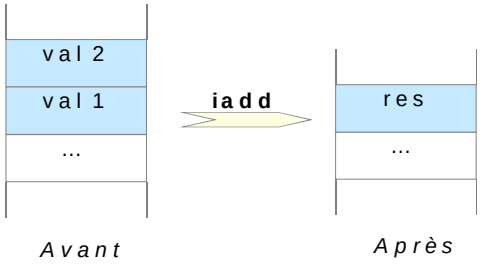

Figure 2-7. Addition de deux entiers dans la JVM

- **Mise en œuvre concrète : Différentes techniques**

Différentes techniques sont utilisées pour la mise en œuvre d'un sous-système d'exécution : l'interprétation, la compilation à la volée (JIT : Just-In-Time), l'optimisation adaptative ou l'exécution native.

Les premières machines virtuelles Java utilisaient un sous-système d'exécution appelé interprète Java. A chaque appel de méthode Java, l'interprète Java traduit une à une les instructions de bytecode de cette méthode en traitements natifs nécessaires à l'exécution des instructions. L'avantage de cette technique est qu'elle peut être aisément portée sur différentes plates-formes. Son inconvénient est sa lenteur d'exécution, en particulier lorsqu'elle est comparée au temps d'exécution d'un code natif.

[1] Pour plus de détails sur les instructions de la JVM, consulter l'Annexe II.

La seconde génération de JVM utilisait un compilateur Java JIT qui, à chaque premier appel de méthode Java, compilait celle-ci en une méthode native et exécutait le code natif. Toute invocation ultérieure de cette méthode se basait alors sur la version native. L'avantage de cette technique est qu'elle ramène le temps d'exécution d'un code Java au temps d'exécution d'un code natif. Son inconvénient est le surcoût que peut induire la compilation à la volée. En effet, il est parfois plus coûteux de compiler du code Java et d'exécuter le code natif résultant que de directement interpréter ce même code Java, surtout lorsque celui-ci n'est exécuté que rarement[1].

La technique d'optimisation adaptative pallie à ce problème en combinant les deux techniques précédentes : l'interprétation de bytecode et la compilation JIT. L'optimisation adaptative se base initialement sur l'interprétation de bytecode, tout en surveillant l'exécution du programme Java au cours de son exécution. Cette surveillance permet de construire des informations relatives au comportement du programme, telles que les méthodes Java les plus fréquemment appelées. Ces informations sont ensuite utilisées pour choisir les méthodes qui sont compilées en code natif puis exécutées en version native.

Une autre mise en œuvre possible du sous-système d'exécution de la JVM est celle basée sur l'exécution native. Avant le lancement de l'exécution d'une application, le code de l'application est compilé pour produire un code dans le langage natif de la machine sur laquelle va s'exécuter l'application. Ainsi, le temps d'exécution de l'application est ramené au temps d'exécution du code natif et le surcoût dû à la compilation lors de l'exécution est annulé.

- **Instance d'exécution : Thread Java**

La machine virtuelle Java supporte l'exécution de plusieurs *threads* (processus légers) à la fois. Le rôle d'un thread est d'*exécuter* le code d'une application : soit du bytecode, soit des méthodes natives (voir section 2.2.3 de ce chapitre). Chaque thread qui exécute une application Java est une instance séparée du sous-système d'exécution de la JVM.

2.2.3. Structures de données d'exécution

La spécification de la machine virtuelle Java définit des types de données et des sous-systèmes mais également des structures de données d'exécution. Les principales structures de données d'exécution de la JVM sont illustrées par la Figure 2-6. Lorsqu'une JVM exécute un programme, elle a besoin de mémoire pour stocker différentes entités, telles que les instructions et autres informations extraites des classes chargées, les objets instanciés par le programme, les paramètres des méthodes exécutées

[1] 80-20 : 80% du temps est passé dans l'exécution de 20% du code

par le programme, leurs valeurs de retour, leurs variables locales et les résultats de leurs calculs intermédiaires. Pour stocker ces informations, la mémoire utilisée par la JVM est organisée en plusieurs structures de données d'exécution : la *zone de méthodes*, le *tas d'objets*, les *piles Java*. Des *piles de méthodes natives* sont également utilisées pour l'exécution des méthodes natives appelées par des programmes Java.

- ***Zone de méthodes***

Chaque instance de machine virtuelle Java possède une *zone de méthodes*. A chaque fois qu'une classe est chargée, celle-ci est placée dans la zone de méthodes. Cette zone est partagée par tous les threads de la machine virtuelle. Ainsi, tout thread s'exécutant dans la machine virtuelle peut accéder à toute classe chargée dans cette même machine virtuelle.

Dans la zone de méthodes, les informations relatives à chaque classe comprennent, entre autres informations, les variables de classe (variables marquées *static*), les informations sur les méthodes de la classe et le *constant pool*.

Les informations sur une méthode sont son nom, le type de son résultat, le nombre et les types de ses paramètres, ses modificateurs (*public*, *private*, *protected*, *static*, *final*, *synchronized*, *native* ou *abstract*)[1]. Des informations supplémentaires décrivent une méthode non *abstract* et non *native*, telles que le bytecode de la méthode, la table d'exceptions, la taille de la pile d'opérandes et de la table des variables locales.

Le *constant pool* d'une classe est une table de symboles pour les constantes utilisées par la classe. Une constante est utilisée par une classe si elle participe à la définition de la classe, des variables de classe ou d'instance, des descripteurs ou du code des méthodes. Ces constantes peuvent être, soit des valeurs de chaînes de caractère, de constantes entières ou flottantes, soit des noms symboliques de classes, d'interfaces, de méthodes ou de variables de classe ou d'instance.

- ***Tas d'objets***

Chaque instance de machine virtuelle Java possède un *tas d'objets*. A chaque fois qu'une instance de classe ou de tableau est créée, celle-ci est placée dans le tas d'objets. Cette zone est partagée par tous les threads de la machine virtuelle. Ainsi, tout thread s'exécutant dans la machine virtuelle peut accéder à tout objet (instance ou tableau) créé dans cette même machine virtuelle.

La représentation d'un objet doit contenir les valeurs des variables d'instance (données d'instance) ainsi que la possibilité d'accéder aux informations de la classe de

[1] Voir le chapitre 8 de *La spécification du langage Java* [73].

l'objet (un pointeur vers la zone de méthodes). La Figure 2-8 illustre une représentation possible d'un objet dans le tas d'objets.

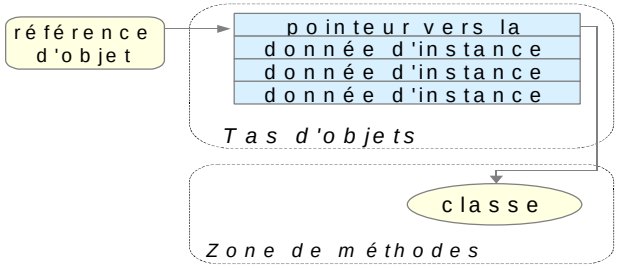

Figure 2-8. Représentation d'une instance de classe dans le tas d'objets

De même que les instances de classe, les tableaux sont créés dans le tas d'objets. Dans le cas d'un tableau, les données d'instances contiennent la taille du tableau et les valeurs de ses éléments. Une classe est associée à chaque tableau, via un pointeur vers la zone de méthodes. La Figure 2-9 illustre la représentation d'un tableau de deux entiers dans le tas d'objets.

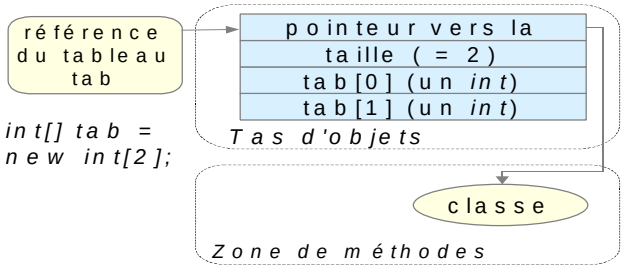

Figure 2-9. Représentation d'un tableau dans le tas d'objets

- ***Pile Java***

Une pile Java est associée à chaque thread, à la création du thread. La pile Java d'un thread décrit l'état des appels de méthodes Java effectués par le thread (méthodes non natives). L'état d'un appel de méthode Java est représenté par une zone appelée *frame Java*.

- ***Frame Java***

Un nouveau *frame* Java est créé et empilé sur la pile Java d'un thread à chaque fois que le thread invoque une méthode Java. Ce *frame* est dépilé lors de la terminaison de la méthode associée. Un *frame* Java inclut les paramètres avec lesquels la méthode a été appelée, le résultat retourné par la méthode (s'il y en a), les variables locales à la

méthode et les résultats des calculs intermédiaires effectués lors de l'exécution de la méthode.

Un *frame* Java comprend les structures de données illustrées par la Figure 2-10 :

- Une *table de variables locales*, contenant les paramètres de la méthode et les variables locales à la méthode. Dans une table de variables locales, les variables sont indicées à partir de 0 et l'accès à une variable locale se fait par son indice.
- Une *pile d'opérandes*, contenant les résultats des calculs intermédiaires effectués lors de l'exécution de la méthode. La manipulation des valeurs sur la pile d'opérandes se fait par les deux opérations d'empilement en sommet de pile et dépilement de la tête de pile.
- Des informations supplémentaires telles qu'un pointeur vers le descripteur de la méthode Java associée au frame Java, un pointeur vers le *constant pool* de la classe de la méthode, un pointeur vers la table d'exceptions de la méthode ou des informations utilisées lors de la terminaison de la méthode telles qu'un pointeur vers le *frame Java* précédent.

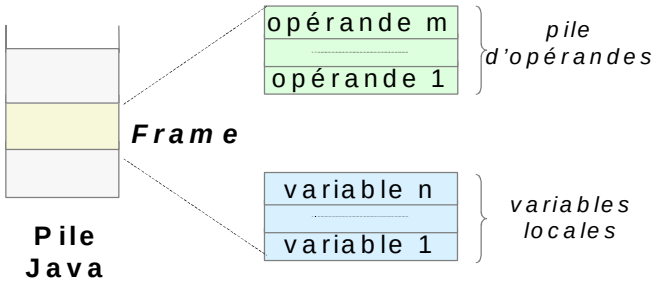

Figure 2-10. Pile Java et frame Java

- ***Registre PC***

A chaque thread est associé un registre compteur ordinal, *PC* (Program Counter). Lorsque le thread exécute une méthode Java, le registre PC contient l'adresse de la prochaine instruction de bytecode exécutée par le thread. Lorsque le thread exécute une méthode native, la valeur du registre PC est indéfinie.

- ***Méthodes natives***

Un programme Java peut utiliser deux sortes de méthodes : les méthodes Java et les méthodes natives. Une méthode Java est écrite dans le langage Java, compilée vers du bytecode puis stockée dans un fichier (fichier *.class* de la classe de la méthode). Une méthode native est généralement écrite dans un autre langage que Java, tel que le langage C ou le langage d'assembleur. Cette méthode est ensuite compilée vers le code natif de la machine sous-jacente puis stockée dans une librairie qui est chargée

dynamiquement lors de l'invocation de la méthode native par un programme Java. Contrairement aux méthodes Java, les méthodes natives sont dépendantes du système sous-jacent. L'interface JNI (Java Native Interface) permet de faire interagir du code Java et du code natif, par l'invocation d'un code natif à partir d'un code Java ou inversement [84].

- ***Pile de méthodes natives***

En plus des structures de données décrites précédemment, et définies par la spécification de la JVM, une application Java utilise d'autres structures de données pour l'exécution de méthodes natives. Lorsqu'un thread Java invoque une méthode native, il laisse sa pile Java de côté et utilise une autre pile appelée *pile de méthodes natives*. La structure de cette pile de méthodes natives n'est pas définie par la spécification de la JVM, elle varie en fonction du système sous-jacent considéré.

La Figure 2-11 illustre les piles associées à un thread, à un instant donné de son exécution. Le thread effectue tout d'abord deux appels imbriqués à des méthodes Java pour lesquelles deux frames Java, A puis B, ont été empilés sur la pile Java. La seconde méthode Java (associée au frame B) appelle une méthode native pour laquelle le frame C est empilé sur la pile de méthodes natives. Cette méthode native appelle une autre méthode native, associée au frame D, qui elle-même appelle une méthode Java, associée au frame E empilé sur la pile Java du thread.

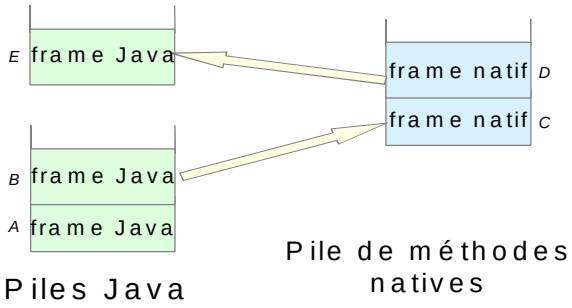

Figure 2-11. Piles d'un thread invoquant des méthodes Java et des méthodes natives

2.3. Mises en œuvre de la machine virtuelle Java

Des mises en œuvre concrètes de la machine virtuelle Java existent sur plusieurs plates-formes et sont soit des mises en œuvre entièrement logicielles, soit une combinaison de mises en œuvre logicielle et matérielle.

La mise en œuvre de JVM la plus répandue est le *Java Development Kit* (JDK) de Sun Microsystems [138], qui est une mise en œuvre de référence puisqu'elle est fournie par le créateur de Java. Le sous-système d'exécution du JDK est basé sur un interprète

Java et un compilateur Java JIT. Le JDK est utilisable sur les plates-formes Linux, Solaris, Mac OS et Windows (NT et 95/98/2000).

Parmi les autres mises en œuvre les plus connues figurent *Kaffe*, *picoJava*, la *KVM* ou la *Java Card*.

Kaffe est une mise en œuvre de JVM distribuée en logiciel libre [75]. Le sous-système d'exécution de Kaffe est basé sur un interprète Java, un compilateur Java à la volée (JIT) ou l'exécution native. Kaffe est utilisable sur plusieurs plates-formes, telles que Linux, Solaris et Windows.

Le microprocesseur *picoJava*, fourni par Sun Microsystems, est une mise en œuvre partiellement matérielle de la JVM [144]. En effet, ce microprocesseur a pour langage machine les instructions de bytecode elles-mêmes.

La *K Virtual Machine* (KVM) est une machine virtuelle Java pour des terminaux physiques minimaux, dotés d'une ressource mémoire réduite, tels que les téléphones et PDA [139]. Le sous-système d'exécution de cette machine virtuelle est basé sur un interprète Java. La KVM présente certaines restrictions par rapport au langage Java, aux API Java et à la spécification de la JVM : les types *float* et *double* et la plupart des sous-classes de la classe *java.lang.Error* ne sont pas supportés.

La *Java Card* est une machine virtuelle Java embarquée sur des cartes à puce [134]. La Java Card ne prend en compte qu'un sous-ensemble du langage Java et de la machine virtuelle Java. Elle ne supporte, par exemple, pas le chargement dynamique de classe, les threads multiples, les types *char*, *double*, *float* et *long*, ni certaines instructions de bytecode. Le sous-système d'exécution de cette machine virtuelle est basé sur un interprète Java.

2.4. Instance d'exécution de la machine virtuelle Java

Une application Java s'exécute au sein d'une instance d'exécution de la machine virtuelle Java. Lorsqu'une application Java est lancée, une instance d'exécution de machine virtuelle Java est créée. Une instance d'exécution de JVM commence l'exécution d'une application en appelant la méthode *main* d'une classe de cette application. La méthode *main* sert de point d'entrée pour le lancement du thread initial de l'application. Par la suite, d'autres threads peuvent être créés par le thread initial. Lorsque l'application se termine, autrement dit lorsque tous ses threads non démons meurent, l'instance d'exécution de la JVM se termine.

2.5. Exemple d'illustration

Pour illustrer les concepts présentés précédemment, voici un exemple de programme Java, son équivalent en bytecode et les structures d'exécution de la JVM nécessaires à l'exécution de ce programme.

- ***Code source Java***

La Figure 2-12 présente le programme Java d'une classe appelée *IntValue*. Les objets de cette classe sont dotés de valeurs entières sur lesquelles les opérations suivantes peuvent être appliquées :

- A chaque objet de la classe *IntValue* est associée une valeur entière stockée dans la variable *value*.
- La classe *IntValue* propose un constructeur qui crée et initialise une nouvelle instance de la classe.
- Les méthodes *getIntValue*, *setIntValue*, *add* et *sub* sont respectivement utilisées pour renvoyer ou affecter la valeur entière associée à un objet et additionner ou soustraire une valeur à la valeur entière associée à un objet. Ici, la méthode *sub* est mise en œuvre en se basant sur la méthode *add*.

Quant à la méthode *main*, elle représente la méthode principale appelée lors du lancement du programme Java. Dans cet exemple, la méthode *main* crée un objet de la classe *IntValue* en initialisant sa valeur entière à *5* puis appelle la méthode *sub* sur cet objet avec l'entier *2* pour argument.

```
class IntValue {

// Variable
int value;

// Constructor
IntValue(int initialValue) {
        value = initialValue;
}

// Methods
int getIntValue() {
        return value;
}

void setIntValue(int newValue)
{
        value = newValue;
}
```

```
void add(int plus) {
0       value =
1           value
2           + plus;
}

int sub(int minus) {
0       add(-minus);
}

// Main method
public static void main(
        String[] args) {
        IntValue val;

0       val = new IntValue(5);
1       val.sub(2);
}

}
```

Figure 2-12. Exemple de code Java

- ***Bytecode***

La compilation du programme Java *IntValue* produit un fichier *IntValue.class*. Le bytecode contenu dans le fichier *IntValue.class* peut être visualisé en utilisant le désassembleur *javap* [130]. La Figure 2-13 donne le bytecode de chaque méthode ou constructeur de la classe *IntValue*.

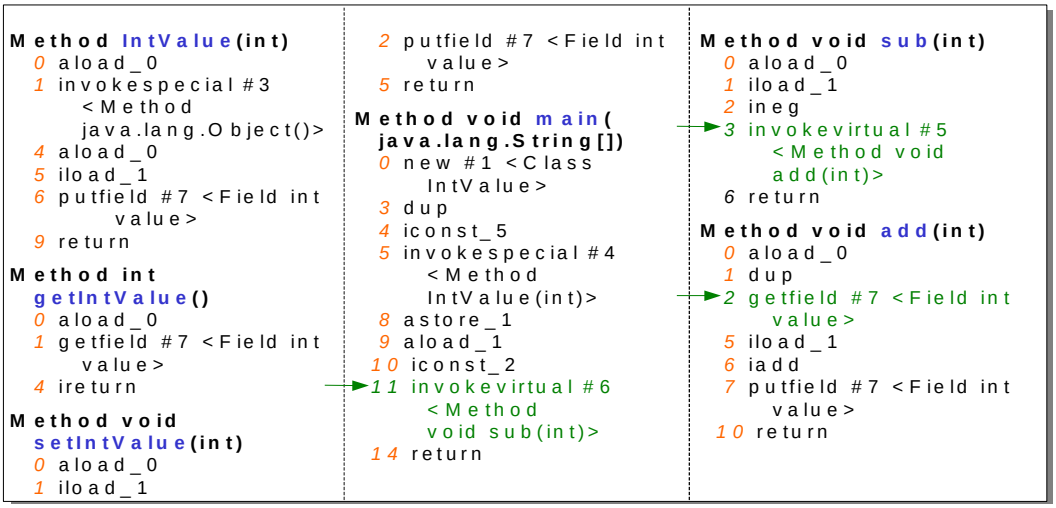

Figure 2-13. Exemple de bytecode

La méthode *getIntValue*, par exemple, comprend trois instructions de bytecode :

- D'abord, l'instruction *aload_0* qui lit la référence Java de l'objet sur lequel est appliquée la méthode *getIntValue*.
- Puis l'instruction *getfield* qui lit la valeur de la variable *value* de l'objet sur lequel est appliquée la méthode *getIntValue*.
- Et enfin l'instruction *ireturn* qui retourne la valeur de la variable *value* de l'objet sur lequel est appliquée la méthode *getIntValue*.

De plus, dans le bytecode présenté par la Figure 2-13, chaque instruction est précédée de la valeur du registre PC qui lui correspond. Cette valeur représente une valeur relative par rapport au début du code de la méthode considérée. Ainsi, dans la méthode *getIntValue*, la valeur relative du PC est :

- *0* pour l'instruction *aload_0*,
- *1* pour l'instruction *getfield* (car l'instruction de bytecode précédente - *aload_0* - est codée sur 1 octet)
- et *4* pour l'instruction *ireturn* (car l'instruction de bytecode *aload_0* est codée sur 1 octet et l'instruction *getfield* est codée sur 3 octets)[1].

- ***Exécution et structures d'exécution***

Lorsque l'exécution du programme Java *IntValue* est lancée, un thread Java est créé et une pile Java lui est associée. Ce thread appelle tout d'abord la méthode *main*, qui appelle la méthode *sub* qui elle-même appelle la méthode *add* (voir la Figure 2-12). Dans la suite, nous décrirons l'état d'exécution du thread lorsque son exécution arrive :

- A la ligne *1* de la méthode *main*, à la ligne *0* de la méthode *sub* et à la ligne *1* de la méthode *add* (voir la Figure 2-12).

[1] Pour plus de détails sur les instructions de bytecode, consulter l'Annexe II.

- A une valeur de PC de *11* dans la méthode *main*, de *3* dans la méthode *sub* et de *2* dans la méthode *add* (voir la Figure 2-13).

Le thread commence tout d'abord par appeler la méthode *main* et un premier frame Java est associé à cet appel de méthode et est empilé sur la pile Java du thread. Ce frame a deux variables locales : le paramètre de la méthode identifié par *args* (instance de la classe *String*) et la variable locale à la méthode identifiée par *val* (instance de la classe *IntValue*). Initialement, la variable locale *args* est à *null* et la variable *val* n'est pas initialisée. De plus, la pile d'opérandes du frame est vide, telle qu'illustrée par l'étape (a) de la Figure 2-14.

L'exécution de la première instruction de bytecode de la méthode *main*, l'instruction *new*, crée une instance de la classe *IntValue* et empile sa référence Java sur la pile d'opérandes du frame (étape (b) de la Figure 2-14). L'exécution de l'instruction *dup* provoque la duplication de la valeur en sommet de pile d'opérandes (étape (c) de la Figure 2-14). L'instruction *iconst_5* empile la valeur entière 5 sur la pile d'opérandes (étape (d) de la Figure 2-14). Les deux valeurs en sommet de pile d'opérandes, la référence Java et l'entier 5, sont en fait les paramètres du constructeur de la classe *IntValue*, constructeur appelé par l'instruction *invokespecial*. Le constructeur *IntValue* prend, en effet, deux paramètres : l'objet sur lequel porte le constructeur et un entier. L'appel au constructeur provoque alors l'exécution du constructeur puis le dépilement de ses paramètres (étape (e) de la Figure 2-14). L'instruction *astore_1* dépile la référence Java en sommet de pile d'opérandes et la stocke dans la variables locale d'indice 1, la variable *val* (étape (f) de la Figure 2-14). L'instruction *aload_1* lit ensuite la référence Java stockée dans la variable locale d'indice 1 et l'empile sur la pile d'opérandes (étape (g) de la Figure 2-14). Et l'instruction *iconst_2* empile la valeur entière 2 sur la pile d'opérandes (étape (h) de la Figure 2-14). Les deux valeurs en sommet de pile d'opérandes, la référence Java et l'entier 2, représentent les paramètres de la méthode *sub*, appelée par l'instruction *invokevirtual*. En effet, la méthode d'instance *sub* a pour paramètres la référence de l'objet sur lequel porte la méthode et un entier.

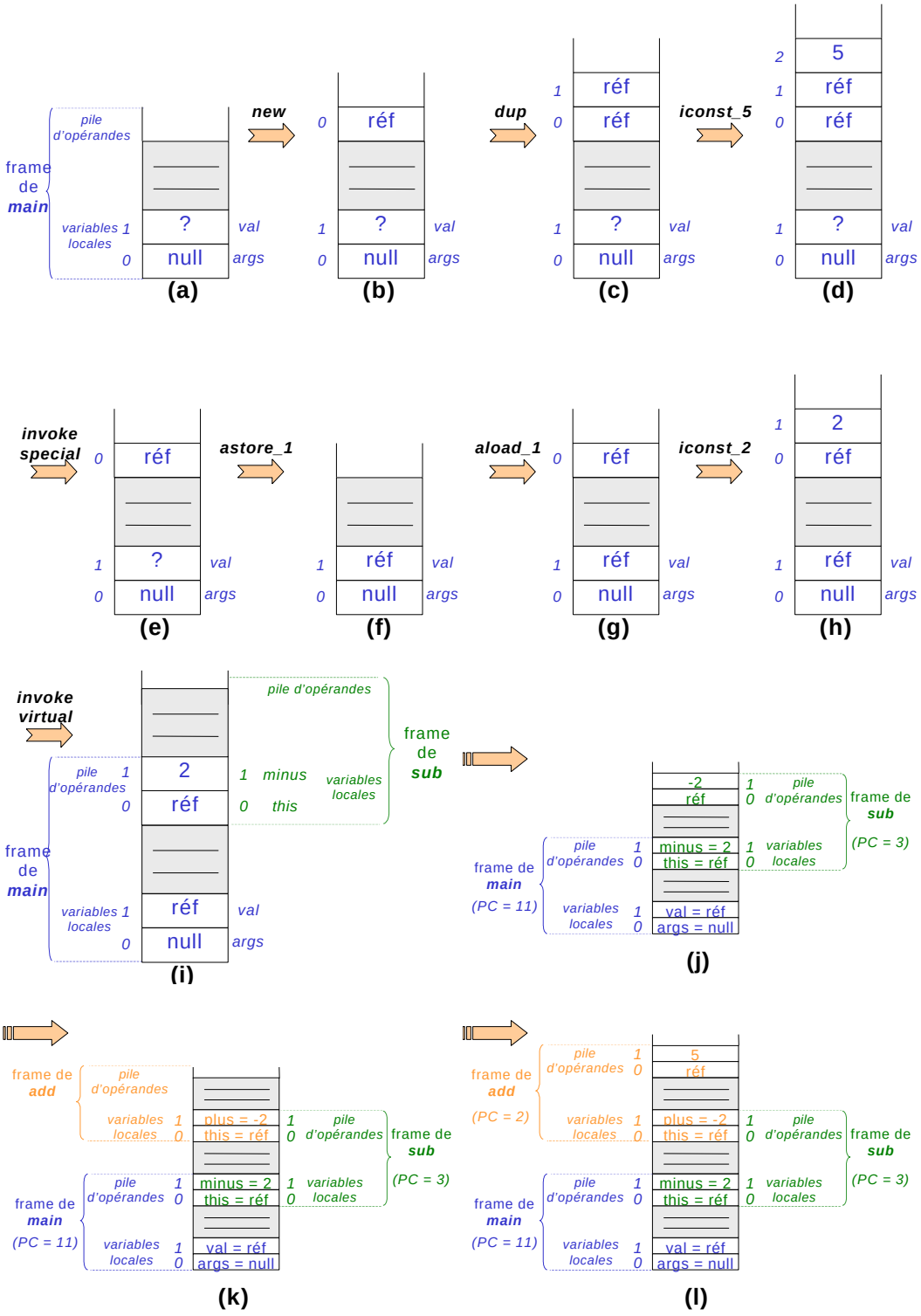

Figure 2-14. Etapes d'exécution du thread

L'instruction *invokevirtual* provoque l'empilement d'un second frame Java sur la pile Java du thread (étape (i) de la Figure 2-14). Ce frame a deux variables locales : l'objet sur lequel porte l'appel de la méthode *sub*, identifié par *this* (instance de la classe *IntValue*) et le paramètre de la méthode identifié par *minus* (de type *int*). Ici, les

variables locales du frame de la méthode *sub* correspondent à la pile d'opérandes du frame de la méthode *main*.

Lorsque l'exécution de la méthode *sub* arrive à l'instruction de bytecode correspondant au registre PC relatif *3* (voir Figure 2-14), la pile d'opérandes du frame est constituée des deux valeurs suivantes (étape (j) de la Figure 2-14) :

- la référence de l'objet sur lequel va porter l'appel de la méthode *add* et qui correspond à la référence *this*
- et le paramètre utilisé pour l'appel de la méthode *add* et qui correspond à l'entier *–2*.

Ces deux valeurs ont été respectivement empilées sur la pile d'opérandes du frame par :

- l'instruction *aload_0* qui lit la valeur de la variable locale d'indice *0* et l'empile sur la pile d'opérandes
- et les deux instructions successives *iload_1* et *ineg* qui calculent la négation de l'entier contenu dans la variable locale d'indice *1* et l'empilent sur la pile d'opérandes.

Au PC relatif *3*, l'instruction *invokevirtual* provoque l'empilement d'un troisième frame Java sur la pile Java du thread (étape (k) de la Figure 2-14). Ce frame a deux variables locales : l'objet sur lequel porte l'appel de la méthode *add*, identifié par *this* (instance de la classe *IntValue*) et le paramètre de la méthode identifié par *plus* (de type *int*). De même que pour les frames précédents, les variables locales du frame de la méthode *add* correspondent à la pile d'opérandes du frame de la méthode *sub*.

Lorsque l'exécution de la méthode *add* arrive à l'instruction de bytecode correspondant au registre PC relatif *2* (voir la Figure 2-13), la pile d'opérandes du frame est constituée des deux valeurs suivantes (étape (l) de la Figure 2-14) :

- la référence de l'objet dont la variable d'instance *value* va être modifiée
- et la valeur actuelle de la variable d'instance *value* de l'objet et qui correspond à l'entier *5*.

Ces deux valeurs ont été respectivement empilées sur la pile d'opérandes du frame par :

- l'instruction *aload_0* qui lit la valeur de la variable locale d'indice *0* et l'empile sur la pile d'opérandes
- et les deux instructions suivantes *dup*[1] et *getfield* qui dupliquent la référence à l'objet, lisent la valeur de sa variable *value* et l'empilent sur la pile d'opérandes.

[1] La duplication de la référence Java est ici nécessaire à l'exécution ultérieure de l'instruction *putfield*.

3. Conclusion

Dans ce chapitre, nous avons tout d'abord introduit Java en tant que langage de programmation, collection d'API riches et abstraction d'environnement homogène. Nous avons, par la suite, décrit la machine virtuelle Java avec ses systèmes et structures de données d'exécution.

Dans le chapitre suivant, nous expliquerons comment nous avons utilisé les systèmes de la JVM et comment nous avons extrait certaines informations fournies par les structures de données d'exécution de la JVM pour concevoir et mettre en œuvre nos services de mobilité et de persistance des threads Java.

CHAPITRE 3 - MISE EN ŒUVRE DE NOS SERVICES DE MOBILITE ET DE PERSISTANCE

Après avoir introduit l'environnement Java et la machine virtuelle Java dans le chapitre précédent, voici une description de la mise en œuvre de nos services.

Ce chapitre commence, tout d'abord, par présenter les points communs entre la mobilité et la persistance des threads. Il décrit ensuite notre extension de la machine virtuelle Java avec des mécanismes de capture et de restauration de l'état d'exécution des threads [22] [23]. Ces mécanismes sont utilisés comme base de construction de services de plus haut niveau : des services de mobilité et de persistance des threads Java [24] [25].

1. Points communs entre la mobilité et la persistance des threads

Nous rappelons que nos choix de conception ont porté sur :

- l'environnement Java, puisqu'il fournit l'abstraction d'un environnement homogène sur des plates-formes hétérogènes
- et le thread (flot d'exécution) comme entité mobile ou persistante, pour une mobilité et une persistance fortes.

Cette section décrit la mobilité et la persistance des threads Java et présente les points communs entre ces deux fonctionnalités.

1.1. Mobilité des threads

Rendre un thread Java mobile consiste à transférer l'exécution du thread d'un site source vers un site destination. Ce transfert de l'exécution se traduit par deux types de traitements : des traitements qui s'effectuent sur le site source et des traitements qui s'effectuent sur le site destination.

- ***Sur le site source***

1. Suspendre momentanément l'exécution du thread. Cette suspension est nécessaire pour éviter que l'exécution du thread n'évolue au cours de la capture de l'état.
2. *Capturer* l'état courant de l'exécution du thread.
3. Arrêter définitivement l'exécution suspendue du thread ou la poursuivre, selon le type de mobilité désiré (migration, clonage à distance).
4. Envoyer l'état capturé du site source vers le site destination.

- ***Sur le site destination***

1. Recevoir l'état d'exécution.
2. *Restaurer* l'état d'exécution reçu.
3. Lancer l'exécution du thread restauré. Ce thread commencera son exécution au point où elle a été interrompue au moment de la capture.

1.2. Persistance des threads

La persistance d'un thread est basée sur deux opérations : la sauvegarde du thread puis la reprise du thread.

- ***Sauvegarde***

1. Suspendre l'exécution du thread momentanément pour éviter que l'exécution du thread n'évolue au cours de la capture de l'état.
2. *Capturer* l'état courant de l'exécution du thread.

3. Arrêter définitivement l'exécution suspendue du thread ou la poursuivre, selon l'utilisation souhaitée.
4. Stocker l'état capturé sur un support persistant (fichier sur disque).

- ***Reprise***

1. Lire l'état d'un thread à partir d'un support persistant.
2. *Restaurer* l'état d'exécution du thread à partir de l'état lu.
3. Lancer l'exécution du thread restauré. Cette exécution commencera au point où l'exécution a été interrompue au moment de la sauvegarde.

Ainsi, la mobilité et la persistance des threads sont toutes deux basées sur des mécanismes communs : la capture et la restauration de l'état d'exécution des threads.

La *capture* de l'état d'exécution d'un thread Java consiste à extraire les informations qui décrivent l'état courant de l'exécution du thread à partir des structures de données d'exécution de la JVM. Ces informations sont ensuite utilisées pour construire une structure de données propre au thread et qui décrit son état courant d'exécution. C'est cette structure de données qui est ensuite envoyée vers un site distant pour mettre en œuvre la mobilité, ou stockée dans un fichier pour mettre en œuvre la persistance.

Quant à la *restauration* de l'exécution d'un thread Java, elle consiste à reconstruire et à ré-initialiser les informations qui décrivent l'état d'exécution du thread dans les structures d'exécution de la JVM.

Ainsi, la mobilité et la persistance des threads Java sont une utilisation possible de mécanismes de plus bas niveau : des mécanismes de capture et de restauration de l'état d'exécution des threads. Mais ces mécanismes de bas niveau peuvent également être utilisés pour mettre en œuvre tout autre service que la mobilité ou la persistance.

2. Problèmes abordés

Nous adressons, tout au long de ce chapitre, trois points importants de notre contribution, points relatifs à :

- La portabilité de l'état capturé.
- Le respect des performances de Java.
- Notre contribution en termes de nouvelles structures de données et nouveaux sous-systèmes dans la JVM.

Nos services de capture d'état, et donc de mobilité/persistance de threads Java, sont des services portables, utilisables sur des plates-formes hétérogènes. La principale difficulté concernant la portabilité de l'état capturé est que la représentation de la pile Java des threads dans la JVM est non portable. Nous proposons, à cet effet, deux techniques de transformation de la pile :

- une technique basée sur une extension de l'interprète Java
- et une technique basée sur une analyse dynamique de flot.

  Ces deux techniques sont présentées dans la section 4.5 de ce chapitre.

Concernant le respect des performances de Java, nous proposons des services de capture d'état qui sont utilisables même en présence de compilation Java à la volée (JIT). Ceci n'est possible qu'avec l'utilisation d'une technique de dés-optimisation à la volée du code compilé, technique que nous utilisons et décrivons dans la section 4.6 de ce chapitre.

Finalement, nous avons mis en œuvre et utilisé les différentes techniques présentées précédemment, à travers la proposition de nouvelles structures de données d'exécution et de nouveaux sous-systèmes dans la machine virtuelle Java., tels que décrits dans les sections 4.2, 4.3 et 4.4 de ce chapitre. De plus amples informations concernant les détails de mise en œuvre de nos services de capture/restauration et de mobilité/persistance sont présentées dans la section 4.7 de ce chapitre.

La suite de ce chapitre présente, tout d'abord, la structure de l'état d'exécution des threads Java puis décrit notre extension de la machine virtuelle Java de mécanismes de capture et de restauration avant de présenter la mise en œuvre de nos services de mobilité et de persistance.

3. Etat d'exécution des threads Java

Cette section décrit la structure de l'état d'exécution des threads Java dans la JVM puis présente les caractéristiques de cet état d'exécution.

3.1. Structure de l'état d'exécution d'un thread Java

L'état d'exécution d'un thread Java est constitué de structures de données qui décrivent l'exécution d'un code Java ou l'exécution d'un code natif par le thread. Nous appelons *état d'exécution Java* la partie de l'état d'exécution du thread qui représente l'exécution de méthodes Java et nous appelons *état d'exécution native* la partie de l'état d'exécution du thread qui représente l'exécution de méthodes natives. En ce qui concerne les méthodes Java compilées à la volée vers du code natif, nous les plaçons à part, dans un *état d'exécution compilée*. Dans la suite, nous présentons les structures de données qui constituent l'état d'exécution Java, l'état d'exécution native et l'état d'exécution compilée.

3.1.1. Etat d'exécution Java

Un thread Java est, avant tout, caractérisé par sa priorité. La priorité d'un thread Java est un entier compris entre les deux valeurs *Thread.MIN_PRIORITY* et *Thread.MAX_PRIORITY*. L'état d'exécution Java d'un thread est constitué d'autres

structures de données : la pile Java, le tas d'objets et la zone de méthodes, illustrés par la Figure 3-1.

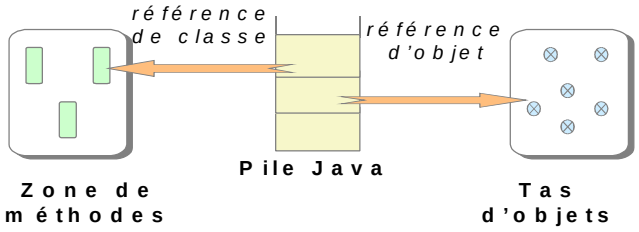

Figure 3-1. Etat d'un thread Java

- ***Pile Java***

Une *pile Java* est associée à chaque thread de la JVM ; elle décrit l'état d'avancement de l'exécution du thread. La pile Java contient des frames Java, chaque frame étant associé à l'appel d'une méthode Java par le thread. Un frame est empilé sur la pile Java d'un thread à chaque appel d'une méthode Java par le thread et dépilé à la terminaison de la méthode. Un frame contient, entre autres informations, les informations suivantes :

- la table des variables locales à la méthode Java associée au frame,
- la pile d'opérandes qui contient les résultats des calculs intermédiaires effectués par la méthode
- le registre compteur ordinal (PC) qui indique la prochaine instruction de bytecode à exécuter dans la méthode associée au frame
- et d'autres informations telles qu'un pointeur vers le descripteur interne de la méthode Java associée au frame Java, un pointeur vers le constant pool de la classe de la méthode associée au frame, un pointeur vers la table d'exceptions de la méthode et un pointeur vers le frame précédent sur la pile Java.

- ***Tas d'objets***

Une autre structure de données qui fait partie de l'état d'exécution Java d'un thread est le *tas d'objets du thread*. Nous appelons tas d'objets d'un thread Java l'ensemble des objets Java utilisés par le thread. Un objet Java est utilisé par un thread si l'objet représente une variable locale à une méthode Java en cours d'exécution par le thread ou si l'objet est le résultat d'un calcul intermédiaire effectué par une méthode Java en cours d'exécution par le thread. Le tas d'objets d'un thread est donc un sous-ensemble du tas d'objets de la JVM dans laquelle s'exécute ce thread.

- ***Zone de méthodes***

La troisième structure de données constituant l'état d'exécution Java d'un thread est la *zone de méthodes du thread*. Nous appelons zone de méthodes d'un thread Java l'ensemble des méthodes Java, et donc des classes, utilisées par le thread. Une méthode Java est utilisée par un thread si elle est en cours d'exécution par le thread (si un frame Java lui est associée sur la pile Java du thread). La zone de méthodes d'un thread est un sous-ensemble de la zone de méthodes de la JVM dans laquelle s'exécute ce thread.

3.1.2. Etat d'exécution native

En plus de l'état d'exécution Java, un thread Java peut avoir un état d'exécution native. L'état d'exécution native d'un thread Java décrit l'exécution de code natif par le thread. Cet état est constitué de code natif et d'une pile de méthodes natives.

Un *code natif* correspond à des méthodes natives d'origine (non obtenues par compilation à la volée). Le code natif est généralement utilisé pour optimiser certains traitements et est représenté dans un format dépendant de la plate-forme sous-jacente.

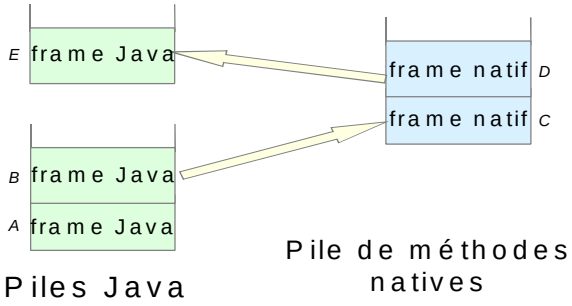

Figure 3-2. Pile de méthodes natives

Quant à *la pile de méthodes natives* elle sert à décrire l'état d'exécution des méthodes natives appelées par le thread. De même que pour le code natif, cette pile est généralement représentée dans un format dépendant de la plate-forme sous-jacente. La Figure 3-2 illustre l'état d'exécution d'un thread Java. Ce thread a, tout d'abord, appelé deux méthodes Java *A* et *B*. Deux frames Java sont alors empilés sur la pile Java du thread. La méthode Java *B* appelle ensuite une méthode native *C* qui elle-même appelle une autre méthode native *D*. Deux frames sont empilés sur la pile de méthodes natives du thread. La méthode native *D* appelle finalement une méthode Java *E*, ce qui provoque l'empilement d'un frame sur la pile Java du thread.

3.1.3. Etat d'exécution compilée

Nous appelons *état d'exécution compilée* d'un thread l'état qui correspond à l'exécution de méthodes Java compilées à la volée et transformées en code natif. Un état d'exécution compilée est initialement un état d'exécution Java qui, pour des raisons

d'optimisation du temps d'exécution, est transformé en exécution native. Nous distinguons ici l'état d'exécution compilée de l'état d'exécution native car ces deux parties sont traitées différemment dans la suite de ce chapitre.

3.2. Caractéristiques de l'état d'exécution des threads Java

Après avoir décrit le structure de l'état d'exécution d'un thread Java, voici les caractéristiques de cet état. L'état d'exécution d'un thread Java est un état interne à la machine virtuelle Java et un état non portable[1], tel que nous le décrivons dans les sections suivantes.

3.2.1. Etat interne à la JVM

Dans l'état actuel de la machine virtuelle Java et de ses API, l'état d'exécution des threads Java est interne à la JVM. Un programme Java ne peut pas directement accéder à l'état d'exécution d'un thread Java ni initialiser l'état d'exécution d'un thread avec certaines valeurs. L'état d'exécution d'un thread Java ne peut donc pas être directement capturé et un état d'exécution ne peut pas être directement restauré dans un nouveau thread Java.

3.2.2. Etat non portable

Un état d'exécution de thread est *portable* s'il peut être directement utilisé sur des plates-formes différentes. Cette condition est nécessaire pour la construction de services de capture et de restauration de threads utilisables sur des plates-formes hétérogènes.

Mais même si la machine virtuelle Java fournit l'abstraction d'un environnement homogène, l'état d'exécution de ses threads n'est pas entièrement portable sur des plates-formes hétérogènes. En effet, une partie de l'état d'exécution des threads Java est constituée de structures de données portables tandis qu'une autre partie est représentée dans un format natif, non portable. Le tas d'objets et la zone de méthodes d'un thread Java sont respectivement constitués d'objets Java et de classes Java, donc de structures de données portables. Alors que la pile Java, la pile de méthodes natives et le code natif sont représentés dans un format natif, dépendant de la plate-forme sous-jacente.

4. Mise en œuvre de la capture et de la restauration

Cette section décrit notre extension de la machine virtuelle Java de nouveaux sous-systèmes et structures de données d'exécution pour l'ajout de mécanismes de capture et de restauration de l'état d'exécution des threads Java.

[1] Ces caractéristiques concernent, en particulier, les threads du JDK de Sun Microsystems.

4.1. Conception de la capture de l'état d'exécution des threads Java

Connaissant les caractéristiques de l'état d'exécution des threads Java, nous aborderons, tout au long de cette section :

- L'identification des informations capturées à partir de l'état d'exécution courant des threads Java.
- La solution au problème d'inaccessibilité de l'état des threads Java.
- Les solutions au problème de non portabilité de l'état des threads Java.
- La vérification de la cohérence de l'état capturé des threads.

Lors de la capture de l'état d'exécution d'un thread Java, quelles sont les informations capturées à partir de cet état ? Comment sont résolus les problèmes d'inaccessibilité et de non portabilité de l'état des threads Java ? Comment est vérifiée la cohérence de l'état capturé d'un thread ? Ce sont ces questions, et leurs réponses, que nous abordons dans les sections suivantes.

L'état capturé d'un thread Java doit décrire les structures de données représentant l'état d'exécution courant du thread. L'état capturé contient donc des informations sur la priorité, la pile Java, le tas d'objets, la zone de méthodes du thread et, comme nous le verrons plus loin, l'état d'exécution compilée du thread.

4.1.1. Etat accessible

Une des caractéristiques de l'état d'exécution des threads Java est son inaccessibilité aux programmes Java. Pour pallier au problème d'inaccessibilité de l'état d'exécution, nous avons étendu la machine virtuelle Java. Notre extension de la JVM permet d'en extraire l'état d'exécution des threads Java et de le rendre visible par les programmes Java. Cette extension permet également d'initialiser un thread Java avec un état donné préalablement capturé. Notre extension permet donc de capturer l'état d'exécution des threads Java et de restaurer ces threads.

4.1.2. Etat portable

Une des caractéristiques de l'état d'exécution des threads Java est sa non portabilité. Pour que l'état capturé d'un thread Java soit portable, il est nécessaire que la structure de données construite lors de la capture et décrivant l'état capturé ne contienne que des informations représentées de façon indépendante de toute plate-forme. L'état capturé par nos services est un objet Java, instance de la classe *ThreadState* fournie par notre package *threadpack*[1].

[1] Voir l'interface de nos services dans l'Annexe I.

Les structures de données portables, qui font partie de l'état courant de l'exécution du thread Java, peuvent être directement intégrées à l'état capturé du thread. C'est le cas de la priorité, du tas d'objets et de la zone de méthodes du thread. Mais pour ce qui est des structures de données non portables de l'état courant du thread, celles-ci ne peuvent être intégrées à l'état capturé que si elles sont traduites vers un format portable. C'est le cas de la pile Java et de l'état d'exécution native du thread. Dans la suite, nous présentons les traitements relatifs à ces structures de données. Quant à l'état d'exécution compilée d'un thread, qui est également une structure non portable, sa gestion est abordée plus loin, dans la section 4.6 de ce chapitre.

- ***Pile Java***

Dans la pile Java d'un thread, tous les frames Java ne sont pas obligatoirement pris en compte lors de la capture d'état. En effet, lorsque la capture de l'état d'exécution est décidée par le thread dont l'état est capturé, une méthode de capture d'état est appelée par une des méthodes de l'application du thread. La pile Java du thread est alors constituée de frames associés aux méthodes de l'application puis de frames associés aux méthodes de capture d'état. Ici, la pile Java est constituée d'un *état applicatif* et d'un *état de capture*. Un exemple de pile Java de thread est donné par la Figure 3-3. L'état applicatif décrit l'exécution des méthodes de l'application du thread, méthode *m1* et *m2*, et l'état de capture décrit l'exécution des méthodes de capture de l'état du thread, méthodes *captureAndSend* et *capture*. Dans ce cas, seul l'état applicatif du thread est pris en compte dans l'état capturé du thread. Donc seules les frames Java des méthodes *m1* et *m2* feront partie de l'état capturé du thread.

Lorsque la capture de l'état d'un thread est forcée par un thread externe, l'état d'exécution courant du thread n'est constitué que d'un état applicatif. Dans ce cas, la totalité de l'état d'exécution du thread est prise en compte dans l'état capturé.

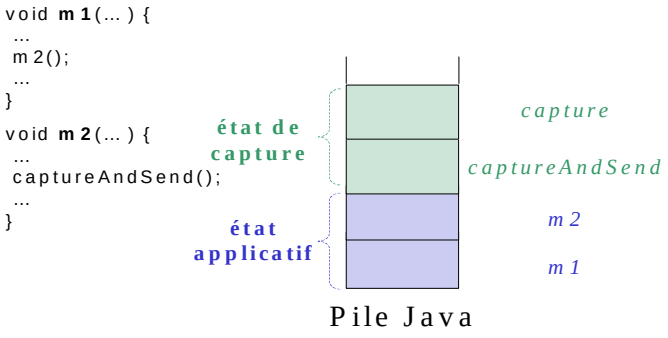

Figure 3-3. Etat applicatif et état de capture d'un thread

Rendre la pile Java d'un thread portable revient à rendre portable l'ensemble des frames Java qui sont sur cette pile (frames associés à l'état applicatif du thread). Rendre un frame Java portable consiste à traduire, vers un état portable, les informations natives

concernant le registre PC, la table des variables locales, la pile d'opérandes et divers pointeurs.

- ***Pointeur vers le descripteur de méthode Java***

Un frame Java est associé à une méthode Java et contient un pointeur vers le descripteur interne de cette méthode. Le pointeur vers le descripteur de la méthode étant une information native, l'information portable construite ici concernera la méthode Java elle-même. Cette information est constituée de la référence de la classe Java à laquelle appartient la méthode et d'un identificateur de la méthode dans la classe (l'indice de la méthode dans la table de méthodes de la classe).

- ***Registre PC***

Le registre PC d'un frame Java indique la prochaine instruction de bytecode à exécuter dans la méthode Java associée au frame. Ce registre contient une adresse dans la mémoire physique sous-jacente (dans la zone de méthodes de l'instance de JVM sous-jacente). Traduire cette valeur native en une valeur portable consiste à calculer le déplacement de l'instruction indiquée par le registre PC par rapport au début du bytecode de la méthode concernée.

- ***Divers pointeurs***

Un frame Java contient divers pointeurs : un pointeur vers la table d'exceptions de la méthode, un pointeur vers le constant pool de la classe de la méthode et un pointeur vers le frame précédent. Pour que le pointeur vers la table d'exceptions et le pointeur vers le constant pool puissent être reconstruits, il suffit que la méthode Java associée au frame et la classe de cette méthode soient connues lors de la restauration. Quant au pointeur vers le frame précédent, celui-ci est obtenu lors de la construction d'une nouvelle pile Java à la restauration du thread.

- ***Table de variables locales et pile d'opérandes***

Rendre la table de variables locales et la pile d'opérandes portables revient à traduire les valeurs natives des variables locales et des résultats intermédiaires vers des valeurs Java. Pour traduire une valeur native en une valeur Java, il faut connaître le type de cette valeur. Mais les structures de table de variables locales et de pile d'opérandes ne donnent aucune information sur le type des valeurs qu'elles contiennent. Un mot de quatre octets, décrivant une variable locale ou un résultat intermédiaire, peut aussi bien représenter un entier qu'une référence d'objet.

La portabilité des tables de variables locales et des piles d'opérandes de la pile Java d'un thread repose donc sur l'existence d'un mécanisme de reconnaissance des types des valeurs contenues dans ces structures. Nous proposons deux techniques de reconnaissance des types : une reconnaissance de types à l'interprétation de bytecode et

une reconnaissance de types par analyse de flot. Ces deux techniques sont discutées plus en détail dans la section 4.5 de chapitre.

- ***Etat d'exécution native***

Une demande de capture de l'état d'exécution d'un thread Java peut intervenir alors que le thread a des méthodes natives empilées. Dans ce cas, l'état d'exécution du thread contient un état d'exécution native, constitué de code natif et de pile de méthodes natives. Mais alors, cet état n'est pas portable sur des plates-formes hétérogènes.

Nous avons choisi de ne pas prendre en compte l'état d'exécution native dans l'état capturé d'un thread Java. Ce choix est justifié par le fait que nous plaçons nos mécanismes Java de capture/restauration d'état des threads au même niveau que les mécanismes système Java existants. Nous avons conçu nos mécanismes en adhérant à l'esprit Java : portabilité du code Java et aucune contrainte sur le code natif.

Les mécanismes de capture/restauration d'état proposés ne prennent donc pas en compte l'état d'exécution native des threads Java. Mais ils proposent un comportement pas défaut qui autorise la capture de l'état d'exécution d'un thread Java, même en présence de méthodes natives. Dans ce comportement par défaut, l'état capturé est la *plus grande partie portable et cohérente* de l'état courant du thread[1]. Ceci se traduit par une capture d'état qui ne considère, dans l'état capturé d'un thread, que les frames Java qui précèdent l'appel à la première méthode native en cours (méthode native qui correspond au frame le plus en bas de pile de méthodes natives).

La Figure 3-4 illustre la capture de la plus grande partie portable et cohérente de l'état d'exécution courant d'un thread Java. Ici, la demande de capture d'état intervient alors que l'état d'exécution courant du thread est constitué de deux frames Java sur la pile Java (frames *A* et *B*), puis de deux frames natifs sur la pile de méthodes natives (frames *C* et *D*) et d'un frame Java sur la pile Java (frame *E*). L'état capturé du thread est alors constitué des deux frames Java *A* et *B* qui précèdent le frame de la première méthode native (frame *C*).

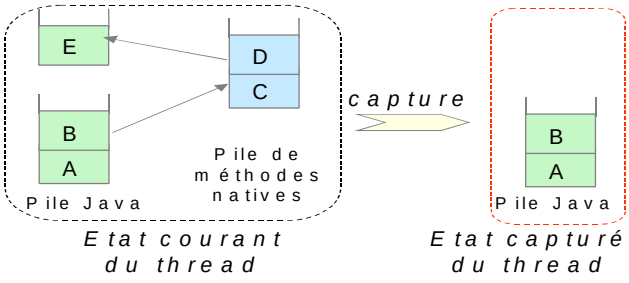

Figure 3-4. Plus grand état portable et cohérent

[1] La cohérence de l'état capturé est discutée dans la section 4.1.3 de ce chapitre.

Avec ce comportement par défaut, l'état capturé n'est certes qu'un état partiel de l'exécution courante du thread et la mobilité ou la persistance sous-jacentes du thread ne sont donc que partiellement fortes. A la restauration du thread, celui-ci devra reprendre son exécution à partir de la méthode native associée au frame C. Mais cette approche a l'avantage de permettre d'effectuer une capture de l'état d'exécution d'un thread Java, même si le thread a des méthodes natives en cours d'exécution.

Il est important de noter que les méthodes dont les frames n'ont pas été pris en compte dans l'état capturé (frames E, D ou C) peuvent modifier des objets Java utilisés par les méthodes dont les frames ont été pris en compte. Ceci pose alors un problème de cohérence de l'état capturé :

- Il faut alors que le programmeur qui utilise nos mécanismes de capture d'état ait la connaissance de cette contrainte et gère ses captures d'état en conséquence.
- Ou il faut disposer d'un système de gestion de la cohérence des objets partagés. Une gestion possible est d'effectuer une copie des objets utilisés par les frames Java A et B (partagés avec les frames C, D ou E), juste avant l'empilement du premier frame natif C. Ainsi, lors de la capture, les objets effectivement pris en compte dans l'état capturé ne sont pas les objets utilisés par les frames A et B mais leurs copies. Pour certains types d'objets, tels que les objets représentant des ressources physiques telles que les fichiers, les copies d'objets doivent être accompagnées des copies de ressources (copie de fichiers).

Dans cette section, nous avons décrit le comportement par défaut de nos mécanismes de capture d'état, en présence d'exécution native. Par exécution native, nous entendons ici l'exécution d'un code qui est initialement natif et non un code Java compilé à la volée vers un code natif. Notre traitement de l'état d'exécution compilée est basé sur une technique de dés-optimisation dynamique de code compilé ; ceci est abordé plus en détail dans la section 4.6 de ce chapitre.

Par ailleurs, nous avons eu connaissance d'un récent rapport technique de Sun Microsystems, où une proposition de services de persistance des threads Java est faite. Dans ce rapport, le comportement par défaut que nous avons présenté plus haut est également abordé : « *…This situation is handled by capturing the state of the thread such that the stack is cut back to the frame of the calller of the native method …* » [72].

4.1.3. Etat cohérent

Dans les sections précédentes, nous avons décrit les solutions apportées aux problèmes d'inaccessibilité et de non portabilité de l'état d'exécution des threads java. Mais ajouté au fait qu'il soit accessible et portable, l'état d'exécution capturé d'un thread doit être cohérent. La cohérence de l'état est garantie :

- en définissant l'unité d'exécution du thread, à la fin de laquelle l'état d'exécution du thread peut être capturé
- et en garantissant que l'état d'exécution du thread n'évolue par durant une opération de capture.

Pour que la capture de l'état d'exécution d'un thread Java et que la restauration ultérieure de cet état soient cohérentes, nous avons choisi de n'effectuer une opération de capture qu'à la fin d'un cycle de l'interprète Java. Ainsi, la capture de l'état d'exécution d'un thread se fera à la fin de l'interprétation d'une instruction de bytecode par le thread et la restauration de l'état se fera au début de l'interprétation de l'instruction de bytecode suivante. Ce choix suppose évidemment que le sous-système d'exécution de la JVM est basé sur un interprète Java. Nous avons opté pour le cycle de l'interprète Java comme unité d'exécution à la fin de laquelle l'état d'un thread peut être capturé, pour les deux raisons suivantes :

- L'existence d'une unité d'exécution, l'interprétation d'une instruction de bytecode, fournie par un sous-système d'exécution de la JVM, l'interprète Java.
- L'interprétation d'une instruction de bytecode est une unité à grain assez fin pour fournir un mécanisme de capture d'état de threads au-dessus duquel seront construites une mobilité et une persistance fortes des threads.

D'autre part, il faut garantir que l'état d'exécution d'un thread Java n'évolue pas au cours de la capture. Si la capture de l'état d'exécution est décidée par le thread, l'état applicatif du thread ne peut pas évoluer au cours de la capture puisque, pendant la capture, le thread n'effectue pas autre chose que la capture.

Si la capture de l'état d'exécution d'un thread est forcée par un thread externe, il faut s'assurer que l'état du thread n'évolue pas durant la capture. Une solution possible est d'utiliser les primitives Java de suspension momentanée puis de reprise de l'exécution d'un thread : la méthode *suspend* et la méthode *resume*, fournies par la classe *Thread*. La méthode *suspend* peut être appelée par le thread externe, sur le thread à capturer, au début de l'opération de capture d'état. Et la méthode *resume* peut être appelée par le thread externe, sur le thread capturé, à la fin de l'opération de capture. Mais l'inconvénient de cette solution est que, pendant l'opération de capture, n'importe quel thread tiers peut appeler la méthode *resume* sur le thread à capturer et remettre en question la cohérence de l'état capturé. Finalement, nous avons choisi de garantir une exclusion mutuelle entre l'interprétation d'une instruction de bytecode par le thread à capturer et l'opération de capture d'état par un thread externe. Ceci est discuté dans la section 4.4 de ce chapitre.

4.1.4. Conclusion

Dans cette section, nous avons abordé des questions concernant l'état capturé d'un thread Java, son extraction de la JVM, sa portabilité et sa cohérence :

- Pour extraire l'état d'exécution des threads Java de la JVM, nous avons étendu la machine virtuelle. Ceci est discuté plus en détail dans la section 4.2 de ce chapitre.
- Pour garantir la portabilité de l'état capturé d'un thread, il est nécessaire d'avoir un mécanisme de reconnaissance des types des valeurs contenues dans les tables de variables locales et dans les piles d'opérandes de la pile Java du thread. Ces mécanismes sont décrits dans la section 4.5 de ce chapitre.
- Pour garantir la cohérence de l'état capturé d'un thread, il est nécessaire que le sous-système d'exécution de la JVM soit basé sur un interprète Java. De plus, dans le cas d'une capture forcée, l'opération de capture par un thread externe et l'interprétation d'une instruction de bytecode par le thread dont l'état est capturé doivent être mutuellement exclusives. Ceci est présenté dans la section 4.6 de ce chapitre.

4.2. Extension de la JVM

L'architecture de notre extension de la JVM est illustrée par la Figure 3-5. Cette extension propose une nouvelle API représentée par notre package *threadpack*. La nouvelle API fournit des services de mobilité et de persistance des threads Java et des mécanismes de capture et de restauration de l'état d'exécution des threads Java. Ces services ont été décrits précédemment et leur interface est présentée dans l'Annexe I.

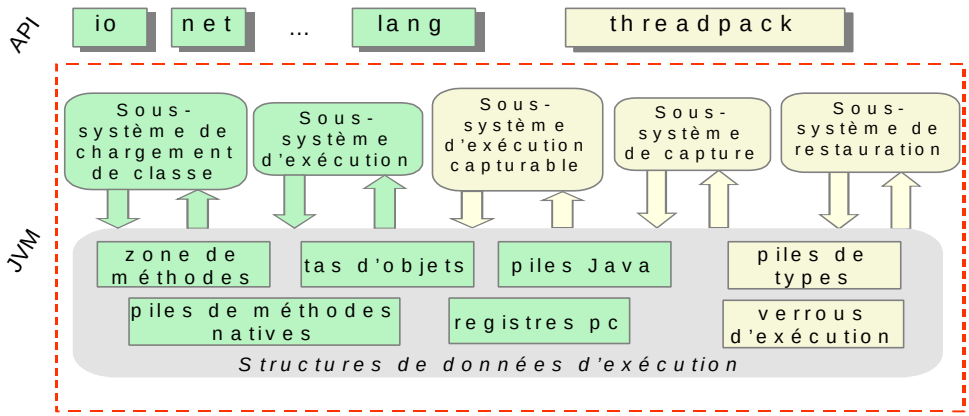

Figure 3-5. Architecture de la JVM étendue

La mise en œuvre de la nouvelle API Java repose sur la définition de nouveaux sous-systèmes dans la JVM :

- Le *sous-système d'exécution capturable*. Chaque JVM étendue possède un sous-système d'exécution capturable. Ce sous-système gère l'exécution des threads Java dont l'état d'exécution peut être capturé.
- Le *sous-système de capture*. Chaque JVM étendue possède un sous-système de capture. Un sous-système de capture fournit les mécanismes nécessaires à la capture de l'état d'exécution de threads Java. Ces mécanismes sont représentés par les primitives de capture d'état, *capture* ou *captureAndSend*, proposées par la classe *ThreadStateManagement*.
- Le *sous-système de restauration*. Chaque JVM étendue possède un sous-système de restauration. Un sous-système de restauration fournit les opérations nécessaires à la restauration de l'exécution de threads Java. Ces opérations sont représentées par les primitives de restauration d'état, *restore* ou *receiveAndRestore*, proposées par la classe *ThreadStateManagement*.

Notre extension de la machine virtuelle Java repose également sur la définition de nouvelles structures de données d'exécution, nécessaires à la capture de l'état d'exécution des threads Java.

Les sections suivantes 4.3 et 4.4 décrivent respectivement les nouvelles structures de données d'exécution de la JVM et les sous-système d'exécution capturable, sous-système de capture et sous-système de restauration.

4.3. Nouvelles structures de données d'exécution

En plus des structures de données d'exécution décrites par la spécification de la machine virtuelle Java [85], notre extension de la JVM apporte de nouvelles structures de données d'exécution : la *pile de types*, le *frame de types* et le *verrou d'exécution*.

4.3.1. Pile de types

Une *pile de types* est utilisée pour assurer la portabilité de l'état capturé d'un thread Java. Pour que l'état d'un thread Java soit portable, il faut procéder à la reconnaissance des types des valeurs contenues dans les tables de variables locales et dans les piles d'opérandes de la pile Java du thread. Ces types sont alors stockés dans une pile de types.

Une pile de types est associée à un thread Java dont l'état d'exécution est capturé. Cette pile de types est associée au thread soit à la création du thread, soit au moment de la capture de l'état d'exécution du thread ; ceci dépend de la technique de reconnaissance de types utilisée (voir la section 4.5 de ce chapitre). La pile de types d'un thread décrit les types des valeurs manipulées par les méthodes Java exécutées par le thread. Les types des valeurs manipulées par une méthode Java sont donnés par un *frame de types*, tel qu'illustré par la Figure 3-6.

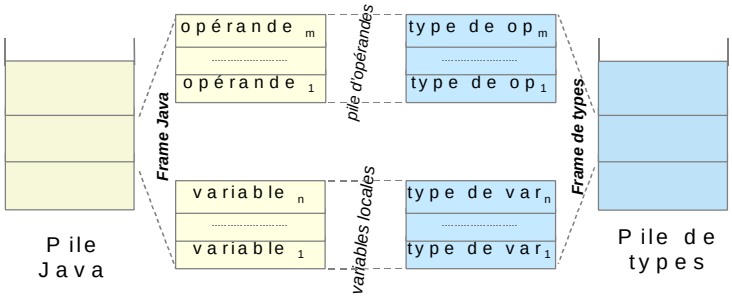

Figure 3-6. Correspondance entre les pile de types/frames de types et pile Java/frames Java

4.3.2. Frame de types

Un *frame de types* est associé à chaque frame Java de la pile Java du thread. Un frame de types contient les informations suivantes :

- Une *table de variables locales* qui a le même nombre d'entrées que la table de variables locales du frame Java associé et qui donne les types des variables locales.
- Une *pile d'opérandes* qui contient le même nombre d'entrées que la pile d'opérandes du frame Java associé et qui donne les types des résultats intermédiaires.
- Des informations supplémentaires telles que le pointeur vers le frame de types précédent.

4.3.3. Verrou d'exécution

Un *verrou d'exécution* est utilisé pour garantir la cohérence de l'état capturé d'un thread Java lorsque la capture est forcée par un thread externe. En effet, pour qu'un état capturé de façon forcée soit cohérent, il faut que l'opération de capture forcée par un thread externe et l'interprétation d'une instruction de bytecode par le thread dont l'état est capturé soient mutuellement exclusives.

De ce fait, un verrou est associé à un thread Java dont l'état d'exécution peut être capturé de façon forcée, à la création du thread. Ce verrou est le même que le verrou associé à tout objet Java et utilisé pour la synchronisation des accès aux objets (voir le chapitre 17 de la spécification du langage Java [73]).Un verrou d'exécution est détenu soit par le thread propriétaire du verrou, lorsque celui-ci est dans une unité d'exécution (interprétation d'une instruction de bytecode), soit par un thread externe, lorsque celui-ci effectue une opération de capture d'état forcée. Le verrou d'exécution associé à un thread est rendu par le thread propriétaire du verrou, à la fin d'une unité d'exécution du thread, et par un thread externe, à la fin d'une opération de capture d'état.

4.4. Nouveaux sous-systèmes

Après avoir présenté les structures de données d'exécution apportées par notre extension de la JVM, voici une description des sous-systèmes ajoutés à cette extension de la JVM. En plus des sous-systèmes proposés par la spécification de la machine virtuelle Java, et qui sont le sous-système de chargement de classes et le sous-système d'exécution, notre extension de la JVM propose :

- un sous-système d'exécution capturable,
- un sous-système de capture
- et un sous-système de restauration.

Chaque JVM étendue possède un *sous-système d'exécution capturable*. Un sous-système d'exécution capturable est, avant tout, un sous-système d'exécution : un mécanisme responsable d'exécuter un bytecode. Ce qui différencie ce sous-système du sous-système d'exécution décrit dans les spécifications de la machine virtuelle Java est qu'avec un sous-système d'exécution capturable, l'état d'exécution d'une instance du sous-système (thread) peut être capturé puis restauré dans une nouvelle instance du sous-système.

Nous avons conçu et expérimenté plusieurs sous-systèmes d'exécution capturable. Ces sous-systèmes se différencient par les solutions adoptées pour gérer la cohérence et la portabilité de l'état capturé :

- **Cohérence**. Tous les sous-systèmes d'exécution capturable expérimentés sont basés sur un interprète Java pour garantir la cohérence de l'état capturé. De plus, pour proposer une capture d'état forcée cohérente, un sous-système d'exécution capturable doit être synchronisé avec le sous-système de capture sous-jacent.
- **Portabilité**. Pour garantir la portabilité de l'état capturé, un seul problème persiste : la traduction de la structure native de pile Java vers une structure portable. Pour ce faire, nous proposons deux approches présentées dans la section 4.5 de ce chapitre : une traduction lors de l'interprétation du bytecode ou une traduction à la capture d'état (n'affectant pas l'interprétation). Ainsi, un sous-système d'exécution capturable est soit basé sur un interprète Java étendu, soit basé sur un interprète Java standard.

Nous avons ainsi conçu et expérimenté quatre sous-systèmes d'exécution capturable :

- Le premier sous-système d'exécution capturable que nous avons conçu est un *interprète Java étendu* d'une gestion de pile de types. Ce sous-système ne propose qu'une capture décidée de l'état des threads Java.
- Pour fournir une capture forcée cohérente de l'état d'exécution des threads Java, nous avons expérimenté un *interprète Java étendu synchronisé.*

- Finalement, nous avons basé les deux derniers sous-systèmes d'exécution capturable sur un *interprète Java standard* et un *interprète Java standard synchronisé*, le premier pour une capture d'état décidée et le second pour une capture forcée.

Dans la suite, nous décrivons chacun de ces sous-systèmes et présentons les sous-systèmes de capture et de restauration qui leur sont associés.

4.4.1. Interprète Java étendu

Le premier sous-système d'exécution capturable expérimenté est un interprète Java étendu d'une gestion de pile de types. A la création d'une instance de ce sous-système, un thread, une pile de types est associée au thread, en plus de sa pile Java. Cette pile de types évolue au fur et à mesure que l'exécution du thread avance, en suivant l'approche décrite dans la section 4.5.1 de ce chapitre. L'exécution d'un thread instance de ce sous-système d'exécution capturable est constituée des étapes suivantes :

Etapes d'exécution :
1. Choisir l'instruction de bytecode courante. S'il n'y a plus d'instructions, l'exécution du thread se termine.
2. Interpréter l'instruction de bytecode, en utilisant la pile Java du thread.
3. *Reconnaître le type des valeurs manipulées par l'instruction de bytecode, en utilisant la pile de types du thread.*
4. Avancer à l'instruction suivante.
5. Aller à l'étape 1.

Dans cet interprète Java étendu, le cycle d'interprétation d'une instruction de bytecode est augmenté d'une nouvelle opération : la reconnaissance du type des valeurs manipulées par l'instruction de bytecode interprétée.

Cette extension de l'interprète Java induit un surcoût sur l'exécution du thread. Pour ne pas pénaliser l'ensemble des threads Java, nous avons proposé une nouvelle classe de threads, la classe *CapturableThread*. Cette classe caractérise les threads Java dont l'état d'exécution peut être capturé et les différencie des threads non concernés par la capture d'état. Ainsi, un thread Java de la classe *CapturableThread* est une instance de notre sous-système d'exécution capturable alors qu'un thread de la classe standard *Thread* est une instance du sous-système d'exécution de la JVM.

- ***Sous-système de capture***

Un thread dont l'état peut être capturé avec ce sous-système de capture doit être un thread de la classe *CapturableThread* (basé sur l'interprète Java étendu). Le sous-système de capture proposé ici fournit une capture décidée par le thread dont l'état d'exécution est capturé et produit un objet *ThreadState*.

Au moment de la capture de l'état d'exécution du thread, la traduction de la structure native de pile Java du thread vers une structure portable est donnée par la pile de types associée au thread. Cette pile de types a été préalablement construite par le sous-système d'exécution capturable sous-jacent et peut alors servir à la construction d'un état portable.

- ***Sous-système de restauration***

Le sous-système de restauration associé à l'interprète Java étendu crée un nouveau thread Java de la classe *CapturableThread*, à partir d'un objet *ThreadState*. La priorité du thread est initialisée et une pile Java, un tas d'objets et une zone de méthodes sont associés au thread. De plus, une pile de types est construite et associée au thread. L'exécution du thread peut alors commencer, celle-ci sera une instance du sous-système d'exécution capturable (interprète Java étendu).

Mais si après la restauration, le thread ne doit plus subir d'opération de capture, il peut être restauré sous la forme d'un thread de la classe *Thread* (instance de l'interprète standard) et éviter ainsi le surcoût induit par l'interprète étendu. Dans ce cas, la restauration du thread ne reconstruit pas de pile de types à associer au thread et le lancement de l'exécution du thread sera basé sur l'interprète Java standard.

4.4.2. Interprète Java étendu synchronisé

Ce second sous-système d'exécution capturable n'est pas fondamentalement différent du précédent, il se différencie de lui par le fait qu'à un thread *CapturableThread* est également associé un verrou d'exécution. Ce verrou est associé au thread, à sa création. L'exécution d'un thread de la classe *CapturableThread* consiste alors à :

Etapes d'exécution :

1. Choisir l'instruction de bytecode courante. S'il n'y a plus d'instructions, l'exécution du thread se termine.
2. *Prendre le verrou d'exécution associé au thread.*
3. Interpréter l'instruction de bytecode, en utilisant la pile Java du thread.
4. *Reconnaître le type des valeurs manipulées par l'instruction de bytecode, en utilisant la pile de types du thread.*
5. Avancer à l'instruction suivante.
6. *Rendre le verrou d'exécution associé au thread.*
7. Aller à l'étape 1.

Avec ce sous-système d'exécution capturable, en plus de la reconnaissance de types, l'interprétation d'une instruction de bytecode est verrouillée par le verrou d'exécution associé au thread.

- ***Sous-système de capture***

Un thread dont l'état peut être capturé avec ce sous-système de capture doit être un thread de la classe *CapturableThread*. Ce sous-système de capture propose deux modes d'initiation de la capture : une capture décidée par le thread dont l'état d'exécution est capturé et une capture forcée par un thread externe. Dans le cas d'une capture forcée, l'opération de capture est verrouillée : elle commence par la prise du verrou d'exécution associé au thread à capturer et se termine par la relâche du verrou.

- ***Sous-système de restauration***

Le sous-système de restauration associé à l'interprète Java étendu synchronisé est similaire au sous-système présenté dans la section 4.4.1 de ce chapitre. La seule différence entre ces deux sous-systèmes est que, lorsqu'un thread de la classe *CapturableThread* est restauré, son exécution est lancée avec une instance de l'interprète Java étendu synchronisé.

4.4.3. Interprète Java standard

Le troisième sous-système d'exécution capturable expérimenté n'est autre que l'interprète Java standard. Une instance de ce sous-système est un thread de la classe *Thread*. Dans ce cas, l'exécution d'un thread consiste à :

Etapes d'exécution :

1. Choisir l'instruction de bytecode courante. S'il n'y a plus d'instructions, l'exécution du thread se termine.
2. Interpréter l'instruction de bytecode, en utilisant la pile Java du thread.
3. Avancer à l'instruction suivante.
4. Aller à l'étape 1.

- ***Sous-système de capture***

Un thread dont l'état peut être capturé avec ce sous-système de capture est n'importe quel thread Java instance de la classe *Thread* (basé sur l'interprète Java standard). Le sous-système de capture proposé ici fournit une capture décidée par le thread dont l'état d'exécution est capturé.

Au moment de la capture de l'état d'exécution du thread, la traduction de la structure native de pile Java du thread vers une structure portable est effectuée en construisant une pile de types associée au thread. La construction de cette pile de types est basée sur l'approche décrite dans la section 4.5.2 de ce chapitre. Une fois la pile de types construite, celle-ci peut servir à la construction d'un état portable, sous la forme d'un objet Java *ThreadState*.

- ***Sous-système de restauration***

Ce sous-système de restauration crée un nouveau thread Java de la classe *Thread*, à partir d'un objet *ThreadState*. La priorité du thread est initialisée et une pile Java, un tas d'objets et une zone de méthodes sont associés au thread. L'exécution du thread peut alors commencer, celle-ci sera une instance de l'interprète Java standard.

4.4.4. Interprète Java standard synchronisé

Le quatrième sous-système d'exécution capturable se différencie du précédent par le fait qu'il est synchronisé avec le sous-système de capture sous-jacent (un verrou d'exécution est associé au thread instance de ce sous-système). Une instance du sous-système d'exécution capturable est un thread de la classe *CapturableThread*, pour ne pas pénaliser les threads non concernés par la capture d'état. L'exécution d'un thread de la classe *CapturableThread* passe par les étapes suivantes :

Etapes d'exécution :

1. Choisir l'instruction de bytecode courante. S'il n'y a plus d'instructions, l'exécution du thread se termine.
2. *Prendre le verrou d'exécution associé au thread*
3. Interpréter l'instruction de bytecode, en utilisant la pile Java du thread.
4. Avancer à l'instruction suivante.
5. *Rendre le verrou d'exécution associé au thread.*
6. Aller à l'étape 1.

- ***Sous-système de capture***

Ce sous-système de capture propose deux modes d'initiation de la capture : une capture décidée et une capture forcée. Un thread dont l'état peut être capturé de façon décidée est un thread Java quelconque (objet *Thread*). Alors qu'un thread dont l'état est capturé de façon forcée doit être un thread de la classe *CapturableThread*. Dans le cas d'une capture forcée, l'opération de capture est verrouillée : elle commence par la prise du verrou d'exécution associé au thread à capturer et se termine par la relâche du verrou.

- ***Sous-système de restauration***

Ce sous-système de restauration crée un nouveau thread Java, à partir d'un état précédemment capturé (objet *ThreadState*). Le thread restauré est :

- Soit un thread de la classe *CapturableThread*, si le thread doit subir, par la suite, une opération de capture forcée. Un verrou d'exécution est alors associé au thread.
- Soit un thread de la classe *Thread*, si le thread ne subit plus de capture forcée.

4.5. Hétérogénéité : Construction de la pile de types

Après avoir présenté notre extension de la machine virtuelle Java pour l'ajout de services de capture/restauration de l'état d'exécution des threads, revenons au problème de non portabilité de l'état d'un thread Java. En effet, de toutes les structures de données constituant l'état d'exécution d'un thread Java, les tables de variables locales et piles d'opérandes se trouvant sur la pile Java d'un thread restent non portables. A cet effet, nous avons introduit une nouvelle structure de données d'exécution : la pile de types.

Nous proposons deux approches de construction de la pile de types associée à un thread Java : une construction de la pile à l'interprétation du bytecode et une construction de la pile au moment de la capture de l'état d'exécution d'un thread. Nous avons mis en œuvre et expérimenté ces deux approches, que nous décrivons ci-dessous.

4.5.1. Construction de la pile de types à l'interprétation : Interprète étendu

Cette approche associe une pile de types à un thread Java dont l'état va être capturé, à la création du thread. Tout comme la pile Java du thread, la pile de types évolue avec le thread, au fur et à mesure que celui-ci interprète des instructions de bytecode [22][23]. Les types des variables locales ou des résultats intermédiaires manipulés par une instruction de bytecode sont reconnus grâce au typage des instructions. En effet, les instructions de bytecode s'appliquent à des valeurs d'un certain type [85].

L'exemple illustré par la Figure 3-7 présente un programme Java et le bytecode qui lui est associé. Ce programme affecte la valeur entière *5* à la première variable locale de la méthode *m*. Le thread Java qui exécute ce programme a une pile Java et une pile de types qui lui sont associées, telles qu'illustrées par la Figure 3-8.

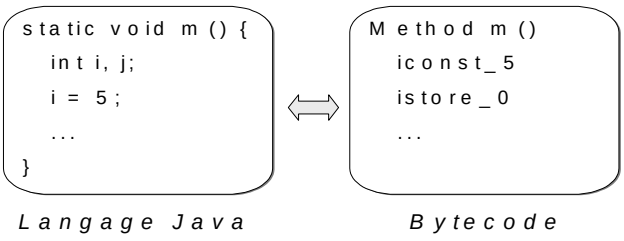

Figure 3-7. Exemple de programme Java

La pile Java contient un frame Java qui représente l'état d'exécution de la méthode *m*. Avant l'interprétation de l'instruction *istore_0*, le frame a deux variables locales qui ne sont pas initialisées et un résultat intermédiaire de valeur entière 5 (précédemment empilé par l'instruction *iconst_5*). L'interprétation de l'instruction *istore_0* dépile la valeur entière en sommet de pile d'opérandes et la stocke dans la variable locale

d'indice 0. L'interprétation de cette instruction de bytecode permet donc de connaître le type de la première variable locale : le type *int*.

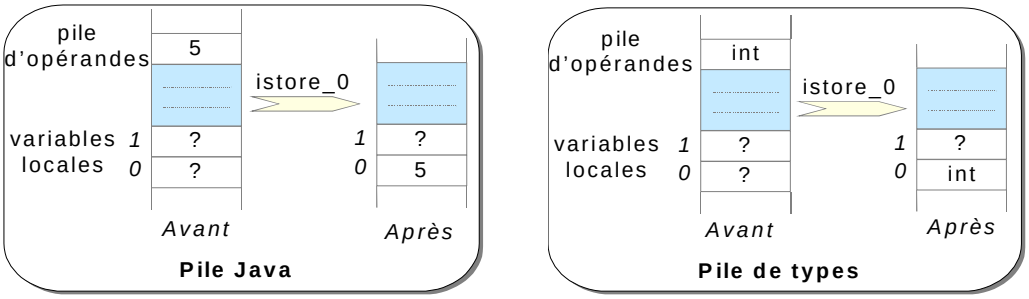

Figure 3-8. Interprétation de bytecode / Reconnaissance de types

La pile de types du thread contient un frame de types associé à la méthode *m*. Avant l'interprétation de l'instruction *istore_0*, le frame a deux variables locales de types inconnus et un type *int* sur la pile d'opérandes (précédemment reconnu grâce à l'instruction *iconst_5*). L'interprétation de l'instruction *istore_0* dépile le type *int* en sommet de pile d'opérandes et le stocke dans la variable locale d'indice 0 : cette variable a pour type *int*.

Ainsi, avec cette approche de reconnaissance de types, la pile de types d'un thread Java évolue en parallèle à la pile Java du thread.

4.5.2. Construction de la pile de types à la capture : Analyse de flot

Avec la seconde approche de construction de la pile de types, la pile de types associée à un thread Java n'est pas construite au fur et à mesure que le thread s'exécute mais au moment de la capture de l'état du thread. Ainsi, lors de la capture, un frame de types est construit et associé à chaque méthode Java en cours d'exécution par le thread (chaque frame Java sur la pile Java du thread) [24][25].

La construction du frame de types associé à une méthode Java est basée sur un algorithme de marquage du graphe de flots d'exécution possibles de la méthode. Le principe de cet algorithme est de partir d'un frame de types initial (associé à la première instruction de la méthode) et de calculer, de proche en proche, la frame de types des instructions de la méthode, jusqu'à trouver la frame de types associé à l'instruction d'arrêt dans la méthode.

Mais avant de donner l'algorithme de parcours du graphe de flots, prenons un exemple de programme Java, illustré par la Figure 3-9. Cette figure présente un exemple de programme source d'un méthode Java *m*. Cette méthode est constituée de 4 blocs : le bloc initial 1, deux blocs conditionnels 2 et 3 puis le bloc final 4. De plus, cette méthode possède deux variables entière *i* et *j*, qui sont connues par l'ensemble des blocs, et une variable locale *k*, qui est connue par le bloc 2 comme étant un entier et par le bloc 3 comme étant un flottant. La Figure 3-9 présente également le bytecode correspondant à

la méthode *m* et le graphe de flots d'exécution possibles de la méthode. Le graphe de flots montre qu'arrivée à l'instruction de bytecode d'adresse 3 (fin du bloc 1), l'exécution de la méthode peut soit aller à l'instruction d'adresse 6 (bloc 2), soit à l'instruction d'adresse 11 (bloc 3). Ces deux flots possibles se rejoignent ensuite à l'instruction d'adresse 13 (bloc 4).

D'autre part, deux propriétés invariantes assurent la correction du bytecode [45] :

> **Propriétés :** A n'importe quel point du programme et quel que soit le chemin suivi pour atteindre ce point :
> - P1 : Les piles d'opérandes construites en suivant chaque chemin contiennent des valeurs de mêmes types.
> - P2 : Les variables locales construites en suivant chaque chemin sont de mêmes types ou inutilisées si leurs types diffèrent.

D'après ces deux propriétés, arrivé au début du bloc 4 et quel que soit le chemin suivi à l'exécution de la méthode :

- La pile d'opérandes du frame Java de la méthode est vide (avant l'exécution de *j = 4* par le programme Java).
- Les variables locales *i* et *j* sont reconnues comme étant des entiers parce qu'elles ont préalablement été utilisées et la variable locale *k* est inutilisée car même si son type diffère dans les blocs 2 et 3, elle reste interne à ces deux blocs et inconnu du bloc 4.

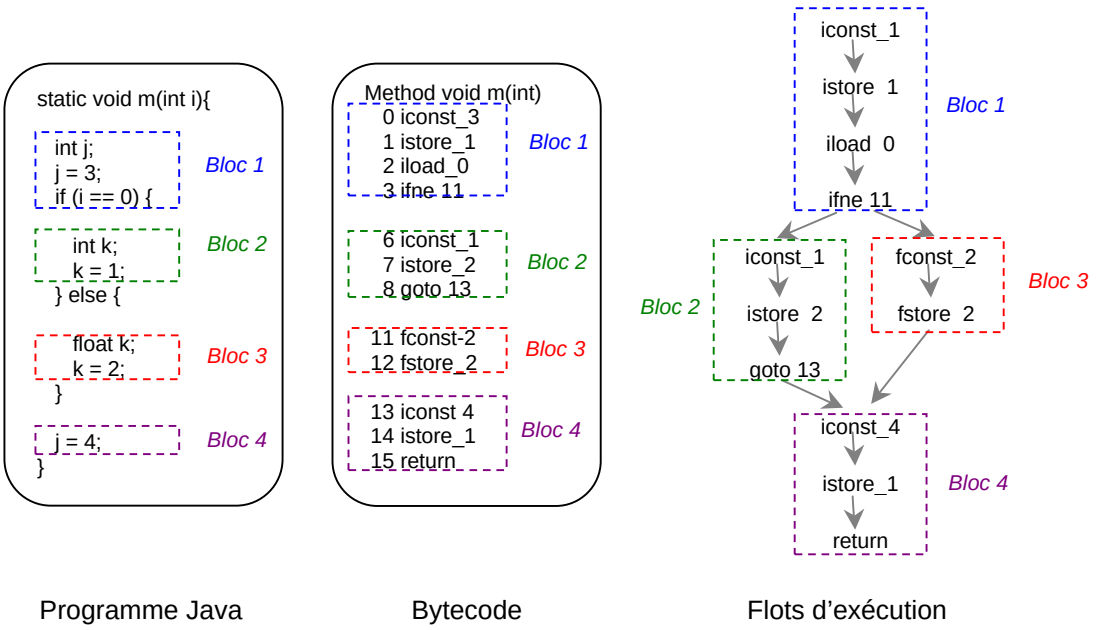

Figure 3-9. Un programme Java, son bytecode et ses flots d'exécution

L'algorithme de parcours des flots d'exécution d'une méthode Java est, en partie, inspiré de l'algorithme de vérification du bytecode Java, décrit dans la section 4.9.2 de

la spécification de la machine virtuelle Java [85]. Les étapes principales de notre algorithme sont :

1. Initialement, un frame de types est construit ; ce frame décrit l'état des types des valeurs (variables locales et résultats intermédiaires) manipulées par la méthode au lancement de la méthode. Ainsi, la pile d'opérandes de ce frame de types est vide et sa table de variables locales a pour nombre d'entrées le nombre de variables locales à la méthode. Toutes les variables locales sont identifiées comme inutilisées sauf celles qui correpondent aux paramètres de la méthode et dont les types peuvent être reconnus grâce à la signature de la méthode.

   Ce frame de types est alors associé à la première instruction de bytecode de la méthode et cette instruction est *marquée*. Initialement, toutes les instructions de la méthode sont identifiées comme étant *non traitées*.
2. Choisir une instruction de bytecode marquée et non traitée. Cette instruction est désignée, dans la suite, par *instruction courante*.
3. A partir du frame de types associé à l'instruction courante et en émulant cette instruction (grâce au typage des instructions), construire un nouveau frame de types.
4. Déterminer l'ensemble des successeurs de l'instruction courante :
   - L'instruction *goto* a pour successeur la destination du *goto*.
   - Une instruction *if* a pour successeurs la destination du *if* et l'instruction suivante dans le code.
   - Les autres instructions ont pour successeur l'instruction suivante dans le code.
   - Si une instruction fait partie d'un bloc de code qui peut lever une exception Java, le traitant de l'exception est un successeur de l'instruction.
5. Si parmi les successeurs de l'instruction courante se trouve l'instruction d'arrêt dans la méthode, l'algorithme s'arrête. L'instruction d'arrêt est désignée par le registre *PC* du frame Java associé à la méthode. Le frame de types construit par émulation de l'instruction courante représente alors le frame de types à associer à la méthode.

   Dans le cas particulier de la dernière méthode appelée par le thread (méthode dont le frame Java est en sommet de pile Java), l'instruction d'arrêt est émulée et son effet et reporté dans le frame de types à associer à la méthode. Ceci est nécessaire pour que la restauration ultérieure de l'état d'exécution du thread commence à la suite de la dernière instruction exécutée par le thread.
6. Si l'instruction d'arrêt dans la méthode ne fait pas partie des successeurs de l'instruction courante, pour chaque instruction dans l'ensemble des successeurs :
   - Si l'instruction n'est pas marquée, la marquer et lui associer le frame de types construit. Puis aller à l'étape 7.

- Si l'instruction est marquée, fusionner le frame de types qui lui est déjà associé avec le frame de types construit :
  - La pile d'opérandes reste inchangée car c'est la même dans les deux frames de types (propriété P1 de correction du bytecode).
  - Une variable locale a soit le même type dans les tables de variables locales des deux frames, soit des types différents et dans ce cas, elle est identifiée comme inutilisée (propriété P2 de correction du bytecode).

7. Noter que l'instruction courante est traitée et aller à l'étape 2.

Dans la suite, nous discutons la correction et la terminaison de l'algorithme proposé.

Montrer que l'algorithme est correct revient à montrer que le frame de types construit à la fin de l'algorithme et associé à une méthode Java reflète effectivement les types des valeurs manipulées par la méthode, à son point d'arrêt. La preuve de la correction de l'algorithme peut se faire par un raisonnement par récurrence.

Montrer que l'algorithme se termine consiste à montrer que l'algorithme finit par construire un frame de types associé à une méthode. Le frame de types associé à une méthode est construit lorsque l'instruction d'arrêt dans la méthode est atteinte (étape 5 de l'algorithme). Prouver que l'algorithme se termine revient donc à prouver qu'il existe un flot d'exécution qui atteint l'instruction d'arrêt dans une méthode, à partir de la première instruction de la méthode. Ceci est le cas du flot suivi lors de l'exécution effective de la méthode : notre algorithme de parcours du graphe de flots se termine.

4.5.3. Conclusion

Nous fournissons des mécanismes de capture d'état, et donc de mobilité/persistance des threads, qui sont utilisables sur des plates-formes hétérogènes. Pour ce faire, nous proposons deux techniques : une première technique basée sur l'interprétation de bytecode et une seconde technique inspirée de l'algorithme de vérification de bytecode.

Dans un premier temps, nous avons mis en œuvre la première technique sur le JDK 1.1.3 de Sun Microsystems. Nous avons ensuite effectué un portage de cette première technique sur le JDK 1.2.2 puis mis en œuvre la seconde technique également sur ce JDK. Ces deux mises en œuvre sont actuellement opérationnelles et disponibles à : *http://sirac.inrialpes.fr/~bouchena/JavaThread/*

4.6. Performances : Méthodes Java compilées à la volée

Dans la section précédente, nous avons présenté deux solutions pour traiter le problème de non portabilité de la pile Java d'un thread. La pile Java est, en effet, constituée de frames de méthodes Java exécutées ou, plus exactement, interprétées par le thread. Mais le sous-système d'exécution de la JVM sous-jacente peut être basé sur un compilateur Java à la volée (compilateur JIT). Dans ce cas, le code des méthodes

Java est dynamiquement traduit en code natif, code dont l'exécution est plus performante. Pour des raisons de performances, la plupart des JVM actuelles sont dotées de compilateurs JIT. L'exécution d'une méthode compilée à la volée ne se base alors plus sur la pile Java du thread mais sur sa pile de méthodes natives. Or cette structure de pile n'est pas portable. Dans la suite, nous décrivons les étapes successives que nous avons suivies pour le traitement de l'état d'exécution compilée.

4.6.1. Pas de compilation à la volée

Nous avons présenté, dans la section précédente, deux techniques pour traiter la pile Java d'un thread :

- une première technique basée sur un interprète Java étendue
- et une seconde technique basée sur une analyse de flot.

Du fait que la première technique soit basée sur un interprète Java étendu, l'utilisation de cette technique impose que le sous-système d'exécution sous-jacent soit un interprète de bytecode et non un compilateur à la volée. Les threads Java dont l'état d'exécution est capturé en suivant cette technique ne peuvent donc pas bénéficier de la compilation à la volée.

Quant à la seconde technique, elle n'est pas basée sur un interprète étendu et peut, en principe, bénéficier de la compilation à la volée. Mais lorsque nous avons effectué nos expérimentations sur le JDK 1.2.2 de Sun Microsystems [138], nous nous sommes rendus compte que le compilateur JIT sous-jacent compilait automatiquement toutes les méthodes Java exécutées en code natif. Par conséquent, la pile java d'un thread reste toujours vide. Or pour que la seconde technique puisse effectuer une analyse de flot des méthodes Java en cours d'exécution par le thread, un parcours des frames de la pile Java est nécessaire. Nous nous sommes donc retrouvés dans une situation où :

- Il n'est, en principe, pas impossible de bénéficier de la compilation JIT lors de l'exécution d'un thread.
- Mais lors de la capture de l'état Java d'un thread, cet état ne reflète que l'exécution de méthodes compilées.

Donc dans un premier temps, pour pouvoir utiliser cette technique, il a fallu désactiver la compilation JIT.

4.6.2. Compilation à la volée puis dés-optimisation à la volée

Interdire la compilation JIT pour pouvoir utiliser la seconde technique a deux conséquences majeures sur les performances des applications :

- Une baisse importante des performances normales des applications Java voulant bénéficier de nos mécanismes.

- Une baisse des performances des autres applications Java s'exécutant sur la même JVM, applications non concernées par nos mécanismes.

Or pour que des mécanismes Java soient effectivement utilisables, ils ne doivent pas annuler les efforts faits en matière de compilation JIT.

Par ailleurs, la portabilité est une des caractéristiques qui sont à la base de l'esprit Java. Un code compilé est à l'origine un code Java qui est portable et même s'il est compilé, ce code correspond toujours à un code portable. La compilation à la volée n'empêche, en effet, pas un programmeur de continuer de bénéficier de la portabilité de son code Java en rendant, par exemple, ce code mobile.

Pour que les mécanismes que nous proposons adhèrent à cet esprit, ils doivent permettre au programmeur de continuer de bénéficier de la portabilité de l'état Java capturé même si l'exécution effective sous-jacente est une exécution de code compilé. Nos mécanismes doivent donc prendre en compte l'état d'exécution du code compilé.

Mais autoriser la compilation JIT pose un problème de non portabilité de l'état d'exécution. La solution que nous proposons consiste à construire un état d'exécution Java (portable) sémantiquement équivalent à l'état d'exécution du code compilé. Construire un état d'exécution Java qui décrit une méthode compilée à la volée revient à construire le frame Java et à retrouver le code Java qui auraient été utilisés si la méthode Java d'origine n'avait pas été compilée :

- Le code Java équivalent au code natif de la méthode compilée n'est autre que le code de la méthode Java d'origine.
- Le frame Java équivalent au frame natif doit être construit, avec sa table de variables locales, sa pile d'opérandes, son registre PC. La principale difficulté ici est de calculer la valeur du PC de bytecode à partir de la valeur du PC du code natif.

Une fois le frame Java construit, notre technique d'analyse de flot, décrite dans la section 4.5.2 de ce chapitre, est appliquée pour produire l'état actuel de l'exécution de la méthode Java. Cet état est ensuite intégré à l'état d'exécution global du thread, état produit en résultat de la capture.

Initialement, lorsque nous avons commencé nos travaux, nous n'avions pas à notre disposition des moyens/outils pour mettre en œuvre la solution décrite plus haut. Mais le compilateur HotSpot, intégré à la dernière version du JDK de Sun Microsystems, utilise de tels outils [91]. En effet, HotSpot emploie des techniques de dés-optimisation qui permettent de convertir un frame de méthode compilée en un frame Java. Ceci permet de revenir, au cours de l'exécution d'un code compilé, à la version interprétée de ce code. HotSpot emploie ces techniques lorsqu'il faut « défaire » la compilation et

l'expansion de méthode (*method inlining*) pour prendre en compte un nouveau code Java chargé dynamiquement.

4.6.3. Conclusion

Nous proposons des mécanismes de capture de l'état d'exécution des threads Java, mécanismes qui répondent à deux exigences de Java : les performances et la portabilité.

- *Performances*. Nos mécanismes permettent aux applications Java de bénéficier des techniques d'optimisation de l'exécution (compilation JIT) proposées par la JVM.
- *Portabilité*. Nos mécanismes prennent en compte la totalité de l'état d'exécution Java des threads, quelle que soit la nature du sous-système d'exécution sous-jacent. L'état capturé est ainsi un état portable qui reflète l'état d'exécution des méthodes Java interprétées et l'état des méthodes compilées à la volée.

Pour garantir ces deux conditions, la réalisation de nos mécanismes de capture d'état est basée sur des techniques de dés-optimisation, techniques utilisées jusqu'alors dans le domaine de la compilation. L'application de ces techniques aux mécanismes de capture d'état est, à notre connaissance, une première utilisation du genre. L'intégration de ces techniques à nos services de capture d'état est actuellement en cours, elle se base sur le compilateur Java JIT de HotSpot [91].

4.7. Bilan de la mise en œuvre

Les sections précédentes ont présenté les nouvelles structures de données et les nouveaux sous-systèmes apportés par notre extension de la machine virtuelle Java et nécessaires à la construction de mécanismes de capture/restauration de l'état d'exécution des threads Java. Cette section donne, à présent, quelques détails sur la mise en œuvre de notre extension de la machine virtuelle Java. Elle décrit la modularité et l'environnement de mise en œuvre, le problème de sécurité et la fourniture de nos services de capture/restauration comme une librairie externe à la JVM.

4.7.1. Modularité de la mise en œuvre

Même si nos services de capture et de restauration de l'état d'exécution des threads Java ont été mis en œuvre en étendant la machine virtuelle Java, cette extension de la machine virtuelle s'est faite :

- Sans modification du langage Java.
- Sans modification du compilateur Java.
- Sans modification des API Java existantes.
- Sans modification de la mise en œuvre des API Java existantes.

L'extension de la machine virtuelle s'est faite de façon modulaire, en proposant une nouvelle API Java et une mise en œuvre de cette API dans le JDK 1.2.2 (Java 2 SDK)

de Sun Microsystems [138]. La modularité de notre extension de la JVM a deux conséquences directes :

- Le portage de nos services sur d'autres mises en œuvre de la machine virtuelle Java est simplifié.
- Les performances des services Java existants restent inchangées puisque la mise en œuvre de ces services n'a pas été modifiée. Ceci est discuté plus en détail dans la section 3 du Chapitre 5

4.7.2. Environnement de mise en œuvre

Nos services de capture/restauration de l'état d'exécution des threads Java et nos services de mobilité/persistance des threads Java ont été intégrés au JDK 1.2.2 (Java 2 SDK), sous une licence de Sun Microsystems [140]. Le JDK que nous avons expérimenté peut être utilisé sur les plates-formes suivantes :

- Solaris 2.5.1, 2.6 sur architecture Sparc,
- Solaris 2.5.1, 2.6 sur architecture x86
- et Windows (NT/95/98).

Nos services de mobilité/persistance des threads Java et nos mécanismes de capture/restauration de l'état d'exécution des threads sont, de ce fait, utilisables sur toutes ces plates-formes.

Nous proposons actuellement deux extensions du JDK 1.2.2, disponibles sur *http://sirac.inrialpes.fr/~bouchena/JavaThread/*. Ces deux extensions se différencient par le type de sous-système d'exécution capturable mis en œuvre :

- une première extension met en œuvre l'interprète Java étendu synchronisé (voir la section 4.4.2 de ce chapitre)
- et une seconde extension met en œuvre l'interprète Java standard synchronisé (voir la section 4.4.4 de ce chapitre).

La mise en œuvre de ces extensions a abouti à :

- 2500 lignes de code Java pour chacune des deux extensions (soit 0,2 % du code Java total de la JVM),
- 12000 lignes de code C pour la première extension (soit 2% du code C total de la JVM)
- et 17500 lignes de code C pour la seconde extension (soit 3% du code C total de la JVM).

Nous avons également mis en œuvre et expérimenté nos services avec deux autres approches qui sont présentées dans la section 3 du Chapitre 5. Ces deux approches sont basées sur :

- un interprète Java étendu non synchronisé
- et un interprète Java standard non synchronisé.

4.7.3. Sécurité des services de mobilité et de persistance

Une extension de la machine virtuelle Java n'est utilisable que si elle ne viole pas la sécurité qui est un aspect important de Java. De ce fait, il est légitime de se demander si notre extension de la machine virtuelle n'affecte pas la sécurité de Java.

Notre extension de la JVM se présente sous la forme de deux services de base : un service qui permet de capturer l'état d'exécution d'un thread Java et un service pour restaurer un état d'exécution dans un nouveau thread Java. Voici quelques questions (et leurs réponses) sur l'éventuelle insécurité introduite par nos services.

Peut-on capturer l'état d'exécution d'un thread Java pour lequel « on n'aurait pas le droit » de capturer l'état (thread interne à la JVM, par exemple) ? *Avec notre service de capture, l'état d'exécution d'un thread Java ne peut être capturé que si la référence du thread en question (référence d'objet Java) est connue.*

Peut-on construire un état d'exécution erroné ou modifier l'état d'exécution capturé d'un thread Java pour le restaurer dans un nouveau thread qui est alors dans un état incohérent ? L'état d'exécution d'un thread Java est représenté par une instance de la classe *ThreadState*. Or, la classe *ThreadState* est une classe finale (qui ne peut avoir de sous-classe), qui ne propose pas de constructeur *public* et dont les variables de classe et d'instance sont marquées *private* (inaccessible par d'autres classes). Un objet *ThreadState* ne peut donc ni être construit ni être modifié « manuellement ». La seule manière d'instancier un objet *ThreadState* est d'effectuer une opération de capture d'état. L'état construit est alors cohérent et non modifiable.

Notre extension de la machine virtuelle avec des services de capture/restauration d'état d'exécution respecte donc la sécurité de Java. Mais une ouverture de nos services pour la mise en œuvre, par exemple, de services de débogage des applications devra permettre de modifier certaines informations de l'état d'exécution. Dans ce cas, il faudra considérer l'aspect sécuritaire plus finement.

4.7.4. Extension de la JVM : Librairie externe à la JVM

Nos services de mobilité et de persistance des threads Java et nos mécanismes de capture et de restauration de l'état d'exécution des threads sont proposés par une machine virtuelle Java étendue. L'utilisation de ces services ne peut se faire que sur une JVM étendue ; ce qui revient à dire que, pour utiliser nos services, le programmeur d'une application doit installer la JVM étendue. Ceci réduit, quelque peu, la diffusion de ces services.

Un moyen de rendre nos services de mobilité et de persistance plus facilement diffusables est de fournir ces services, non pas comme une partie intégrante d'une JVM, mais comme une extension externe qui peut être ajoutée à une JVM déjà installée. Cette

extension externe doit pouvoir accéder aux structures qui décrivent l'état d'exécution des threads Java et qui sont internes à la JVM.

L'ajout d'une extension externe à la JVM est possible via l'interface du compilateur JIT de la JVM [135]. En effet, l'interface du compilateur JIT est une interface fournie par la JVM pour l'intégration d'un compilateur Java JIT. Cette interface se présente sous la forme de points d'entrée dans la JVM et permet d'accéder aux structures internes à la JVM. Il est donc possible, à travers cette interface, d'accéder aux structures décrivant l'état d'exécution des threads Java. Ainsi, nos services de mobilité et de persistance des threads peuvent être fournis dans une extension externe à la JVM, une extension qui accède aux structures internes de la JVM via l'interface du compilateur JIT.

Il est important de noter que toute extension, même externe, reste dédiée à une mise en œuvre particulière de JVM. Par exemple, une extension du JDK 1.2.2 est utilisable sur tous les JDK 1.2.2 préalablement installés mais pas sur les autres implantations de JVM.

4.7.5. Transparence à l'exécution

Après avoir décrit la mise en œuvre des services de capture et de restauration de l'état d'exécution des threads Java et présenté la construction des services de mobilité et de persistance des threads, il est légitime de se demander comment sont gérés les objets partagés entre plusieurs threads, la synchronisation des accès aux objets partagés ou les variables de classes, par nos services de mobilité et de persistance des threads. Ces aspects concernent, de façon générale, la gestion de la transparence de la mobilité et de la persistance à l'exécution des applications.

Un de nos choix de conception était de ne pas proposer une politique de gestion de la transparence à l'exécution mais de permettre au programmeur d'une application d'adopter sa propre politique en fonction de la spécificité de son application. Nous justifions ce choix par le fait que nos services de mobilité et de persistance des threads sont conçus dans la machine virtuelle Java, une machine centralisée. Nous pensons que ce n'est pas au niveau de ces services ni au niveau de la machine virtuelle que les aspects relatifs au partage dans un contexte distribué doivent être gérés.

Nos services proposent cependant un comportement par défaut et la possibilité d'adapter ce comportement. Un exemple de comportement par défaut, dans le cas de la mobilité d'un thread, est le transfert automatique par sérialisation de tous les objets Java atteignables par le thread. Mais il est également possible d'adapter ce comportement. En effet, du fait que nos services de mobilité et de persistance soient, en partie, basés sur les mécanismes Java de sérialisation d'objets et de chargement dynamique de classes, il

est possible d'adapter le comportement des services de mobilité et de persistance des threads en spécialisant les mécanismes de sérialisation et de chargement dynamique.

5. Interface de nos services de capture et de restauration

Après avoir décrit la mise en œuvre des mécanismes de capture et de restauration de l'état d'exécution des threads Java, voici une présentation de l'interface proposée par ces mécanismes.

Nos services de capture et de restauration de l'état d'exécution des threads Java sont proposés dans un package Java appelé *java.lang.threadpack*[1] et plus particulièrement par la classe *ThreadStateManagement*. Le package *threadpack* est constitué de plusieurs classes et interfaces, telles qu'illustrées par la Figure 3-10 :

- La classe *MobileThreadManagement* fournit les services de mobilité des threads Java.
- La classe *PersistentThreadManagement* propose les services de persistance des threads. La mise en œuvre de ces deux premières classes est basée sur les autres classes et interfaces du package *threadpack.*
- La classe *ThreadStateManagement* fournit les services nécessaires à la capture et à la restauration de l'état d'exécution des threads Java.
- La classe *CapturableThread* représente les threads Java dont l'état d'exécution peut être capturé de façon forcée.
- La classe *ThreadState* représente l'état d'exécution des threads Java.
- Les interfaces *SendInterface* et *ReceiveInterface* sont utilisées pour spécialiser les opérations de capture d'état et de restauration d'état aux besoins des applications qui les utilisent.

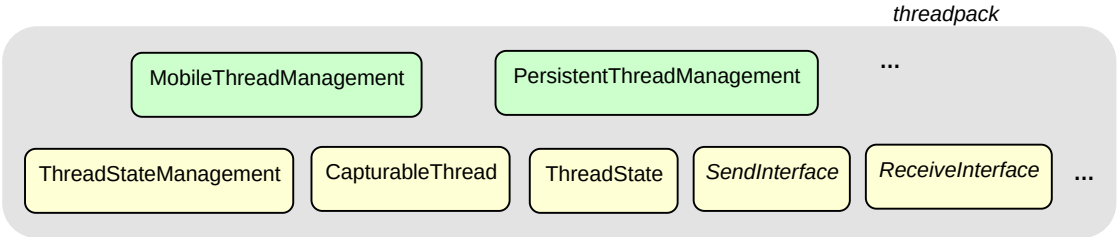

Figure 3-10. Package *threadpack*

[1] Le package *java.lang.threadpack* est décrit dans la section 1 de l'Annexe I.

Une partie de l'interface des services de capture et de restauration de l'état d'exécution des threads Java, représentés par la classe *ThreadStateManagement*, est donnée par la Figure 3-11[1].

java.lang.threadpack
Class ThreadStateManagement
public final class **ThreadStateManagement** extends Object
The ThreadStateManagement class provides several useful services for the capture and restoration of Java thread state.

Method Summary	
static ThreadState	capture() Captures the state of the current Java thread and returns it as a ThreadState object.
static ThreadState	capture(CapturableThread thread) Captures the state of a Java thread and returns it as a ThreadState object.
static Thread	restore (ThreadState threadState) Creates a new Java thread, initializes it with a previously captured state and starts it.
static void	captureAndSend(SendInterface stateTransfItf, boolean toStop) Captures the state of the current Java thread and sends it (to a remote node or on disk) by calling the sendState method of the SendInterface argument.
static void	captureAndSend(CapturableThread thread, SendInterface stateTransfItf, boolean toStop) Captures the state of a Java thread and sends it (to a remote node or on disk) by calling the sendState method of the SendInterface argument.
static Thread	receiveAndRestore(ReceiveInterface stateTransfItf) Receives the state of a Java thread by calling the receiveState method of the ReceiveInterface argument, creates a new Java thread, initializes it with the received state and starts it.

Figure 3-11. Services de capture et de restauration de l'état d'exécution des threads Java

La première méthode *capture* extrait l'état d'exécution du thread Java courant et le stocke dans un objet Java retourné en résultat de la méthode. L'objet Java retourné est un objet de la classe *ThreadState*, dont les instances représentent l'état d'exécution de threads Java.

Cette première méthode *capture* fournit une capture d'état décidée par le thread dont l'état est capturé. Pour fournir une capture d'état forcée par un autre thread que le thread dont l'état est capturé, la classe *ThreadStateManagement* propose une autre méthode

[1] L'interface complète des services de capture/restauration d'état est décrite dans la section 4 de l'Annexe I.

capture. Cette seconde méthode suspend momentanément l'exécution d'un thread Java, passé en paramètre à la méthode, extrait son état d'exécution courant et le retourne sous la forme d'un objet *ThreadState*. Mais contrairement à la première méthode *capture*, qui s'applique à une instance de la classe *Thread* (le thread Java courant), cette seconde méthode *capture* s'applique à un thread Java instance de la classe *CapturableThread*. La classe *CapturableThread* représente les threads Java dont l'état d'exécution peut être capturé de façon forcée.

La méthode duale à ces méthodes *capture* est la méthode *restore*. Cette méthode, qui prend en paramètre un objet *ThreadState*, crée un nouveau thread Java, initialise son état avec les informations contenues dans l'objet *ThreadState* et lance l'exécution du thread. Le thread restauré est retourné en résultat de la méthode.

Avec ces mécanismes de capture et de restauration d'état, il est possible de construire des services de plus haut niveau, tels que le service de mobilité ou le service de persistance des threads. En considérant le cas de la mobilité, par exemple, une mise en œuvre naïve du service est que le thread effectue, dans son programme, un appel à la méthode *capture* suivi d'un transfert de l'état capturé vers un site distant. Mais avec cette solution, le thread Java créé sur le site distant commencera son exécution par une nouvelle émission de son état sur le réseau (puisque son exécution reprend au point où elle a été interrompue au moment de la capture).

Une solution possible à ce problème est de permettre d'attacher à l'opération de capture de l'état d'un thread un traitement particulier à effectuer après la capture, de telle sorte que ce traitement post-capture ne fasse pas partie de l'application du thread. C'est ce qui est proposé par la méthode *captureAndSend*.

En effet, en plus de la capture de l'état d'exécution du thread Java courant, la première méthode *captureAndSend* de la classe *ThreadStateManagement* (Figure 3-11) permet de spécifier le traitement à effectuer sur l'état capturé. Ce traitement peut, par exemple, être l'envoi de l'état vers un site distant pour des besoins de mobilité, le cryptage de l'état envoyé sur le réseau pour une mobilité sécurisée, l'écriture de l'état sur disque pour une mise en œuvre de la persistance ou tout autre traitement spécifique aux besoins de l'application utilisant le mécanisme de capture d'état. Cette spécialisation du traitement à effectuer sur l'état capturé est rendue possible grâce à la généricité de la méthode *captureAndSend* et grâce, plus particulièrement, au paramètre de type *SendInterface*[1]. Une classe qui implante notre interface *SendInterface* fournit une méthode *sendState*, dans laquelle est spécifié le traitement à effectuer sur l'état capturé. C'est cette méthode *sendState* qui est appelée par la méthode *captureAndSend*.

[1] L'interface *SendInterface* est décrite dans la section 5 de l'Annexe I.

Le dernier paramètre de la méthode *captureAndSend* est un booléen qui détermine si l'exécution du thread Java dont l'état a été capturé doit être arrêtée ou poursuivie. Ce booléen est, par exemple, mis à *vrai* dans le cas de la mise en œuvre de la migration de thread (arrêt de l'exécution du thread sur le site source) ou à *faux* dans pour le clonage de thread à distance (exécution poursuivie sur le site source). La mise en œuvre de la méthode *captureAndSend* est décrite par la Figure 3-12.

```
public static void captureAndSend(
      SendInterface stateTransfItf, boolean toStop) {
      ThreadState state ;

      // Thread state capture
      state = ThreadStateManagement.capture();

      // Thread state processing
      stateTransfItf.sendState(state);

      //Resuming or stopping the thread
      if (toStop)
            Thread.currentThread().stop();
}
```

```
public static Thread receiveAndRestore(
      ReceiveInterface stateTransfItf) {
      ThreadState state;
      Thread      thread;

      // Thread state processing
      state = stateTransfItf.receiveState();

      // Thread state restoration
      thread = ThreadStateManagement.restore(state);

      return thread;
}
```

Figure 3-12. Méthodes *captureAndSend* et *receiveAndRestore*

Nous avons décrit la première méthode *captureAndSend* de la classe *ThreadStateManagement*. Cette méthode effectue une capture d'état décidée par le thread Java dont l'état est capturé. La seconde méthode *captureAndSend* permet d'effectuer une capture forcée ; elle prend, en paramètre, le thread Java dont l'état d'exécution est capturé de façon forcée : un thread de la classe *CapturableThread*.

La méthode *receiveAndRestore*, qui est une méthode duale aux méthodes *captureAndSend*, est proposée pour garantir une uniformité des interfaces des services de capture/restauration. La méthode *receiveAndRestore* permet de spécifier le traitement à effectuer sur un état d'exécution avant de procéder à sa restauration. Ce traitement peut, par exemple, être la réception d'un état envoyé sur le réseau, le décryptage d'un état reçu via le réseau, la lecture d'un état à partir du disque ou tout autre traitement répondant aux besoins de l'application utilisant le mécanisme de restauration. La spécialisation du traitement sur l'état à restaurer est spécifiée par l'objet générique de type *ReceiveInterface*[1], passé en paramètre à la méthode *receiveAndRestore*. Un objet de ce type fournit une méthode *receiveState*, qui spécifie le traitement à effectuer sur l'état à restaurer. C'est cette méthode qui est appelée par la méthode *receiveAndRestore*, tel que décrit par la Figure 3-12. Finalement, l'état traité est restauré dans un nouveau thread Java qui est retourné en résultat de la méthode *receiveAndRestore*.

[1] L'interface *ReceiveInterface* est décrite dans la section 5 de l'Annexe I.

6. Interface de nos services de mobilité et de persistance

Voici, à présent, une description de l'interface de nos services de mobilité et de persistance des threads Java, services construits au-dessus des mécanismes de capture/restauration d'état.

6.1. Services de mobilité

Nos services de mobilité des threads Java sont proposés par la classe *MobileThreadManagement*, dont une partie de l'interface est décrite par la Figure 3-13[1].

java.lang.threadpack
Class MobileThreadManagement
public final class **MobileThreadManagement** extends Object
The MobileThreadManagement class provides several useful services for making Java threads mobile.

Method Summary	
static void	go(String host, int port) Transfers the excution of the current thread from the local host to a target host identified by a name and a port number. The execution of the thread is resumed on both the target host and the local host.
static void	goAndStop(String host, int port) Transfers the excution of the current thread from the local host to a target host identified by a name and a port number. The execution of the current thread is stopped on the local host and resumed on the target host.
static void	go(CapturableThread thread, String host, int port) Transfers the excution of the specified thread from the local host to a target host identified by a name and a port number. The execution of the thread is resumed on both the target host and the local host.
static void	goAndStop(CapturableThread thread, String host, int port) Transfers the excution of the specified thread from the local host to a target host identified by a name and a port number. The execution of the specified thread is stopped on the local host and resumed on the target host.
static Thread	arrive(int port) Receives a mobile thread and resumes its execution on a host identified by a port number on the underlying host.

Figure 3-13. Services de mobilité des threads Java

La première méthode *go* crée, sur une JVM destination, un thread identique au thread courant s'exécutant sur la JVM locale. Par thread identique, nous entendons un thread qui a le même état d'exécution que le thread courant (même pile d'exécution, mêmes données, même code). Cette méthode est similaire à la commande UNIX *fork* qui crée

[1] L'interface complète de nos services de mobilité est décrite dans la section 2 de l'Annexe I.

un processus UNIX fils identique au processus UNIX père [55], sauf que la méthode *go* crée le thread identique sur une machine distante. Cette méthode *go* a deux paramètres : un nom de machine et un numéro de port. Le nom de la machine identifie la machine sur laquelle s'exécute la JVM destination et le numéro de port identifie le port sur lequel la JVM destination est en attente d'un thread mobile. Après l'appel à cette méthode, le thread courant et le thread nouvellement créé poursuivent leur exécution, l'un sur la JVM locale et l'autre sur la JVM destination.

Si le thread courant de la JVM locale souhaite arrêter son exécution après avoir appelé la méthode *go* dans son programme, une solution naïve est de proposer que le thread appelle, dans son programme, une primitive d'arrêt juste après l'appel à la méthode *go*. L'inconvénient de cette solution est que le thread identique, créé sur la JVM destination, commencera son exécution par l'appel de cette même primitive d'arrêt et son exécution prendra fin dès son arrivée sur la JVM destination. La méthode *goAndStop* remédie à ce problème. Après l'appel à cette méthode, le thread nouvellement créé poursuit son exécution sur la JVM destination et le thread courant sur la JVM locale est détruit.

Ces deux premières méthodes *go* et *goAndStop* fournissent une mobilité décidée par le thread courant : les méthodes sont appelées par le thread courant et appliquées sur ce même thread. Pour fournir une mobilité forcée, initiée par un thread externe au thread mobile, la classe *MobileThreadManagement* propose deux autres méthodes *go* et *goAndStop*. Ces deux méthodes prennent en paramètre le thread mobile, en plus du nom et du numéro de port de la JVM destination. Mais contrairement aux deux premières méthodes *go* et *goAndStop* qui s'appliquent sur le thread Java courant, instance de la classe *java.lang.Thread*, ces deux autres méthodes *go* et *goAndStop* s'appliquent sur des threads Java instances de la classe *java.lang.threadpack.CapturableThread*. La classe *CapturableThread*, sous-classe de la classe *Thread*, caractérise les threads Java qui peuvent être rendus mobiles ou persistants de façon forcée. La classe *CapturableThread* fournit la même interface que la classe *Thread* mais se différencie d'elle par sa mise en œuvre.

La méthode duale à ces méthodes *go* et *goAndStop* est la méthode *arrive*. Cette méthode, qui est exécutée sur la JVM d'arrivée (JVM destination), se met en attente d'un thread mobile. Cette attente se fait sur un numéro de port passé en paramètre à la méthode. Une fois le thread mobile arrivé, son exécution est lancée et sa référence est retournée en résultat de la méthode.

6.2. Services de persistance

Nos services de persistance des threads Java sont proposés par la classe *PersistentThreadManagement*. Une partie de l'interface de cette classe est décrite par la Figure 3-14[1].

java.lang.threadpack
Class PersistentThreadManagement
public final class **PersistentThreadManagement** extends Object
The PersistentThreadManagement class provides several useful services for making Java threads persistent.

Method Summary	
static void	store(String fileName) Stores the excution of the current thread on a file identified by a name. The execution of the current thread is resumed.
static void	storeAndStop(String fileName) Stores the excution of the current thread on a file identified by a name. The execution of the current thread is stopped.
static void	store(CapturableThread thread, String fileName) Stores the excution of the specified thread on a file identified by a name. The execution of the current thread is resumed.
static void	storeAndStop (CapturableThread thread, String fileName) Stores the excution of the specified thread on a file identified by a name. The execution of the current thread is stopped.
static Thread	load(String fileName) Loads a thread's excution from a file identified by a name. The loaded execution is resumed.

Figure 3-14. Services de persistance des threads Java

La première méthode *store* sauvegarde une image de l'état d'exécution du thread courant sur un fichier. Le fichier de sauvegarde est identifié par un nom passé en paramètre à la méthode. Après l'appel à cette méthode, le thread courant poursuit son exécution.

Pour les mêmes raisons que celles présentées dans le cas de la mobilité, si le thread courant souhaite arrêter son exécution après avoir appelé le service de sauvegarde, il doit faire appel à la méthode *storeAndStop* au lieu de la méthode *store* (voir la section 6.1 de ce chapitre). L'appel à cette méthode provoque donc la sauvegarde du thread courant sur un fichier puis la terminaison de ce thread.

Ces deux premières méthodes *store* et *storeAndStop* permettent d'effectuer une sauvegarde décidée par le thread courant. Pour effectuer une sauvegarde forcée, initiée par un thread externe au thread persistant, la classe *PersistentThreadManagement* propose deux autres méthodes *store* et *storeAndStop*. Ces deux méthodes prennent en

[1] L'interface complète de nos services de persistance est décrite dans la section 3 de l'Annexe I.

paramètre le thread persistant, en plus du nom du fichier de sauvegarde. Mais contrairement aux deux premières méthodes *store* et *storeAndStop* qui s'appliquent sur une instance de la classe *java.lang.Thread*, ces deux autres méthodes s'appliquent sur des instances de la classe *CapturableThread*.

La reprise d'un thread à partir d'une image sur un fichier se fait par la méthode *load*. Cette méthode prend en paramètre le nom du fichier utilisé pour la sauvegarde, crée un nouveau thread ayant le même état d'exécution que le thread sauvegardé et lance l'exécution de ce thread. Ce thread nouvellement créé est retourné en résultat de la méthode *load*.

7. Construction des services de mobilité et de persistance

Après avoir présenté l'interface de nos services de mobilité et de persistance des threads Java, voici une description de la mise en œuvre de ces services au-dessus de la capture/restauration d'état.

7.1. Services de mobilité

La mobilité d'un thread a été présentée dans la section 1.1 de ce chapitre, comme consistant, tout d'abord, à :

1. suspendre l'exécution du thread,
2. capturer son état d'exécution,
3. arrêter ou poursuivre l'exécution du thread, selon l'utilisation de la mobilité
4. et envoyer l'état capturé vers un site distant, à travers le réseau.

La mise en œuvre de ces opérations est décrite par la Figure 3-15, où :

- la suspension de l'exécution et la capture de l'état sont effectuées par la méthode *ThreadStateManagement.capture*,
- l'arrêt éventuel de l'exécution est réalisé par la méthode *Thread.stop*
- et l'envoi de l'état est effectué par la méthode *SendInterface.sendState*.

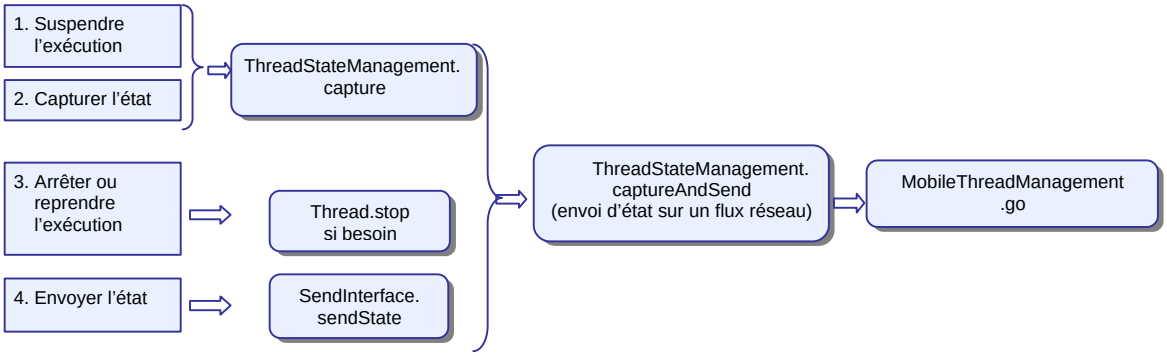

Figure 3-15. Mise en œuvre de la mobilité de thread Java / Site source

Ces traitements sont, en fait, regroupés dans la méthode *ThreadStateManagement.captureAndSend* qui est spécialisée pour l'envoi de l'état d'exécution capturé vers un flux réseau sortant. Cette spécialisation n'est autre que la méthode *MobileThreadManagement.go*[1], dont la mise en œuvre est illustrée par la Figure 3-16. La mise en œuvre de la méthode *go* est basée sur un appel de la méthode *captureAndSend*, avec un paramètre de type *SendingMobileState*. La classe *SendingMobileState* implante l'interface *SendInterface* et fournit donc une méthode *sendState*. Ici, la méthode *sendState* envoie un état de thread Java sur un flux réseau sortant, reliant le site local à un site distant identifié par un nom de machine et un numéro de port. L'envoi de l'état du thread est basé sur la sérialisation d'objets Java. Du fait que la sérialisation Java soit adaptable, celle-ci peut être spécialisée par le programmeur de l'application mobiles pour ne pas transférer toutes les informations contenues dans l'état capturé d'un thread ou pour proposer sa propre politique de transfert d'état.

```
public static void go(String host, int port) {
      SendingMobileState sendSt;


      // A SendInterface for sending thread
      // state over the network
      sendSt = new SendingMobileState(host, port);


      // Capturing the state of the current thread,
      // sending it over the network and resuming
      // the thread
      ThreadStateManagement.captureAndSend(sendSt,
                                           false);


}
```

```
public class SendingMobileState
      implements SendInterface {


public SendingMobileState(String host, int port) {
      // Getting an output stream connected
      // to the host with the specified name
      // and at the specified port number

      ...
}
public void sendState(ThreadState state) {
      // Writing the state on the network
      // output stream

      ...
} }
```

Figure 3-16. Méthode *go* et classe *SendingMobileState*

De façon symétrique, l'arrivée d'un thread mobile sur un site destination a été présentée dans la section 1.1 de ce chapitre et consiste à :

1. recevoir l'état d'exécution à partir du réseau,
2. le restaurer dans un nouveau thread
3. et lancer l'exécution du thread restauré.

La Figure 3-17 montre que :

- la réception de l'état est effectuée par la méthode *ReceiveInterface.receiveState*
- et la restauration et le lancement de l'exécution sont effectuées par la méthode *ThreadStateManagement.restore*.

[1] L'interface des services de mobilité est décrite dans la section 2 de l'Annexe I.

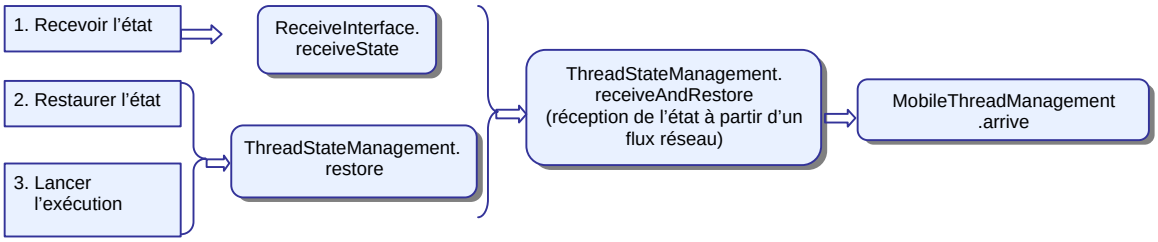

Figure 3-17. Mise en œuvre de la mobilité de thread Java / Site destination

Ces traitements sont regroupés dans la méthode *ThreadStateManagement.receiveAndRestore*, spécialisée ici pour la réception d'un état d'exécution à partir d'un flux réseau entrant. Cette spécialisation produit la méthode *MobileThreadManagement.arrive*, décrite par la Figure 3-18. La réalisation de la méthode *arrive* est faite par un appel à la méthode *receiveAndRestore*, spécialisée avec un paramètre de type *ReceivingMobileState*. La classe *ReceivingMobileState*, qui implante l'interface *ReceiveInterface*, propose une méthode *receiveState* qui reçoit un état de thread Java à partir d'un flux réseau entrant. L'état reçu est reconstruit en utilisant la dé-sérialisation d'objets Java et le chargement dynamique de classes Java. Les mécanismes de chargement et de sérialisation/dé-sérialisation étant adaptables, ceux-ci peuvent être spécialisés par le programmeur de l'application mobile, pour des besoins particuliers de l'application.

```
public static Thread arrive(int port) {
      ReceivingMobileState receiveSt;

      // A ReceiveInterface for receiving thread
      // state from the network
      receiveSt = new ReceivingMobileState(port);

      // Receiving a thread state from the network,
      // restoring it and starting it
      return ThreadStateManagement.receiveAndRestore(
                  receiveSt);

}
```

```
public class ReceivingMobileState
      implements ReceiveState {

public ReceivingMobileState(int port) {
      // Getting an input stream from
      // the network, at the specified
      // port number
      ...
}
public ThreadState receiveState() {
      // Reading a thread state from
      // the network input stream and
      // returning it
      ...
}
}
```

Figure 3-18. Méthode *arrive* et classe *ReceivingMobileState*

7.2. Services de persistance

De même que pour les services de mobilité, les services de persistance des threads Java, proposés par la classe *PersistentThreadManagement*, sont basés sur les mécanismes de capture/restauration de l'état d'exécution des threads.

Considérons, par exemple, la méthode *PersistentThreadManagement.store*. Cette méthode est mise en œuvre avec la méthode *ThreadStateManagement.captureAndSend*, spécialisée pour l'envoi de l'état capturé sur un flux disque sortant. Et de façon symétrique, la méthode *PersistentThreadManagement.load* est construite avec la

méthode *ThreadStateManagement.receiveAndRestore*, spécialisée pour la lecture d'un état de thread à partir d'un flux disque entrant.

8. Conclusion

Dans ce chapitre, nous avons tout d'abord présenté la mise en œuvre complète de nos services de capture et de restauration de l'état d'exécution des threads Java. Nous avons, en particulier, décrit la mise en œuvre de nos hypothèses de conception, hypothèses vérifiant que la capture d'état est :

- forte (voir la section 4.1.3 de ce chapitre)
- décidée ou forcée (voir la section 4.4 de ce chapitre)
- et utilisable sur des plates-formes hétérogènes (voir les sections 4.1.2 et 4.5 de ce chapitre).

Par ailleurs, notre mise en œuvre repose sur une extension de la machine virtuelle Java, à travers la définition de nouvelles structures de données et de nouveaux sous-systèmes pour la machine virtuelle. Cette extension de la JVM est justifiée par :

- le besoin d'accéder aux structures de données de l'état d'exécution des threads Java, structures internes à la JVM
- tout en proposant les performances les plus avantageuses.

Les principales difficultés techniques apportées par la machine virtuelle Java, pour la mise en œuvre des services de capture/restauration, concernent la non portabilité totale de l'état d'exécution Java et la non portabilité de l'état d'exécution des méthodes Java compilées à la volée :

- La non portabilité totale de l'état d'exécution Java a exigé la transformation de la structure non portable de pile Java en structure portable. A cet effet, nous avons proposé et mis en œuvre deux techniques : une technique basée sur l'extension de l'interprète Java et une technique basée sur l'analyse des flots d'exécution.
- Pour faire face au problème de non portabilité de l'état d'exécution des méthodes Java compilées à la volée, nous avons utilisé les techniques de dés-optimisation de code à la volée. Ces techniques permettent de reconstruire un état Java portable à partir d'un état natif non portable.

Nous avons également illustré la généricité et l'adaptabilité de nos mécanismes de capture et de restauration de l'état des threads Java. Cette adaptabilité a été mise en avant à travers la spécialisation de ces mécanismes pour des besoins de mobilité des threads ou pour une mise en œuvre de la persistance des threads.

Une de nos hypothèses de conception était de proposer des services de mobilité et de persistance (et donc de capture/restauration) sur un système existant et répandu. Ce

choix de conception a été fait pour favoriser la diffusion de nos services. A cet effet, nous avons choisi de baser nos services sur la plate-forme universelle Java. Mais notre hypothèse n'est que partiellement vérifiée puisque la mise en œuvre de nos services n'est pas dédiée à toutes les plates-formes Java existantes mais à une mise en œuvre particulière de la JVM (le JDK 1.2.2 dans notre cas). Ce choix de mise en œuvre a été fait pour concilier la faisabilité des services avec leurs performances. Mais pour vérifier cette hypothèse, il faudrait spécifier une nouvelle interface fournies par toutes les implantations de JVM. Cette interface donnerait accès à toutes les informations/opérations nécessaires à la construction de services de capture/restauration d'état des threads Java. Ceci n'est actuellement pas le cas mais risque peut-être de changer puisque les concepteurs de Java se sont récemment intéressés à ce type de fonctionnalités :

« *The persistence of threads, which supports the persistence of active execution state, is considered by many to be unnecessary and, indeed, somewhat eccentric. … As we move closer towards a world of concurrently active applications composed of independently developed and complex parts, with 7*24 uptime requirements, support for persistent threads become more important. … The current generation of Java virtual machines has paid close attention to the exact identification of object references to support garbage collection, but fall some way short of the requirements for persistent threads.* »

Mick Jordan et Malcom Atkinson, Sun Microsystems, décembre 2000 [72].

Finalement, nous avons conçu et mis en œuvre des services de capture/restauration de l'état d'exécution des threads Java, services sur lesquels sont construites des fonctionnalités de mobilité et de persistance des threads. Dans le chapitre suivant, nous décrivons des exemples d'utilisation de ces services ainsi que des expérimentations faites avec ces services.

CHAPITRE 4 - EXPERIMENTATIONS

Nos services de mobilité et de persistance des threads Java sont opérationnels. Ils sont actuellement utilisés par :

- le projet de recherche SUMA, à l'université Simon Bolivar, au Vénézuéla
- et par divers utilisateurs, testeurs, évaluateurs, étudiants et chercheurs, dans divers pays tels que l'Allemagne, l'Australie, le Canada, la Chine, les Etats-Unis, la France, l'Irlande, le Japon ou Singapour.

Ce chapitre illustre, tout d'abord, l'utilisation de nos services de mobilité et de persistance des threads Java à travers des exemples de programmes de migration de thread, de clonage de thread à distance, de sauvegarde puis de reprise de thread. Il présente ensuite deux expérimentations :

- une des applications que nous avons élaborées à des fins de démonstration de nos services de mobilité : une application graphique fractale mobile
- et l'utilisation de nos services de persistance à travers la construction d'un service de sauvegarde/reprise sur la plate-forme de calculs parallèles SUMA.

1. Exemples introductifs

Dans cette première section, nous illustrons :

- l'utilisation de nos services de mobilité à travers des exemples de migration d'application ou de clonage d'application à distance
- et l'utilisation de nos services de persistance à travers des exemples de sauvegarde puis de reprise d'application.

1.1. Quelques remarques et notations

Pour illustrer l'utilisation de nos services de mobilité et de persistance, nous présentons des exemples de programmes accompagnés de leurs traces d'exécution. Pour des raisons de clarté, les programmes Java présentés n'incluent pas la gestion des exceptions. Quant aux traces d'exécution, voici quelques remarques et notations les concernant. Chaque trace d'exécution présente :

- La commande de lancement du programme associé.
- Les messages affichés par le programme exécuté. L'exécution d'un programme pouvant impliquer plusieurs threads, différentes couleurs sont utilisées pour représenter les messages affichés par différents threads.
- Chaque message affiché commence par le nom de la méthode dans laquelle le message a été affiché. Un message contient éventuellement :
  - un des termes *Begin* ou *End* pour indiquer respectivement le début ou la fin de la méthode Java affichant le message,
  - le niveau d'itération d'un calcul itératif de la méthode, indiqué par "*iter = niveau*"
  - le niveau de récursivité d'un calcul récursif de la méthode, indiqué par "*rec = niveau*"
  - les nom, priorité et groupe du thread qui exécute la méthode, indiqués par l'expression "*thread = Thread[nom, priorité, groupe]*",
  - les nom, priorité et groupe du thread mobile qui arrive sur une machine destination, indiqués par l'expression "*Arrived thread = Thread[nom, priorité, groupe]*",
  - les nom, priorité et groupe du thread cloné, indiqués par l'expression "*Cloned thread = Thread[nom, priorité, groupe]*",
  - les nom, priorité et groupe du thread sauvegardé, indiqués par l'expression "*Checkpointed thread = Thread[nom, priorité, groupe]*",
  - les nom, priorité et groupe du thread restauré, indiqués par l'expression "*Recovered thread = Thread[nom, priorité, groupe]*".

Prenons l'exemple de message affiché suivant : "*run: Begin, thread = Thread[Thread-0,5,main]*". Ce message signifie que l'exécution de la méthode *run* vient de commencer et que cette méthode est exécutée par le thread de nom *Thread-0*, qui est de priorité *5* et qui fait partie du groupe de threads de nom *main*.

1.2. Utilisation des services de mobilité

Voici quelques exemples illustrant l'utilisation de nos services de mobilité des threads Java. Ces exemples présentent deux cas de mobilité : la migration de threads Java et le clonage de threads Java à distance.

1.2.1. Migration de thread

La migration de thread est le déplacement de l'exécution d'un thread d'une machine source vers une machine destination, de telle sorte que l'exécution du thread soit définitivement arrêtée sur la machine source pour être poursuivie sur la machine destination. Cet exemple de migration de thread illustre une migration décidée par le thread migrant. Autrement dit, c'est le thread migrant qui appelle notre service de mobilité des threads Java sur lui-même.

```
/*
 * Command line: java Source <target host name> <target port>
 */

public class Source {


public static void main(String[] args) {
    String   host = args[0];
    int      port = Integer.parseInt(args[1]);
    Thread   cur  = Thread.currentThread();
    MyThread myThr ;


    System.out.println("main: Begin, thread = " + cur);

    // Thread creation
    myThr = new MyThread(host, port)
    myThr.start();

    System.out.println("main: End, thread = " + cur);
}
}
```

```
/*
 * Command line: java Target <local port number>
 */

import java.lang.threadpack.*;

public class Target {

public static void main(String[] args) {
    int    port     = Integer.parseInt(args[0]);
    Thread thr, cur = Thread.currentThread();

    System.out.println("main: Begin, thread = " + cur);
    System.out.println("main: Waiting for an incoming
      thread...");

    // Thread reception
    thr = MobileThreadManagement.arrive(port);

    System.out.println("main: Arrived thread = " + thr) ;
    System.out.println("main: End, thread = " + cur);

} }
```

```
import java.lang.threadpack.*;
public class MyThread extends Thread
    implements java.io.Serializable {
// Target machine
private transient String host;
private transient int port;

public MyThread(String host, int port) {
    this.host = host;  this.port = port; }

public void run() {
    int res;
    System.out.println("run: Begin, thread = " + this);
    res = modif(5);
    System.out.println("run: End, Result returned from modif = " + res +
        ", thread = " + this);}

public int modif(int val) {
    int local_var =2 * val;
    System.out.println("modif: Before migration, thread = " + this);
    // Migration
    MobileThreadManagement.goAndStop(host, port);
    System.out.println("modif: After migration, thread = " + this);
    return local_var; } }
```

Figure 4-1. Migration de thread Java – Programme

Dans cet exemple, deux machines virtuelles Java (JVM) sont lancées : une JVM source et une JVM destination. Un thread est créé et lancé sur la JVM source. Au cours

de son exécution, ce thread migre de la JVM source vers la JVM destination où il poursuit son exécution. L'exemple est constitué des trois classes illustrées par la Figure 4-1 :

- La classe *Target* lance la JVM destination qui se met en attente d'un thread mobile, en appelant la méthode *arrive* de la classe *MobileThreadManagement*.
- La classe *Source* lance la JVM source, laquelle crée un thread de la classe *MyThread* et lance son exécution.
- La classe *MyThread* est une sous-classe de la classe *Thread*, elle représente l'application exécutée par le thread migrant. Un thread de la classe *MyThread* commence par exécuter la méthode *run*, qui appelle la méthode *modif* qui elle-même appelle notre service de mobilité des threads Java : la méthode *goAndStop* de la classe *MobileThreadManagement*.

Un thread Java étant à la fois un flot d'exécution et un objet Java, la migration d'un thread implique la migration de l'objet Java associé. La migration de l'objet *MyThread* étant basée sur la sérialisation Java, la classe *MyThread* doit implanter l'interface *java.io.Serializable*.

D'autre part, cet exemple de migration de thread est basé sur un système de chargement de classes sur chacune des machines visitées par le thread migrant. Ce système doit être capable de retrouver et de charger les classes applicatives du thread migrant :

- Une première proposition pour un tel système est le système de chargement de classes Java par défaut. Avec un tel système, les classes applicatives du thread doivent se trouver sur chaque machine visitée par le thread (par copie des classes ou par partage des classes via un système de gestion de fichiers répartis tel que NFS).
- Un autre système de chargement de classes peut être un système de chargement de classes via le réseau. Ce système charge les classes applicatives du thread à partir de la machine de départ du thread (machine source), via un serveur web par exemple.

Les remarques concernant la sérialisation de l'objet *MyThread* et le système de chargement des classes applicatives du thread sont également valables pour les exemples qui suivent.

La trace d'exécution de l'exemple de migration de thread Java est présentée par la Figure 4-2. Elle décrit le déplacement du thread *MyThread* de la JVM source vers la JVM destination :

- Sur la console de la JVM source :

- Il y a tout d'abord les messages verts affichés par le thread principal de nom *main*. Ce thread exécute le programme de la classe *Source* qui crée et lance un thread de la classe *MyThread*, puis se termine.
- Il y a ensuite les messages bleus affichés par le thread *MyThread* de nom *Thread-0*. Ce thread commence effectivement par appeler la méthode *run* qui appelle la méthode *modif*, avant de migrer vers la JVM destination. L'exécution de ce thread se termine et la JVM source s'arrête.

- Sur la console de la JVM destination :
  - Les messages verts sont les messages affichés par le thread principal de nom *main*. Ce thread se met en attente et dès qu'un thread mobile arrive, il affiche son nom puis se termine.
  - Le thread mobile, qui a pour nom *Thread-0*, poursuit son exécution par l'affichage des messages bleus. Il poursuit l'exécution de la méthode *modif* puis de la méthode *run* avant de se terminer.

Console de la JVM source

```
% java Source <host name> <port number>
 main: Begin, thread = Thread[main,5,main]
 main: End, thread = Thread[main,5,main]
 run: Begin, thread = Thread[Thread-0,5,main]
 modif: Before migration, thread = Thread[Thread-0,5,main]
%
```

Console de la JVM destination

```
% java Target <port number>
 main: Begin, thread = Thread[main,5,main]
 main: Waiting for an incoming thread...
 main: Arrived thread = Thread[Thread-0,5,main]
 main: End, thread = Thread[main,5,main]
 modif: After migration, thread = Thread[Thread-0,5,main]
 run: End, Result returned from modif = 10, thread = Thread[Thread-0,5,main]
 %
```

Figure 4-2. Migration de thread Java – Trace d'exécution

La migration du thread Java a donc bien mené au transfert de l'exécution du thread de la JVM source à la JVM destination. Arrivé sur la JVM destination, le thread migrant a commencé son exécution au point où celle-ci a été interrompue sur la JVM source, en préservant l'imbrication des appels de méthodes et les valeurs des variables locales aux méthodes (la variable *local_var* locale à la méthode *modif*). De plus, la prise en compte de la mobilité s'est faite de façon transparente à la programmation de l'application mobile, puisque le seul changement apporté à l'application est l'appel à la primitive de migration.

1.2.2. Clonage de thread à distance

Le clonage d'un thread Java à distance est la création, sur une JVM distante, d'un clone de thread se trouvant sur la JVM locale. Le clonage de thread à distance peut se traduire par une migration de thread dans laquelle l'exécution sur la machine source

n'est pas arrêtée mais poursuivie. Smith et al. proposent une primitive *fork* de clonage à distance [122].

Notre exemple de clonage de thread illustre un clonage forcé dans lequel un thread externe au thread cloné appelle notre service de mobilité des threads Java sur le thread cloné.

```
/*
 * Command line: java Source <target host name> <target port>
 */
import java.lang.threadpack.*;
public class Source {

public static void main(String[] args) {
    String   host = args[0];
    int      port = Integer.parseInt(args[1]);
    Thread   cur  = Thread.currentThread();
    MyThread myThr;
    System.out.println("main: Begin, thread = " + cur);

    // Thread creation
    myThr=new MyThread(10000);
    myThr.start();  cur.yield();

    // Thread migration
    MobileThreadManagement.go(myThr, host, port);

    System.out.println("main: End, Cloned thread = " +
      myThr); }}
```

```
/*
 * Command line: java Target <local port number>
 */
import java.lang.threadpack.*;

public class Target {

public static void main(String[] args) {
    int    port     = Integer.parseInt(args[0]);
    Thread thr, cur = Thread.currentThread();

    System.out.println("main: Begin, thread = " + cur);
    System.out.println("main: Waiting for an incoming
      thread...");

    // Thread reception
    thr = MobileThreadManagement.arrive(port);

    System.out.println("main: Arrived thread = " + thr);
    System.out.println("main: End, thread = " + cur);} }
```

```
import java.lang.threadpack.*;
public class MyThread extends CapturableThread {
// Maximum loop count
private int maxCount;
public MyThread(int cnt) {
  this.maxCount = cnt;
}
public void run() {
  System.out.println("run: Begin, thread = " + this);
  count();
  System.out.println("run: End , thread = " + this);
}
public void count() {
  int i;
  System.out.println("count: Begin,thread = " + this);
  // A long computation
  for (i = 0; i < maxCount; i++) {
    System.out.println("count: iter = " + i + ",thread = " +
      this);
  }
  System.out.println("count: End,thread = " + this); } }
```

Figure 4-3. Clonage de thread Java à distance – Programme

Dans cet exemple, deux machines virtuelles Java sont lancées : une JVM source et une JVM destination. Un thread est créé et lancé sur la JVM source. Au cours de l'exécution du thread, un clone de ce thread est créé sur la JVM destination. Le thread cloné et le thread clone poursuivent alors leur exécution : l'un sur la JVM source et l'autre sur la JVM destination. Cet exemple est constitué des trois classes présentées par la Figure 4-3 :

- La classe *Target* lance la JVM destination qui se met en attente d'un thread mobile, en appelant la méthode *arrive* de la classe *MobileThreadManagement*.
- La classe *Source* lance la JVM source, laquelle crée un thread de la classe *MyThread* et lance son exécution. C'est le programme de cette classe *Source* qui demandera le clonage distant forcé du thread *MyThread*, en appelant la méthode *go* de la classe *MobileThreadManagement* sur le thread *MyThread*.

- La classe *MyThread* représente l'application exécutée par le thread qui sera cloné à distance. Cette classe est une sous-classe de la classe *CapturableThread* puisque le clonage distant va être forcé par un autre thread. Un thread de la classe *MyThread* commence par exécuter la méthode *run*, qui appelle la méthode *count*. Cette dernière méthode effectue un calcul itératif plus ou moins long, en fonction de la valeur de la variable d'instance *maxCount* de l'objet *MyThread*.

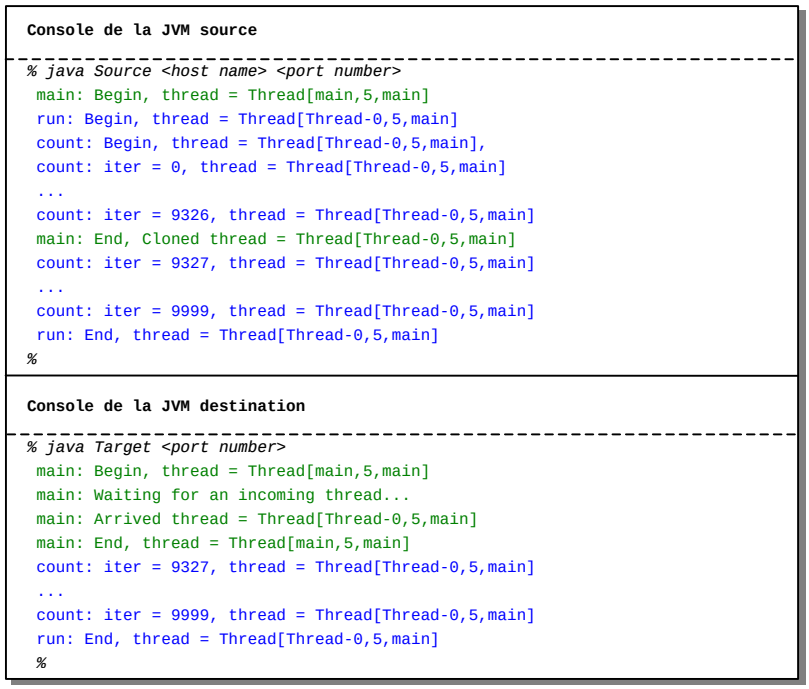

Figure 4-4. Clonage de thread Java à distance – Trace d'exécution

La trace d'exécution de cet exemple est présentée par la Figure 4-4. Elle décrit le clonage du thread *MyThread* de la JVM source sur la JVM destination :

- Sur la console de la JVM source :
  - Le premier message vert est un message affiché par le thread principal de nom *main*. Ce thread exécute le programme de la classe *Source* qui crée un thread de la classe *MyThread* et lui passe la main.
  - La première suite de messages bleus est affichée par le thread *MyThread* de nom *Thread-0*. Ces messages correspondent à l'appel de la méthode *run* puis à l'appel de la méthode *count* qui effectue une partie du calcul itératif.
  - Au cours de l'exécution de la méthode *count*, l'exécution du thread *MyThread* est interrompue pour passer la main au thread principal. Un second message vert est alors affiché par le thread principal. Ce message correspond à l'ordre de clonage distant effectué par le thread principal sur le thread *MyThread*. Le thread principal se termine.
  - Le thread *MyThread* reprend alors son exécution en affichant la seconde suite de messages bleus. Ces messages correspondent à la suite du calcul effectué

par la méthode *count* puis à la suite de la méthode *run*, avant la terminaison du thread.

- Sur la console de la JVM destination :
  - Les messages verts sont les messages affichés par le thread principal de nom *main*. Ce thread se met en attente d'un thread clone. Lorsque le thread clone arrive, il affiche son nom puis se termine.
  - Le thread clone, qui a pour nom *Thread-0*, poursuit son exécution par l'affichage des messages bleus. Il poursuit le calcul effectué par la méthode *count* puis la méthode *run*, avant de se terminer.

Le clonage du thread Java à distance a donc bien mené à la création d'un thread Java clone sur une JVM distante. Après le clonage, le thread initial et le thread clone s'exécutent en parallèle, le premier sur la JVM locale et le second sur la JVM distante. Cet exemple illustre la mobilité forcée et la mobilité forte à travers la préservation de l'état du calcul récursif et la préservation de la valeur de la variable d'instance *maxCount*. Cet exemple illustre également la transparence de la mobilité à la programmation de l'application mobile puisque le seul changement apporté à la classe *MyThread* est le fait qu'elle hérite de la classe *CapturableThread*.

1.3. Utilisation des services de persistance

Voici un exemple illustrant l'utilisation de nos services de persistance des threads Java. Cet exemple présente les deux opérations sur lesquelles est basée la persistance d'un thread : la sauvegarde du thread et la reprise du thread.

1.3.1. Sauvegarde d'un thread Java

La sauvegarde d'un thread est l'opération de stockage de l'état courant de l'exécution du thread sur un support persistant tel qu'un fichier sur disque. Cet exemple de sauvegarde de thread illustre une sauvegarde forcée par un autre thread que le thread sauvegardé.

Dans cet exemple, une JVM est lancée et un thread Java y est créé et lancé. Au cours de l'exécution du thread, une sauvegarde du thread est effectuée puis l'exécution du thread est arrêtée. L'arrêt du thread permet de simuler une panne survenue après une opération de sauvegarde. Les deux classes qui constituent cet exemple sont présentées par la Figure 4-5 :

- La classe *Checkpointing* lance la JVM qui crée un thread de la classe *MyThread* et lance son exécution. C'est le programme de cette classe qui ordonnera la sauvegarde du thread *MyThread*, en appelant la méthode *storeAndStop* de la classe *PersistentThreadManagement*.
- La classe *MyThread* représente l'application exécutée par le thread qui sera sauvegardé. Cette classe est une sous-classe de la classe *CapturableThread*

puisque la sauvegarde se fera de façon forcée. Un thread de la classe *MyThread* commence par exécuter la méthode *run*, qui appelle la méthode récursive *count*. L'exécution du thread *MyThread* sera donc plus ou moins longue, en fonction du niveau de récursivité du premier appel de la méthode *count*.

```
/*
 * Command line: java Checkpointing <file name>
 */

import java.lang.threadpack.*;

public class Checkpointing {

public static void main(String[] args) {
    String   file = args[0];
    MyThread myThr;
    Thread   curr = Thread.currentThread();

    System.out.println("main: Begin, thread = " + curr);

    // Creating a thread
    myThr = new MyThread(1000);
    myThr.start();
    curr.yield();

    // Thread checkpointing
    PersistentThreadManagement.storeAndStop(myThr, file);

    System.out.println("main: End, Checkpointed thread = "
      + myThr);
} }
```

```
import java.lang.threadpack.*;
public class MyThread extends CapturableThread {

// Recursion depth
private transient int recurs;

public MyThread(int rec) {
    this.recurs = rec;
}
public void run() {
    System.out.println("run: Begin, thread = " + this);
    count(recurs);
    System.out.println("run: End, thread = " + this);
}
// Recursive method
public void count(int rec) {
    System.out.println("count: Begin, rec = " + rec + ", thread = " +
        this);
    if (rec > 0)
        count(rec - 1);
    System.out.println("count: End, rec = " + rec + ", thread = " +
        this);
} }
```

Figure 4-5. Sauvegarde de thread – Programme

La trace d'exécution de cet exemple est présentée par la Figure 4-6 :

- Le premier message vert est un message affiché par le thread principal de la JVM, de nom *main*. Ce thread exécute le programme de la classe *Checkpointing* qui crée un thread de la classe *MyThread* et lui passe la main.
- La première suite de messages bleus est affichée par le thread *MyThread* de nom *Thread-0*. Ces messages correspondent à l'appel de la méthode *run* puis aux appels récursifs de la méthode *count*.
- Au cours d'une exécution de la méthode *count*, l'exécution du thread *MyThread* est interrompue pour passer la main au thread principal. Un second message vert est alors affiché par le thread principal. Ce message correspond à l'ordre de sauvegarde du thread *MyThread*. La sauvegarde sur fichier est alors effectuée puis le thread *MyThread* est arrêtée et le thread principal se termine.

  La reprise du thread à partir de cette sauvegarde est présentée par l'exemple de la section suivante.

```
% java Checkpointing <file name>
 main: Begin, thread = Thread[main,5,main]
 run: Begin, thread = Thread[Thread-0,5,main]
 count: Begin, rec = 1000, thread = Thread[Thread-0,5,main]
 count: Begin, rec = 999, thread = Thread[Thread-0,5,main]
 …
 count: Begin, rec = 856, thread = Thread[Thread-0,5,main]
 main: End, Checkpointed thread = Thread[Thread-0,5, main]
%
```

Figure 4-6. Sauvegarde de thread – Trace d'exécution

1.3.2. Reprise d'un thread Java

La reprise d'un thread est la restitution, à partir d'une image persistante, de l'exécution du thread. Le thread repris commence son exécution au point où celle-ci a été interrompue au moment de la sauvegarde.

Dans cet exemple, une JVM est lancée et appelle une opération de reprise d'un thread à partir d'une image persistante stockée sur un fichier. L'image persistante est l'image construite par l'exemple de sauvegarde de thread, présenté précédemment. Cette reprise peut, par exemple, illustrer un redémarrage après panne. Les deux classes qui constituent cet exemple sont :

- La classe *MyThread* qui représente l'application exécutée par le thread persistant. Cette classe est la même classe *MyThread* présentée par la Figure 4-5 car cet exemple de reprise est une reprise du thread sauvegardé dans l'exemple précédent.
- La classe *Recovery* qui est présentée par la Figure 4-7. Cette classe lance la JVM et ordonne la reprise du thread à partir d'un fichier, en appelant la méthode *load* de la classe *PersistentThreadManagement*.

```
/*
 * Command line: java Recovery <file name>
 */

import java.lang.threadpack.*;

public class Recovery {

public static void main(String[] args) {
    String file      = args[0];
    Thread thr, curr = Thread.currentThread();

    System.out.println("main: Begin, thread = " + curr);

    // Thread Recovery
    thr = PersistentThreadManagement.load(file);

    // Recovered thread
    System.out.println("main: End, Recovered thread = " +
      thr);

}
}
```

Figure 4-7. Reprise d'un thread Java – Programme

La trace d'exécution de cet exemple, présentée par la Figure 4-8, montre que la reprise du thread commence au point où l'exécution a été interrompue lors de la sauvegarde :

- Les messages verts sont affichés par le thread principal de la JVM, de nom *main*. Ce thread exécute le programme de la classe *Recovery* qui demande la reprise d'un thread à partir d'un fichier. Le thread principal se termine alors que le thread restauré poursuit son exécution.
- Les messages bleus sont affichés par le thread restauré *MyThread*, de nom *Thread-0*. Ces messages correspondent à la suite de l'exécution des appels

récursifs de la méthode *count* puis à la suite de l'exécution de la méthode *run* avant la terminaison du thread.

```
% java Recovery <file name>
 main: Begin, thread = Thread[main,5,main]
 main: End, Recovered thread = Thread[Thread-0,5,main]
 count: Begin, rec = 855, thread = Thread[Thread-0,5,main]
 ...
 count: Begin, rec = 0, thread = Thread[Thread-0,5,main]
 count: End, rec = 0, thread = Thread[Thread-0,5,main]
 count: End, rec = 1, thread = Thread[Thread-0,5,main]
 ...
 count: End, rec = 1000, thread = Thread[Thread-0,5,main]
 run: End, thread = Thread[Thread-0,5,main]
%
```

Figure 4-8. Reprise d'un thread Java – Trace d'exécution

Le thread restauré commence ainsi son exécution au point où celle-ci a été interrompue au moment de la sauvegarde, tout en préservant l'imbrication des appels récursifs effectués.

1.4. Conclusion

Les exemples présentés précédemment ont permis d'illustrer la mobilité et la persistance fortes, décidées ou forcées et transparentes à la programmation des applications les utilisant.

Dans les exemples présentés, la force de la mobilité et de la persistance a pu être constatée à travers les comportements suivants. A l'arrivée d'un thread mobile (migrant ou cloné) sur une nouvelle machine ou à la reprise d'un thread persistant :

- Le thread mobile commence son exécution, sur la machine d'arrivée, au point où il a été interrompu, sur la machine de départ. Et le thread persistant commence son exécution, au moment de la reprise, au point où il a été interrompu, au moment de la sauvegarde.
- L'imbrication des appels de méthodes en cours d'exécution par le thread est préservée.
- Les valeurs des variables locales aux méthodes en cours d'exécution sont préservées (exemple de la section 1.2.1).
- Les valeurs des variables d'instance sont préservées (exemple de la section 1.2.2).

De plus, la mobilité et la persistance proposées peuvent être décidées ou forcées :

- Mobilité/Persistance décidées. Un thread peut initier sa propre mobilité ou sa propre persistance (exemple de la section 1.2.1).
- Mobilité/Persistance forcées. Un thread peut initier la mobilité ou la persistance d'un autre thread (exemples des sections 1.2.2 et 1.3.1).

D'autre part, la mobilité et la persistance fournies sont transparentes à la programmation de l'application. Elles sont, en effet, proposées par des services dédiés et n'ont pas exigé une restructuration du programme de l'application.

Après cette introduction à l'utilisation de nos services de mobilité et de persistance des threads Java, voici une description de deux applications construites au-dessus de nos services : la courbe fractale du Dragon, basée sur nos services de mobilité, et la plate-forme de metacomputing SUMA, basée sur nos services de persistance.

2. Courbe fractale du Dragon : Application récursive mobile

Après avoir présentés quelques exemples d'utilisation de nos services de mobilité et de persistance des threads, voici des expérimentations effectuées avec ces services. La première expérimentation met en œuvre la mobilité d'une application récursive graphique, la courbe fractale du *Dragon* [145]. Lors de la visualisation de la courbe fractale du Dragon, une forme de dragon apparaît à un certain niveau de récursivité. Nous avons utilisé nos services de mobilité pour déplacer cette application, au cours de son exécution, d'un site à un autre. Ainsi, l'affichage de la courbe fractale, qui commence sur un site, se poursuit sur un autre, puis sur un troisième et ainsi de suite.

Dans cette section, nous présentons la courbe du Dragon avant de décrire la conception et la mise en œuvre de l'application de démonstration de la mobilité.

2.1. Définition de la courbe du Dragon

La courbe fractale du Dragon est définie comme suit :

Définition : Une courbe du Dragon est caractérisée par deux points P et Q et un degré de récursivité n. C'est entre ces deux points que la courbe est dessinée :

- La courbe du Dragon d'ordre 1 est un vecteur entre deux points P et Q.
- La courbe du Dragon d'ordre n est la courbe du Dragon d'ordre n-1 entre les points P et R, suivie de la même courbe d'ordre n-1 entre les points Q et R.
  Ici, R est un point à droite du vecteur PQ tel que PRQ est un triangle isocèle rectangle en R. Donc, si P et Q ont pour coordonnées respectives (x, y) et (z, t), les coordonnées (u, v) du point R sont :
  $u = (x + z)/2 + (t - y)/2$
  $v = (y + t)/2 - (z - x)/2$

Donc étant donnés un certain degré de récursivité *n* et deux points P et Q de coordonnées respectives (x, y) et (z, t), l'algorithme récursif de la courbe fractale du Dragon se présente comme suit :

Algorithme :

```
Dragon (entiers : n, x, y, z, t)  // Courbe d'ordre n entre les points P et Q
    entiers : u, v
    Si (n = 1) alors
        Dessiner une ligne entre les deux points (x, y) et (z, t)
    Sinon
        u  := (x + z + t - y)/2
        v  := (y + t - z + x)/2
        Dragon(n-1, x, y, u, v)  // Courbe d'ordre n-1 entre les points P et R
        Dragon(n-1, z, t, u, v)  // Courbe d'ordre n-1 entre les points Q et R
```

2.2. Conception de l'expérimentation

La Figure 4-9 illustre un système distribué constitué de trois sites. Une courbe fractale du Dragon de degré 15, caractérisée par les points P(100,100) et Q(200,200), commence son affichage sur le premier site. Au cours de cette exécution, l'affichage de la courbe est transféré vers un second site où il se poursuit, puis vers un troisième site.

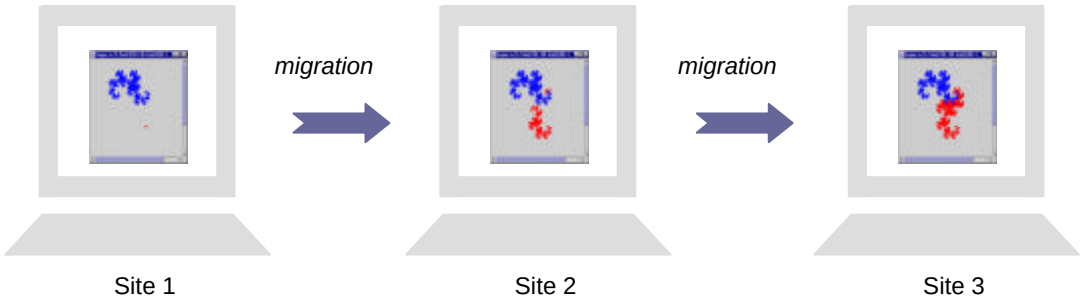

Figure 4-9. Courbe du Dragon mobile

Cette expérimentation est constituée de quatre composants logiciels qui sont le *dragon*, son *interface graphique*, le *site* sur lequel il s'exécute et le *gestionnaire de noms* des dragons :

- Un *dragon* est un composant logiciel qui calcule la courbe du Dragon. Ce composant s'exécute sur un composant *site* et est doté d'un nom unique dans le système distribué de sites.
- Une *interface graphique* est associée à chaque composant dragon. L'interface graphique a pour rôle d'afficher la courbe calculée par le composant dragon auquel elle est associée.
- Un *site* est un composant logiciel qui héberge l'exécution de plusieurs composants dragons. Un composant site propose plusieurs opérations, telles que la création d'un nouveau dragon, la terminaison d'un dragon, la migration d'un dragon vers un autre composant site ou la réception d'un dragon migrant venant d'un autre composant site.
- Un *gestionnaire de noms* est un composant logiciel qui gère l'unicité des noms attribués aux composants dragons s'exécutant dans le système distribué de sites.

Les différents composants logiciels constituant cette expérimentation ont été mis en œuvre comme suit :

- Un composant *dragon* est représenté par un thread Java qui exécute l'algorithme de la courbe du Dragon. Pour autoriser une mobilité forcée du dragon, la classe de ce thread est une sous-classe de notre classe *CapturableThread*.
- Un composant *interface graphique* est représenté par un objet Java graphique associé à un composant dragon.
- Un composant *site* est représenté par une instance de JVM qui héberge l'exécution des threads représentant les composants dragons. Pour fournir une mobilité forte aux dragons, le composant site doit être une instance de notre JVM étendue. De plus, le composant site doit gérer la correspondance entre les noms attribués aux composants dragons s'exécutant sur ce site et les références de threads Java représentant ces dragons.
- Un composant *gestionnaire de noms* est représenté par une instance de JVM standard. Un gestionnaire de noms maintient la liste des noms des composants dragons existants.

Ainsi, le lancement de l'affichage de la courbe du Dragon sur le premier site se traduit par la création d'un composant dragon sur ce site et l'association d'une interface graphique à ce composant. De plus, un nom est attribué au composant dragon par le gestionnaire des noms ; ce nom est désigné ici par *NomDragon*. Le lancement de l'exécution de ce composant dragon provoque alors l'affichage de la courbe sur le premier site.

Au cours de cet affichage, une requête de migration est faite sur ce premier site : elle concerne la dragon de nom *NomDragon* et a pour destination le second site. Cette requête est traitée par le composant site 1, comme suit :

1. La recherche du thread représentant le composant dragon qui a pour nom *NomDragon*.
2. L'envoi du thread et du nom du composant dragon au second site, en utilisant notre service de mobilité forcée des threads.

Le composant site 2 reçoit alors un thread migrant (un composant dragon) et un nom de composant dragon, en utilisant notre service de réception de threads mobiles. Le composant dragon migrant possède alors une interface graphique contenant les derniers affichages effectués avant sa migration. Ce dragon peut alors poursuivre son calcul et son affichage de la courbe à partir du point où il a été interrompu au moment de la migration.

Par la suite, une seconde requête de migration du dragon de nom *NomDragon* est faite sur le second site, pour une migration vers le troisième site …

2.3. Conclusion

A travers cette expérimentation, nous avons pu vérifier des hypothèses de conception de la mobilité qui sont relatives à :

- la force de la mobilité,
- son mode d'initiation forcée
- et sa transparence pour l'application mobile.

Concernant la force de la mobilité, celle-ci est mise en évidence à travers la continuité de l'affichage de la courbe sur les sites d'arrivée de l'application mobile. En effet, arrivée sur son site destination, l'application récursive mobile n'a pas perdu les contextes imbriqués de ses appels récursifs et a pu ainsi poursuivre son exécution au point où elle avait été interrompue sur le site de départ.

Pour ce qui est du mode d'initiation forcé de la mobilité, celui-ci est illustré par le fait que la migration d'un composant dragon est initiée par un autre composant : le composant site qui l'héberge. En effet, c'est le thread principal de la JVM représentant le composant site qui ordonne la migration du thread représentant le composant dragon.

Quant à la transparence de la mobilité pour l'application mobile, elle se décline sous deux formes : une transparence à la programmation de l'application mobile et une transparente à l'exécution de l'application mobile.

La mobilité d'un composant dragon n'a, en effet, pas exigé de restructuration du programme de ce composant ni une modification de l'algorithme de la courbe du Dragon. Cette transparence à la programmation est quasi-totale, à une exception près. En effet, pour autoriser la mobilité forcée, la classe du composant dragon doit être une sous-classe de notre classe *CapturableThread*. Mais cette exception à la transparence totale de la mobilité peut être aisément contournée en utilisant un pré-processeur du code du composant dragon.

Pour ce qui est de la transparence à l'exécution de l'application, celle-ci a été abordée sous deux angles : l'identification de l'entité mobile et la gestion des entrées/sorties de l'entité mobile.

- *Identification de l'entité mobile*. La solution adoptée pour identifier l'entité mobile repose ici sur une identification globale et unique des composants dragons dans un système constitué de plusieurs sites. Cette identification globale permet de désigner l'entité mobile, le composant dragon, de la même manière, avant et après sa migration.
- *Gestion des entrées/sorties de l'entité mobile*. L'expérimentation présentée met en œuvre une interface graphique (entrée/sortie) associée à un composant dragon (thread mobile). Puisque le thread qui représente le dragon fait référence au composant interface graphique, l'interface graphique est automatiquement

transférée avec le dragon migrant[1]. Ce transfert de l'interface est basé sur une sérialisation de l'objet interface, sur le site source, puis une dé-sérialisation de l'objet, sur le site destination. Mais même si la dé-sérialisation reconstruit bien l'objet Java interface graphique, ce n'est pas pour autant que cette interface est visible à l'utilisateur. En effet, la reconstruction de l'objet interface ne provoque pas le ré-affichage de l'interface. La dé-sérialisation du composant interface graphique doit donc être suivie d'un rafraîchissement de l'interface. Ceci est effectué en spécialisant la méthode de dé-sérialisation du composant interface graphique pour faire appel à une primitive de rafraîchissement de l'interface.

3. SUMA : Plate-forme de sauvegarde/reprise de calculs parallèles

La seconde expérimentation présentée illustre l'utilisation de nos services de persistance des threads sur une plate-forme de calculs parallèles SUMA [63]. SUMA (Scientific Ubiquitous Metacomputing Architecture) est un système de méta-calcul dédié au calcul scientifique Java et, plus particulièrement, au calcul parallèle à haute performance. Un système de méta-calcul est un système distribué dans lequel l'accès aux ressources hétérogènes du système est fait de manière uniforme. SUMA est ainsi basé sur la machine virtuelle Java pour l'exécution des calculs et sur *mpiJava* pour l'interaction entre les calculs parallèles ; *mpiJava* est une interface Java pour MPI [17].

Les calculs parallèles sont généralement des calculs longs et, après l'occurrence d'une panne, la reprise de ces calculs depuis le début peut être coûteuse. Un mécanisme de sauvegarde/reprise d'application est donc particulièrement utile dans le cas d'une application parallèle. La plate-forme SUMA propose un mécanisme de sauvegarde/reprise de calculs parallèles, en se basant sur nos services de persistance des threads Java.

3.1. Pourquoi avoir choisi nos services de persistance ?

Les concepteurs du mécanisme de sauvegarde/reprise de calculs parallèles sur SUMA ont comparé les différents services de persistance existants dans Java et ont opté pour nos services. Les raisons pour lesquelles ils ont choisi nos services sont présentées dans [31], elles concernent :

- **Portabilité de la sauvegarde**. La sauvegarde d'un calcul doit produire une structure de données portable sur des plates-formes hétérogènes. Une sauvegarde effectuée sur un type de machine doit pouvoir être reprise sur un autre type de

[1] Comportement par défaut du service de mobilité.

machine, en particulier, dans le cas d'une panne définitive de la machine. La structure de données produite par une sauvegarde de calcul peut, par exemple, être un objet Java qui est ensuite restauré sur une autre JVM. Cette condition de portabilité exclut donc les mécanismes de persistance dépendant d'une architecture ou d'un système d'exploitation, tels que le mécanisme de Howell, proposé comme une couche logicielle entre le système d'exploitation et la JVM [67].

- **Persistance forte**. La persistance d'un calcul doit prendre en compte l'état d'exécution du calcul. La persistance doit donc être une persistance forte, basée sur les flots d'exécution (threads Java) et non sur les objets Java, telle que proposée par le système PJama [11].
- **Granularité de la sauvegarde**. La persistance doit se faire sur la base d'un seul calcul, donc d'un thread Java, et non sur l'ensemble de la JVM, telle que proposée par le système Merpati [129].
- **Sauvegarde forcée**. Pour permettre une automatisation de la sauvegarde des calculs, la sauvegarde doit pouvoir être forcée.
- **Transparence de la persistance**. Pour permettre une transparence de la persistance à la programmation d'un calcul persistant, la prise en compte de la persistance ne doit pas induire de modification manuelle du programme du calcul.
- **Performances de la persistance**. Pour un minimum de surcoût sur les performances des calculs parallèles, la persistance d'un calcul ne doit pas reposer sur une injection de code dans le code du calcul, telle que l'approche suivie par Fünfrocken dans [53].

Ces conditions de portabilité, de force, de granularité fine, d'automatisation, de transparence et de performance des services de persistance sont vérifiées par nos services.

3.2. Sauvegarde de calculs parallèles dans SUMA

Il existe deux catégories principales de sauvegarde de calculs parallèles : une sauvegarde coordonnée et une sauvegarde non coordonnée [32]. Dans la sauvegarde coordonnée, les processus qui exécutent les calculs parallèles doivent se synchroniser pour effectuer une sauvegarde globale cohérente. Alors que dans la sauvegarde non coordonnée, les processus parallèles sont sauvegardés de façon plus ou moins indépendante les uns des autres et c'est au moment de la reprise qu'un état global cohérent est construit à partir des sauvegardes effectuées individuellement.

Une sauvegarde globale dans un système constitué de plusieurs processus est l'ensemble des sauvegardes effectuées sur ces processus individuellement. Une sauvegarde globale cohérente est définie par les concepteurs de SUMA comme étant une sauvegarde :

- qui ne contient pas de message *orphelin*
- et qui ne contient pas de message *en transit*.

Un message *orphelin* est un message dont la réception est prise en compte dans la sauvegarde globale mais dont l'émission n'est pas prise en compte dans cette sauvegarde.

Un message *en transit* est un message dont l'émission est prise en compte dans la sauvegarde globale mais dont la réception n'est pas prise en compte dans cette sauvegarde.

Pour illustrer ces définitions, un exemple d'exécution distribuée est donné par la Figure 4-10. Cet exemple illustre l'exécution de trois processus P1, P2 et P3 par une succession d'évènements. Un évènement est soit un évènement local à un processus (évènement e_1^1 local au processus P1), soit un événement d'envoi de message (évènement e_1^2 d'envoi d'un message m1 du processus P1 au processus P2), soit un évènement de réception de message (évènement e_2^2 de réception du message m1 par le processus P2). La sauvegarde globale S1, qui prend en compte les évènements e_1^1, e_2^1, e_2^2 et e_3^1, n'est pas une sauvegarde cohérente car le message m1 y est orphelin. La sauvegarde S2, qui prend en compte les évènements e_1^1, e_1^2, e_2^1, e_2^2, e_3^1 et e_3^2, n'est pas une sauvegarde cohérente car le message m2 y est en transit. Mais si la sauvegarde S2 prend en compte le message m2, cette sauvegarde devient cohérente puisque, lors de la reprise, le message m2 pourra être réémis.

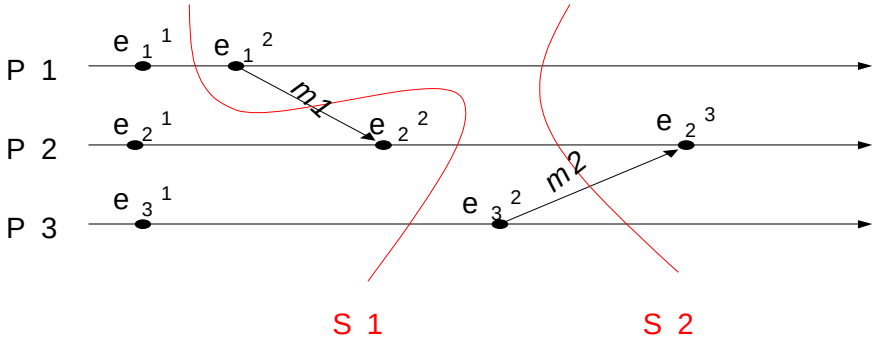

Figure 4-10. Sauvegarde globale

Dans SUMA, la sauvegarde des calculs parallèles est une sauvegarde non coordonnée : les processus parallèles sont sauvegardés indépendamment les uns des autres. Une sauvegarde effectuée individuellement sur un processus utilise nos services de sauvegarde des threads Java. De plus, chaque sauvegarde individuelle est estampillée d'une horloge logique et conserve le message en transit émis par le processus sauvegardé. Ceci permettra de reconstruire un état global cohérent lors de la reprise.

Le protocole de sauvegarde non coordonnée adopté par les concepteurs de SUMA est une combinaison de deux autres protocoles présentés dans [97] et [98] ; il est décrit en détail dans [32].

3.3. Reprise des calculs parallèles dans SUMA

Si une panne survient au cours de l'exécution de calculs parallèles, une exception est levée dans le système SUMA. Suite à cette exception, le système lance une opération de reprise automatique. La reprise construit un état global cohérent à partir des états de processus parallèles sauvegardés séparément. Chaque processus ayant effectué plusieurs sauvegardes successives, il faut déterminer la sauvegarde à prendre en compte pour chaque processus, pour que l'état global construit soit cohérent. De plus, l'état construit lors de la reprise doit être l'état global cohérent le plus récent.

L'opération de reprise est détaillée dans [32] et [31] ; le principe général de son algorithme est le suivant :

- déterminer une estampille *a* commune à toutes les sauvegardes de processus effectuées séparément,
- déterminer, pour chaque processus, la dernière sauvegarde dont l'estampille est inférieure ou égale à l'estampille commune *a*
- restaurer chacun de ces processus, en utilisant nos services de reprise des threads Java
- et réémettre les messages en transit stockés dans les sauvegardes individuelles des processus.

3.4. Conclusion

L'utilisation de nos services de persistance des threads Java dans la plate-forme SUMA a donc permis de construire un mécanisme de plus haut niveau : un mécanisme de sauvegarde/reprise de calculs parallèles. Un tel mécanisme est particulièrement intéressant pour la mise en place de la tolérance aux pannes.

A travers cette utilisation de nos services de persistance, la question de transparence de la persistance à l'exécution de l'application persistante a été abordée à travers la gestion des entrées/sorties de l'application persistante. En effet, la sauvegarde/reprise des applications parallèles dans SUMA repose sur une politique de sauvegarde des messages en transit pour une éventuelle réémission de ces messages lors d'une reprise ultérieure.

4. Conclusion

Dans ce chapitre, nous avons donné des exemples d'utilisation de nos services de mobilité et de persistance des threads Java. Nous avons ensuite décrit une application de

démonstration de nos services de mobilité et présenté l'utilisation de nos services de persistance pour la mise en place de la sauvegarde/reprise d'applications dans une plate-forme de calculs parallèles.

La présentation de l'application de la mobilité et de l'utilisation de la persistance dans la plate-forme SUMA a permis de mettre en évidence certaines hypothèses de conception de nos services de mobilité et de persistance. Ces hypothèses concernent :

- la force de la mobilité et de la persistance,
- le mode d'initiation forcé de la mobilité et de la persistance,
- la transparence de la mobilité et de la persistance à la programmation des applications
- et la transparence de la mobilité et de la persistance à l'exécution des applications mobiles/persistantes. Cette transparence a été mise en évidence à travers des exemples de gestion des problèmes d'identification de l'entité mobile et de gestion des entrées/sorties de l'entité mobile/persistante (interface graphique, communications en cours).

Tout au long de ces premiers chapitres, nous avons abordé toutes les hypothèses de conception de nos services de mobilité et de persistance des applications, excepté une hypothèse : l'hypothèse concernant les performances de ces services. Cette hypothèse est discutée dans le chapitre suivant.

CHAPITRE 5 - EVALUATION

Ce chapitre présente les performances de nos services de mobilité et de persistance des threads Java et les performances de nos mécanismes de capture/restauration de l'état d'exécution des threads. Ces performances résultent de l'évaluation de la latence de nos services, des surcoûts qu'ils induisent sur les applications mobiles/persistantes et des surcoûts induits sur les applications non concernées par la mobilité/persistance.

De plus, une comparaison qualitative et quantitative de nos services avec des services Java proposés par d'autres projets est donnée à la fin du chapitre.

1. Introduction

Les mesures de performances présentées concernent deux techniques de mise en œuvre de nos services de capture/restauration de l'état des threads Java et donc, deux techniques de mise en œuvre de nos services de mobilité/persistance des threads Java. Ces techniques sont désignées dans la suite par :

- IJES (Interprétation Java Etendue Synchronisée). Cette technique est basée sur une gestion de pile de types qui se fait en parallèle à l'interprétation du bytecode exécuté par le thread. Elle est présentée dans la section 4.5.1 du Chapitre 3. Cette mise en œuvre propose une capture décidée et une capture forcée de l'état des threads Java ; ces threads devant être des instances de la classe *CapturableThread*.
- IJSS (Interprétation Java Standard Synchronisée). Cette technique est basée sur une reconnaissance de types et une construction de pile de types, via une analyse des flots d'exécution du thread. Elle est décrite dans la section 4.5.2 du Chapitre 3. Avec cette mise en œuvre, les threads Java dont l'état d'exécution peut être capturé de façon décidée sont des threads de la classe *Thread* et les threads dont l'état peut être capturé de façon forcée sont des threads de la classe *CapturableThread*.

La suite de cette section décrit les paramètres d'évaluation et l'environnement d'évaluation de nos services.

1.1. Paramètres d'évaluation

La durée du traitement effectué par une opération de capture ou de restauration de l'état d'un thread dépend de la taille de l'état d'exécution du thread au moment de la capture. Avec nos services de capture/restauration de l'état des threads Java, l'état considéré d'un thread est son état Java. Dans ce cas, la taille de l'état d'un thread dépend de la taille des trois structures de données qui constituent l'état Java du thread et qui sont la pile Java, le tas d'objets et la zone de méthodes.

La taille de la pile Java d'un thread dépend du nombre et de la taille des frames Java contenus dans cette pile :

- Le nombre de frames Java varie en fonction du nombre d'appels de méthodes imbriquées par le thread.
- La taille d'un frame Java varie en fonction de la taille de sa table de variables locales ou de sa pile d'opérandes. La taille de la table de variables locales dépend évidemment du nombre de variables locales à la méthode Java associée au frame. Et la taille de la pile d'opérandes dépend de la longueur des calculs intermédiaires effectués par la méthode associée au frame.

Quant à la taille du tas d'objets d'un thread, elle dépend du nombre d'objets Java utilisés par le thread. Ces objets sont désignés par des variables locales ou des résultats intermédiaires dans les méthodes Java du thread[1].

La taille de la zone de méthodes d'un thread dépend du nombre de classes Java utilisées par le thread et donc, des méthodes Java exécutées par le thread[2].

Finalement, pour effectuer les mesures de performances de nos services de capture/restauration d'état et de mobilité/persistance des threads, nous avons considéré les paramètres suivants :

- Le nombre de frames sur la pile Java du thread.
- Le nombre d'objets dans le tas d'objets de thread.

Des mesures de performances supplémentaires permettraient de prendre en compte un troisième paramètre : le nombre de classes dans la zone de méthodes du thread.

1.1.1. Variation du nombre de frames

Pour faire varier le nombre de frames sur la pile Java d'un thread, nous avons mis en œuvre une méthode Java récursive. Cette méthode effectue le calcul récursif correspondant à la fonction factorielle et prend en paramètre le degré de récursivité du calcul. Lorsque l'exécution de la méthode récursive arrive à la profondeur maximale de récursivité, un de nos services est appelé : le service de capture d'état pour la mesure du coût d'une capture, le service de restauration d'état pour la mesure du coût d'une restauration et ainsi de suite.

Pour minimiser l'erreur dans la mesure du coût d'un service, plusieurs appels successifs au service sont effectués. Le temps total de cette exécution est divisé par le nombre d'appels au service, pour obtenir le coût d'un appel au service.

De plus, la mesure du coût d'un service est elle-même effectuée plusieurs fois et une moyenne des différents coûts résultants (excepté le coût maximum et le coût minimum) est calculée pour obtenir le coût moyen d'un appel au service.

Par ailleurs, dans le cas de la mesure du coût d'une migration de thread entre deux sites, un problème d'asynchronisme entre les horloges des deux sites se pose. Pour pallier à ce problème, notre programme de mesure du coût de la migration fait faire des aller-retour au thread migrant, pour mesurer le temps initial et le temps final sur un même site.

[1] La variation de la taille du tas implique donc une variation de la taille des frames de la pile Java.

[2] La variation de la taille de la zone de méthodes implique une variation du nombre de frames Java.

1.1.2. Variation du nombre d'objets

Pour faire varier le nombre d'objets dans le tas d'objets d'un thread, nous avons réalisé un générateur de programmes Java. Ce générateur de programme prend en paramètre un nombre *N* et construit un programme Java doté d'une méthode Java qui contient et manipule *N* objets Java (*N* variables locales de types *Integer*). Cette méthode appelle ensuite le service dont le coût doit être mesuré : le service de mobilité, de persistance ou de capture d'état.

Les remarques, présentées dans la section précédente, concernant la minimisation de l'erreur et le calcul du coût moyen d'un service sont également valables ici.

Les résultats des mesures de performances qui prennent en compte le nombre d'objets sont présentés dans l'évaluation comparative quantitative, dans la section 5.2 de ce chapitre.

1.2. Environnement d'évaluation

Nous avons évalué les performances de nos services sur deux types de plates-formes : une plate-forme Windows NT et une plate-forme Solaris. Notre environnement d'évaluation est constitué des éléments suivants :

- JDK 1.2.2
- Solaris 2.6, Sun Ultra-1 (Sparc Ultra-1 167 MHz, 64 Mo RAM)
- Microsoft Windows NT 4 (PIII 600 MHz, 256 Mo RAM)
- Ethernet 100 Mb/s.

D'autre part, nous avons exécuté nos programmes d'évaluation de performances en désactivant le compilateur Java JIT, afin d'éviter une erreur de comparaison due à des optimisations ayant lieu dans certains cas et pas dans d'autres.

De plus, les programmes Java d'évaluation se basent sur le système de chargement dynamique de classes par défaut. Ceci suppose bien sûr que les classes utilisées par les programmes d'évaluation sont accessibles sur tous les sites sur lesquels s'exécutent ces programmes (site de départ et site d'arrivée d'un thread mobile, site sur lequel un thread persistant est sauvegardé et site sur lequel le thread persistant est repris).

Les sections suivantes présentent les résultats de l'évaluation de performances de nos services sur la plate-forme Windows NT. Les résultats obtenus sur la plate-forme Solaris confirment un même comportement sur les deux plates-formes mais n'apportent pas plus d'information. Elles sont données, à titre indicatif, dans l'Annexe III.

Dans la suite de ce chapitre, nous présentons les résultats de l'évaluation de la latence de nos services (section 2), de l'éventuel surcoût qu'ils induisent sur les applications mobiles/persistantes (section 3) et de l'absence de surcoût sur les applications non concernées par la mobilité/persistance (section 4). La section 5

présente ensuite une comparaison de nos services avec d'autres techniques de mise en œuvre.

2. Latence de nos services

Cette section présente les résultats de l'évaluation de la latence de la capture et de la restauration de l'état d'exécution d'un thread Java ainsi que la latence de la migration, de la sauvegarde et de la reprise d'un thread Java.

2.1. Latence de la capture et de la restauration d'état

La Figure 5-1 présente deux courbes : la courbe de la latence de la capture et la courbe de la latence de la restauration, en suivant l'approche IJSS. La première courbe illustre la variation du temps nécessaire à une opération de capture de l'état d'un thread Java lorsque le nombre de frames sur la pile Java du thread varie. La seconde courbe représente la variation du coût d'une opération de restauration de l'état d'un thread Java en fonction du nombre de frames Java présents dans cet état.

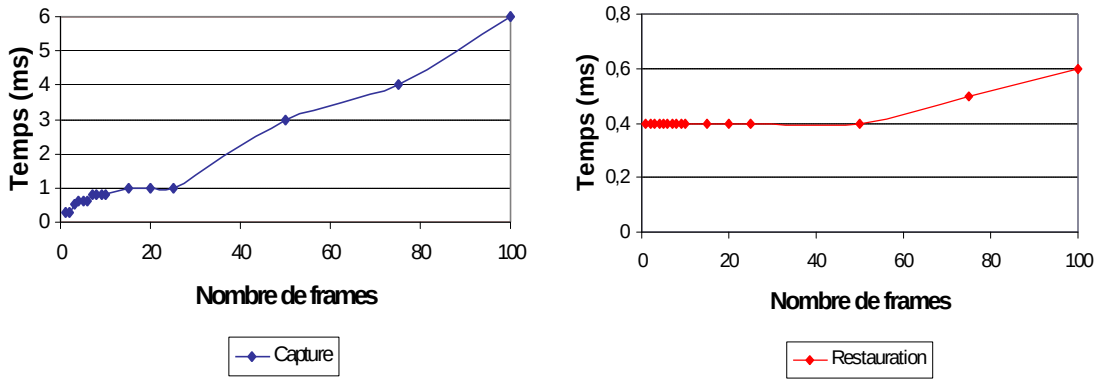

Figure 5-1. Capture et Restauration d'état / IJSS / Windows NT

D'après la première courbe, le coût d'une opération de capture d'état de thread ne dépasse pas la milli-seconde, lorsque la pile Java du thread ne contient pas plus de 25 frames. Ce coût atteint ensuite 3 ms à 50 frames puis 6 ms à 100 frames. Et selon la seconde courbe, une opération de restauration d'état de thread coûte 0,4 ms si le nombre de frames ne dépasse pas 50. Ce coût augmente ensuite, avec l'augmentation du nombre de frames, pour atteindre 0,6 ms lorsque le nombre de frames est égal à 100. Les coûts de ces deux opérations sont raisonnables, surtout lorsque le nombre de frames est inférieur à 25.

La Figure 5-2 présente les mêmes courbes d'évaluation que la figure précédente mais en suivant l'approche IJES. Avec l'approche IJES, le coût d'une opération de capture d'état de thread reste toujours inférieur à la milli-seconde lorsque la pile Java du thread ne contient pas plus de 25 frames. Ce coût augmente ensuite, avec l'augmentation du

nombre de frames, sans jamais dépasser 4 ms, si le nombre de frames est inférieur à 100. Quant au coût d'une opération de restauration d'état, il est de 0,5 ms lorsque le nombre de frames est inférieur à 20 puis atteint et ne dépasse pas 0,6 ms tant que le nombre de frames ne dépasse pas 100.

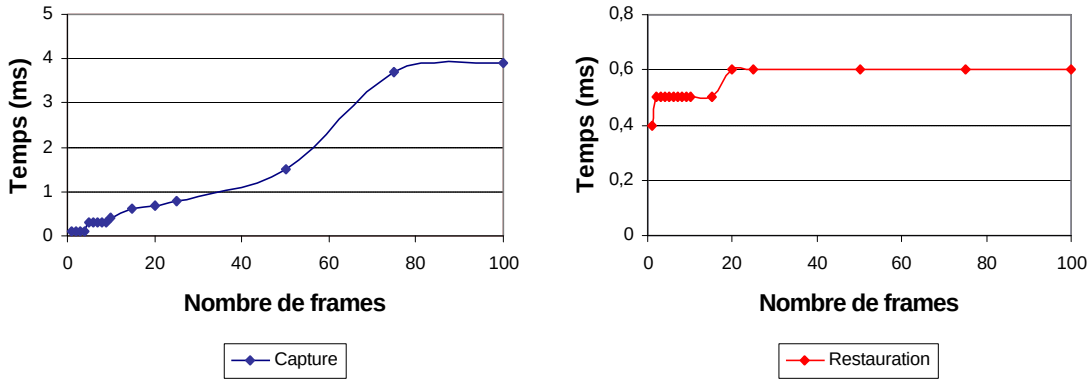

Figure 5-2. Capture et Restauration d'état / IJES / Windows NT

Pour comparer les latences de la capture/restauration avec les deux approches IJSS et IJES, nous avons regroupé les courbes présentées dans les deux figures précédentes pour obtenir la Figure 5-3. Cette comparaison montre que les courbes de capture se comportent de la même manière, avec les deux approches, lorsque la pile Java du thread ne contient pas plus de 25 frames. Ce coût est ensuite globalement supérieur avec l'approche IJSS. Ceci est dû au surcoût induit par la construction de la pile de types du thread lors de la capture d'état avec IJSS (voir la section 4.4.2 du Chapitre 3). Quant à l'opération de restauration, son coût est globalement supérieur avec l'approche IJES. Ce surcoût est dû à la reconstruction de la pile de types à associer au thread lors de la restauration d'état avec IJES (voir la section 4.4.4 du Chapitre 3).

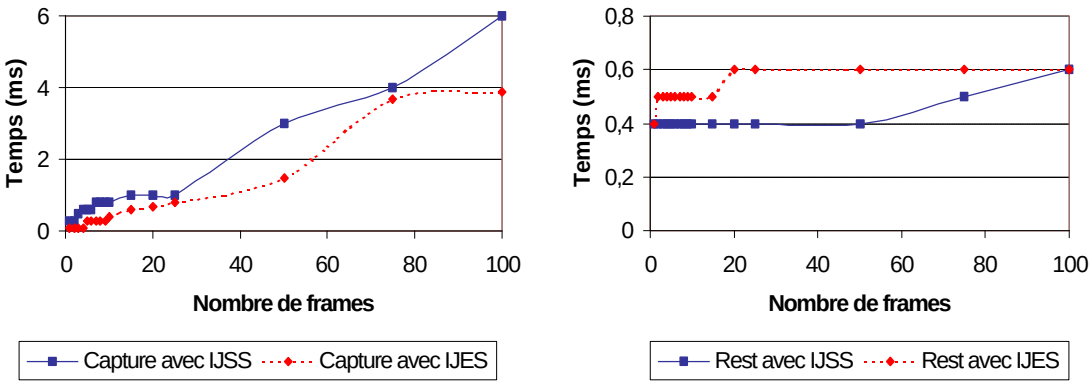

Figure 5-3. Comparaison de la capture et de la restauration avec IJSS et IJES / Windows NT

2.2. Latence de la mobilité

La Figure 5-4 présente une courbe de latence de la migration d'un thread Java entre deux sites. La migration d'un thread étant composée des opérations de capture, de transfert puis de restauration d'état, nous présentons également la courbe du coût de transfert de l'état capturé d'un thread Java d'un site à un autre. Ces courbes résultent de l'évaluation de la latence du service de migration proposé par l'approche IJSS. Les deux courbes illustrent la variation des performances en fonction du nombre de frames sur la pile Java du thread mobile.

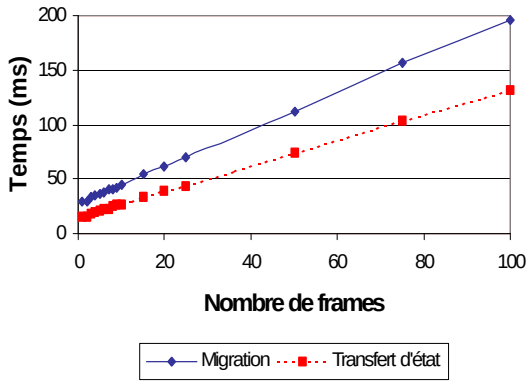

Figure 5-4. Migration de thread / IJSS / Windows NT

Nous constatons que le coût d'une opération de migration de thread entre deux sites varie linéairement : il va de 29 ms, lorsqu'il y a un frame sur la pile Java du thread, à 196 ms, lorsque le nombre de frames atteint 100. Nous constatons également que la majorité du temps requis pour une migration de thread est passée dans le transfert de l'état capturé du thread vers le site destination. Le temps total nécessaire à une opération de migration peut alors être sensiblement réduit en réduisant le temps de transfert de l'état d'un thread sur le réseau. La réduction du temps de transfert de l'état peut, par exemple, être faite en utilisant l'externalisation d'objets Java au lieu d'utiliser la sérialisation d'objets ; l'externalisation peut en effet être jusqu'à 40% plus rapide que la sérialisation [136].

L'évaluation de la latence de la migration avec l'approche IJES est illustrée par la Figure 5-5. Cette figure présente deux courbes qui sont quasiment superposées : une courbe qui décrit le coût d'une opération de migration de thread et une courbe qui donne le temps de transfert de l'état capturé du thread vers un site distant.

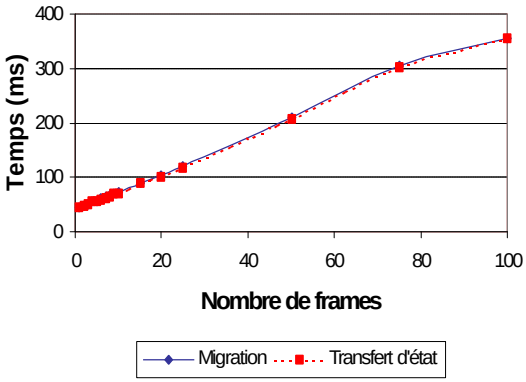

Figure 5-5. Migration de thread / IJES / Windows NT

Avec l'approche IJES, le temps nécessaire à une migration de thread varie entre 45 ms et 356 ms, lorsque le nombre de frames sur la pile Java du thread migrant va de 1 à 100. Ici, au moins 97% du temps de migration est passé dans le transfert de l'état capturé vers le site distant. Comme dans l'approche précédente, ce temps peut réduit en utilisant l'externalisation d'objets Java ou en déléguant le transfert d'état à un autre thread. De plus, la différence entre les coûts de migration et de transfert est moins importante avec cette approche qu'avec l'approche IJSS. Ceci est expliqué dans la suite, lors de la comparaison des deux approches.

Une comparaison de la latence de la migration de thread avec les approches IJSS et IJES est présentée dans la Figure 5-6. Cette comparaison montre qu'une migration de thread avec l'approche IJSS est plus performante qu'une migration avec l'approche IJES. Rappelons ici que le temps de migration mesuré concerne une migration décidée, exécutée par le thread migrant. Tous les traitements relatifs à la migration (capture, transfert et restauration d'état) sont donc exécutés par le thread migrant. Or avec l'approche IJES, le thread migrant est une instance de l'interprète Java étendu qui gère une pile de types associée au thread. Ainsi, les opérations de capture, de transfert et de restauration d'état sont elles-mêmes exécutées par l'interprète Java étendu, d'où le surcoût par rapport à la migration avec l'approche IJSS.

Un moyen d'annuler ce surcoût est de considérer, dans l'interprète Java étendu, deux catégories de traitements :

- Les traitements de l'application exécutée par le thread Java. C'est l'état d'exécution de ces traitements qui sera pris en compte lors de la capture d'état. Ces traitements seront donc exécutés en se basant sur une pile de types, en plus de la pile Java du thread.
- Les traitements relatifs aux services fournis par notre package *threadpack*. Ces traitements seront exécutés sans faire appel à la pile de types du thread.

Cette solution est en cours d'intégration à nos services de capture et de restauration de l'état d'exécution des threads.

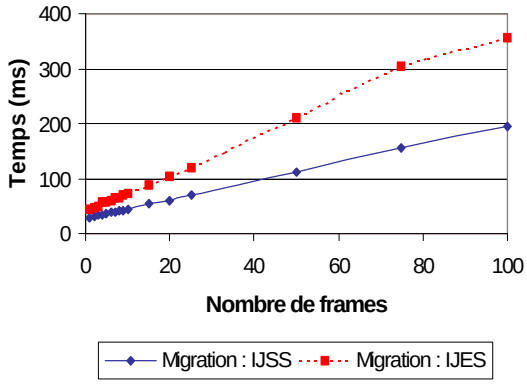

Figure 5-6. Comparaison de la migration avec IJSS et IJES / Windows NT

2.3. Latence de la sauvegarde/reprise

Cette section présente deux évaluations : une évaluation de la latence de la sauvegarde et une évaluation de la latence de la reprise, avec l'approche IJSS. Dans la Figure 5-7, la première évaluation présente une courbe qui illustre la variation du temps nécessaire à la sauvegarde d'un thread Java et une courbe qui représente la variation du temps nécessaire à l'écriture de l'état capturé du thread sur un fichier[1]. Ces deux variations sont données en fonction du nombre de frames sur la pile Java du thread persistant.

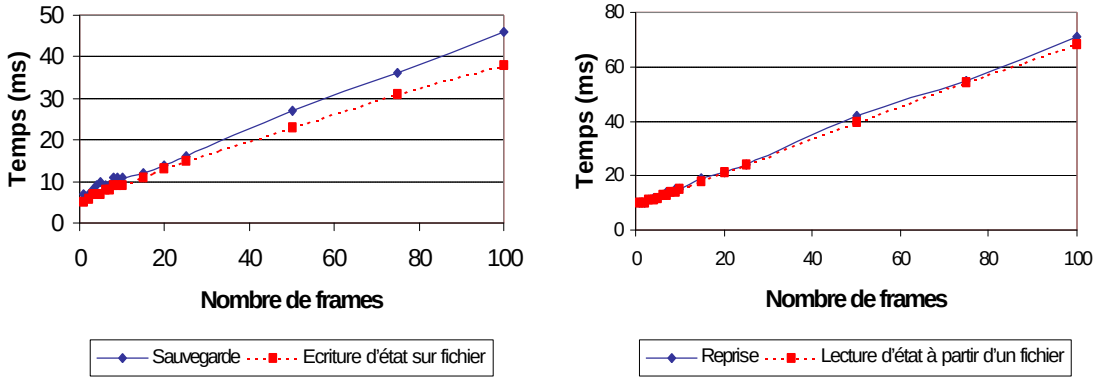

Figure 5-7. Sauvegarde et Reprise de thread / IJSS / Windows NT

La seconde évaluation de la Figure 5-7 présente une courbe qui illustre le coût d'une opération de reprise de thread Java et une courbe qui représente la variation du coût

[1] Sauvegarde de thread = Capture d'état + Ecriture d'état sur fichier

d'une opération de lecture d'état à partir d'un fichier[1]. Ces deux variations sont également données en fonction du nombre de frames sur la pile Java du thread.

Ici, la sauvegarde et la reprise d'un thread varient toutes les deux de façon linéaire. Nous constatons, par ailleurs, que plus de 80% du temps de sauvegarde est passé dans l'écriture de l'état capturé sur un fichier. Ce temps peut être sensiblement réduit en effectuant des écritures asynchrones sur le fichier et en remplaçant la sérialisation par l'externalisation d'objets. De façon symétrique, plus de 90% du temps de reprise est passé dans la lecture d'un état précédemment sauvegardé, à partir d'un fichier.

La Figure 5-8 décrit les mêmes évaluations de performances que la figure précédente, mais en utilisant l'approche IJES. Les services de sauvegarde/reprise de l'approche IJES ont les mêmes comportements que ceux proposés par l'approche IJSS (rapport entre la sauvegarde et l'écriture sur fichier et rapport entre la reprise et la lecture du fichier). Mais les coûts résultant ici et ceux présentés par l'approche IJSS ne sont pas les mêmes.

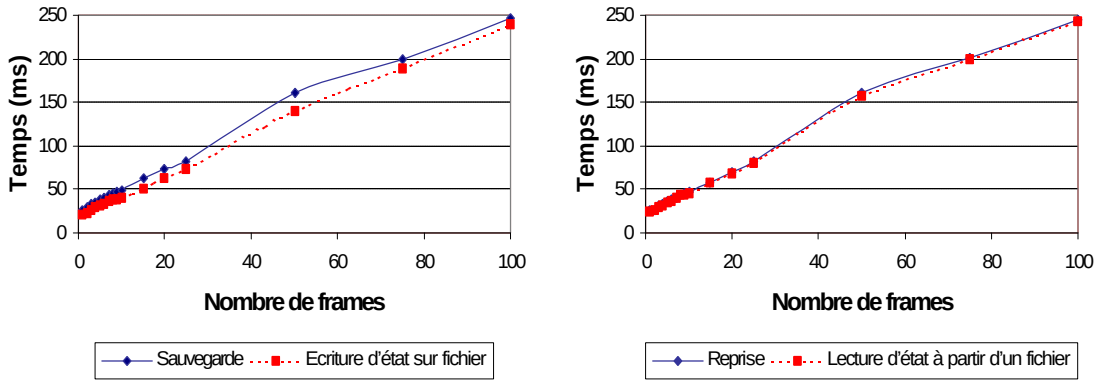

Figure 5-8. Sauvegarde et Reprise de thread / IJES / Windows NT

Une comparaison des coûts des services de sauvegarde/reprise des approches IJSS et IJES est présentée dans la Figure 5-9. Cette comparaison montre que les services de sauvegarde/reprise proposés par l'approche IJSS sont beaucoup plus performants que ceux proposés par l'approche IJES. Cette différence de performances est due au surcoût induit par l'interprète Java étendu sur lequel repose l'approche IJES. En effet, avec l'approche IJES, les opérations relatives à la sauvegarde et à la reprise sont basées sur l'interprète Java étendu, d'où le surcoût avec l'approche IJES. Ce surcoût peut être réduit de la même manière que pour la mobilité.

[1] Reprise de thread = Lecture d'état à partir d'un fichier + Restauration d'état

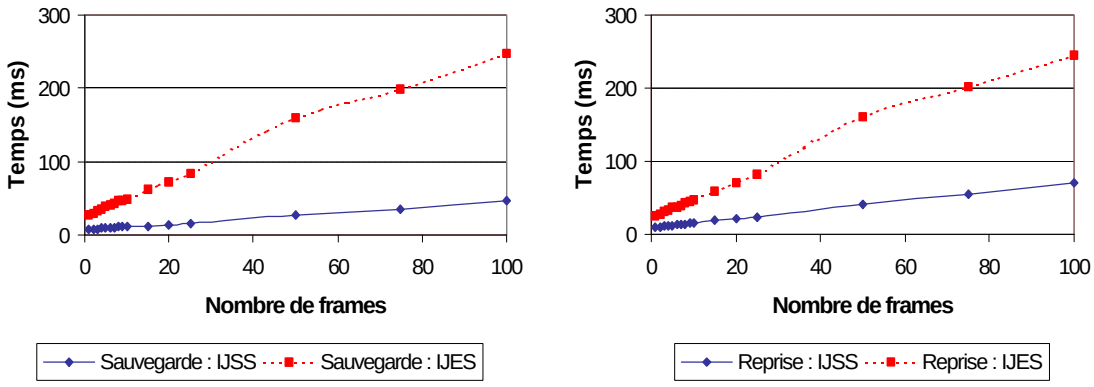

Figure 5-9. Comparaison de la sauvegarde et de la reprise avec IJSS et IJES / Windows NT

3. Surcoût sur les performances des applications mobiles/persistantes

En plus de la mesure de la latence de nos services, nous nous sommes intéressés à l'évaluation de l'éventuel surcoût induit par nos services sur les threads Java dont l'état d'exécution peut être capturé. Nous avons évalué ce surcoût dans le cas où la capture d'état est décidée et dans le cas où la capture est forcée.

A cet effet, nous avons utilisé un benchmark de machine virtuelle Java appelé SciMark 2.0 [110]. SciMark 2.0 est un benchmark de calcul numérique scientifique Java. Il est constitué de cinq calculs : la transformée de Fourier rapide (FFT), la résolution de systèmes linéaires par la méthode Jacobi Successive Over-Relaxation (SOR), l'intégration Monte Carlo, la factorisation de matrice LU et la multiplication de matrices (sparse matmut). Ce benchmark présente également un résultat global (composite score). Tous les résultats sont donnés en Mflops (Millions d'opérations de flottants par seconde). Autrement dit, plus le score résultant d'un calcul est important, plus la JVM sous-jacente est performante pour ce type de calcul.

Les calculs abordés par ce benchmark sont des calculs de petite taille pour éviter les problèmes engendrés par le manque d'espace mémoire et se concentrer sur les performances de calcul de la JVM.

Pour évaluer le surcoût induit par nos services sur les threads dont l'état peut être capturé, nous avons effectué plusieurs mesures avec le benchmark SciMark 2.0, telles qu'illustrées par la Figure 5-10. Du fait que nos extensions de l'interprète Java soient basées sur un interprète écrit en C et pour pouvoir évaluer le surcoût induit par ces extensions, nous avons basé toutes les mesures présentées dans cette section sur un interprète Java écrit en C. Considérons, tout d'abord, les quatre premiers cas qui représentent :

- *JVM standard, Interprète en asm*. Cette mesure concerne l'interprète Java standard de la JVM, écrit en assembleur. Elle nous sert de référence à laquelle nous comparons les résultats obtenus avec nos services. Dans le cas de cette mesure, le benchmark est exécuté par un thread Java standard (instance de la classe *Thread*). Cette mesure est représentée par la barre verte de la figure.
- *IJSS/mode décidé*. Cette mesure évalue l'approche IJSS, dans le cas où une capture d'état peut être décidée. Ici, le benchmark est exécuté par un thread standard de la classe *Thread* (thread capturable de façon décidée), sur notre JVM étendue en suivant l'approche IJSS. Il est important de noter que ce thread se base sur l'interprète standard de la JVM. Ce résultat est représenté par la barre jaune de la figure.
- *IJSS/mode forcé*. Cette mesure évalue l'approche IJSS, où une capture d'état de thread peut être forcée. Dans ce cas, le benchmark est exécuté par un thread de la classe *CapturableThread*, sur notre JVM étendue avec l'approche IJSS. Pour faire exécuter le benchmark par un thread *CapturableThread*, nous avons apporté un changement au benchmark SciMark 2.0 pour lancer son exécution avec un thread *CapturableThread* au lieu d'un thread *Thread*. Un thread *CapturableThread* est basé sur un interprète Java synchronisé. Ce résultat est décrit par la barre jaune-à-points-noirs de la figure.
- *IJES/mode décidé ou forcé*. Cette mesure concerne notre JVM étendue en suivant l'approche IJES, où la capture d'état de thread peut être effectuée de façon décidée ou de façon forcée. Nous proposons une seule mesure pour le mode décidé et pour le mode forcé puisque dans les deux cas, le benchmark est exécuté par un thread *CapturableThread*, sur notre JVM étendue avec l'approche IJES. Ce thread se base sur un interprète Java étendu. Ce résultat est décrit par la barre bleue de la figure.

Les résultats de cette évaluation montrent que :

- Le mode de capture décidée avec l'approche IJSS n'induit aucun surcoût sur les performances des threads dont l'état est capturé. Ceci s'explique par le fait qu'avec l'approche IJSS, la capture décidée est applicable à des threads Java standards, instances de la classe *Thread*.
- Le mode de capture forcée avec l'approche IJSS induit une perte de performances de 67% par rapport à l'interprète Java standard. Cette perte de performances est due à la synchronisation de l'interprète IJSS pour autoriser les captures forcées. Cette synchronisation est basée sur la synchronisation Java qui souffre de la faiblesse de ses performances. De nouvelles mises en œuvre de machines virtuelles Java proposent des implantations optimisées de la

synchronisation [14] ; c'est le cas de la machine Jalapeño d'IBM [3] ou de l'environnement HotSpot de Sun Microsystems [91].

- Quant à l'approche IJES (mode décidé ou forcé), elle induit une perte de performances de 70% par rapport à l'interprète Java standard. Cette réduction des performances a plusieurs causes :
  - La synchronisation Java utilisée dans l'interprète IJES est peu performante.
  - En plus de l'interprétation des instructions de bytecode, l'interprète IJES effectue une gestion de types via la construction d'une pile de types.

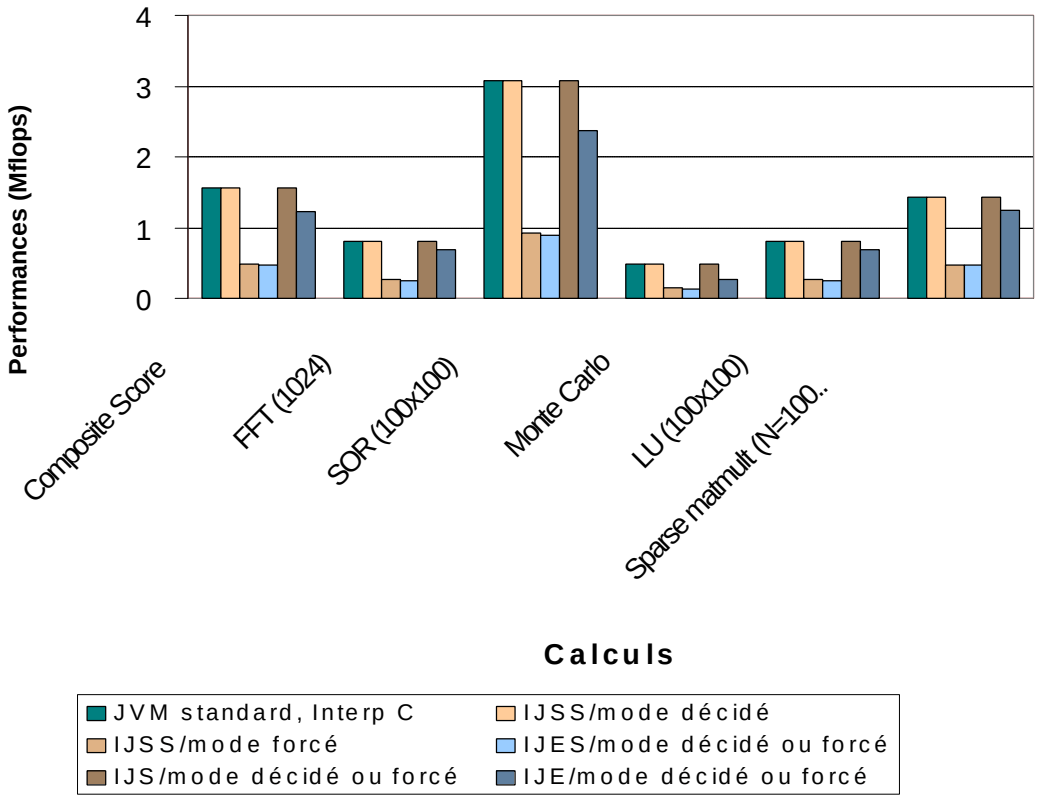

Figure 5-10. Surcoût sur les performances des applications mobiles ou persistantes SciMark 2.0 / Windows NT

Après l'obtention de ces premiers résultats, nous avons cherché à éliminer l'important surcoût induit par la synchronisation Java utilisée dans les interprètes. Nous avons alors apporté une modification aux premières mises en œuvre de nos services. Dans cette modification, la synchronisation Java est remplacée par l'utilisation des primitives *suspend* et *resume* de la classe *Thread* [1]:

- L'approche IJSS ainsi modifiée donne une mise en œuvre désignée dans la suite par IJS : interprète Java standard (voir la section 4.4.3 du Chapitre 3). Puisque la synchronisation de l'interprète Java a été éliminée dans IJS, il n'y a pas lieu

[1] Cette solution est présentée dans la section 4.1.3 du Chapitre 3.

d'avoir une classe particulière *CapturableThread* (dont les instances se différenciaient des instances de *Thread* par le fait que leur interprète était synchronisé). Avec IJS, tout thread de la classe *Thread* peut subir une opération de capture d'état, décidée ou forcée. Les résultats de l'évaluation du surcoût induit par IJS sur les performances des applications mobiles/persistantes sont présentés par la barre jaune-rayée de la Figure 5-10. Ces résultats sont obtenus par l'exécution du benchmark SciMark 2.0 par un thread *Thread* (interprète Java standard).

- L'approche IJES modifiée produit IJE : interprète Java étendu (voir la section 4.4.1 du Chapitre 3). Comme dans IJES, avec IJE, les threads Java dont l'état peut être capturé de façon décidée ou de façon forcée sont des threads de la classe *CapturableThread*. Les résultats de l'évaluation des surcoûts induits IJE sont donnés par la barre bleue-rayée de la Figure 5-10. Ces coûts résultent de l'exécution du benchmark SciMark 2.0 par un thread *CapturableThread* (interprète Java étendu).

Ces deux dernières évaluations montrent que l'élimination de la synchronisation permet de :

- Réduire le surcoût induit par l'approche IJE sur les performances des applications mobiles/persistantes. Ce surcoût est, en effet, passé de 70% à 20%, il reflète le traitement relatif à la gestion du typage.
- Eliminer tout surcoût induit par l'approche IJS sur les performances des applications mobiles/persistantes, que la mobilité/persistance soit décidée ou qu'elle soit forcée.

Dans la section 4.1.3 du Chapitre 3, nous avons expliqué pourquoi nous avons décidé, dans un premier temps, de ne pas utiliser les primitives *suspend* et *resume* de la classe *Thread*. En effet, l'utilisation de ces primitives pour la capture forcée risque de remettre en question la cohérence de l'état capturé : un thread externe peut ordonner la reprise d'un thread qui est suspendu pour une opération de capture forcée. En réponse à ce problème, nous proposons de mettre en œuvre et d'expérimenter une nouvelle classe de threads qui surcharge les primitives *suspend* et *resume* pour que la reprise d'un thread (*resume*) ne se fasse que si le thread en question ne subit pas une opération de capture.

4. Surcoût sur les performances des applications non mobiles/persistantes

Dans le cadre de l'évaluation des performances de nos services de mobilité/persistance, nous nous sommes intéressés à un troisième coût : l'éventuel surcoût induit par nos services sur les applications non mobiles et non persistantes. En

effet, du fait que nos services soient intégrés à la machine virtuelle Java, ceux-ci ont pu altérer le comportement « normal » des applications non concernées par la mobilité/persistance.

Pour effectuer cette évaluation, nous avons utilisé le benchmark SciMark 2.0 dans cinq cas, tels qu'illustrés par la Figure 5-11[1] :

- *JVM standard/Thread*. La JVM standard nous sert de référence pour les mesures suivantes. Cette JVM est lancée avec un thread *Thread* qui exécute le benchmark (barre verte).
- *IJSS/Thread*. Cette mesure concerne notre JVM étendue avec l'approche IJSS, sur laquelle est lancé un thread *Thread* qui exécute le benchmark. Cette mesure sert à évaluer le surcoût induit par nos services sur les applications non concernées par la mobilité, persistance, capture ou restauration (barre jaune).
- *IJES/Thread*. Cette mesure est la même que la précédente, sauf qu'elle concerne notre JVM étendue avec l'approche IJES (barre bleue).
- *IJS/Thread*. Cette évaluation concerne notre JVM étendue avec l'approche IJS. La JVM est lancée avec un thread *Thread* qui exécute le benchmark (barre jaune-rayée).
- *IJE/Thread*. Cette mesure concerne la JVM étendue avec l'approche IJE ; le benchmark est exécuté par un thread *Thread* (barre bleue-rayée).

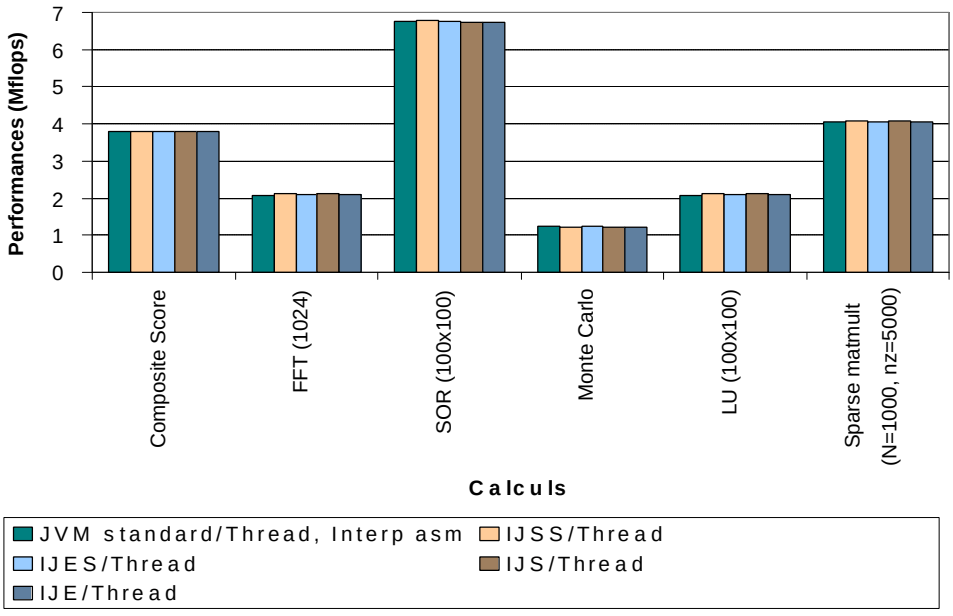

Figure 5-11. Surcoût sur les performances des applications non mobiles et non persistantes / SciMark 2.0 / Windows NT

[1] Toutes les mesures effectuées se basent sur l'interprète Java par défaut de la JVM (interprète écrit en assembleur).

Les résultats de ces mesures de performances montrent que l'ajout de nos services de mobilité/persistance/capture/restauration à la machine virtuelle Java n'affecte pas les performances des applications Java non concernées par ces services. Ceci est dû au fait que nos extensions de la JVM n'ont pas modifié les API Java existantes ni leur mise en œuvre (voir la section 4.7.1 du Chapitre 3).

5. Comparaison avec d'autres travaux

Dans cette section, nous nous intéressons à comparer nos travaux avec d'autres travaux sur la mobilité et la persistance fortes Java, en donnant tout d'abord une comparaison qualitative puis une comparaison quantitative.

5.1. Evaluation qualitative

Les travaux sur la mobilité et la persistance fortes Java ont vu le jour avec la plate-forme à agents mobiles Sumatra en 1997 [2] puis la plate-forme à agents mobiles Wasp en 1998 [53]. Lorsque nos travaux ont débuté, en 1998, Sumatra et Wasp étaient, à notre connaissance, les seuls projets adressant le problème de la mobilité/persistance fortes Java. Puis sont venus s'ajouter à eux d'autres travaux, tels que notre approche IJE [22][23] ou la plate-forme JavaGo [118] en 1999, notre approche IJS [24][25], Brakes [147], JavaGoX [116], Merpati [129], Nomads [142] et Moba [80] en 2000 et récemment un projet de migration d'agents proposé par Illmann et al. [70] en 2001.

Les travaux dans le domaine de la mobilité et de la persistance fortes Java peuvent être classés dans deux catégories :

- les travaux effectués au niveau du langage
- et les travaux effectués au niveau de la machine virtuelle.

Les travaux effectués au niveau du langage présentent l'avantage de ne pas modifier le système Java ; les techniques et solutions proposées sont ainsi utilisables sur toutes les plates-formes Java existantes. Mais le principal inconvénient de cette approche est le surcoût non négligeable qu'elle induit sur les performances des applications voulant bénéficier de la mobilité ou de la persistance.

Les travaux effectués au niveau de la machine virtuelle Java sont une solution au problème de performances. Mais il est important de noter que ces travaux, qui apportent une extension ou une modification à la machine virtuelle, proposent des solutions qui ne sont utilisables que sur des JVM étendues ou modifiées.

Dans la suite, nous décrivons les différentes techniques mises en œuvre au niveau du langage ou au niveau de la machine virtuelle et les comparons aux approches que nous proposons.

5.1.1. Travaux au niveau du langage

Mettre en œuvre des services de mobilité/persistance fortes Java au niveau du langage consiste à fournir une primitive de mobilité/persistance et un pré-processeur du programme de l'application. Le pré-processeur transforme le code de l'application en y injectant un code supplémentaire, soit dans le code Java source tel que le projet Wasp ou le projet JavaGo, soit dans le bytecode tel que proposé par Brakes ou par JavaGoX.

La principale difficulté avec cette approche est de pouvoir parcourir la pile Java du thread sous-jacent, une structure de données qui n'est pas accessible par les programmes Java. Deux techniques de parcours et de capture de la pile Java sont proposées : une technique basée sur l'utilisation d'exceptions Java (Wasp, JavaGo et JavaGoX) et une technique basée sur l'utilisation de *contrôle-retour* (Brakes).

Un pré-compilateur qui suit une technique basée sur les exceptions ajoute, au code de chaque méthode exécutée par le thread, la gestion et la propagation d'une exception particulière que l'on nommera *exception de capture*. La gestion de l'*exception de capture* dans une méthode se traduit par la sauvegarde de l'état d'exécution courant de la méthode dans un objet Java (appelé *contexte*). Une fois la sauvegarde de l'état d'une méthode effectuée, l'exception est propagée vers la méthode appelante. La propagation de l'*exception de capture* permet ainsi de parcourir la pile Java du thread et de capturer l'état de chaque méthode en cours d'exécution.

Quant à la technique du *contrôle-retour*, elle propose un pré-processeur qui injecte un code à la suite de chaque invocation de méthode dans le programme de l'application. Ce code teste si une opération de capture est en cours (*contrôle*) et si c'est le cas, l'état de la méthode est sauvegardé et la méthode est quittée (*retour*). L'exécution passe ensuite à la méthode appelante qui effectue une sauvegarde de son état avant de passer à sa méthode appelante et ainsi de suite jusqu'à parcourir la pile Java en entier.

Finalement, quelle que soit la technique adoptée pour la capture, elle aboutit toujours à la construction d'un objet Java (*contexte*) qui décrit l'état de chaque méthode en cours d'exécution par le thread. Cet objet *contexte* est, par la suite, utilisé pour la restauration de l'exécution. Restaurer une application consiste alors à relancer le programme modifié de l'application (programme modifié par le pré-processeur). Dans ce programme modifié, chaque méthode commence par tester si une opération de restauration est en cours. Si c'est le cas, l'état de la méthode est restauré à partir de l'objet *contexte* et un positionnement sur la dernière instruction exécutée dans la méthode est effectué.

La principale motivation de cette approche est qu'elle ne modifie pas la machine virtuelle Java et qu'elle est, de ce fait, portable sur toutes les JVM existantes. De plus, elle ne pose pas de restriction par rapport à la compilation Java JIT et peut ainsi bénéficier de cette optimisation à l'exécution.

Mais cette approche présente deux inconvénients majeurs :

- le surcoût sur les performances des applications
- et l'incomplétude de l'approche.

En effet, malgré le fait qu'elle supporte les compilateurs Java JIT, cette approche induit un surcoût non négligeable sur les performances normales des applications qui veulent bénéficier de la mobilité et de la persistance [116]. Ce surcoût est dû au code Java ou au bytecode injecté dans le programme de l'application.

Par ailleurs, une approche mise en œuvre au niveau du langage est souvent incomplète. Dans le cas du projet Wasp, par exemple, la solution proposée ne permet pas de capturer les valeurs des résultats intermédiaires dans une méthode. Considérons l'exemple de programme mobile donné par la Figure 5-12. A l'arrivée du programme sur son site destination, la méthode *m2* poursuit son exécution, se termine, retourne à la méthode *m1* pour effectuer l'opération d'addition mais la valeur du premier opérande, la valeur 5, n'a pas été sauvegardée. Le traitement des résultats intermédiaires, bien que possible, ajoute à la complexité de la solution.

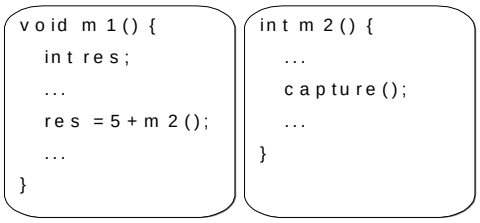

Figure 5-12. Non prise en compte des résultats intermédiaires

Une autre limitation est mise en évidence par les auteurs de la plate-forme JavaGoX [116] et par les auteurs de la plate-forme Brakes [147]. Elle concerne la clause *finally* et l'impossibilité d'effectuer une opération de migration au sein de ce type de clause. Ceci est dû au fait que la mise en œuvre de cette clause au niveau de bytecode exige la manipulation d'une valeur de type *returnAddress*[1]. Mais cette valeur représente une adresse dans la mémoire physique de la machine sous-jacente : la valeur n'est donc pas directement portable sur une autre machine. Ceci nous amène à une constatation importante : la mise en œuvre de la capture d'état au niveau du langage ne traite pas complètement le problème de l'hétérogénéité. Une capture d'état basée sur cette approche ne prend donc pas en compte la totalité de l'état d'exécution Java de l'application.

D'autre part, des quatre projets présentés dans cette section, seul Brakes propose une mobilité forcée à ses agents. Pour proposer une telle mobilité, Brakes définit un

[1] Voir la section 2.2.1 du Chapitre 2.

nouveau middleware dans lequel un unique thread Java s'exécute. Ce thread peut héberger plusieurs *tâches* (abstraction d'agent). Brakes fournit son propre système d'exécution et d'ordonnancement des tâches ; ce système garantit qu'il n'y a qu'une seule tâche qui s'exécute à la fois et il se base sur les outils de capture/restauration pour effectuer le changement de contexte entre les tâches. Ainsi, lorsqu'une tâche exécute une demande de capture forcée sur une autre tâche, la tâche qui initie la capture est forcément l'unique tâche active et la tâche dont l'état doit être capturé est forcément suspendue et son état a précédemment été capturé et sauvegardé (lors d'un changement de contexte entre les tâches). Finalement, Brakes fournit ses services de mobilité/capture forcée dans un nouveau middleware au-dessus de la JVM, alors que nous proposons des services de mobilité/capture forcée au même niveau que les services système de la JVM. Nos services sont applicables à des threads Java s'exécutant sur n'importe quel middleware alors que pour utiliser les services de Brakes, une application doit s'appuyer sur le middleware proposé.

5.1.2. Travaux au niveau de la machine virtuelle

Les travaux sur la mobilité et la persistance fortes Java, effectués au niveau de la machine virtuelle, sont de trois types :

- des travaux au niveau de l'interprète Java,
- des travaux basés sur une analyse de flot
- ou des travaux basés sur l'interface de débogage fournie par la JVM.

- ***Travaux au niveau de l'interprète Java***

L'approche qui consiste à mettre en œuvre les mécanismes de mobilité/persistance en se basant sur une extension de l'interprète Java est une des approches que nous avons expérimentées ; elle est présentée dans la section 4.5.1 du Chapitre 3. Cette approche consiste à étendre l'interprète Java de traitements relatifs au typage des valeurs contenues dans la pile Java d'un thread (approche IJE).

La principale motivation de cette approche est qu'elle permet de prendre en compte la totalité de l'état d'exécution Java d'une application. Elle traite, par exemple, la clause *try-finally* qui n'est pas prise en compte dans l'approche au niveau langage. Mais elle présente deux inconvénients majeurs :

- Elle induit un surcoût sur les performances des applications voulant bénéficier de la mobilité/persistance. Ce surcoût est dû aux traitements supplémentaires relatifs à la gestion du typage des valeurs contenues dans la pile Java d'un thread.
- La capture d'état proposée ne prend pas en compte l'état d'exécution compilée de l'application ; puisque s'il y a compilation à la volée, l'exécution du code se base sur une exécution native et non sur l'interprète Java étendu. Les

informations sur le typage ne sont alors pas construites et la capture de cet état d'exécution ne peut donc pas se faire. Autrement dit, les applications Java qui souhaitent utiliser une telle capture d'état ne peuvent pas bénéficier de la compilation Java JIT ; leurs performances à l'exécution sont alors dégradées.

Cette approche a initialement été utilisée dans la plate-forme à agents mobiles Sumatra, qui fournit la primitive *go* de mobilité forte à ses agents [2]. La mobilité dans Sumatra est obligatoirement décidée par l'agent mobile lui-même. Par ailleurs, les mécanismes sous-jacents de capture/restauration d'état ne sont pas disponibles dans l'API proposée par Sumatra, Sumatra est ainsi dédiée à la mobilité. Le projet Sumatra a pris fin mais ses services de mobilité forte ont été intégrés à la plate-forme à agents mobiles D'Agents [56].

- ***Travaux basés sur une analyse de flot***

La seconde approche de mise en œuvre au niveau de la machine virtuelle Java est basée sur une analyse du flot d'exécution d'un thread, au moment de la capture de son état. Nous avons expérimenté cette approche qui est présentée dans la section 4.5.2 du Chapitre 3 (approche IJS).

Cette approche présente plusieurs avantages :

- Elle annule le surcoût sur les performances des applications voulant bénéficier de la mobilité ou de la persistance puisqu'elle n'ajoute aucun traitement supplémentaire lors de la capture de l'état d'exécution de l'application.
- La totalité de l'état d'exécution Java d'une application est prise en compte lors de la capture d'état. Donc, contrairement à l'approche au niveau langage, elle traite la clause *try-finally*.
- Contrairement à l'approche basée sur un interprète, cette approche peut être utilisée en présence d'un compilateur Java JIT, à condition de procéder, au moment de la capture, à une dés-optimisation du code compilé à la volée puis à une analyse de flot. C'est la solution que nous proposons dans la section 4.6 du Chapitre 3 et qui a deux conséquences directes :
  - La totalité de l'état d'exécution Java est pris en compte lors de la capture (méthodes Java interprétées et méthodes Java compilées à la volée).
  - Aucune baisse de performances n'est induite ni sur les applications voulant bénéficier de la mobilité/persistance ni sur les applications non concernées par ces services.

L'approche qui consiste à analyser le flot d'exécution des threads est également suivie par le projet Merpati qui fournit des services très similaires aux nôtres [129]. Il existe, en effet, plusieurs points communs entre Merpati et nos travaux : fournir des services génériques, utilisables pour la mobilité et la persistance, fournir des points

d'entrées qui permettent de spécialiser ces services (nos interfaces *SendInterface* et *ReceiveInterface* et les interfaces de Merpati *MigrOutProtocol* et *MigrInProtocol*), la mise en œuvre de l'approche basée sur l'extension de l'interprète Java et de l'approche basée sur une analyse de flot. Mais il y a deux points majeurs qui différencient Merpati de nos travaux : la granularité et les performances.

Dans Merpati, une opération de migration se traduit obligatoirement par la migration de tous les threads qui s'exécutent sur la JVM et une opération de sauvegarde se traduit par la sauvegarde de l'ensemble des threads de la JVM. Les auteurs de Merpati justifient ce point en faisant l'hypothèse qu'une instance de JVM correspond à une seule application Java et que la mobilité ou persistance d'une application se traduit par la mobilité ou persistance de tous les threads de la JVM. Contrairement à Merpati, nos services de mobilité et de persistance sont à grain plus fin puisqu'ils sont applicables sur des instances de threads. Nous ne faisons, en effet, pas d'hypothèse sur la définition d'une application Java. Nos services peuvent, par ailleurs, être utilisés sur un groupe de threads ou sur l'ensemble des threads.

Un second point qui différencie Merpati de notre mise en œuvre concerne les performances à l'exécution ou, plus précisément, le fait que Merpati ne fournisse pas le troisième avantage cité précédemment. En effet, contrairement à nos travaux, Merpati n'adresse pas le problème de l'état d'exécution compilée et interdit, de ce fait, l'utilisation de compilateur Java JIT. Comme nous l'avons présenté précédemment (section 4.6 du Chapitre 3), cette restriction affecte sensiblement les performances de la machine virtuelle et peut réduire la diffusion et l'utilisation des services proposés.

- ***Travaux basés sur l'interface de débogage***

Cette troisième approche consiste à mettre en œuvre les mécanismes de mobilité/persistance des threads à travers l'interface de débogage fournie par la JVM, interface connue sous le nom de JVMDI (Java Virtual Machine Debug Interface) [141]. La JVMDI est une interface de programmation qui fournit des outils permettant d'inspecter l'état d'exécution d'applications Java. Cette interface est normalement utilisée pour la mise en œuvre d'outils de débogage des applications Java. Mais du fait qu'elle donne accès aux structures internes d'exécution de la JVM, elle peut également être utilisée pour la mise en œuvre de la capture/restauration de l'état des threads Java. Cette solution est proposée par Illmann et al. pour la mise en œuvre de leur service de migration de threads [70].

La JVMDI fournit des primitives qui permettent, par exemple, de parcourir la pile Java d'un thread, d'accéder aux frames de la pile Java ou d'accéder aux variables locales et au registre PC. Illmann et al. ont construit leurs mécanismes de capture d'état

au-dessus de ces primitives. Mais l'interface JVMDI ne suffit pas à elle seule pour construire les mécanismes de capture/restauration d'état puisque :

- Elle ne permet pas d'accéder à la pile d'opérandes d'un frame Java. Les résultats intermédiaires contenus dans les piles d'opérandes ne sont ainsi pas pris en compte par le service de migration de threads.
- Elle ne fournit pas de primitive pour l'initialisation du registre PC. Pour pouvoir initialiser ce registre, Illmann et al. ont dû :
  - étendre la machine virtuelle Java pour lui ajouter la primitive manquante
  - ou augmenter le programme de l'application pour y ajouter un traitement relatif au registre PC (une partie du mécanisme est ainsi mise en œuvre au niveau du langage).

La motivation initiale de cette approche est d'extraire, de la JVM, les informations relatives à l'état d'exécution des threads et ceci, sans modifier la machine virtuelle. Mais l'interface JVMDI n'a pas suffi pour atteindre ce but. Mais l'inconvénient majeur de cette approche est que pour bénéficier des services proposés, une JVM doit être lancée avec une option de débogage, ce qui :

- désactive la compilation Java JIT
- et impose une surcharge de traitement à l'exécution, surcharge due au débogage.

Ceci induit alors un surcoût très important sur les performances de toutes les applications qui s'exécutent sur cette JVM, que ces applications soient mobiles ou non concernées par la mobilité.

- ***Autres travaux***

En plus des projets cités précédemment, il existe deux autres projets qui adressent le problème de mobilité forte Java : les plates-formes à agents mobiles Moba [80] et Nomads [142]. Moba est basé sur une extension de la machine virtuelle Java et Nomads repose sur une nouvelle mise en œuvre de JVM. Aucun des deux projets ne propose une extension de l'interprète Java, ni une analyse de flot.

Moba propose un système de reconnaissance des références d'objets Java contenues dans la pile Java d'un thread. Ce système est basé sur une inférence de type à partir d'une valeur, par comparaison de la valeur de la référence avec l'intervalle des adresses mémoire couvertes par le tas d'objets de la JVM. Mais Moba ne propose actuellement pas de solution globale au problème de typage : les valeurs de types primitifs sont déplacées telles quelles, ce qui suppose l'utilisation de machines hétérogènes.

Concernant le problème de typage dans Nomads, la nouvelle JVM sur laquelle repose Nomads pose certaines restrictions qui violent la portabilité des données Java. En effet, alors que la machine virtuelle Java définit un format portable et abstrait pour ses données, la JVM associée à Nomads restreint le format des données Java à une

représentation particulière. Par exemple, avec la machine virtuelle associée à Nomads, une valeur entière doit obligatoirement être représentée sur 4 octets et n'est, de ce fait, pas portable sur des plates-formes physiques où les entiers sont représentés sur 8 octets. La machine virtuelle sous-jacente à Nomads pose également des restrictions sur le format des flottants, en exigeant que ceux-ci aient la même représentation sur toutes les plates-formes.

Finalement, l'incomplétude de la gestion du typage proposée par Moba et les restrictions imposées par Nomads font que les services fournis ne sont pas entièrement portables sur des plates-formes hétérogènes.

5.1.3. Conclusion

Un récapitulatif des différents projets sur la mobilité/persistance fortes Java est donné par le Tableau 5-1. Une évaluation de chacun des projets est faite en fonction du mode d'initiation proposé pour la mobilité/persistance, de la granularité considérée, de la complétude de l'état d'exécution Java pris en compte par les mécanismes de capture, de l'éventuel surcoût induit sur les performances des applications mobiles/persistantes, de la compatibilité des mécanismes fournis avec la compilation Java JIT et de l'utilisabilité des mécanismes sur des environnements Java existants.

Sur les six critères d'évaluation énoncés précédemment, l'approche IJS se présente comme l'approche la plus complète puisque :

- Elle propose deux modes d'initiation de la mobilité et de la persistance : des mobilité et persistance décidées ou des mobilité et persistance forcées et ceci sans imposer de modèle/middleware particulier tel que proposé par Brakes. Fournir les deux modes d'initiation permet d'adresser un plus large spectre d'applications. La mobilité décidée peut ainsi être utilisée pour la mise en œuvre de plates-formes à agents mobiles et la persistance forcée peut servir à l'automatisation des sauvegardes des applications, pour des besoins de tolérance aux pannes.
- L'unité de mobilité ou de persistance est le thread, les services proposés sont ainsi applicables à grain fin.
- La totalité de l'état d'exécution Java est pris en compte par le service de capture (pile d'opérandes, registre PC, clause *try-finally*, etc.).
- L'approche IJS n'induit aucun surcoût sur les performances des applications mobiles/persistantes car elle se base sur les threads Java standards, instances de la classe *Thread*.

Projet	**Technique**	**Utilisabilité/ env Java existant**	**Mode d'initiation**	**Granularité**	**Complétude de l'état Java considéré**	**Sans surcoût à l'exécution**	**Compatibilité JIT**
Wasp	Pré-processeur de langage source Java	oui ☺	décidé	un thread ☺	non	non	oui ☺
JavaGo	Pré-processeur de langage source Java	oui ☺	décidé	un thread ☺	non	non	oui ☺
JavaGoX	Pré-processeur de bytecode	oui ☺	décidé	un thread ☺	non	non	oui ☺
Brakes	Pré-processeur de bytecode	oui ☺	décidé ou forcé/un middleware 😐	un thread ☺	non	non	oui ☺
Sumatra/ D'Agents	JVM/ interprète	non	décidé	un thread ☺	oui ☺	non	non
IJE	JVM/ interprète	non	décidé ou forcé ☺	un thread ☺	oui ☺	non	non
IJS	JVM/ analyse de flot	non	décidé ou forcé ☺	un thread ☺	oui ☺	oui ☺	oui ☺
Merpati	JVM/ interprète	non	décidé ou forcé ☺	tous les threads	oui ☺	non	non
	JVM/ analyse de flot	non	décidé ou forcé ☺	tous les threads	oui ☺	oui ☺	non
Illmann et al.	JVMDI/ extension de JVM	non	décidé ou forcé ☺	un thread ☺	non	non	non
	JVMDI/ augmentation de code	oui ☺	décidé ou forcé ☺	un thread ☺	non	non	non

Tableau 5-1. Comparaison des projets traitant la mobilité/persistance fortes Java

- Cette approche est compatible avec la compilation Java JIT. En effet, en plus de la prise en compte de l'état d'exécution Java des threads, cette approche permet de prendre en compte l'état d'exécution compilée de ces threads. Ceci a deux conséquences directes :
  - Le problème d'hétérogénéité est traité dans sa globalité puisque tout ce qui a attrait à l'exécution Java (interprétation ou compilation à la volée) est pris en compte par la solution proposée.
  - Les performances des applications Java sont préservées, ces applications peuvent continuer à bénéficier des optimisations de la compilation à la volée. Ce n'est qu'à l'invocation d'un de nos services (capture, mobilité, persistance d'application) qu'un temps de latence est induit sur l'application concernée.

L'approche IJS a une restriction : ses services ne sont disponibles que sur une JVM étendue. Ceci peut quelque peu restreindre l'utilisabilité des services proposés.

Dans l'état actuel de notre mise en œuvre, la prise en compte de l'état d'exécution Java est finalisé et la prise en compte de l'état d'exécution compilée est en cours d'intégration. En ce qui concerne l'état d'exécution native, nous avons fait le choix de ne pas le prendre en compte dans la mise en œuvre de nos services[1]. Nous proposons, cependant, un comportement par défaut en cas de présence de méthodes natives en cours d'exécution.

5.2. Evaluation quantitative

Après avoir décrit et comparé les fonctionnalités des différents systèmes de mobilité forte Java, nous nous intéressons dans cette section à comparer les performances de ces systèmes. Nous avons ainsi, en plus de nos systèmes IJE et IJS, installé et configuré les systèmes JavaGoX, Brakes et JavaGo. Cette évaluation comparative ne prend malheureusement pas en compte le système proposé par Illmann et al., dont les mécanismes de migration ne sont actuellement pas encore stables. Nous avons, par la suite, construit les programmes d'évaluation correspondant à chaque système, en respectant le modèle de programmation et les contraintes imposées par le système. Nos mesures de performances comparent :

- la variation de la latence de la migration de thread en fonction du nombre d'objets Java utilisés par le thread
- et le surcoût induit par le système sur les performances des applications Java.

D'autre part, chaque système propose son propre mécanisme de transfert d'état lors d'une opération de migration. Pour des raisons d'homogénéité, nous avons basé le transfert d'état, avec chaque système, sur la sérialisation/dé-sérialisation d'objets Java.

[1] Voir la section 4.1.2 du Chapitre 3.

Pour ce faire, nous avons apporté des modifications aux mécanismes de transfert d'état des différents sous-systèmes.

De plus, pour permettre la comparaison des performances des différents systèmes, lorsque certains systèmes ne supportent pas la compilation à la volée, nous avons effectué les mesures de performances en désactivant la compilation Java JIT.

Finalement, l'évaluation de performances des différents systèmes a été effectuée avec le JDK 1.2.2, sur un processeur P III/600 MHz, avec le système Windows NT.

5.2.1. Comparaison des temps de latence de la mobilité

La Figure 5-13 présente, pour chaque système, la variation de la latence de la migration d'un thread en fonction du nombre d'objets Java utilisés par le thread au moment de la migration. Ces évaluations ont été effectuées comme suit :

- Avec les systèmes JavaGoX et Brakes, avant le lancement du programme exécuté par le thread migrant, celui-ci est augmenté avec le pré-processeur de bytecode fourni par le système. Le pré-processeur insère du code dans le programme pour construire l'état du thread au cours de son exécution. Ainsi, au moment de la migration du thread, l'état d'exécution est déjà construit, il peut être directement transféré puis restauré à son arrivée, grâce à une ré-exécution partielle du programme augmenté. De ce fait, le temps de latence d'une migration est constitué ici du temps de transfert de l'état et du temps de restauration du thread. Cette évaluation a été effectuée sur la version standard du JDK 1.2.2.
- Avec le système JavaGo, le programme du thread est augmenté avec un pré-processeur de code source Java. Ce pré-processeur suit le même principe que le pré-processeur de JavaGoX ou Brakes, sauf qu'il agit au niveau du programme source Java. L'évaluation de JavaGo a également été faite sur la version standard du JDK 1.2.2.
- Avec les systèmes IJE et IJS, le programme du thread est exécuté sur nos versions étendues du JDK 1.2.2. Le temps de latence d'une migration est ici constitué du temps de capture d'état, de son temps de transfert et du temps de restauration du thread.

La Figure 5-13 montre que JavaGoX et Brakes évoluent de la même manière. Ceci est dû au fait que les deux systèmes soient basés sur une même technique : un pré-processeur de bytecode. La principale différence entre ces deux systèmes réside dans leur technique de capture d'état puisque JavaGoX se base pour cela sur une gestion des exceptions Java alors que Brakes se base sur une technique de contrôle-retour[1]. Mais

[1] Voir la section 5.1.1 de ce chapitre.

cette différence n'est pas perceptible dans la latence de migration puisque, avec ces deux systèmes, le temps de capture est reporté à l'exécution du programme alors que la migration n'est constituée que du transfert et de la restauration d'état.

Quant à JavaGo, l'évaluation de ses performances montre un comportement curieux. En effet, la latence de migration avec JavaGo, qui est basé sur un pré-processeur de code source, est parfois plus faible que la latence avec les systèmes basés sur un pré-processeur de bytecode (JavaGoX ou Brakes). Ceci est le cas, par exemple, lorsque le nombre d'objets Java utilisés par le thread est égal à 1, 5, 10 ou 25. Ceci s'explique par le fait qu'avec ce nombre d'objets, l'état capturé avec JavaGo est plus petit que l'état capturé avec JavaGoX ou Brakes. Ceci est montré par la Figure 5-14 qui donne, pour chaque système, la variation de la taille de l'état capturé en fonction du nombre d'objets utilisés par le thread. Ainsi, pour un nombre d'objets entre 1 et 25, la latence du transfert d'état, et donc de la migration de thread, est plus faible avec JavaGo qu'avec JavaGoX ou Brakes. Ceci n'est plus vrai lorsque le nombre d'objets utilisés est 50 ou 100.

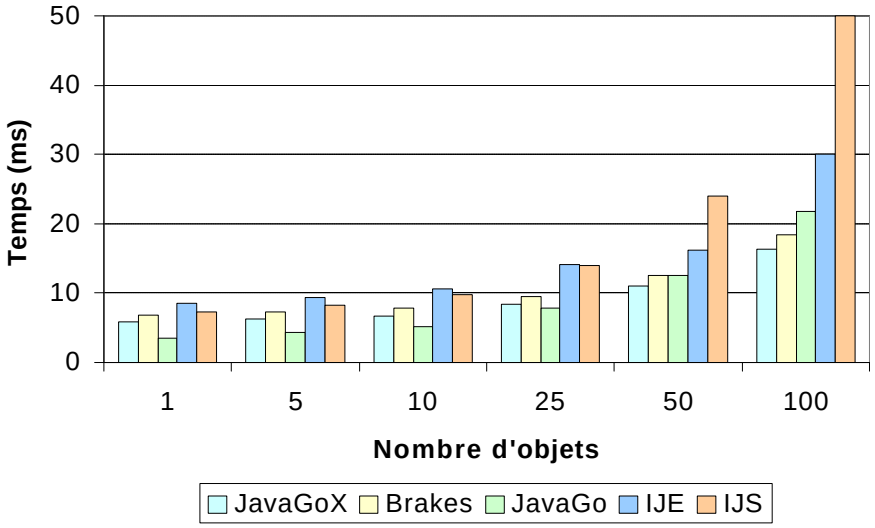

Figure 5-13. Latence de la migration avec différents systèmes en fonction du nombre d'objets

Quant aux systèmes IJE et IJS, leurs temps de latence évoluent de la même manière lorsque le nombre d'objets utilisés par le thread est faible. L'écart se creuse ensuite entre les deux systèmes lorsque le nombre d'objets augmente. Ceci est principalement dû à la différence entres les techniques mises en œuvre pour capturer l'état d'un thread dans les deux systèmes :

- Avec IJE, la capture d'état a lieu au cours de l'exécution du thread (par un interprète java étendu).
- Avec IJS, la capture d'état a lieu au moment de la migration (par une analyse de flot).

Ainsi, plus l'état à capturer est grand, plus la différence entre les latences de migration des deux systèmes est grande. Par ailleurs, les systèmes IJE et IJS présentent des temps de latence de migration les plus importants. Ceci est dû au fait que, contrairement aux autres systèmes, IJE et IJS prennent en compte la totalité de l'état d'exécution Java du thread (voir la section 5.1.3 de ce chapitre). Ceci est mis en évidence par la Figure 5-14 où la taille de l'état capturé avec IJE ou IJS est plus importante.

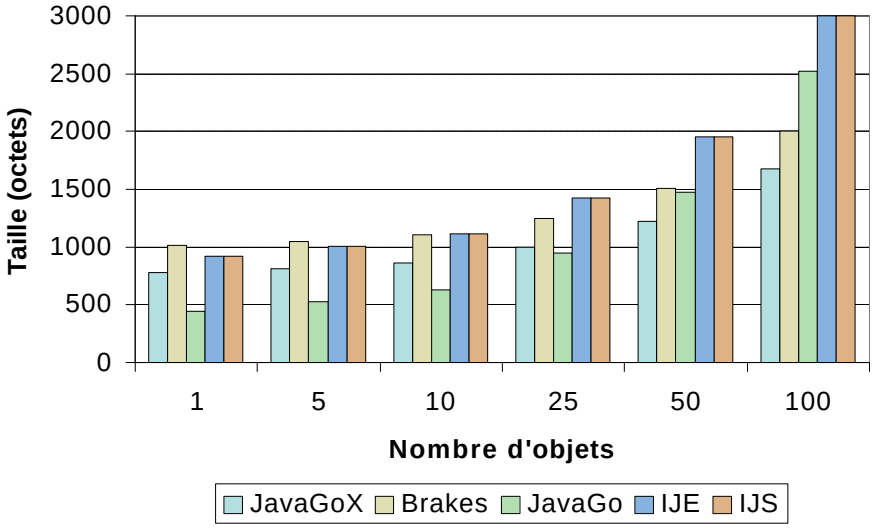

Figure 5-14. Taille de l'état transporté lors d'une opération de migration (en fonction du nombre d'objets)

5.2.2. Comparaison des surcoûts sur les performances des applications mobiles

La Figure 5-15 ou encore le Tableau 5-2 donnent, pour chaque système, le surcoût induit par le service proposé sur le temps de calcul de la fonction Fibonacci(30) et Fibonacci(20). Les temps de calcul mesurés n'incluent pas d'opération de capture d'état. Ces évaluations ont été effectuées comme suit :

- Le programme est augmenté avec le pré-processeur JavaGo, JavaGoX ou Brakes, avant d'être exécuté. L'exécution est basée ici sur la compilation Java JIT. L'utilisation de ces pré-processeurs induit un surcoût non négligeable sur les performances de l'application à cause du code inséré dans le programme du thread. Ce surcoût est plus ou moins important, selon que la mise en œuvre est faite au niveau du langage source Java (JavaGo) ou au niveau du bytecode (JavaGoX, Brakes).
- L'approche IJE induit un surcoût très important sur les performances du programme. Ce surcoût correspond au temps nécessaire à la gestion du typage en parallèle à l'interprétation du bytecode. Ce système présente l'inconvénient de ne pouvoir bénéficier de la compilation Java à la volée.

- Quant à l'approche IJS, elle n'induit aucun surcoût car aucun traitement supplémentaire n'est effectué à l'exécution et tout coût est reporté à la capture. De plus, cette approche peut bénéficier de la compilation Java JIT, en se basant sur les techniques de dés-optimisation à la volée décrites dans la section 4.6du Chapitre 3.

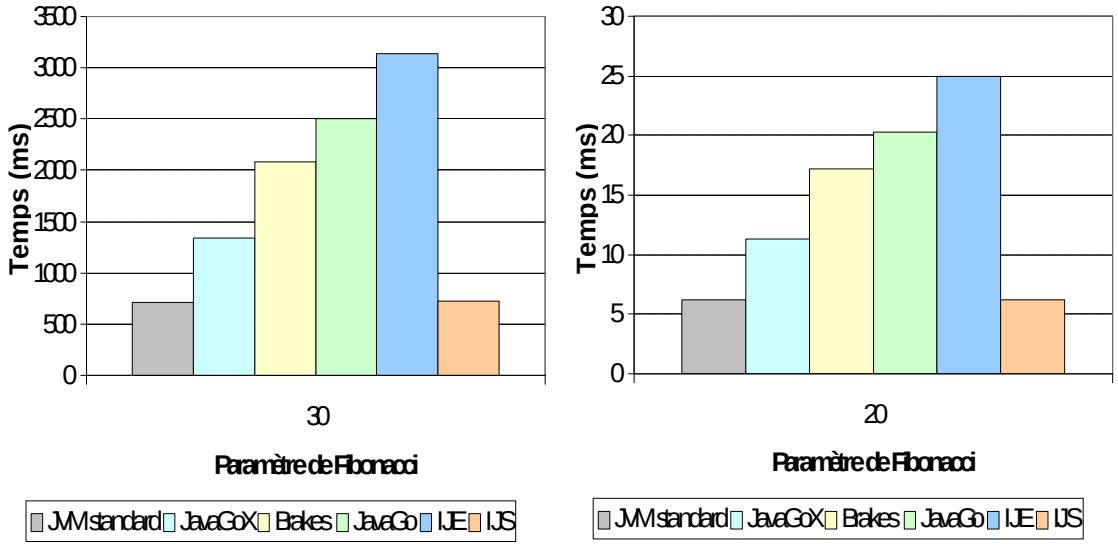

Figure 5-15. Surcoût induit par les différents systèmes sur le calcul de Fibonacci(30) et Fibonacci(20)

		Fibonacci(30)	Fibonacci(20)
Système	JavaGoX	+87%	+83%
	Brakes	+190%	+177%
	JavaGo	+250%	+227%
	IJE	+338%	+303%
	IJS	+0%	+0%

Tableau 5-2. Surcoût induit par les différents systèmes sur le calcul de Fibonacci

5.3. Conclusion

L'évaluation des performances des différents systèmes a permis de montrer que :

- Les systèmes JavaGoX, Brakes, JavaGo et IJE présentent les temps de latence de migration les plus faibles mais induisent un surcoût important sur les performances des applications (entre 83% et 338%). Ce surcoût persistera même en présence de la compilation Java JIT.
- Le système IJS propose, quant à lui, un temps de latence de migration plus important mais présente l'avantage de n'induire aucun surcoût sur les

performances normales des applications. Ce comportement adhère à notre choix de conception qui privilégie le besoin d'annuler tout surcoût sur le besoin de diminuer la latence (voir la section 6 du Chapitre 1). Par ailleurs, le système IJS présente l'avantage d'être compatible avec la compilation JIT et de permettre ainsi aux applications Java de bénéficier de l'optimisation de leur exécution. Ainsi, une application qui souhaite bénéficier de la mobilité ou de la persistance ne payera qu'au moment où elle fait appel à ces services.

6. Conclusion

Dans ce chapitre, nous avons tout d'abord présenté les temps de latences obtenu avec nos premières mises en œuvre (IJSS et IJES). L'évaluation de ces latences a montré que :

- La capture et la restauration d'état présentent des coûts raisonnables qui sont de l'ordre de la milli-seconde.
- La migration, la sauvegarde et la reprise d'un thread présentent des coûts plus élevés qui sont de l'ordre de la dizaine et de la centaine de milli-secondes.

L'évaluation du surcoût induit par nos services sur les applications voulant utiliser ces services montre que :

- L'approche qui consiste à étendre l'interprète Java induit un surcoût non négligeable sur les performances de l'application.
- L'approche basée sur une analyse de flot n'induit aucune perte de performances de l'application dans le cas où la prise en compte de la capture forcée n'est pas basée sur la synchronisation Java. Nous avons, de ce fait, proposé de nouvelles mises en œuvre de nos services : les approches IJS et IJE.

Par ailleurs, à travers nos mesures de performances, nous avons montré que l'intégration de nos services à la machine virtuelle Java n'affecte pas les performances des autres services de la machine virtuelle. Ce comportement est une conséquence directe de la modularité de la mise en œuvre de nos services dans la JVM.

Finalement, une comparaison des fonctionnalités et des techniques de mise en œuvre de plusieurs projets traitant de la mobilité et de la persistance fortes Java est donnée en fin de ce chapitre. Elle montre que l'approche qui consiste à étendre la JVM présente l'avantage d'être :

- complète puisqu'elle prend en compte les différents cas non traités au niveau du langage, tels que la prise en compte de la clause *try-finally* ou la proposition d'une solution générale pour la capture d'état forcée,
- plus performante puisqu'elle n'induit aucun surcoût sur les performances des applications, lorsque la technique mise en œuvre est IJS,

- compatible avec la compilation Java JIT puisqu'il est possible, grâce aux techniques de dés-optimisation dynamique, de prendre en compte l'état d'exécution résultant de la compilation JIT.

CONCLUSIONS

1. Rappel des objectifs

Les travaux menés durant cette thèse concernent l'étude, la proposition et la réalisation de services système pour la mobilité et la persistance des applications – code, données et exécution – dans des environnements hétérogènes.

Nous avons choisi l'environnement Java comme plate-forme de base pour la conception de nos services de mobilité et de persistance pour deux raisons :

- Java est tout d'abord une machine virtuelle qui cache l'hétérogénéité des processeurs et systèmes d'exploitation sous-jacents et fournit l'abstraction d'un environnement homogène. Cette machine garantit ainsi la portabilité de son code et de ses données sur des plates-formes de natures variées. Ceci nous procure une bonne base pour la mise en place de la portabilité des applications, avec leur code, données et exécution.
- De plus, avec la généralisation des environnements hétérogènes et la démocratisation de l'utilisation d'Internet, Java a connu une grande diffusion et est actuellement implanté sur la plupart des systèmes. Ceci nous procure un large champ d'applications et d'expérimentations des services proposés.

La machine virtuelle Java fournit ainsi l'abstraction d'un processeur et d'un système d'exploitation universels. Elle définit, en effet, son propre jeu d'instructions – *bytecode* –, son système de gestion de la mémoire – *tas d'objets* – et son système de gestion des flots de contrôle – *threads* –. Son code et ses données sont ainsi universellement portables, mais l'exécution de ses applications ne l'est pas.

Notre objectif était de proposer les services nécessaires pour que la machine virtuelle Java soit une machine effectivement universelle, avec un code, des données mais également une *exécution* des applications qui soient universellement portables.

2. Bilan et évaluation de la réalisation

Dans cette perspective, nous avons proposé et mis en œuvre une extension de la machine virtuelle Java. Cette extension fournit les mécanismes nécessaires pour rendre l'exécution des applications Java (threads) portable. Ces mécanismes de bas niveau ont, par la suite, été utilisés pour la construction de nouveaux services Java : des services pour la mobilité et la persistance des applications. Une application peut ainsi, au cours de son exécution, migrer vers un site distant pour y poursuivre son exécution, créer une application clone qui s'exécute sur le même site qu'elle ou sur un autre site, sauvegarder son exécution courante sur un support persistant et poursuivre, par la suite, son exécution à partir de cette sauvegarde persistante. Ces services ont divers domaines d'applications, tels que la tolérance aux pannes, la reconfiguration dynamique

d'applications réparties ou la répartition dynamique de charge dans un système distribué. Les services proposés sont actuellement opérationnels et disponibles via une nouvelle API Java ; ils sont accessibles via : *http://sirac.inrialpes.fr/Logiciel*

Deux axes principaux ont dirigé la conception de nos services : respecter l'abstraction d'homogénéité fournie par Java et respecter les performances des applications Java :

- Java garantit la portabilité du code et des données. Nos services garantissent, de plus, la portabilité de l'exécution des applications.
- Les services proposés ne remettent pas en cause pas les efforts faits en matière d'optimisation de l'exécution du code Java et restent utilisables même en présence de compilateurs Java JIT.

Pour garantir la portabilité de l'exécution des applications, nous avons proposé, mis en œuvre et expérimenté deux techniques :

- La première technique est basée sur une extension de l'interprète Java. L'extension de l'interprète a pour but de construire, au fur et à mesure de l'interprétation du bytecode de l'application, les informations nécessaires à la construction d'une représentation portable de l'exécution de l'application. L'expérimentation et l'évaluation de cette technique ont montré qu'elle présentait l'avantage d'être simple à mettre en œuvre mais qu'elle induisait un surcoût non négligeable sur les performances de l'application.
- La seconde technique est basée sur une analyse de flot d'exécution de l'application, analyse similaire au traitement effectué lors de la vérification de bytecode par un système de chargement de classes Java. Contrairement à la première solution, avec cette seconde approche, l'exécution de l'application (interprétation de bytecode) n'est augmentée d'aucun traitement supplémentaire et tout traitement relatif à la construction d'une représentation portable est fait de façon ponctuelle, par application de l'algorithme. Les performances de l'application restent alors inchangées et tout coût supplémentaire est reporté dans la construction de la représentation portable de l'exécution.

Un autre principe qui a guidé la conception de nos services est le respect des performances des applications Java et, plus particulièrement, des efforts faits en matière de compilation Java JIT. Ainsi, les applications qui font appel à nos services peuvent continuer à utiliser la compilation JIT. Ceci a introduit une difficulté supplémentaire : garantir la portabilité d'une exécution qui a subi une optimisation faisant d'elle une exécution non portable. La solution que nous avons adoptée consiste à défaire le travail effectué par le compilateur JIT, autrement dit dés-optimiser dynamiquement l'optimisation effectuée par le compilateur.

La validation de nos travaux et des services qui en résultent a été faite par des expérimentations d'applications. Une première application a été élaborée à des fins de démonstration de nos services de mobilité. Elle met en œuvre la mobilité du calcul récursif d'une courbe fractale et de la mobilité de la visualisation de cette courbe. La seconde application est implantée au-dessus de nos services de persistance des applications. Elle a été mise en œuvre par un groupe de recherche de l'université Simón Bolívar, au Venezuela. Cette application met en œuvre un système de sauvegarde/reprise de calculs parallèles, sur une plate-forme de metacomputing appelée SUMA.

Ces trois dernières années ont vu naître plusieurs projets de recherche étudiant la mobilité et la persistance des applications Java : nous en avons dénombré une dizaine. De tous ces projets, notre proposition est l'unique projet qui allie la complétude de la conception au souci de performances. Les travaux mis en œuvre au niveau du langage de programmation Java présentent l'inconvénient d'avoir plusieurs limitations techniques et d'induire un surcoût sur les performances normales des applications et ceci, malgré le fait qu'ils bénéficient de la compilation Java JIT. Quant aux autres travaux mis en œuvre au niveau de la machine virtuelle, ils souffrent d'une altération considérable des performances des applications puisque aucun ne supporte la compilation JIT. Nous proposons des services qui sont mis en œuvre au niveau de la machine virtuelle Java, qui n'altèrent pas les performances des applications et qui proposent une solution pour la prise en compte des optimisations résultant de la compilation Java JIT. La principale limitation de nos services résulte du fait qu'ils sont proposés comme une extension de la machine virtuelle et qu'ils ne soient donc pas directement utilisables sur toutes les plates-formes Java existantes. Mais l'objectif de nos travaux était justement d'étendre les abstractions fournies par la machine virtuelle pour qu'elles prennent en compte l'*exécution* des applications, dans le but de rendre cette exécution universellement portable.

3. Perspectives

Cette contribution propose les extensions et les services nécessaires pour que la machine virtuelle Java soit une machine effectivement universelle, avec un code, des données mais également une *exécution* des applications qui soient universellement portables. Elle constitue une base de travail qui peut être complétée de diverses manières.

Une première direction concerne l'investigation et l'expérimentation de nouvelles applications telles que l'administration des applications réparties, la répartition dynamique de charge dans un système distribué ou la construction d'environnements

reconfigurables. Cette direction suit la thématique abordé par le projet des Machines Virtuelles Virtuelles (VVM – Virtual Virtual Machines) [49] et s'insère dans la continuité et l'évolution actuelle du projet Sirac vers le projet Sardes.

Un second champ d'activité est l'étude des aspects qui ont été volontairement laissés de côté durant nos travaux. Ces aspects concernent, plus particulièrement, la transparence de la mobilité ou de la persistance à l'exécution des applications (cf. la section 4.8.2 du Chapitre 1). Un exercice technique intéressant consisterait à expérimenter l'utilisation de nos services avec les diverses solutions envisageables pour les problèmes de transparence à l'exécution. Ceci permettrait d'évaluer la complexité de la mise en place de diverses politiques de gestion de la transparence de la mobilité et de la persistance à l'exécution des applications mobiles/persistantes Java.

REFERENCES BIBLIOGRAPHIQUES

[1] B. Aaron et A. Aaron. *ActiveX Technical Reference*. Prima Pub, avril 1997.

[2] A. Acharya, M. Ranganathan et J. Salz. Sumatra: A Language for Resource-aware Mobile Programs. *Mobile Object Systems: Towards the Programmable Internet, Lecture Notes in Computer Science*, Numéro 1222, avril 1997.
http://www.cs.umd.edu/~acha/

[3] B. Alpern, C. R. Attanasio, J. J. Barton, M. G. Burke, P.Cheng, J.-D. Choi, A. Cocchi, S. J. Fink, D. Grove, M. Hind, S. F. Hummel, D. Lieber, V. Litvinov, M. F. Mergen, T. Ngo, J. R. Russell, V. Sarkar, M. J. Serrano, J. C. Shepherd, S. E. Smith, V. C. Sreedhar, H. Srinivasan et J. Whaley. The Jalapeño Virtual Machine. *IBM System Journal*, FStraVolume 39, Numéro 1, février 2000.
http://www.research.ibm.com/jalapeno/

[4] P. Amaral, C. Jacquemont, P. Jensen, R. Lea et A. Mirowski. Transparent Object Migration in COOL-2. *European Conference on Object-Oriented Programming (ECOOP'92)*, Utrecht, Pays-Bas, juin 1992.

[5] B. Andersen. Load Balancing in the Fine-Grained Object-Oriented Language Ellie. *Proceedings of the Workshop on Dynamic Object Placement and Load Balancing in Parallel and Distributed Systems Programs*, pages 97 – 102, juin 1992.

[6] K. Arnold, J. Gosling et D. Holmes. *The Java Programming Language, Third Edition*. Addison-Wesley, 2000.

[7] Y. Artsy et R. Finkel. Designing a Process Migration Facility: The Charlotte Experience. *IEEE Computer*, pages 47 – 56, septembre 1989.

[8] M. P. Atkinson, K. Chisholm et P. Cockshot. PS-algol: An Algol with a Persistent Heap. *ACM SIGPLAN Notices*, Volume 17, 1982.

[9] M. P. Atkinson et O. P. Buneman. Types and Persistence in Database Programming Languages. *ACM Computing Surveys*, Volume 19, Numéro 2, juin 1987.

[10] M. P. Atkinson et R. Morrison. Orthogonally Persistent Object Systems. *VLDB Journal*, Volume 4, 1995.

[11] M.P. Atkinson, L. Daynes, M.J. Jordan, T. Printezis et S. Spence. An Orthogonally Persistent Java. *ACM Sigmod Record*, Volume 25, Numéro 4, décembre 1996.
http://www.dcs.gla.ac.uk/pjava

[12] M. P. Atkinson. Persistence and Java - A Balancing Act. *Lecture Notes in Computer Science*, Numéro 1944, 2001.

[13] G. Attardi, A. Baldi, U. Boni, F. Carignani, G. Cozzi, A. Pelligrini, E. Durocher, I. Filotti, W. Qing, M. Hunter, J. Marks, C. Richardson et A. Watson. Techniques for dynamic software migration. *Proceedings of the 5th Annual ESPRIT Conference (ESPRIT'88)*, Volume 1, Bruxelles, Belgique, november 1988.

[14] D. Bacon, R. Konuru, C. Murthy et M. Serrano. Thin Locks: Featherweight Synchronization for Java. *Proceedings of the ACM Conference on Programming Language Design and Implementation*, ACM SIGPLAN Notices, Volume 33, Numéro 6, Montréal, Canada, juin 1998.
http://www.research.ibm.com/people/d/dfb/

[15] A. Baggio. *Adaptable and Mobile-Aware Distributed Objects*. Thèse de Doctorat, Université Pierre et Maris Curie, Paris, France, juin 1999.

[16] R. Bagrodia, W. W. Chu, L. Kleinrock et G. Popek. Vision, Issues and Architecture for Nomadic Computing. *IEEE Personal Communications Magazine*, Volume 2, Numéro 6, 1995.

[17] M. Baker, B. Carpenter, S. Hoon Ko et X. Li. mpiJava: A Java Interface to MPI. *First UK Workshop on Java for High Performance Network Computing*, Euro-Par'98, 1998.

[18] A. Barak, O. Laden et A. Braverman. The NOW MOSIX and its Preemptive Process Migration Scheme. *Bulletin of the IEEE Technical Commitee on Operating Systems and Application Environments*, Volume 7, Numéro 2, 1995.
http://www.mosix.cs.huji.ac.il

[19] A. Barak et A. Braveman. Memory Ushering in a Scalable Computing Cluster. *Journal of Microprocessors and Microsystems*, Volume 22, Numéro 3, août 1998.
http://www.mosix.cs.huji.ac.il

[20] 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*, volume 1, numéro 3, 1998.
http://mole.informatik.uni-stuttgart.de

[21] A. D. Birrell et B. J. Nelson. Implementing Remote Procedure Calls. *ACM Transactions on Computer Systems*, pages 35 - 59, février 1984.

[22] S. Bouchenak, D. Hagimont et X. Rousset de Pina. Capture et Restauration du Contexte d'Exécution d'un Thread dans l'Environnement Java. *1ère Conférence Française sur les Systèmes d'Exploitation (CFSE'1), Chapitre Français de l'ACM*, Rennes, France, 8 – 11 juin 1999.
http://sirac.inrialpes.fr/~bouchena/Publications/

[23] S. Bouchenak, D. Hagimont. Pickling Threads State in the Java System. *Technology of Object-Oriented Languages and Systems – Europe (TOOLS Europe'2000)*, Mont Saint Michel / Saint Malo, France, 5 – 8 juin, 2000.
http://sirac.inrialpes.fr/~bouchena/Publications/

[24] S. Bouchenak et D. Hagimont. Approaches to Capturing Java Thread State. *Middleware'2000*, session de posters, New York, Etats-Unis, 4 – 8 avril 2000. http://sirac.inrialpes.fr/~bouchena/Publications/

[25] S. Bouchenak. Making Java Applications Mobile or Persistent. *6th USENIX Conference on Object-Oriented Technologies and Systems (COOTS'01)*, San Antonio, Texas, USA, 29 janvier – 2 février 2001.
http://sirac.inrialpes.fr/~bouchena/Publications/

[26] G. Cabillic et I. Puaut. Stardust: An Environment for Parallel Programming on Networks of Heterogeneous Workstations. *Journal of Parallel and Distributed Comuting*, Volume 40, Numéro 1, janvier 1997.

[27] V. Cahill, S. Baker, G. Starovic and C. Horn. Generic Runtime Support for Distributed Persistent Programming. *Proceedings of the 1993 Conference on Object-Oriented Systems, Languages and Applications*, 1993.

[28] V. Cahill, P. Nixon, B. Tangney et F. Rabhi. Object Models for Distributed or Persistent Programming. *The Computer Journal*, Volume 40, Numéro 8, 1997.

[29] M. Campione et K. Walrath. *The Java™ Tutorial Second Edition: Object-Oriented Programming for the Internet*. Addison-Wesley, 1998.

[30] L. Cardelli. Mobile Computation. *Mobile Object Systems - Towards the Programmable Internet, Lecture Notes in Computer Science*, Volume 1222, 1997.

[31] Y. Cardinale et E. Hernández. Checkpointing Facility on a Metasystem. *European Conference on Parallel Computing (Euro-Par'2001)*, Manchester, Royaume-Uni, janvier 2001.
http://suma.ldc.usb.ve/

[32] Y. Cardinale et E. Hernández. Parallel Checkpointing Facility on a Metasystem. *Parallel Computing (Paro'2001)*, Naples, Italie, septembre 2001.
http://suma.ldc.usb.ve/

[33] P. Chan, R. Lee, D. Kramer et D. Kramer. The Java Class Libraries Second Edition, Volume 1: Supplement for Java 1.2. Addison-Wesley, 1999.

[34] D. Chess, C. Harrison et A. Kershenbaum. Mobile Agents: Are They a Good Idea? *T.J. Watson Research Center*, IBM Research Division, mars 1995.
http://www.research.ibm.com/iagents/publications.html

[35] G. Clavel, N. Mirouze, S. Munerot, E. Pichon et M. Soukal. *Java - La synthèse. Vers la maturité avec Java 2*. Dunod, 1999.

[36] B. J. Cox. Object-Oriented Programming : An Evolutionary Approach. Addison-Wesley, 1986.

[37] D. H. Craft. A Study of Pickling. *Journal of Object Oriented Programming*, Volume 5, Numéro 8, 1993.

[38] A. Dearle, D. Hulse et A. Farkas. Persistent Operating System Support for Java. *Proceedings of the 1st International Workshop on Persistence and Java*, 1996.
http://www.dcs.st-and.ac.uk/Rsch/persistent.html

[39] A. Dearle et D. Hulse. Operating System Support for Persistent Systems: Past, Present and Future. *Software - Practice and Experience*, Volume 30, Numéro 4, 2000.
http://www.dcs.st-and.ac.uk/Rsch/persistent.html

[40] N. De Palma, L. Bellissard, M. Riveill. Dynamic Reconfiguration of Agent-Based Applications. *Third European Research Seminar on Advances in Distributed Systems (ERSADS'99)*, Ile de Madère, Portugal, avril 1999.
http://sirac.inrialpes.fr/~depalma

[41] B. Dimitrov and V. Rego. Arachne: A Portable Threads System Supporting Migrant Threads on Heterogeneous Network Farms. *Proceedings of IEEE Parallel and Distributed Systems*, Volume 9, Numéro 5, 1998.

[42] F. Douglis et J. Ousterhout. Transparent Process Migration: Design Alternatives and the Sprite Implementation. *Software Practice and Experience*, Volume 21, Numéro 8, pages 757 – 785, août 1991.

[43] F. Douglis et B. Marsh. The Workstation as a Waystation: Integrating Mobility into Computing Environment. *The Third Workshop on Workstation Operating System (IEEE)*, avril 1992.
http://www.douglis.org/fred

[44] E. N. Elnozahy, D. B. Johnson et Y. M. Wang. A Survey of Rollback-Recovery Protocols in Message Passing Systems. *Rapport Technique CMU-CS-96-181, School of Computer Science, Carnegie Mellon University*, octobre 1996.

[45] J. Engel. *Programming for the Java(TM) Virtual Machine*. Addison-Wesley, 1999.

[46] M. R. Eskicioglu. Design Issues of Process Migration Facilities in Distributed Systems. *Scheduling and Load Balancing in Parallel and Distributed Systems*, IEEE Computer Society Press, mars 1995.
ftp://menaik.cs.ualberta.ca/pub/rasit/issues.ps.gz

[47] S. I. Feldman et C. B. Brown. IGOR: A System for Program Debugging via Reversible Execution. *Proceedings of the ACM SIGPLAN/SIGOPS Workshop on Parallel and Distributed Debugging*, Madison, WI, Etats-Unis, mai 1988.

[48] B. Folliot, P. Sens, P-G Reverdy. Plate-forme de Répartition de Charge et de Tolérance aux Fautes pour Applications Parallèles en Environnement Réparti. *Calculateurs Parallèles*, Volume 7, Numéro 4, 1995.

[49] B. Folliot, I. Piumarta, F. Riccardi. A Dynamically Configurable, Multi-Language Execution Platform. *Proceedings of the 8th ACM SIGOPS European Workshop*, Sintra, Portugal, septembre 1998.
http://www-sor.inria.fr/projects/vvm/

[50] D. Freedman. Experience Building a Process Migration Subsystem for Unix. *USENIX Winter Conference*, Dallas, TX, Etats-Unis, janvier 1991.

[51] B. Ford et J. Lepreau. Evolving Mach 3.0 to a Migrating Thread Model. *Proceedings of the Winter 1994 USENIX Conference*, janvier 1994.
http://www.sleepless.com/people/baford/

[52] A. Fuggetta, G.P. Picco et G. Vigna. Understanding Code Mobility. *IEEE Transactions on Software Engineering*, Volume 24, Numéro 5, mai 1998.
http://www.cs.ucsb.edu/~vigna/listpub.html

[53] 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 1998. http://www.informatik.tu-darmstadt.de/~fuenf

[54] J. D. Gibson et E. M. Gibson. *The Mobile Communications Handbook*. CRC press, Springer, 1999.

[55] GNU Project web server. *Processes*. GNU Project web server , 2001.
http://www.gnu.org/manual/glibc-2.0.6/html_chapter/libc_23.html

[56] R. S. Gray, G. Cybenko, D. Kotz, R. A. Peterson et D. Rus. D'Agents: Applications and Performance of a Mobile-Agent System. Soumis à *Software-Practice and Experience*, novembre 2000.
http://agent.cs.dartmouth.edu/papers/

[57] R. Guerraoui. Strategic Directions in Object-Oriented Programming. *ACM Computing Surveys*, Volume 28 , Numéro 4, 1996.

[58] A. Hac. A Distributed Algorithm for Performance Improvement Through File Replication, File migration and process Migration. *IEEE Transactions on Software Engineering*, Volume 15, Numéro 11, novembre 1989.

[59] D. Hagimont, P. Y. Chevalier, A. Freyssinet, S. Krakowiak, S. Lacourte, J. Mossière, X. Rousset de Pina. Persistent Shared Object Support in the Guide System: Evluation and Related Work. *Proceedings of the 1994 Conference on Object-Oriented Systems*,

Languages and Applications, 1994.
http://sirac.inrialpes.fr/~hagimont

[60] D. Hagimont et F. Boyer. A Configurable RMI Mechanism for Sharing Distributed Java Objects. *IEEE Internet Computing*, Volume 5, Numéro 1, janvier 2001.
http://sirac.inrialpes.fr/~hagimont

[61] C. G. Harrison. Data or Computation: Which should we move? *4th International Agent Systems and Applications/Mobile Agents Symposium (ASA/MA'2000)*, Présentateur invité, Zurich, Suisse, septembre 2000.

[62] A. A. Helal, B. Haskell et J. L. Carter. *Any time, anywhere computing: Mobile computing concepts and technology*. Kluwer Academic Publ, 1999.

[63] E. Hernández, Y. Cardinale, C. Figueira et A. Teruel. Suma: A Scientific Meta-computer. *Parallel Computing (ParCo'99)*, Delft, Pays-Bas, août 1999. http://suma.ldc.usb.ve/

[64] C. Hofmeister et J. M. Purtilo. Dynamic Reconfiguration in Distributed Systems : Adapting Software Modules for replacement. *Proc. of the 13th International Conference on Distributed Computing Systems*, Pittsburgh, Etats-Unis, mai 1993.

[65] F. Hohl. *The Mobile Agent List*, une liste des plates-formes à agents mobiles les plus connues, octobre 1999.
http://www.informatik.uni-stuttgart.de/ipvr/vs/projekte/mole/mal/mal.html

[66] G. Hong, S. J. Ahn, S. C. Han, T. Park, H. Y. Yeom et Y. Cho. Kckpt: Checkpoint and Recovery Facility on UnixWare Kernel. *Proceedings of the 15th International Conference on Computers and Their Applications (ISCA)*, mars.2000.
http://ssrnet.snu.ac.kr/IC.html

[67] J. Howell. Straigthforward Java Persistence Through Checkpointing. *Advances in Persistent Object Systems*, pages 322-334, 1999.

[68] Y. Huang, C. Kintala et Y-M. Wang. Software Tools and Libraries for Fault-Tolerance. *IEEE Technical Commitee on Operating Systems and Application Environments*, Volume 7, Numéro 4, 1995.

[69] IBM Tokyo Research Labs. *Aglets Workbench: Programming Mobile Agents in Java*, 1996.
http://www.trl.ibm.co.jp/aglets

[70] T. Illmann, T. Krueger, F. Kargl et M. Weber. Transparent Migration of Mobile Agents Using the Java Debugger Architecture. *The Fifth IEEE International Conference on Mobile Agents (MA'2001)*, Atlanta, Géorgie, Etats-unis, 2 – 4 décembre 2001, à paraître.
http://cia.informatik.uni-ulm.de/

[71] D. Johansen, R. van Renesse et F. B. Schneider. Operating System Support for Mobile Agents. *Proceedings of the 5th IEEE Workshop on Hot Topics in Operating Systems*, Ile d'Orcas, Washington, Etats-Unis, mai 1995.
http://www.tacoma.cs.uit.no/

[72] M. Jordan et M. P. Atkinson. Orthogonal Persistence for the Java Platform: Specification and Rationale. *Rapport Technique SMLI TR-2000-94, Sun Microsystems*, décembre 2000.

[73] B. Joy, G. Steele, J. Gosling et G. Bracha. *The Java Language Specification, Second Edition (Java Series)*. Addison-Wesley, 2000.

[74] E. Jul, H. Levy, N. Hutchinson et A. Black. Fined-Grained Mobility in the Emerald System. *ACM Transactions on Computer Systems*, Volume 6, Numéro 1, pages 109 – 133, février 1988.
ftp://ftp.diku.dk/pub/diku/dists/emerald/

[75] *Kaffe.org*, 2001. http://www.kaffe.org.

[76] A. Kaplan. *Name Management: Models, Mechanisms and Applications*. Thèse de Doctorat, University of Massacusetts, MA, Etats-Unis, mai 1996.

[77] A. M. Kermarrec, A. Rowstron, M. Shapiro and P. Druschel. The IceCube Approach to the Reconciliation of Diverging Replicas. *Proceedings of the 20th Annual ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing (PODC)*, Newport, Rhode Island, Etats-Unis, août 2001.

[78] F. Knabe. *Language Support for Mobile Agent*. Thèse de Doctorat, School of Computer Science, Carnegie Mellon University, 1995.
http://www.cs.virginia.edu/~knabe/

[79] B. W. Lampson. Hints for Computer System Design. *SIGOPS Operating Systems Review*, Volume 15, Numéro 5, octobre 1983.
http://research.microsoft.com/lampson

[80] G. von Laszewski, K. Shudo etY. Muraoka. Grid-based Asynchronous Migration of Execution Context in Java Virtual Machines. *European Conference on Parallel Computing (Euro-Par'2000)*, 29 août – 1 septembre, Munich, Allemagne.
http://www-unix.mcs.anl.gov/~laszewsk/papers/europar2000/

[81] J. L. Lawall et G. Muller. Efficient Incremental Checkpointing of Java Programs. *International Conference on Dependanble Systems and Networks (DSN'2000)*, New York, Etats-Unis, juin 2000.
http://www.irisa.fr/compose/muller/

[82] C. A. Lazere et D. E. Shasha. *Out of Their Minds : The Lives and Discoveries of 15 Great Computer Scientists*. Copernicus (Springer-Verlag), 1995.

[83] S. Lewis. *The Art and science of Smalltalk*. Prentice Hall, 1995.

[84] S. Liang. *The Java Native Interface: Programme's Guide and Specification (Java Series)*. Addison-Wesley, 1999.

[85] T. Lindholm et F. Yellin. *The Java Virtual Machine Specification (2nd Ed) (Java Series)*. Addison-Wesley, 1999.

[86] A. Lindström, A. Dearle, R. di Bona, S. Norris, J. Rosenberg et F. Vaughan. Persistence in the Grasshopper Kernel. *Proceedings of the 18th Australasian Computer Science Conference*, Adelaide, Australie, 1995.
http://os.dcs.st-and.ac.uk/GH/Grasshopper.html

[87] M. J. Litzkow et M. Solomon. Supporting Checkpointing and Process Migration Outside the UNIX Kernel. *USENIX Winter Conference*, pages 283 – 290, San Francisco, CA, Etats-Unis, janvier 1992.

[88] C. Lu, A. Chen et W. S. Liu. Protocols for Reliable Process Migration. *In IEEE INFOCOM'87*, San Francisco, CA, Etats-unis, mars 1987.

[89] F. Matthes et J. W. Schmidt. Persistent Threads. *Proceedings of the 20th International Conference on Very Large Data Bases (VLDB'94)*, Santiago, Chili, 1994. http://www.sts.tu-harburg.de/projects/Tycoon/

[90] B. Matthiske, F. Matthes et J. Schmidt. On Migrating Threads. *Proceedings of the Second International Workshop on Next Generation Information Technologies and Systems*, Naharia, Israel, 27 – 29 juin1995.
http://www.sts.tu-harburg.de/projects/Tycoon/

[91] S. Meloan. *The Java HotSpot Performance Engine: An In-Depth Look*. Sun Microsystems, juin 1999.
http://developer.java.sun.com/developer/technicalArticles/Networking/HotSpot/

[92] D. Milojicic, F. Douglis, Y. Paindaveine, R. Wheeler et S. Zhou. *Process Migration*. Rapport Technique TOG RI, Mars 1997.
http://www.camb.opengroup.org/RI/java/moa

[93] D. Milojicic, F. Douglis et R. Wheeler. *Mobility: Processes, Computers, and Agents*. Addison-Wesley, 1999.

[94] M. Mira da Silva and M. Atkinson. Combining Mobile Agents with Persistent Systems: Opportunities and Challenges. *Proceedings of The 2nd ECOOP Workshop on Mobile Object Systems*, Linz, Autriche, juillet 1996

[95] P. Monday, J. Carey et M. Dangler. *San Francisco Component Framework: An Introduction*. Addison-Wesley, 2000.

[96] D. Mosberger. Memory Consistency Models. *SIGOPS Operating Systems Review*, Volume 27, Numéro 1, 1993.

[97] A. Mostefaoui et M. Raynal. Efficient Message Logging for Uncoordinated Checkpointing Protocols. *Rapport Technique numéro 1018, IRISA*, France, juin 1996.

[98] R. Netzer, J-M. Helary, A. Mostefaoui et M. Raynal. Communication-Based Prevention of Useless Checkpoints in Distributed Computations. *Rapport Technique numéro 1105, IRISA*, France, mai 1997.

[99] D. A. Nichols. Using Idle Workstations in a Shared Computing Environment. *Proceedings of the 11th ACM Symposium on Operating Systems Principles (SOSP)*, novembre 1987.

[100] M. Nuttall. A Brief Survey of Systems Providing Process or Object Migration Facilities. *Operating System Review*, volume 28, numéro 4, pages 64 – 79, octobre 1994.

[101] ObjectSpace. *Voyager*. 2001.
http://www.objectspace.com/products/voyager/

[102] M. O'Connor, B. Tangney, V. Cahill et N. Harris. Microkernel Support for Migration. *Distributed Systems Engineering Journal*, 1993.

[103] I. Oueichek. *Conception et Réalisation d'un Noyau d'Administration pour un Système Réparti à Objets Persistants*. Thèse de Doctorat, Institut National Polytechnique de Grenoble, France, octobre 1996.

[104] V. Paxson. A Survey of Support for Implementing Debuggers, octobre 1990.
ftp://ftp.ee.lbl.gov:papers/debugger_support.ps.Z

[105] H. Peine et T. Stolpmann. The Architecture of the Ara Platform for Mobile Agents. *Proceeding of the First International Workshop on Mobile Agents MA'97*, Berlin, Allemagne, 7 – 8 avril 1997.
http://wwwagss.informatik.uni-kl.de/Projekte/Ara/

[106] S. Perret. Agents mobiles pour l'accès nomade à l'informatique répartie dans les réseaux à grande envergure. Thèse de Doctorat, Université Joseph Fourier, 1997.

[107] J. S. Plank, M. Beck, G. Kingsley et K. Li. Libckpt: Transparent Checkpointing under Unix. Proceedings of USENIX Technical Conference, 1992.
http://www.cs.utk.edu/~plank

[108] G. Pomberger et G. Blaschek. Object Orientation and Prototyping in Software Engineering. Prentice Hall, 1996.

[109] M. L. Powell et B. P. Miller. Process Migration in DEMOS/MP. *Proceedings of the 9th ACM Symposium on Operating System Principles (SOSP'83)*, pages 110 – 119, novembre 1983.

[110] R. Pozo et B. Miller. *SciMark 2.0 Documentation*, 2000.
http://math.nist.gov/scimark2

[111] M. Ranganathan, A. Acharya, S. D. Shamik et J. Salz. Network-Aware Mobile Programs. *Proceedings of the USENIX 1997 Annual Technical Conference*, Anaheim, CA, Etats-Unis, janvier 1997.

[112] J. E. Richardson, M. J. Carey et D. T. Schuh. The design of the E Programming Language. *ACM Transactions on Programming Languages Systems*, Volume 15, 1993.

[113] J. V. E. Ridgway, C. Thrall et J. C. Wileden. Toward Assessing Approaches to Persistence for Java. *Proceedings of the 2nd International Workshop on Persistence and Java*, San Francisco, CA, Etats-Unis, août 1997.

[114] R. Riggs, J. Waldo, A. Wollrath et K. Bharat. Pickling State in the Java System. *Proceedings of the USENIX Conference on Object-Oriented Technologies (COOTS'96)*, Toronto, Ontario, Canada, juin 1996.
http://www.usenix.org/publications/library/proceedings/coots96/full_papers/riggs/

[115] J. Rosenberg, A. Dearle, D. Hulse, A. Lindstrom et S. Norris. Operating System Support for Persistent and Recoverable Computations. *Communications of the ACM*, Volume 39, Numéro 9, septembre 1996.

[116] T. Sakamoto, T. Sekiguchi, A. Yonezawa. Bytecode Transformation for Portable Thread Migration in Java. *Proceedings of Second International Workshop Mobile Agents 2000 (MA'2000)*, Zurich, Suisse, septembre 2000.
http://www.yl.is.s.u-tokyo.ac.jp/~takas/

[117] J. W. Schmidt et M. Brodie. *Relational Database Systems*. Springer-Verlag, 1983.

[118] T. Sekiguchi, H. Masuhara et A. Yonezawa. A Simple Extension of Java Language for Controllable Transparent Migration and its Portable Implementation. *Coordination Languages and Models, Lecture Notes in Computer Science*, Volume 1594, avril 1999.
http://web.yl.is.s.u-tokyo.ac.jp/amo/

[119] M. Shapiro, P. Gautron et L. Mosseri. Persistence and Migration for C++ Objects. *Proceedings of the European Conference on Object-Oriented Programming*, 1989.

[120] C. M. Shub. Native Code Process-Originated Migration in a Heterogeneous Environment. *Proceedings of the 1990 ACM Annual Conference on Cooperation*, pages 266 – 270, Washington, Etats-Unis, 20 – 22 février 1990.

[121] J. M. Smith. A Survey of Process Migration Mechanisms. *Operating Systems Review*, Volume 22, Numéro 3, pages 28 – 40, juillet 1988. http://www.cis.upenn.edu/~jms/

[122] J. M. Smith et J. Ioannidis. Implementing Remote fork with Checkpoint-Restart. *IEEE Technical Commitee on Operating Systems Newsletter*, Volume 3, Numéro 1, pages 15 - 19, 1989.

[123] P. Smith et N. C. Hutchinson. Heterogeneous process Migration: The Tui System. *Rapport Technique 96-04, British Columbia University*, avril 1996.

[124] J. Srouji, P. Schuster, M. Bach et Y. Kuzmin. A Transparent Checkpoint Facility on NT. *Proceedings of the 2nd USENIX Windows NT Symposium*, août 1998.

[125] J. W. Stamos. *Remote Evaluation*. Thèse MIT/LCS/TR-354, Massachusetts Institute of Technology, Laboratory for Computer Science, Cambridge, Massachusetts, Etats-Unis, janvier 1986.

[126] B. Steensgaard et E. Jul. Object and Native Code Mobility Among Heterogeneous Computers. *Proceedings of the 15th ACM Symposium on Operating System Principles (SOSP'95)*, Copper Mountain Resort, Colorado, Etats-Unis, décembre 1995.

[127] A. F. Straw, F. Mellender et S. Riegel. Object Management in Persistent Smalltalk System. *Software Practice and Experience*, Volume 19, Numéro 8, 1989.

[128] B. Stroustrup. The C++ Programming Language, Third Edition. Addison-Wesley, 2000.

[129] T. Suezawa. Persistent Execution State of a Java Virtual Machine. *Proceedings of the ACM 2000 Java Grande Conference*, San Francisco, CA, Etas-Unis, juin 2000. http://www.ifi.inizh.ch/staff/suezawa

[130] Sun Microsystems. *javap - The Java Class File Disassembler*. Sun Microsystems. http://java.sun.com/products/jdk/1.2/docs/tooldocs/solaris/javap.html

[131] Sun Microsystems, Inc. *Entreprise Java Beans Specification 1.0*, mars 1998. http://java.sun.com/products/ejb

[132] Sun Microsystems. *Applets*. Sun Microsystems, 2000. http://java.sun.com/applets/

[133] Sun Microsystems. *Improving Serialization Performance with Externalizable*. *Technical Tips*, Sun Microsystems, April 2000. http://developer.java.sun.com/developer/TechTips/2000/tt0425.html

[134] Sun Microsystems. *Java Card Technology*. Sun Microsystems, 2000. http://java.sun.com/products/javacard/

[135] Sun Microsystems. *Java JIT Compiler Overview*. Sun Microsystems, 2000. http://www.sun.com/solaris/jit/

[136] Sun Microsystems. *Object Serialization*. Sun Microsystems, 2001. http://java.sun.com/j2se/1.3/docs/guide/serialization/

[137] Sun Microsystems. *Java 2 Platform, Standard Edition, v1.2.2 - API Specification*. Sun Microsystems, 2001.
http://java.sun.com/products/jdk/1.2/docs/api/

[138] Sun Microsystems. *Java 2 SDK, Standard Edition*. Sun Microsystems, 2001. http://java.sun.com/products/jdk/1.2/

[139] Sun Microsystems. *CLDC and the K Virtual Machine (KVM)*. Sun Microsystems, 2001. http://java.sun.com/products/cldc/

[140] Sun Microsystems. *The Java 2 Community Source Developers' Area*. Sun Microsystems, Avril 2001.
http://developer.java.sun.com/developer/products/java2cs/

[141] Sun Microsystems. *Java Virtual Machine Debug Interface Reference*. Sun Microsystems, Avril 2001.
http://java.sun.com/products/jdk/1.2/docs/guide/jvmdi/jvmdi.html

[142] N. Suri, J. M. Bradshaw, M. R. Breedy, P. T. Groth, G. A. Hill et R. Jeffers. Strong Mobility and Fined-Grained Resource Control in NOMADS. *Agent Systems and Applications / Mobile Agents (ASA MA'2000)*, 13 – 15 septembre 2000, Zurich, Suisse.

[143] M. M. Theimer, K. A. Lantz et D. R. Cheriton. Preemptable Remote Execution Facilities for the V-System. *Proceedings of the 10th ACM Symposium on Operating System Principles (SOSP'85)*, décembre 1985.

[144] M. Tremblay, Sun Microelectronics. picoJava: A Hardware Implementation of the Java Virtual Machine. *JavaOne: Sun's Worldwide Java Conference*, 29-31 Mai 1996. http://java.sun.com/javaone/javaone96/

[145] C. Tricot. *Courbes et dimension fractale*. Springer, 1993.

[146] A. Tripathi, N. Karnik, M. Vora, T. Ahmed et R. Singh. Mobile Agent Programming in Ajanta. *Proceedings of the 19th International Confernce on Distributed Computing Systems (ICDCS '99)*, Austin, Texas, Etats-Unis, 31 mai – 4 juin 1999.
http://www.cs.umn.edu/Ajanta/

[147] E. Truyen, B. Robben, B. Vanhaute, T. Coninx, W. Joosen et P. Verbaeten. Portable Support for Transparent Thread Migration in Java. *Proceedings of Second International Workshop Mobile Agents 2000 (MA'2000)*, Zurich, Suisse, septembre 2000. http://www.cs.kuleuven.ac.be/~eddy/research.html

[148] B. J. Walker et R. M. Matthews. Process Migration in AIX's Transparent Computing Facility (TCF). IEEE Technical Commitee on Operating Systems newsletter, Volume 3, Numéro 1, pages 5 – 7, 1989.

[149] Y-M. Wang, Y. Huang, K-P. Vo, P-Y Chung et C. Kintala. Checkpointing and its Applications. *Proceedings of the 25th IEEE Fault-Tolerant Computing Symposium (FTCS'25)*, Pasadena, CA, Etats-Unis, juin 1995.

[150] D. Wong, N. Paciorek, T. Walsh, J. DiCelie, M. Young et B. Peet. Concordia: An Infrastructure for Collaborating Mobile Agents. *First International Workshop on Mobile Agents 97 (MA'97)*, Berlin, Allemagne, 7 – 8 avril 1997.

[151] E. R. Zayas. Attacking the Process Migration Bottleneck. *Proceedings of the 11th ACM Symposium on Operating System Principles (SOSP'87)*, pages 13 – 24, Austin, Texas, Etats-Unis, 8 – 11 novembre 1987.

ANNEXES

Annexe I. Interface des nos services

1. Interface visible du package java.lang.threadpack

Interface Summary	
ReceiveInterface	A class implements the ReceiveInterface interface to indicate the way a captured thread state is received.
SendInterface	A class implements the SendInterface interface to indicate the way a captured thread state is transferred.

Class Summary	
CapturableThread	A CapturableThread object is a Java thread whose current execution state can be captured and therefore restored.
MobileThreadManagement	The MobileThreadManagement class provides several useful services for moving Java threads, during their execution, between different Java virtual machines.
PersistentThreadManagement	The PersistentThreadManagement class provides several useful services for making Java threads persistent.
ThreadState	Instances of the class ThreadState represent execution states of Java threads running in a Java application.
ThreadStateManagement	The ThreadStateManagement class provides several useful services for the capture and restoration of Java thread states.

Exception Summary	
MobilityException	Thrown when thread persistence operations are not supported, such as store, storeAndStop and load.
PersistenceException	Thrown when thread persistence operations are not supported, such as store, storeAndStop and load.
ThreadCaptureException	Thrown when the capture of a thread state is not supported.
ThreadRestorationException	Thrown when the restoration of a thread state is not supported.

2. Interface de nos services de mobilité des threads Java

java.lang.threadpack
Class MobileThreadManagement
public final class **MobileThreadManagement** extends Object
The MobileThreadManagement class provides several useful services for making Java threads mobile.

Method Summary	
static void	go(String host, int port) Transfers the excution of the current thread from the local host to a target host identified by a name and a port number. The execution of the thread is resumed on both the target host and the local host.
static void	go(Socket socket) Transfers the excution of the current thread from the local host to a target host identified by a socket. The execution of the thread is resumed on both the target host and the local host.
static void	go(CapturableThread thread, String host, int port) Transfers the excution of the specified thread from the local host to a target host identified by a name and a port number. The execution of the thread is resumed on both the target host and the local host.
static void	go(CapturableThread Thread thread, Socket socket) Transfers the excution of the specified thread from the local host to a target host identified by a socket. The execution of the thread is resumed on both the target host and the local host.
static void	go(Thread thread, String host, int port) Transfers the excution of the specified thread from the local host to a target host identified by a name and a port number. The execution of the thread is resumed on both the target host and the local host. If the thread argument is the current thread, it may not be a CapturableThread instance. Otherwise, it must be a CapturableThread instance.
static void	go(Thread thread, Socket socket) Transfers the excution of the specified thread from the local host to a target host identified by a socket. The execution of the thread is resumed on both the target host and the local host. If the thread argument is the current thread, it may not be a CapturableThread instance. Otherwise, it must be a CapturableThread instance.
static void	goAndStop(String host, int port) Transfers the excution of the current thread from the local host to a target host identified by a name and a port number. The execution of the current thread is stopped on the local host and resumed on the target host.
static void	goAndStop(Socket socket) Transfers the excution of the current thread from the local host to a target host identified by a socket. The execution of the current thread is stopped on the local host and resumed on the target host.
static void	goAndStop(CapturableThread thread, String host, int port) Transfers the excution of the specified thread from the local host to a target host identified by a name and a port number. The execution of the specified thread is stopped on the local host and resumed on the target host.
static void	goAndStop(CapturableThread thread, Socket socket) Transfers the excution of the specified thread from the local host to a target host identified by a socket. The execution of the specified thread is stopped on the local host and resumed on the target host.
static void	goAndStop(Thread thread, String host, int port) Transfers the excution of the specified thread from the local host to a

	target host identified by a name and a port number. The execution of the specified thread is stopped on the local host and resumed on the target host. If the thread argument is the current thread, it may not be a CapturableThread instance. Otherwise, it must be a CapturableThread instance.
static void	goAndStop(Thread thread, Socket socket) Transfers the excution of the specified thread from the local host to a target host identified by a socket. The execution of the specified thread is stopped on the local host and resumed on the target host. If the thread argument is the current thread, it may not be a CapturableThread instance. Otherwise, it must be a CapturableThread instance.
static Thread	arrive(String host, int port) Receives a mobile thread and resumes its execution on a host identified by a name and a port number.
static Thread	arrive(int port) Receives a mobile thread and resumes its execution on the local host, at the specified port number.
static Thread	arrive(ServerSocket servSocket) Receives a mobile thread and resumes its execution on a host identified by a server socket.
static Thread	arrive(Socket servSocket) Receives a mobile thread and resumes its execution on a host identified by a socket.
static Thread	arrive(String host, int port, URL[] classSource) Receives a mobile thread and resumes its execution on a host identified by a name and a port number. The classes used by the mobile thread are located at the specified URLs.
static Thread	arrive(int port, URL[] classSource) Receives a mobile thread and resumes its execution on the local host, at the specified port number. The classes used by the mobile thread are located at the specified URLs.
static Thread	arrive(ServerSocket servSocket, URL[] classSource) Receives a mobile thread and resumes its execution on a host identified by a server socket. The classes used by the mobile thread are located at the specified URLs.
static Thread	arrive(Socket servSocket, URL[] classSource) Receives a mobile thread and resumes its execution on a host identified by a socket. The classes used by the mobile thread are located at the specified URLs.
static CapturableThread	arriveAsCapturable(String host, int port) Receives a mobile capturable thread and resumes its execution on a host identified by a name and a port number.
static CapturableThread	arriveAsCapturable (int port) Receives a mobile capturable thread and resumes its execution on the local host, at the specified port number.
static CapturableThread	arriveAsCapturable (ServerSocket servSocket) Receives a mobile capturable thread and resumes its execution on a host identified by a server socket.
static CapturableThread	arriveAsCapturable (Socket servSocket) Receives a mobile capturable thread and resumes its execution on a host identified by a socket.
static CapturableThread	arriveAsCapturable (String host, int port, URL[] classSource) Receives a mobile capturable thread and resumes its execution on a host identified by a name and a port number. The classes used by the mobile thread are located at the specified URLs.
static CapturableThread	arriveAsCapturable(int port, URL[] classSource) Receives a mobile capturable thread and resumes its execution on the local host, at the specified port number. The classes used by the mobile thread are located at the specified URLs.

static CapturableThread	arriveAsCapturable (ServerSocket servSocket, URL[] classSource) Receives a mobile capturable thread and resumes its execution on a host identified by a server socket. The classes used by the mobile thread are located at the specified URLs.
static CapturableThread	arriveAsCapturable (Socket servSocket, URL[] classSource) Receives a mobile capturable thread and resumes its execution on a host identified by a socket. The classes used by the mobile thread are located at the specified URLs.

3. Interface de nos services de persistance des threads Java

java.lang.threadpack
Class PersistentThreadManagement
public final class **PersistentThreadManagement** extends Object
The PersistentThreadManagement class provides several useful services for making Java threads persistent.

Method Summary	
static void	store(String fileName) Stores the excution of the current thread on a file identified by a name. The execution of the current thread is resumed.
static void	storeAndStop(String fileName) Stores the excution of the current thread on a file identified by a name. The execution of the current thread is stopped.
static void	store(CapturableThread thread, String fileName) Stores the excution of the specified thread on a file identified by a name. The execution of the current thread is resumed.
static void	store(Thread thread, String fileName) Stores the excution of the specified thread on a file identified by a name. The execution of the current thread is resumed. If the thread argument is the current thread, it may not be a CapturableThread instance. Otherwise, it must be a CapturableThread instance.
static void	storeAndStop (CapturableThread thread, String fileName) Stores the excution of the specified thread on a file identified by a name. The execution of the current thread is stopped.
static void	storeAndStop (Thread thread, String fileName) Stores the excution of the specified thread on a file identified by a name. The execution of the current thread is stopped. If the thread argument is the current thread, it may not be a CapturableThread instance. Otherwise, it must be a CapturableThread instance.
static Thread	load(String fileName) Loads a thread's excution from a file identified by a name. The loaded execution is resumed.
static Thread	load(String fileName, URL[] classSource) Loads a thread's excution from a file identified by a name. The loaded execution is resumed. The classes used by the restored thread are located at the specified URLs.
static CapturableThread	loadAsCapturable(String fileName) Loads a capturable thread's excution from a file identified by a name. The loaded execution is resumed.
static CapturableThread	loadAsCapturable (String fileName, URL[] classSource) Loads a capturable thread's excution from a file identified by a name. The loaded execution is resumed. The classes used by the restored thread are located at the specified URLs.

4. Interface de nos services de capture/restauration d'état des threads Java

java.lang.threadpack
Class ThreadStateManagement
public final class **ThreadStateManagement** extends Object
The ThreadStateManagement class provides several useful services for the capture and restoration of Java thread state.

Method Summary	
static ThreadState	capture() Captures the state of the current Java thread and returns it as a ThreadState object.
static ThreadState	capture(CapturableThread thread) Captures the state of a Java thread and returns it as a ThreadState object.
static ThreadState	capture(Thread thread) Captures the state of a Java thread and returns it as a ThreadState object. If the thread argument is the current thread, it may not be a CapturableThread instance. Otherwise, it must be a CapturableThread instance.
static ThreadState	captureAndStop(CapturableThread thread) Captures the state of a Java thread, stops the execution of the state and returns the state as a ThreadState object.
static ThreadState	captureAndStop(Thread thread) Captures the state of a Java thread, stops the execution of the state and returns the state as a ThreadState object. If the thread argument is the current thread, it may not be a CapturableThread instance. Otherwise, it must be a CapturableThread instance.
static Thread	restore (ThreadState threadState, Class[] classes) Creates a new Java thread, initializes it with a previously captured state and starts it.
static void	captureAndSend(SendInterface stateTransfItf, boolean toStop) Captures the state of the current Java thread and sends it (to a remote node or on disk) by calling the sendState method of the SendInterface argument.
static void	captureAndSend(CapturableThread thread, SendInterface stateTransfItf, boolean toStop) Captures the state of a Java thread and sends it (to a remote node or on disk) by calling the sendState method of the SendInterface argument.
static void	captureAndSend(Thread thread, SendInterface stateTransfItf, boolean toStop) Captures the state of a Java thread and sends it (to a remote node or on disk) by calling the sendState method of the SendInterface argument. If the thread argument is the current thread, it may not be a CapturableThread instance. Otherwise, it must be a CapturableThread instance.
static Thread	receiveAndRestore(ReceiveInterface stateTransfItf) Receives the state of a Java thread by calling the receiveState method of the ReceiveInterface argument, creates a new Java thread, initializes it with the received state and starts it.
static CapturableThread	restoreAsCapturable (ThreadState threadState, Class[] classes) Creates a new capturable Java thread, initializes it with a previously captured state and starts it.

static CapturableThread	receiveAndRestoreAsCapturable (ReceiveInterface stateTransfItf) Receives the state of a Java thread by calling the receiveState method of the ReceiveInterface argument, creates a new capturable Java thread, initializes it with the received state and starts it.
static void	captureAndSend(SendingState stateTransfItf, boolean toStop) Captures the state of the current Java thread and sends it (to a remote node or on disk) by calling the sendState method of the SendingState argument.
static void	captureAndSend(CapturableThread thread, SendingState stateTransfItf, boolean toStop) Captures the state of a Java thread and sends it (to a remote node or on disk) by calling the sendState method of the SendingState argument.
static void	captureAndSend(Thread thread, SendingState stateTransfItf, boolean toStop) Captures the state of a Java thread and sends it (to a remote node or on disk) by calling the sendState method of the SendingState argument. If the thread argument is the current thread, it may not be a CapturableThread instance. Otherwise, it must be a CapturableThread instance.
static Thread	receiveAndRestore(ReceivingState stateTransfItf) Receives the state of a Java thread by calling the receiveState method of the ReceivingState argument, creates a new Java thread, initializes it with the received state and starts it.
static CapturableThread	receiveAndRestoreAsCapturable (ReceivingState stateTransfItf) Receives the state of a Java thread by calling the receiveState method of the ReceivingState argument, creates a new capturable Java thread, initializes it with the received state and starts it.

5. Interface de nos services de spécialisation de la capture et restauration de l'état d'exécution des threads Java

java.lang.threadpack
Interface SendInterface
public interface **SendInterface** extends Object
A class implements the SendInterface interface to indicate the way a captured thread state is processed (sent over the network for thread mobility or stored on disk for persistence purpose).

Method Summary	
abstract void	sendState(ThreadState threadState) Specifies the way a thread state is processed after a capture operation.

java.lang.threadpack
Interface ReceiveInterface
public interface **ReceiveInterface** extends Object
A class implements the ReceiveInterface interface to indicate the way a thread state is processed before a restoration operation (received from the network for thread mobility or read from disk for persistence purpose).

Method Summary	
abstract ThreadState	receiveState() Specifies the way a thread state is processed before a restoration operation.

Annexe II. Instructions de la machine virtuelle Java

L'unité de base de la taille des valeurs de données dans la machine virtuelle Java est le mot (*word*), une taille fixée choisie par les concepteurs de chaque mise en œuvre de machine virtuelle Java. La taille d'un mot doit être suffisante pour pouvoir contenir une valeur de type *byte*, *short*, *int*, *char*, *float*, *returnAddress* ou *reference*. De même que deux mots doivent être suffisants pour contenir une valeur de type *long* ou *double*.

1. Opérations sur les variables locales et la pile d'opérandes

- ***Empilent d'une constante sur la pile d'opérandes***

iconst_m1	iconst_4	lconst_0	bipush
iconst_0	iconst_5	lconst_1	sipush
iconst_1	fconst_0	dconst_0	ldc
iconst_2	fconst_1	dconst_1	ldc_w
iconst_3	fconst_2	aconst_null	ldc2_w

- ***Chargement d'une variable locale sur la pile d'opérandes***

iload	fload_1	lload_3	aload
iload_0	fload_2	dload	aload_0
iload_1	fload_3	dload_0	aload_1
iload_2	lload	dload_1	aload_2
iload_3	lload_0	dload_2	aload_3
fload	lload_1	dload_3	
fload_0	lload_2		

- ***Sauvegarde d'un opérande dans une variable locale***

istore	fstore_1	lstore_3	astore
istore_0	fstore_2	dstore	astore_0
istore_1	fstore_3	dstore_0	astore_1
istore_2	lstore	dstore_1	astore_2
istore_3	lstore_0	dstore_2	astore_3
fstore	lstore_1	dstore_3	wide
fstore_0	lstore_2		

- ***Manipulation d'opérandes***

nop	swap	dup_x1	dup2_x1
pop	dup	dup_x2	dup2_x2
pop2	dup2		

2. Conversion de type

i2l	l2f	f2d	i2b
i2f	l2d	d2i	i2c
i2d	f2i	d2f	i2s
l2i	f2l	d2l	

3. Opérations arithmétiques sur les entiers

iadd	lsub	idiv	lrem
ladd	imul	ldiv	ineg
iinc	lmul	irem	lneg
isub			

4. Opérations logiques

ishl	lshl	iand	land
ishr	lshr	ior	lor
iushr	lrshr	ixor	lxor

5. Opérations arithmétiques sur les flottant

fadd	dsub	fdiv	drem
dadd	fmul	ddiv	fneg
fsub	dmul	frem	dneg

6. Opérations sur les objets et les tableaux

new	newarray	iaload	sastore
putfield	anewarray	laload	iastore
getfield	multianewarray	faload	lastore
putstatic	arraylength	daload	fastore
getstatic	baload	aaload	dastore
checkcast	calod	bastore	aastore
instanceof	saload	castore	

7. Opérations de branchement

ifeq	if_icmpeq	lcmp	if_acmpeq
ifne	if_icmpne	fcmpg	if_acmpne
iflt	if_icmplt	fcmpl	goto
ifle	if_icmple	dcmpl	goto_w
ifgt	if_icmpgt	ifnull	lookupswitch
ifge	if_icmpge	ifninnull	tableswitch

8. Exceptions

athrow

9. Clause *finally*

jsr	jsr_w	ret

10.Invocation et retour de méthode

invokevirtual	invokeinterface	freturn	areturn
invokestatic	ireturn	dreturn	return
invokespecial	lreturn		

11.Synchronisation

monitorenter	monitorexit

Annexe III. Evaluation des performances de nos services sur une plate-forme Solaris

Voici les courbes d'évaluation des performances de nos services de capture/restauration d'état d'un thread Java, de migration d'un thread et de sauvegarde/reprise d'un thread, sur une plate-forme Solaris.

1. Environnement d'évaluation

Nous avons évalué les performances de nos services sur une plate-forme Solaris. Notre environnement d'évaluation est constitué des éléments suivants :

- JDK 1.2.2
- Solaris 2.6, Sun Ultra-1 (Sparc Ultra-1 167 MHz, 64 Mo RAM)
- Ethernet 100 Mb/s.

D'autre part, nous avons exécuté nos programmes d'évaluation de performances en désactivant le compilateur Java JIT, afin d'éviter une erreur de comparaison due à des optimisations ayant lieu dans certains cas et pas dans d'autres.

2. Latence de la capture/restauration d'état

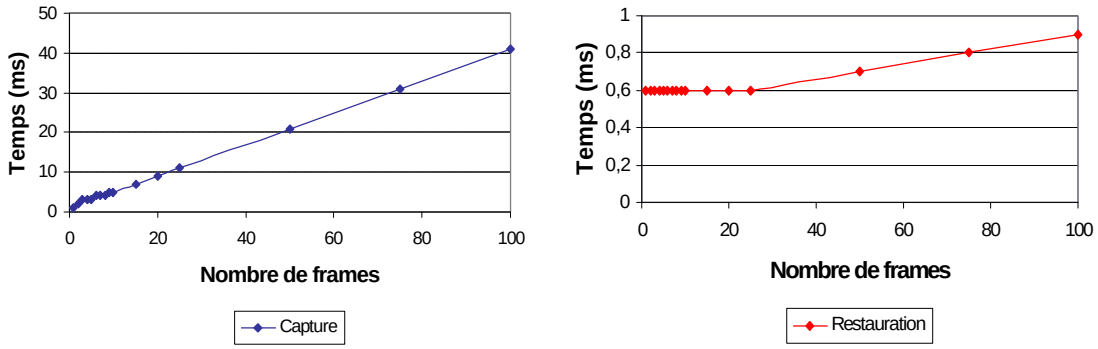

Figure I. Capture et Restauration d'état / IJSS / Solaris

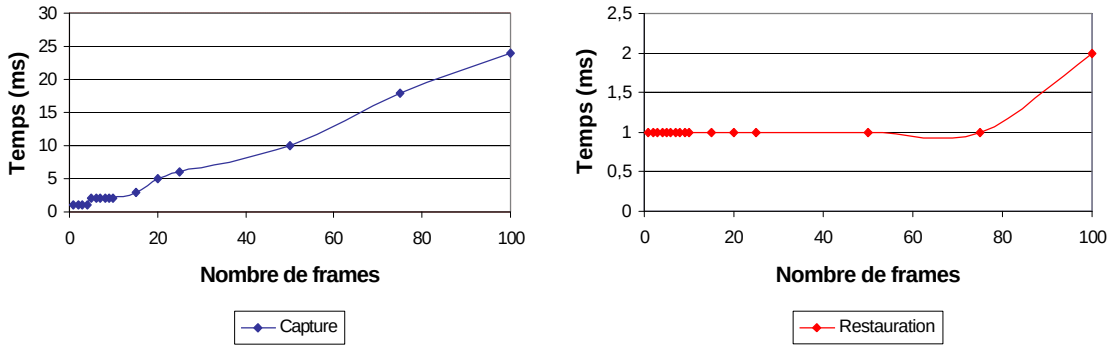

Figure II. Capture et Restauration d'état / IJES / Solaris

3. Latence de la migration de thread

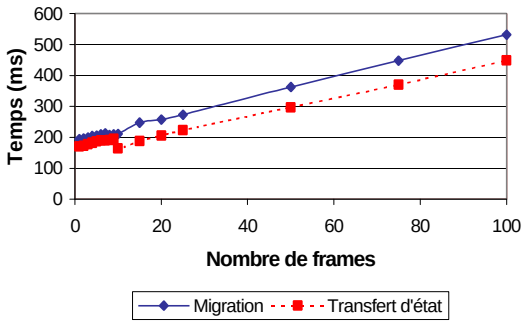

Figure III. Migration de thread / IJSS / Solaris

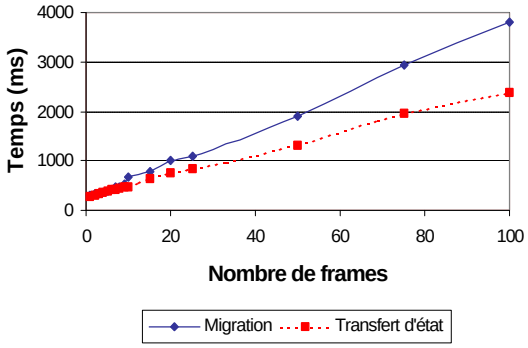

Figure IV. Migration de thread / IJES / Solaris

4. Latence de la sauvegarde/reprise de thread

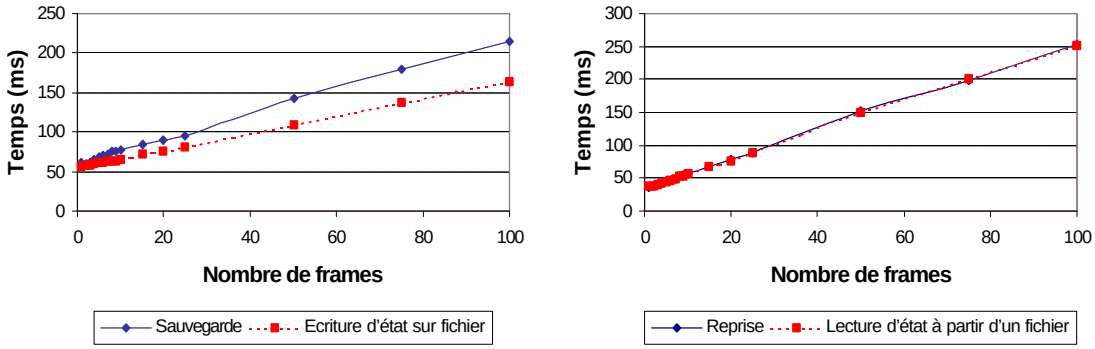

Figure V. Sauvegarde et Reprise de thread / IJSS / Solaris

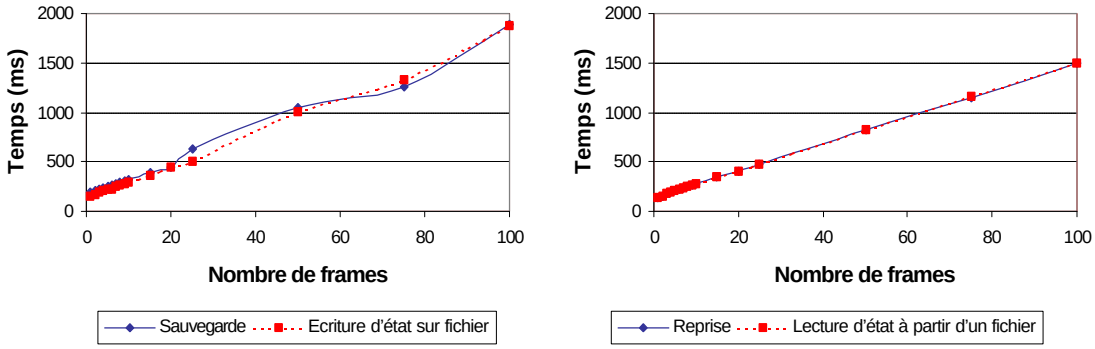

Figure VI. Sauvegarde et Reprise de thread / IJES / Solaris

Résumé

Les travaux de cette thèse portent sur la mobilité et la persistance des applications dans des environnements hétérogènes. Ces fonctions sont utiles à la répartition dynamique de charge dans les systèmes distribués, à la reconfiguration dynamique d'applications réparties ou à la mise en place de techniques de tolérance aux pannes. La mobilité et la persistance des applications ont largement été abordées au niveau du système d'exploitation ou au niveau du langage/modèle de programmation. Mais très peu de travaux ont été menés au niveau des machines virtuelles. L'objet de cette thèse est l'étude et la réalisation de fonctions de mobilité et de persistance dans la machine virtuelle Java.

L'environnement Java fournit des outils pour la mobilité et la persistance du code et des données mais il n'adresse pas le problème de mobilité ni de persistance de l'état d'exécution des processus légers (*threads*). Les travaux de cette thèse portent sur la conception de tels services, une conception guidée par deux principes : la portabilité sur des environnements hétérogènes et le respect des performances des applications. Pour des besoins de performances, plusieurs efforts ont été faits par les concepteurs de Java en matière d'optimisation de l'exécution et de compilation à la volée. Notre solution permet de fournir des fonctions de mobilité et de persistance portables même en présence de compilation à la volée. Ceci est mis en œuvre en reposant, d'une part, sur des techniques d'inférence dynamique du type des données sur la pile d'exécution à partir du code Java exécuté et, d'autre part, sur des techniques de dés-optimisation dynamique du code Java compilé à la volée.

Nos services ont été réalisés via une extension de la machine virtuelle Java de Sun Microsystems. Ils sont opérationnels et ont pu être validés pour des besoins de tolérance aux pannes dans une plate-forme de metacomputing. Notre solution est actuellement l'unique approche qui est complète et qui élimine toute pénalité sur l'exécution des applications.

Mots-clés : mobilité, migration, persistance, sauvegarde, reprise, Java, JVM, threads

Abstract

This work tackles the problem of applications mobility and applications persistence in heterogeneous environments. These functions have many fields of use such as dynamic load balancing in distributed systems, fault tolerance and dynamic reconfiguration of distributed applications. Applications mobility and persistence have been largely addressed at the operating system level and at the programming language/model level. However, few works were done at the virtual machine level. The goal of this thesis is to propose mobility and persistence functions at the Java virtual machine level.

The Java environment provides many useful tools for making code and data mobile or persistent but it does not address threads' execution state mobility and persistence. We provide such functionalities, following two principles: portability across heterogeneous platforms with respect to applications performance. Indeed, regarding performance, an important effort was made by Java's designers in terms of execution optimization and Just-In-Time (JIT) compilation. Our solution allows us to provide mobility and persistence functions that are portable even in the presence of JIT compilation. This is performed by basing our solution, on the one hand, on techniques of dynamic type inference via flow analysis, and on the other hand, on techniques of dynamic de-optimization of compiled Java code.

Our mobility and persistence services were implemented as an extension of Sun Microsystems' Java virtual machine. They are operational and were used in a metacomputing platform for fault tolerance purpose. Nowadays, this solution is the unique approach that is complete without inducing any performance overhead for applications.

Keywords: mobility, migration, persistence, checkpointing, recovery, Java, JVM, threads