MVV : une plate-forme à composants dynamiquement reconfigurables[1]

La machine virtuelle virtuelle

Frédéric Ogel[‡] — Gaël Thomas[†] — Antoine Galland[†⋆] — Bertil Folliot[†]

[‡]*INRIA Rocquencourt,*
Domaine de Voluceau, F-78153 Le Chesnay.
frederic.ogel@inria.fr
[†]*Laboratoire d'Informatique Paris VI, Université Pierre et Marie Curie,*
4, place Jussieu, F-75252 Paris Cedex 05.
gael.thomas@lip6.fr,
bertil.folliot@lip6.fr
[⋆]*Gemplus Research Labs,*
La Vigie, Avenue du Jujubier, ZI Athelia IV, F-13705 La Ciotat Cedex.
antoine.galland@gemplus.com

RÉSUMÉ. *Le nombre toujours croissant de domaine d'application émergent entraîne un nombre croissant de solutions* ad hoc*, rigides et faiblement interopérables. Notre réponse à cette situation est une plate-forme pour la construction d'applications et d'environnements d'exécutions flexibles et interopérables appelée machine virtuelle virtuelle. Cet article présente notre approche, l'architecture de la plate-forme ainsi que les premières applications.*

ABSTRACT. *With the wide acceptance of distributed computing a rapidly growing number of application domains are emerging, leading to a growing number of ad-hoc solutions that are rigid and poorly interoperable. Our response to this situation is a platform for building flexible and interoperable execution environments called the virtual virtual machine. This article presents our approach, the architecture of the VVM and some of its primary applications.*

MOTS-CLÉS : *machine virtuelle, adaptablité, composant, intergiciel.*

KEYWORDS: *virtual machine, adaptability, component, middleware.*

1. Ces travaux sont en partie financés par le projet européen IST COACH (2001-34445).

1. Introduction

Il est un fait que les systèmes d'exploitations dits *traditionnels*, comme *Unix* ou *Windows*, ne répondent pas toujours aux besoins des applications. En effet, non seulement ils appliquent en générale la même politique à toutes les applications (ce qui a pour effet de privilégier certaines classes d'applications au détriment des autres), mais ils se trouvent également rapidement confrontés à des applications nouvelles, dont les besoins sont également nouveaux et ne correspondent pas à ce qu'ils proposent, ou tout simplement à ce qui pouvait être imaginé lors de leur conception.

De fait, ces dernières années ont vues l'émergence rapide d'un grand nombre de nouveaux domaines applicatifs tels que le multimédia sur internet, le déploiement dynamique de services/applications actives, ou encore les applications mobiles. Ces nouveaux domaines d'application ont fait apparaître des limitations dans les environnements d'exécution actuels. Chacun a en effet des besoins et des contraintes qui lui sont propre, liées à sa sémantique et qui ne correspondent pas, en générale, aux abstractions et politiques prédéfinies contenues dans un environnement d'exécution traditionnel. Ces environnements n'offrant aucun support de flexibilité, les développeurs doivent maîtriser, en plus de la complexité du domaine d'application, celle des différents environnements d'exécution cibles et celle, non négligeable, de l'expression de la sémantique de l'application (et donc de ses besoins et contraintes) dans le cadre d'un environnement n'ayant pas été prévu pour et offrant le plus souvent des abstractions et une sémantique inadaptée.

Les conséquences sont immédiates : de nombreuses applications, défavorisées par telle ou telle politique de gestion de ressources, comme l'ordonnancement ou les défauts de page, atteindraient de bien meilleures performances avec d'autres stratégies que celles proposées par le système. Un exemple classique est celui des bases de données qui accèdent à de très grandes quantités d'informations : l'application, voir le SGBD[1], est souvent en mesure de déterminer les prochaines pages à consulter, or le système utilise une politique de défauts de page statique (par exemple LRU) qui n'exploite aucunement les informations dont l'application dispose. Il en résulte bien évidemment des performances en dessous de ce que l'on pourrait obtenir en utilisant les connaissances de l'application.

C'est en constatant ce déséquilibre de plus en plus grand entre les besoins des applications et ce qu'offrent les systèmes qu'est né le concept des systèmes flexibles. L'objectifs de tels systèmes est d'offrir aux applications la possibilité de spécifier *leurs besoins*, ou plus exactement d'adapter la gestion des ressources à ces derniers.

Un constat similaire est également fait dans les différents sous-systèmes constitutifs d'un environnement d'exécution : intergiciels, machines virtuelles et langages de programmation. Des techniques pour la flexibilité ont ainsi été proposées, offrant des solutions plus ou moins *ad hoc*.

1. Système de Gestion de Base de Données.

Par exemple, les systèmes et architectures répartis actuels, s'ils sont en voix de standardisation (comme *CORBA*, *.COM+* ou *EJB*), restent néanmoins complexes, adaptés à certains besoins applicatifs, de par leur héritage du modèle RPC, et ne sont pas prévus pour être spécialisés pour d'autres domaines d'applications ou architectures. Ceci conduit à une prolifération de solutions *ad hoc*, adaptées à un domaine d'application particulier, mais difficilement réutilisables ou interopérables avec l'existant. Ceci concerne les supports d'exécution pour l'embarqué (comme les téléphones mobiles ou les cartes à puces), pour la gestion de la qualité de service (comme le temps réel ou la tolérance aux fautes) et plus généralement pour l'adaptation des services à des besoins non prévus au moment du déploiement de l'application.

Tant que subsistera la décomposition artificielle de l'environnement d'exécution en entités distinctes, et plus particulièrement le découplage système/langage présenté dans [HOW 99], il n'y aura pas de solution complète à la flexibilité logicielle. En effet, les systèmes d'exploitation, même flexibles, n'offre qu'une flexibilité limitée et, du fait de ce découplage, ignorent tous les aspects langage. A l'opposé, les machines virtuelles, comme les langages pour flexibilité, considèrent le système comme une *boîte noire* rigide et fermée. Le cas des intergiciels flexibles est encore pire : ils sont limités à la fois par le système sous-jacent et par leur ignorance du modèle/langage de programmation utilisé par l'application.

Pour répondre à ce problème nous présentons une approche systématique pour la construction d'environnements logiciels adaptables, spécialisables et interopérables réconciliant langage et système : la *machine virtuelle virtuelle* [FOL 00]. La *MVV* fournit un environnement à la fois de programmation et d'exécution dont les objectifs sont : (i) d'adapter le langage de programmation et l'environnement d'exécution à un domaine d'application spécifique (carte à puces, etc.) ; (ii) d'étendre dynamiquement l'environnement d'exécution pour y ajouter/supprimer/changer des fonctionnalités en fonction des conditions opératoires ; (iii) de fournir un support pour l'interopérabilité entre divers domaines d'applications (langages et/ou environnements d'exécution).

Le reste de cet article se décompose comme suit. La section 2 fournit un tour d'horizon des travaux de recherches autour de la flexibilité dans les environnements d'exécution, avant de présenter les principes de base des *MVV* dans la section 3 puis le cœur de la *MVV*, appelée *YNVM*, dans la section 4. Les utilisations de la *YNVM* sur machine nue, au-dessus d'un système d'exploitation et en tant qu'un support d'exécution sont décrites respectivement dans les sections 5, 6 et 7. Nous concluons et présentons les perspectives de ces travaux dans la section 8.

2. Travaux similaires

De nombreux travaux sont menés autour de la flexibilité dans les environnements d'exécution, chacun se focalisant sur un élément constitutif d'un environnement d'exécution (système, machine virtuelle, intergiciel…). Ainsi, le constat de rigidité des sys-

tèmes d'exploitation traditionnels présenté dans l'introduction a motivé de nombreux travaux visant la conception et la mise en œuvre de systèmes d'exploitation *flexibles*.

L'objectif de cette approche système est de rendre aux applications le contrôle de la gestion des ressources ainsi que des services exportés. Pour cela, la flexibilité repose sur la modularisation des stratégies sous forme de librairies ou de modules. Suivant le type de noyau (exo, nano, micro ou monolithique), les extensions sont chargées à l'intérieur du noyau ou au niveau applicatif. On distingue donc principalement deux familles de systèmes flexibles : ceux à base d'extensions *intra-noyau*, chargeant du code applicatif à l'intérieur du noyau *via* un protocole d'extension fixe et garantissant certaines propriétés de sécurité, comme *Spin* [BER 95] ou *Vino* [SEL 96b], et les noyaux *minimaux*, déléguant les services et abstractions du système aux applications sous la forme d'extensions *extra-noyau*, comme *Exokernel* [ENG 95] ou *Think* [FAS 01].

Il est clair que seule la deuxième solution offre un système réellement flexible. En effet, si l'utilisation d'un noyau minimal donne une totale liberté aux applications pour étendre et spécialiser « leur » système d'exploitation, il n'en est pas de même avec l'utilisation d'extensions « intra-noyau » qui repose sur une architecture « tout compris » et un protocole d'extension unique et figé, sous la forme d'un langage de programmation imposé ou de règles de sécurité (signatures, isolation. . .). Néanmoins, la flexibilité ne couvre toutefois que les aspects « système » (principalement la gestion des ressources) et n'offre donc aucun support ou cadre pour la flexibilité des aspects « langage » ou l'interopérabilité.

De la même façon, les intergiciels ont évolué vers plus de flexibilité dynamique. Les plus rigides ont été révisé afin d'y inclure quelques éléments de flexibilité de façon *ad hoc* et limitée, comme le POA[2] et les intercepteurs pour *CORBA* [OMG99]. Des intergiciels plus récent offre une flexibilité plus ou moins dynamique, en permettant la construction dynamique d'un ORB spécifique à l'application, comme *Jonathan* [DUM 98], ou en utilisant la réification de certains éléments de l'ORB afin de permettre à l'application de les modifier, comme dans le projet *OpenORB* [BLA 99] ou le système *2K* [KON 98, ROM 99].

Pour faire face aux domaines émergent autour des environnements embarqués des environnements dédiés ont été proposés, tel que *MultOS* [Mao], *μClinux* [UCL] ou la *KVM* [Jav] de *SUN*. Si ces environnements sont plus ou moins bien adaptés à certains domaines d'applications, il s'agit toujours de solutions *ad hoc*, rigides, et faiblement interopérables.

Le projet *XVM* [HAR 99] propose l'architecture d'une machine virtuelle monolithique extensible, reprenant ainsi le principe des systèmes flexibles à base d'extensions *intra-noyau*. Une application peut alors redéfinir le comportement des primitives et mécanismes de base (tel que la compilation dynamique ou les opérateurs arithmétiques). Mais là encore, afin de pouvoir garantir l'intégrité de la machine virtuelle,

2. Portable Object Adapter.

certains mécanismes demeurent figés, de même que l'architecture globale de la machine virtuelle.

L'utilisation de langage dédiés pour la spécialisation d'environnement d'exécution s'est particulièrement développée ces dernières années. Alors que l'approche dite réflexive a pour principe d'offrir aux applications un cadre général pour exprimer leur sémantique et modifier certains aspects de l'environnement d'exécution, l'approche *DSL*[3] propose d'utiliser des langages restreints à un domaine d'application précis, permettant ainsi de manipuler directement les abstractions de ce domaine au moyen de constructions appropriées. Cette approche a été appliqué à de nombreux domaines, des pilotes de périphériques (*Devil* [MER 00], *GAL* [THI 97]), aux réseaux actifs (*Plan-P* [THI 99]), en passant par les ordonnanceurs (*Bossa* [LAW 02]) et les caches web (*WebCal* [GUL 01], *CacheL* [BAR 99]).

Le principal inconvénient de cette approche, mis à part le coût de mise en œuvre induit par la capture de l'expertise du domaine, est le manque de dynamicité. En effet, l'objectif est ici d'utiliser la connaissance de la sémantique du domaine applicatif pour spécialiser l'environnement d'exécution de l'application et, le cas échéant, pour vérifier des propriétés (terminaison, invariant, etc.). Une fois le programme compilé (et exécuté) plus rien ne peut être fait. Il est notamment impossible de reconfigurer le programme pour réagir à un événement ou introduire un nouveau protocole si cela n'avait pas été prévu dans le programme : le *DSL* et son architecture logicielle ne sont pas flexible. Cette approche prenant le parti de proposer des solutions *ad hoc* adaptées à chaque domaine applicatif, il est concevable que la flexibilité ne soit pas une priorité[4] : l'environnement d'exécution sera nécessairement *parfaitement* adapté aux besoins et contraintes des applications. Alors que les environnements d'exécution flexibles et réflexifs propose de laisser le développeur d'application injecter la sémantique du domaine d'application dans un environnement général, l'approche *DSL* revient à laisser les développeurs d'environnement d'exécution capturer cette sémantique afin de proposer un environnement spécialisé et un langage dédié. Comme nous le verrons dans la section suivante, notre approche est intermédiaire.

Pris séparément, les différents constituants potentiels de l'environnement d'exécution n'offre donc qu'une flexibilité partielle : les systèmes d'exploitation sont de trop bas niveau pour *manipuler* la sémantique applicative et les aspects langage, l'implémentation des langages (compilateur, machine virtuelle) est de trop haut niveau pour *manipuler* la gestion des ressources. Le découplage artificiel [HOW 99] entre langage et système conduit à une fragmentation de l'environnement d'exécution en couches *indépendantes* et partiellement flexibles. Nous proposons donc de réconcilier système et langage sous la forme d'un méta-environnement d'exécution permettant la *manipulation* des définitions de l'ensemble des aspects constitutifs de l'environnement d'exécution.

3. Domain Specific Languages.
4. Excepté pour un éventuel *DSL systèmes flexibles*…

3. Machine virtuelle virtuelle

Les applications tendent de plus en plus à reposer sur l'interaction de nombreux composants hétérogènes et souvent distribués. Pour faire face à cela, un élément de solution a été apporté par les machines virtuelles : l'utilisation d'une représentation intermédiaire du code (ou *bytecode*) offre la portabilité et facilite donc l'interopérabilité et la mobilité. Néanmoins, elles demeurent plus ou moins dédiées à certains domaines applicatifs. L'émergence continuelle de nouveaux domaines d'application entraîne donc une prolifération de solutions *ad hoc*. La machine virtuelle *Java* de SUN, par exemple, est adaptée à des domaines applicatifs ne nécessitant pas de qualité de service et n'ayant pas des ressources limitées (particulièrement la mémoire). L'émergence de l'informatique *nomade* à entraîné l'apparition de nouvelles versions de *Java*, chacune adaptée à des contraintes précises d'architectures, comme *KVM* [Jav] pour les téléphones mobiles et *JavaCard* [CHE 00] pour les cartes à puces, ou à des besoins logiciel, comme *RT Java* [BOL 00] pour le temps réel.

Les machines virtuelles étant donc encore trop rigides, il est nécessaire de les virtualiser sous la forme d'une machine virtuelle virtuelle qu'il suffit de spécialiser dynamiquement pour instancier une machine virtuelle spécifique. Ainsi, au lieu de redévelopper une nouvelle machine virtuelle dédiée pour chaque nouveau domaine applicatif, il suffit de charger dynamiquement un script de spécification qui spécialise et étend la machine virtuelle virtuelle afin d'obtenir la sémantique adéquate.

L'isolation inhérente à l'utilisation de machines virtuelles conduit à une utilisation inefficace des ressources ainsi qu'à une faible interopérabilité. En effet, il est fréquent de trouver sur une machine autant de machine virtuelle s'exécutant que d'applications : *Java*, incorporé dans le navigateur internet, l'éditeur *Emacs*, *Ghostscript* dans les visualiseurs *PostScript* et *Portable Document Format* (*PDF*), *Tcl/Tk* dans les interfaces graphiques, etc. La factorisation de nombreux mécanismes internes[5], communs à la plupart des machines virtuelles, doit permettre une meilleure utilisation des ressources, aussi bien logicielles que matérielles : (i) de nombreux composants redondant sont factorisés (ce qui résulte en une consommation plus faible des ressources) ; (ii) les ressources ne sont plus réparties entre différents processus indépendants et isolés, mais contrôlées par un unique environnement, suivant les sémantiques applicatives.

De plus, cette factorisation entraîne la présence d'un substrat linguistique commun, au plus bas niveau, qui va permettre une interopérabilité complète : aussi bien au niveau applicatif qu'au niveau environnemental.

L'objectif de cette architecture est donc d'être à la fois flexible, dynamique et interopérable sans pour autant sacrifier les performances. Pour ce faire, deux principes sont proposés :

5. Comme le processeur virtuel ou les techniques d'optimisation.

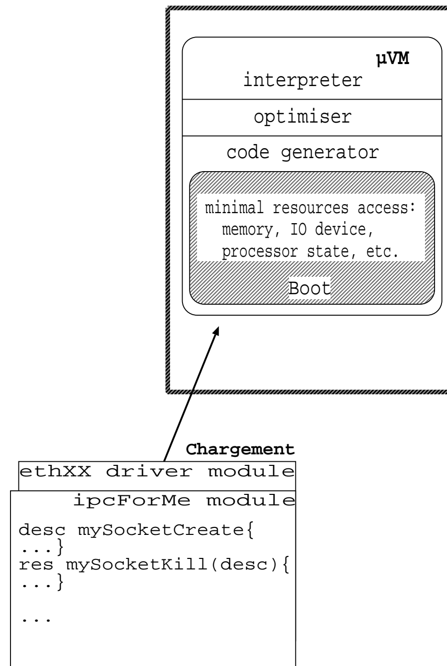

Figure 1. *L'environnement minimal*

1) minimalité de l'environnement d'exécution ;

2) couplage système/langage au cœur de l'architecture.

Le principe de minimalité se justifie par l'objectif de flexibilité. En effet, pour qu'un environnement d'exécution soit complètement flexible, il ne doit imposer aucune abstraction ou modèle prédéfini. Ainsi, une machine virtuelle virtuelle doit proposer un environnement d'exécution aussi minimal que générique.

La flexibilité dans les environnements d'exécutions se concentre, en générale, sur l'adaptation des aspects fonctionnels pour correspondre aux besoins ou contraintes spécifiques d'une application. Néanmoins, il peut également être nécessaire d'adapter les aspects langage à une sémantique applicative particulière, et cela ne peut être fait que si langage et système sont couplés au cœur de l'environnement d'exécution.

La figure 1 représente une *MVV* lors de son installation/initialisation : il s'agit d'un environnement d'exécution minimal[6], ne définissant aucun modèle de gestion de ressources ni aucune abstraction. Cet environnement de base est constitué d'une interface d'accès au matériel (ou, le cas échéant, aux couches basses d'un système

6. Il s'agit en fait d'un environnement d'exécution pour environnement d'exécution.

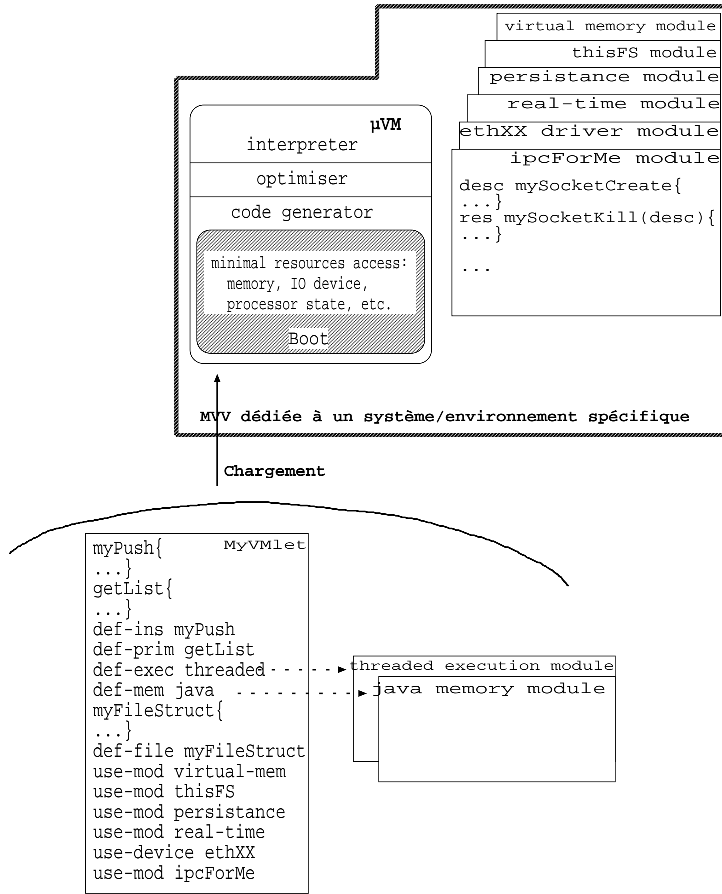

Figure 2. *Construction d'un environnement d'exécution dédié*

d'exploitation sous-jacent) chargée de réifier les ressources matérielles sans ajout de sémantique, ainsi que d'une machine virtuelle générique, constituée, initialement, uniquement d'un compilateur dynamique réflexif, permettant la mise en place dynamique de chaîne de compilation (du lexer/parser jusqu'au générateur de code natif).

Cet environnement minimal est ensuite étendu et spécialisé pour un domaine applicatif particulier (dans l'esprit des *Domain Specific Languages*). Pour cela, une spécification active, appelée *MVlet*, décrivant un environnement d'exécution est chargée, comme illustré sur la figure 2. Elle définit complètement le comportement de cet environnement et agit directement sur la machine virtuelle générique pour la *transformer* en la machine virtuelle cible. En effet, il ne s'agit pas d'empiler des couches d'indirection et de virtualisation : la machine virtuelle générique devient une machine virtuelle dédiée et offre la même sémantique aux applications avec des performances similaires

à une machine virtuelle dédiée native, sans pour autant perdre son interface de machine virtuelle générique. Si la machine virtuelle virtuelle « vide » peut être vue comme une machine virtuelle dédiée à la mise en place de machines virtuelles spécifiques, *via* les *MVlets*, le chargement d'une *MVlet* la transforme en une machine virtuelle dédiée à la fois au domaine applicatif décrit dans la *MVlet* et à la mise en place de machines virtuelles spécifiques, *via* de nouvelles *MVlets*.

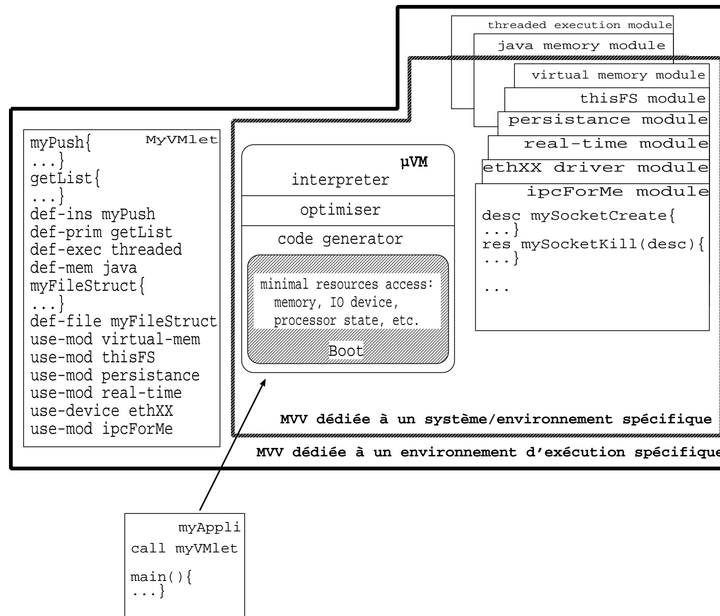

Figure 3. *Chargement d'une application*

En procédant à des reconfigurations au niveau de la machine virtuelle générique (*via* le chargement de *MVlets*), il est possible d'adapter aussi bien les aspects fonctionnels que sémantiques d'un environnement d'exécution. C'est également au niveau de cette machine virtuelle générique que se trouve le substrat linguistique commun qui permet l'interopérabilité entre applications ainsi qu'entre environnements d'exécution en offrant une représentation intermédiaire commune du code et des données.

Une application est typée par un identifiant de *MVlet* permettant d'identifier le chargeur à utiliser et l'environnement d'exécution associé. Elle est donc chargée et exécutée par un environnement d'exécution entièrement spécialisé pour son domaine applicatif, incluant sa sémantique, ses besoins et éventuelles contraintes, comme illustré sur la figure 3.

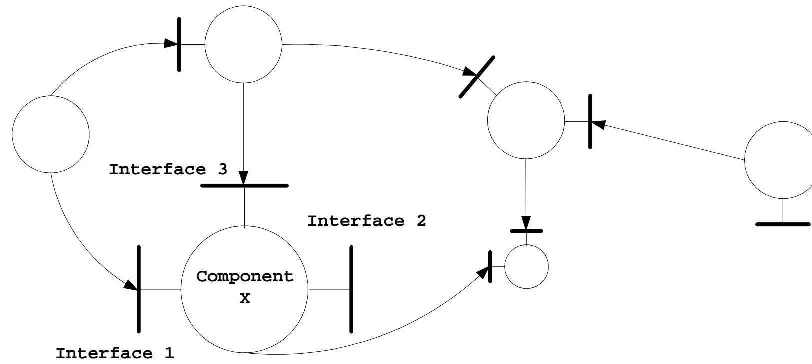

Figure 4. *Composants et interfaces*

4. Environnement d'exécution minimal

Afin de faire face à l'émergence continuelle de nouveaux domaines applicatifs, avec des besoins et contraintes spécifiques, nous proposons donc une plate-forme logicielle entièrement flexible pour la construction dynamique d'environnements d'exécutions adaptables.

Cette architecture est structurée sous la forme d'un ensemble de composants et d'interfaces. Le modèle de composants est basé sur le modèle de référence *ODP* [odp98]. Chaque composant exporte une ou plusieurs interface(s) qui sont autant de points d'accès vers le composant. La figure 4 représente un composant (*X*) exportant trois interfaces (*Interface1*, *Interface2* et *Interface3*) à travers lesquelles il peut être accédé.

La possibilité d'exporter plusieurs interfaces permet à un composant de séparer ses fonctionnalités ou de matérialiser différents aspects de sa sémantique, comme différents niveaux de sécurité ou de qualité de service.

Les interactions entre composants reposent sur la notion de *liaisons*. Ces dernières son réifiées grâce à l'introduction d'*usines à liaisons*. Dès lors les canaux de communication entre composants deviennent flexible et peuvent être dynamiquement adaptés, par exemple en invocation distante après la migration d'un composant ou en invocation distante de groupe pour les composants répliqués dynamiquement.

L'environnement d'exécution minimal, constituant le cœur de la plate-forme et illustré sur la figure 5, est composé de deux parties symbolisant ainsi le couplage système/langage. *THINK*[7] [FAS 01] est un ensemble de composants développés à *France Telecom R&D* et chargés de réifier les composants matériels, comme la *MMU* ou les interruptions. Aucune abstraction n'est définie à ce niveau. Cette partie système peut

7. Pour *THINK Is Not a Kernel*.

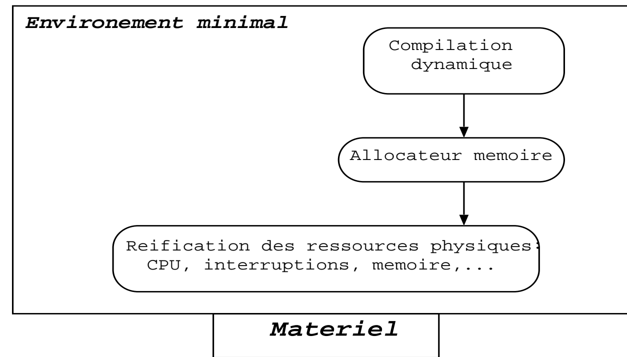

Figure 5. *Environnement d'exécution minimal*

également être assurée par un système d'exploitation traditionnel comme Linux. La partie langage, appelée *YNVM*[8], est un générateur dynamique de code.

4.1. *Une interface vers le matériel :* THINK

Comme son nom l'indique, *THINK* n'est pas un noyau. En effet, les noyaux traditionnels peuvent être vus comme des machines virtuelles définissant un ensemble d'abstractions logicielles et des services associés, tels que les systèmes de fichiers, les processus ou la mémoire. A l'inverse, *THINK* est une librairie d'abstractions matérielles dont l'objectif est l'amorçage de l'environnement et la réification des ressources matérielles, telles que la mémoire physique, la *MMU* ou les périphériques. Aucune abstraction logique n'est définie en sus. Il s'agit donc d'un ensemble de composants, chacun représentant un élément physique et exportant une ou plusieurs interface(s) correspondants aux fonctionnalités matérielles, sans y ajouter la moindre sémantique. *THINK* fournit donc un accès direct aux ressources physiques sans aucune gestion ou aucun contrôle : les composants sont *politiquement neutres*.

En complément des composants réifiant le matériel, *THINK* propose également des pilotes de périphériques, ainsi qu'un *Trader*, des usines à liaisons, ainsi qu'une librairie, appelée *Kortex*, de composants de plus haut niveau (ordonnanceurs, protocoles réseaux, etc.) pour constituer un système flexible. Néanmoins, ces composants peuvent également être définis au-dessus de la *YNVM*, au niveau applicatif. C'est pourquoi, dans le cadre du projet *machine virtuelle virtuelle*, nous n'utilisons *THINK* que comme une couche d'amorçage et d'interfaçage avec le matériel : afin de garantir la minimalité de l'environnement d'exécution minimal, nous repoussons au niveau applicatif tout ce qui peut l'être. Cette partie de l'environnement minimal est en court de

8. Pour *YNVM is Not a Virtual Machine*.

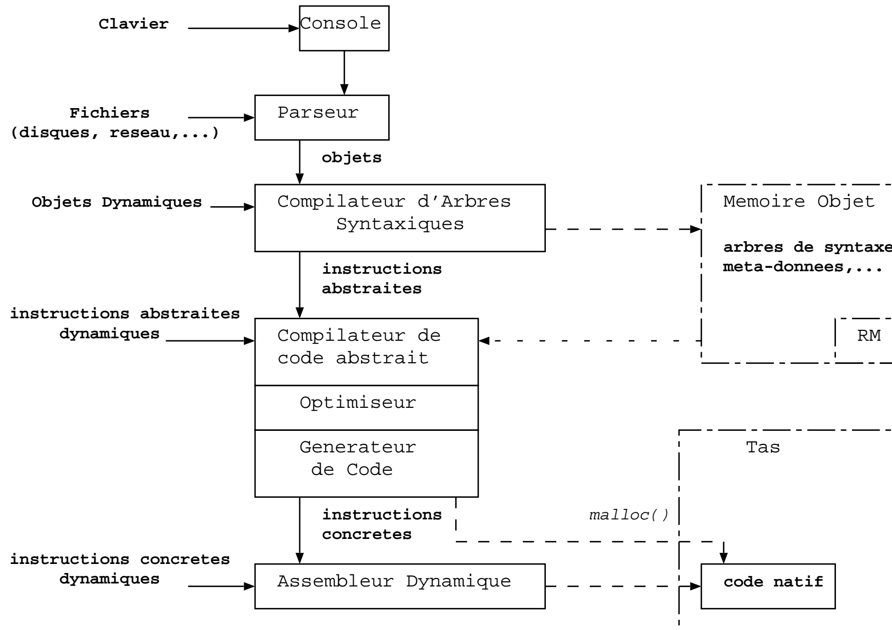

Figure 6. *Structure de la* YNVM

portage sur *Linux*, *via* un module noyau permettant à la *YNVM* d'accèder et d'agir sur l'ensemble des symboles du noyau.

4.2. *Un compilateur dynamique :* YNVM

La *YNVM* correspond à la machine virtuelle générique de l'architecture *MVV*. Son rôle est donc essentiellement la construction dynamique de machines virtuelles et d'environnements d'exécution spécialisés. C'est également à ce niveau que les reconfigurations dynamiques se font. Il s'agit là encore d'un ensemble de composants constituant une chaîne de compilation dynamique entièrement ouverte et dynamiquement flexible, représentée sur la figure 6.

Les composants internes reposent sur une mémoire objet interne avec ramasse-miettes (ou *RM*) utilisée pour le stockage des métadonnées. Le parseur convertit un texte, reçu depuis le disque, le réseau ou une interface de type console, en un arbre de syntaxe abstraite (AST), stocké dans la mémoire objet de la *YNVM*. Le compilateur d'arbres convertit ensuite ces objets en instructions pour une machine abstraite à pile dont le modèle d'exécution et la sémantique sont similaires au *C*. Il maintient également à jour des métadonnées sur l'état du code compilé et applique les éventuelles règles de transformations de code fournies par des *MVlets*. Les métadonnées sont

```
(module dev.console)
(component.define-state fb         ;;composant framebuffer
                        x y        ;;position du curseur
                        cols rows) ;;dimensions
;; définition d'une interface pour le composant
(component.export-methods (putc    char)
                          (putcs   string)
                          (putxycs int int string)))
;; création d'une instance d'interface
(define %default (:component.interface myPutc myPutcs myPutxycs))
(define new (lambda(fb nbcols nbrows)
               (component.new %default fb 0 0 nbcols nbrows)))
;; utilisation du composant
(module global)
(define fb-ptr (myBindingFactory bind
                                  (myTrader lookup ``framebuffer'')))
(define myConsole (dev.console.new fb-ptr 80 25))
(myConsole putcs ``hello world !\n'')
```

Figure 7. *Création d'un composant* console

organisées en espaces de nommages hiérarchiques appelés *modules*. Enfin, le générateur de code traduit la représentation intermédiaire, à base d'instructions abstraites, en instructions concrètes pour la plate-forme locale. Le découplage en deux niveaux d'abstraction (AST et machine à pile) permet deux phases d'optimisation spécifiques et indépendantes de la machine, *via* deux compilations dynamiques : la compilation d'arbres syntaxiques en instructions abstraites puis la compilation de ces instructions abstraites en instructions concrètes. La génération des instructions concrètes se fait non plus dans la mémoire interne, mais dans la mémoire applicative (en général le tas) par l'intermédiaire d'un assembleur dynamique propre à la plate-forme cible. Si chaque expression est traditionnellement lue, compilée puis exécutée, un mécanisme de macro-expansion, appelé *syntaxe*, permet de définir du code exécuté pendant la phase de compilation dynamique. Cela permet notamment la mise en œuvre de mécanismes de réécriture/transformation de code.

Par défaut, le langage d'entrée de la chaîne de compilation est une représentation textuelle d'arbre de syntaxe abstraite de type *Lisp*. Les composants internes de la *YNVM* étant accessible au niveau applicatif, il est possible de redéfinir dynamiquement tout ou partie de la chaîne de compilation dynamique afin de prendre d'autres langages en entrée ou de changer la sémantique d'un langage.

La figure 7 illustre la définition[9] et l'utilisation d'un composant de type *console* depuis un script utilisant le langage par défaut. Un espace de nommage (*dev.module*) est défini pour contenir la définition du composant. Cette définition comprend l'état

9. Pour des raisons de simplification, les fonctions sont supposées être définies ailleurs.

(*i.e.* les attributs), déclaré par *define-state*, une interface (*%default*) et un constructeur (*new*). L'utilisation de ce composant dépendant d'un composant *framebuffer*, une liaison est construite *via* un composant de courtage (*myTrader*) et une usine à liaisons (*myBindingFactory*) avant d'instancier le composant[10].

5. La flexibilité sur machine nue

Afin d'illustrer la flexibilité offerte par l'environnement minimal constitué de la *YNVM* et de *THINK*, nous avons développé deux applications utilisant la flexibilité au plus bas niveau : un mécanisme de démarrage par le réseau et un mini-environnement d'exécution réparti à base de composants mobiles.

Le mécanisme de démarrage se présente comme une mini-pile réseau permettant uniquement le chargement dynamique de script(s) par le réseau. Une fois chargé, le script reconfigure son environnement : par exemple, la pile de démultiplexage réseau est adaptée pour ajouter un service de routage ou un protocole applicatif. L'image « bootable » de base représente environ 120 Ko (correspondant à *THINK*, la *YNVM* et le code de ce mécanisme). A partir de cet environnement basique, un environnement d'exécution complet, ainsi que des applications, sont dynamiquement construits en ne chargeant incrémentalement que les extensions requises. De plus, la compilation dynamique des diverses extensions lors de leur chargement autorise une optimisation poussée du code.

La seconde application est un intergiciel flexible permettant la construction de services répartis à base de composants coopérants. Grâce à la compilation dynamique, les liaisons entre composants distants sont générées dynamiquement au-dessus d'un protocole donné (ou directement au-dessus du pilote de la carte réseau sur un intranet). Lors de la migration d'un composant, il suffit alors de regénérer dynamiquement ses liaisons.

La *YNVM* nous permet de conserver les propriétés des machines virtuelles (code mobile, vérification de code, etc.) tout en offrant des performances comparables à des programmes *C* ainsi qu'une flexibilité dynamique complète. Cette flexibilité permet aux composants d'être mobiles sans être liés à des mécanismes génériques et peu efficaces : le code d'interaction entre deux composants est toujours spécialisé. Les premières mesures de performances, effectuées à l'aide d'un G3 290 Mhz et d'un G3 366 Mhz reliés par un Ethernet 100 Mbits, pour l'invocation de méthode distante montre un gain d'un facteur trois à sept par rapport à divers ORB *CORBA*. Le temps nécessaire à un appel distant synchrone d'une fonction simple[11] est de 143 μs, en moyenne sur la *YNVM* contre des moyennes entre 520 μs et 1250 μs suivant les ORB *via* Corba. De même, la migration d'un composant s'est révélée nettement plus per-

10. Le composant *console* aurait pu également prendre en charge lui-même cette étape de configuration à l'intérieur de son constructeur.
11. Dans ce cas, il s'agit d'une fonction réalisant la somme de deux entiers passés en paramètres.

formantes qu'en *Java* : 176 μs contre plus de 20 ms (pour un objet fonctionnellement équivalent).

Alors qu'un ORB Corba ou une machine virtuelle Java repose sur l'utilisation de mécanismes génériques pour la sérialisation et la communication réseau, notamment en utilisant des protocoles de haut niveau, comme TCP, notre approche permet de spécialiser dynamiquement la communication entre deux composants, limitant ainsi le surcoût. Par exemple, pour la migration de composant, le temps de transit du code est environ 120 μs, d'où un surcoût de sérialisation et d'invocation de méthode inférieur à 60 μs, contre plus de 19 ms de surcoût dans le cas de Java. La flexibilité de la *YNVM* et la minimalité offerte par *THINK* nous permettent d'encapsuler la migration de code, sur un réseau local, dans un protocole situé juste au-dessus du pilote de la carte réseau, éliminant ainsi pratiquement tout surcoût logiciel dans la communication. Le code est donc sérialisé, puis transmis directement dans un paquet Ethernet vers le destinataire. A l'arrivée, le paquet est transmis par le pilote au composant chargé du démultiplexage des paquets Ethernet, qui le déserialise et le transmet à la *YNVM* pour compilation dynamique et évaluation. Le résultat est alors sérialisé et transmis vers la source, *via* le pilote réseau.

De même, dans le cas des appels de méthode à distance, alors que les mécanismes traditionnels reposent sur une compilation statique des souches et des squelettes, la génération dynamique de souche pour les composants, *via* la *YNVM* nous permet de spécialiser la communication : lorsqu'un composant A se lie à un composant distant B, une souche est dynamiquement générée pour assurer la communication. Ainsi, cette souche peut contenir le mapping entre l'interface du composant et les adresses physique des méthodes correspondantes ou laisser cette traduction s'opérer côté serveur. La souche peut ainsi ne transmettre qu'une adresse de fonction et une adresse de composant, directement *via* le pilote réseau. Le type du résultat étant connu de la souche, les besoins en sérialisation sont faible[12].

6. La *YNVM* : support d'exécution flexible

L'utilisation de la *YNVM* comme support d'exécution au-dessus de *Linux* nous a permis d'illustrer l'apport de la flexibilité dynamique dans le contexte de différents domaines d'applications : les résaux actifs, les caches web et les documents actifs.

L'émergence de nouveaux domaines applicatifs à fait naître un besoin de déploiement dynamique de nouveaux services et de nouveaux protocoles. Les réseaux actifs[13] constituent une réponse générique à ce problème. Néanmoins, chaque protocole actif reste figé. L'architecture DARPA est structurée en trois couches : le *NodeOS*, interface encapsulant le système d'exploitation, les environnements d'exécution, des machines

12. Chaque méthode peut avoir sa propre représentation « compacte » du résultat, du moment que la souche et le squelette de la méthode la partage.
13. Un réseau actif propose de transférer non plus uniquement des données, mais également du code : le protocole actif permet donc le déploiement dynamique de services.

virtuelles (le plus souvent Java) exécutant chacun un protocole actif, et enfin les applications actives s'exécutant au-dessus d'un protocole actif (et de la machine virtuelle sous-jacente). Dans *PLAN* [HIC 98], chaque paquet contient, en plus des données, le code de traitement associé. A l'inverse, *ANTS* [TEN 96] repose sur une phase de déploiement préalable du code et utilise ensuite des identifiants dans les paquets pour les associer avec le code de traitement. Ainsi, chaque protocole actif représente un point figé sur le continuum des possibilités, avec ses propres contraintes et hypothèses, généralement fortement liées à la machine virtuelle sous-jacente. Par exemple, les choix en matière de sécurité dans *ANTS* le rendent inadapté au déploiement de firewall. De plus, les services doivent être exprimés à partir de l'API « restreinte » *ANTS*, le code d'un service doit être inférieur a 16 Ko, chaque routine de « forward » doit être suffisamment rapide pour terminer sous peine d'être interrompue et abandonnée, un service ne doit pas échouer dans le cas où une partie du réseau n'est pas active, etc.

Un unique protocole actif ne pouvant convenir à tous les besoins applicatifs, nous proposons la mise en œuvre d'une généralisation de ces protocoles permettant la spécialisation dynamique de protocole (lors du déploiement d'un nouveau service, par exemple), ainsi que l'interopérabilité entre protocole.

Tout en offrant le même degré de portabilité qu'une machine virtuelle classique, *via* l'utilisation de la représentation intermédiaire du code, la *YNVM* offre, de par sa flexibilité dynamique, la possibilité de (re-)définir dynamiquement les protocoles actifs qu'elle héberge, aussi bien concernant le contrôle de ressource que la vérification du code actif ou le mode de déploiement[14].

Nous avons donc développé une *MVlet* décrivant *PLAN* et une autre décrivant *ANTS*. De cette façon, en chargeant ces *MVlet* la *YNVM* devient un routeur actif supportant à la fois *ANTS* et *PLAN*. Un protocole actif supplémentaire est également utilisé pour déployer du code *YNVM*, servant notamment à administrer le routeur. Ainsi, il est possible de modifier dynamiquement le comportement d'un protocole pour un flux ou un service donné ou de déployer de nouveaux protocoles, sous la forme de paquets actifs : la spécification du protocole (une *MVlet*) représentant les données et le code effectuant le chargement, la partie active.

Les documents actifs, ou documents flexibles, sont une approche systématique pour construire des documents qui interagissent avec leur environnement, par analogie avec les réseaux actifs[15]. Un document actif est donc composé d'une partie passive, les données, qui correspond au contenu « classique » du document (texte, image, son, etc.) et une partie active, du code exécutable, chargé du traitement des données. Le code de la partie active permet de modifier certains paramètres du document, comme son format ou sa qualité, ainsi que d'agir sur l'environnement du document, en modifiant les politiques internes par exemple.

14. Intégré, comme *PLAN*, discret, comme *ANTS*, ou une combinaison des deux.
15. D'où l'appellation de « documents actifs ».

Tout comme les réseaux actifs, il est essentiel pour l'environnement chargé de l'exécution de la partie active d'être dynamiquement flexible, afin que les documents puissent l'adapter à leur sémantique.

La *YNVM* constitue un support d'exécution naturel pour la partie active des documents. Les lecteurs gérant la partie passive, telle la machine virtuelle ghostscript, sont utilisés comme des « plug-ins » par la partie active, *via* la *YNVM*.

Plusieurs exemples de documents actifs ont été implantés, en particulier un document actif de type PostScript qui encapsule l'utilisation de ghostscript. Ce document actif permet d'adjoindre à un document PostScript des fichiers (sons, images, etc.) qui sont évalués par le programme adéquat lors du passage de certaines pages et d'adjoindre des notes au format texte qui sont insérées à l'exécution dans le code PostScript. La partie active du document actif est utilisable avec plusieurs types de fichiers joints et s'enrichit à la volée si le format du fichier joint est inconnu.

Cette expérience a montrer que le couplage langage/système à la base de *YNVM* permettait d'offrir un support simple et intuitif pour enrichir l'environnement d'exécution des documents actifs sans perdre en performance.

Victime de son propre succès, le web s'est vu peuplé de systèmes de cache destinés à réduire la charge des serveurs, ainsi que la latence observée par les utilisateurs. Les performances de ces systèmes reposent sur des paramètres hautement dynamiques, tel que l'état du réseau ou le comportement des utilisateurs, conditionnant un tres grand nombre de paramètre de configuration du système, comme la politique d'éviction, la taille, la politique de coopération, la répartition des caches, etc.

Cependant, en raison des fluctuations, généralement aussi soudaine qu'imprévisibles, du trafic, une configuration donnée ne reste pas efficace au cours du temps : pour demeurer efficace, le système de cache et sa configuration devront évoluer et s'adapter pour continuer à correspondre aux conditions réelles d'utilisation.

Malheureusement, la flexibilité de ces systèmes est, le plus souvent, limitée à une paramètrisation statique, *i.e.* la possibilité de choisir parmi certaines valeurs prédéfinies concernant certains paramètres eux aussi prédéfinis, comme trois politiques d'éviction de documents classiques, la taille du cache ou des règles de sécurité [SC].

Certains travaux comme *WebCal* [GUL 01] ou *CacheL* [BAR 99] utilisent l'approche *DSL* pour permettre la spécialisation de cache. Néanmoins ils ne permettent pas la reconfiguration dynamique et donc la prise en compte du caractère dynamique et peu prévisible des fluctuations de certains paramètres comme le célèbre effet *Hot-Spot-Of-The-Week* [SEL 96a].

Une autre approche consiste à structurer le proxy cache sous la forme d'un ensemble d'interfaces et de composants et à utiliser un « design pattern » [GAM 95] encaspulant la notion de stratégie, comme dans [AUB 99]. Ainsi, il devient possible de remplacer certains composants prédéfinis par d'autres composants, respectant la même interface. Il s'agit d'un intermédiaire entre la configuration statique et la flexi-

bilité dynamique complète : si le cache peut réagir dynamiquement à certains événements, il ne peut que réagir à ceux qui ont été prévus lors de sa conception et suivant une stratégie d'adaptation également figée à la conception.

L'utilisation de la *YNVM* comme support d'exécution nous permet de franchir cette limitation et d'assurer une flexibilité complète et réellement dynamique, *i.e.* où aucune partie du cache et de son environnement d'exécution n'est définitivement figée.

Nous avons donc développé une *MVlet* « cache web », appelée *C/NN*[16], qui transforme l'environnement d'exécution en un cache web tout en héritant de la flexibilité dynamique de la *YNVM* et qui peut donc être entièrement reconfiguré *à chaud*. Des tests ont permis de vérifier que la flexibilité dynamique et les performances ne s'excluaient pas forcément : nous avons évalué notre cache sur la base de traces du trafic web de l'*INRIA*, puis nous avons évalué le coût de la reconfiguration dynamique. Pour ce faire nous avons utilisé, pour héberger *Squid* et la *YNVM*, un G3 366 MHz sous LinuxPPC, connecté au client « rejouant » la trace par un réseau Ethernet 100 Mbits.

Le temps de réponse moyen de notre cache est de 0,83 sec., contre un peu plus de 1 sec. pour *Squid* sur les mêmes traces. Le code compilé dynamiquement et exécuté par la *YNVM* est donc comparable, en performance, avec un code C, fonctionnellement équivalent, compilé statiquement. Le temps de traitement d'un *hit*, correspondant à la lecture de la requête, la recherche dans le cache et l'envoi de la réponse, est de 130 μs. Dans le cas d'un *miss*, qui comprend en sus l'envoi d'une requête vers le serveur web ainsi que la réception et mise en cache de la réponse, le temps de traitement est de 300 μs.

Ces temps de traitement permettent d'évaluer l'impact de la flexibilité dynamique sur l'activité première du cache.

Remplacer la politique d'éviction par une politique déjà définie coûte environ 50 μs. Le remplacement en lui-même, correspond à une mise à jour d'une liaison de symbole, soit une affectation. La transition entre deux politiques étant en soit une politique, c'est à son exécution que correspondent les 50 μs[17].

La définition dynamique d'une nouvelle politique, comme celle proposée dans [COL 00], est d'environ 400 μs, soit approximativement le temps de traitement de deux requêtes[18]. Cela comprend la compilation et l'exécution d'un script définissant la nouvelle politique (et donc sa compilation en code natif) ainsi que le remplacement de l'ancienne politique vers la nouvelle (évalué précédemment à 50 μs).

Avoir un cache web dynamiquement reconfigurable n'implique donc pas d'avoir un cache web moins performant. De plus, l'impact des mécanismes supportant cette flexibilité sur l'activité du cache semble suffisamment faible pour autoriser des mises à jour fréquentes, sans dégradations notables.

16. Cache with No Name.
17. Il s'agit dans ce cas de la mise à jour d'un petit nombre de métadonnées sur une partie des documents.
18. En considérant un taux de *hit* très optimiste de 75%…

L'évolution naturelle de ce cache fut l'automatisation de son administration. En effet, les reconfigurations « manuelles » du cache demandent une certaine connaissance de son architecture et sont, de fait, des sources potentielles d'erreur. De plus, la pertinence d'une adaptation décroissant avec le temps, il est important que l'écart entre le moment où la décision de reconfigurer le cache est prise et celui où la reconfiguration devient effective soit faible. De ce fait, seule l'automatisation des reconfigurations permet d'assurer une bonne réactivité.

Pour atteindre ce niveau d'autonomie, nous avons couplé *C/NN* avec une plateforme d'observation à composants : *Pandora* [PAT 00]. *Pandora* se présente comme un ensemble de composants C++ permettant la création dynamique de pile d'observation du réseau. La compatilité de la *YNVM* avec l'*ABI* de la machine, nous permet d'exécuter directement *C/NN* et *Pandora* au-dessus de la *YNVM*, ainsi que de les faire interagir[19].

L'administration du cache se fait alors à travers un ensemble de politiques de reconfiguration, chacune composée d'une sonde d'observation dédiée à un événement[20] et d'une fonction associée, implantant la reconfiguration. Grâce à la réflexion complète de l'environnement d'exécution, les politiques de reconfigurations sont, en tant que politiques, elles-mêmes reconfigurables. Ainsi, une politique de reconfiguration peut également servir à activer/désactiver d'autres politiques. Par exemple, la gestion d'une liste dynamique d'exclusion[21] représente une politique de reconfiguration, dont la sonde surveille les temps de réponse des serveurs et dont la fonction gère l'ajout et le retrait des serveurs dans la liste. Cette politique est « pilotée » par une autre politique, dont la sonde surveille la charge du cache et dont la fonction active la politique de liste d'exclusion, reconfigure la gestion du cache pour utiliser cette liste et se reconfigure elle-même en vue de la désactivation de la politique de liste d'exclusion, lorsque la charge sera redescendue sous un certain seuil.

Un autre exemple concerne la mise en place dynamique de qualité de service : lorsque le nombre de connexions ou la charge passe certains seuils, une stratégie de contrôle du trafic réseau et des connexions est activée. Plus généralement, cela permet d'éviter le surcoût généralement associé au contrôle de qualité, particulièrement avec des stratégies à base de réservation, lorsqu'un système n'est pas chargé.

Ces stratégies d'administration servent également à l'installation de nouveaux algorithmes, protocoles ou services dans le cache.

Contrairement à un environnement traditionnel, la *YNVM*, de par sa flexibilité dynamique, permet le déploiement incrémental de ces stratégies : au fur et à mesure des besoins, en réaction à un imprévu ou simplement à une innovation.

19. *Pandora* fait appel directement à des fonctions et variables de *C/NN* et inversement.
20. Une conjonction d'événements est considérée comme un (macro) événement.
21. Une liste de serveurs « proches » dont on ne cache plus les documents lorsque le cache est trop chargé.

7. *Pomv* : une bibliothèque pour machines virtuelles Java

Java est devenu un standard de fait dans le développement d'applications réparties. Java permet en effet de masquer l'hétérogénéité matériel et système des plates-formes grâce à l'utilisation d'un bytecode générique évalué dans une machine virtuelle Java.

Toutefois, les machines virtuelles Java sont monolithiques et peu ou pas adaptables. De plus, la machine virtuelle de référence de Sun [LIN 96] offre une faible API de réflexion, car limitée à celle du langage Java. Cet aspect monolithiques conduit à implanter des machines virtuelles dédiées en fonction des besoins matériels ou logiciels : KVM [Jav] et Java for SmartCard [CHE 00] sont dédiées à des matériels ayant peu de ressources, PJama [ATK 96] ou Gemstone [Gem] implantent une machine virtuelle à objets persistants, MetaXa [Mic 97] est une JVM réflexive, le Real-Time for Java Expert Group [BOL 00] définit l'API d'une machine virtuelle temps réel, etc.

Le développement de machines virtuelles spécialisées est coûteux en temps de développement alors que ces machines virtuelles Java dédiées restent proches les unes des autres. Ces plates-formes résolvent des problèmes spécifiques mais sont elles-mêmes figées : par exemple une machine virtuelle à objets persistants et temps réel demanderait le développement d'une nouvelle machine virtuelle alors que les deux aspects semblent orthogonaux.

De plus, l'aspect monolithique des machines virtuelles Java ne permet pas d'étendre dynamiquement le traitement des attributs[22]. Considérons la vérification de bytecode JVM (qui est un module important du modèle de sécurité Java [GON 99]). Elle est implantée suivant plusieurs techniques [LER 03], chacune d'entre elles reposant sur un algorithme différent, adapté à son domaine applicatif (stations de travail, systèmes embarqués, cartes à puce, etc.). Quelques-uns modifient l'application[23]) avant son chargement dans la JVM afin de la prétraiter et ainsi d'optimiser sa vérification. Il est donc nécessaire d'adapter aussi le chargeur en fonction du type de vérification utilisée. Cette solution est retenue dans les KVM. Un *prévérifieur* adjoint une preuve de type PCC [NEC 97] appelée `StackMap`, cet attribut de code Java ne peut être interprété que par les JVM qui implantent cette fonctionnalité (au niveau du chargeur, et de la vérification).

Un autre cas d'étude concerne le modification de la sémantique d'un bytecode par le biais d'un attribut Java. Nous avons redéfinit la sémantique du bytecode `new` pour allouer un objet dans la pile ou dans le tas tout en respectant la politique de sûreté des pointeurs de Java. Cette exemple s'appuie sur des techniques d'analyse d'échappement (*escape analysis*) [WHA 99, CHO 99, BLA 03]. Cette technique détermine, par analyse statique de code, la durée de vie des variables objets. Les objets qui peuvent être détruit à la sortie d'une méthode sont gérés par *allocation en pile* et non plus par

22. Un attribut est un nom suivis d'une structure de méta-données donnant des informations sur un champ, une méthode ou une classe.
23. Plus exactement, le fichier `.class` la contenant.

le ramasse-miettes. Cela permet d'améliorer les performances globales de la machine virtuelle avec la même sûreté de fonctionnement.

Nous nous proposons d'aborder le problème des machines virtuelles Java dédiées différemment en introduisant les applications actives. Une application active est constituée de deux entités : une application Java (la partie fonctionnelle de l'application active) et un script actif (la partie non fonctionnelle). Le script actif s'occupe d'instancier la machine virtuelle Java dédiée à l'application (figure 8). Cette architecture permet à l'application de spécifier les mécanismes systèmes (dans le sens OS et JVM) nécessaires à son bon fonctionnement. Cette approche est différente des approches méta-objet : les méta-objets ne s'exécutent pas au même niveau que la machine virtuelle alors que les scripts actifs peuvent la modifier en profondeur.

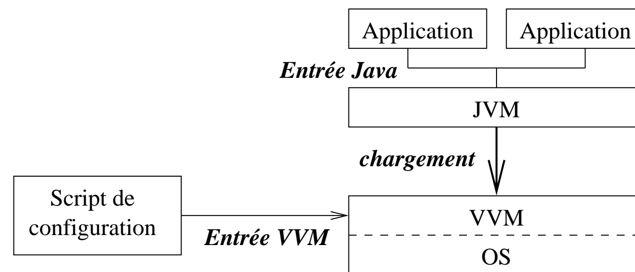

Figure 8. *Les applications actives*

Les applications actives Java sont un cas particulier de documents actifs : le document est l'application Java et la partie active, le script de configuration qui instancie une plateforme dédiée à l'application.

Pour construire des applications actives, nous disposons d'un certain nombre de module génériques regroupés dans une bibliothèque appelée *Pomv* pour plateforme ouverte pour machines virtuelles. Ces modules offrent une API clair, ce qui permet à une application active de remplacer globalement un module au chargement de la machine virtuelle Java. Les modules génériques offrent aussi une API d'adaptation, ce qui permet à l'application active de spécialiser le comportement des modules en fonction des ses besoins logiciels, que ce soit au démarrage de l'application ou en cours d'exécution de l'application.

Les modules génériques correspondent à la spécification de Sun [LIN 96] et sont développés au-dessus du prototype de machine virtuelle virtuelle *YNVM*, *Pomv* est donc une *MVlet*.

La *YNVM* offre les mécanismes d'adaptation nécessaire à l'adaptation de *Pomv* et dispose d'un compilateur JIT[24] (le VPU) utilisé intensivement dans *Pomv*. Les scripts actifs sont écrit dans le langage de la *YNVM*.

24. Just in Time.

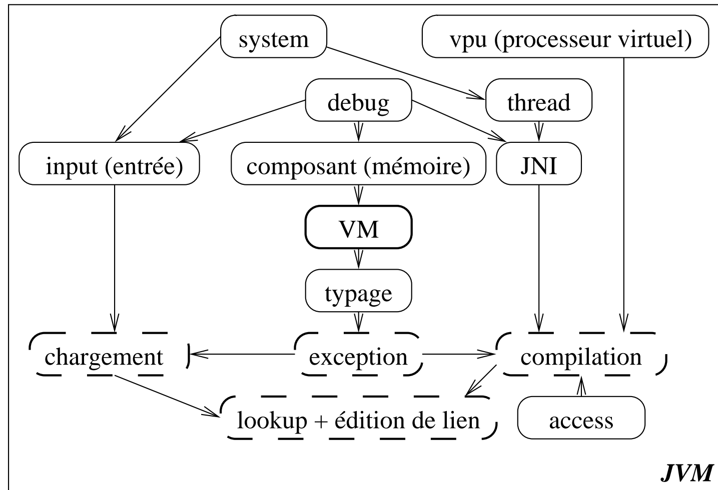

Figure 9. *Les modules de* Pomv

Les modules de *Pomv* suivent un arbre de dépendance représentée sur la figure 9.

7.1. *Module de nommage*

Les symboles Java sont nommés grâce aux symboles de la *YNVM* : les classes sont projetées vers des modules MVV, les méthodes et attributs vers des symboles. Par exemple, la méthode `Object *clone()` de la classe `Java.lang.Object` est projetée vers le symbole MVV `:Java.lang.Object.clone_sign` où `sign` est la signature la méthode. Cette technique de nommage permet à un script actif d'interagir directement avec les classes chargées dans la machine virtuelle Java.

7.2. *Module de définition de machine virtuelle (VM)*

Ce module permet de définir les structures de base nécessaires à l'exécution d'une machine virtuelle. Ce module est le pivot des autres modules et ne peut que difficilement être remplacé. On trouve dans ce module une structure représentant la machine virtuelle : le cycle de vie d'une classe, la gestion des erreurs et le compilateur de bytecode Java.

Seules les parties les plus intéressantes de la structure de machine virtuelle sont données ici :

– `Input *open(char *name)` : ouvre une description de classe, par défaut cette fonction essaye de trouver un fichier description dans le système de fichier ;

– int (*load)(VM *, Class *, Input *) : la fonction de chargement d'une classe, c'est grâce à cette fonction qu'on peut implanter un chargement incrémental de byte-codes depuis le réseau par exemple ;

– int (*check-error)(VM *src, Object *excp) : cette fonction permet de vérifier si une exception excp levée par la machine virtuelle src doit être attrapée en un point particulier, par défaut, cette fonction réalise l'algorithme d'attrapement Java ;

– void *(*compile)(Class *, Method *, int access) : cette fonction compile la méthode passée en argument et renvoie un pointeur vers celle-ci, on peut donc modifier totalement le compilateur grâce à cette fonction ;

– char reserved[32] : des octets réservés pour une application active particulière.

Chaque pointeur de fonction de ce module est un point d'entrée vers un autre module. Une application active qui modifie une de ces fonctions spécialise un des aspects de la machine virtuelle Java. Cette structure couplé avec les modules de gestion d'erreur et d'édition de lien devrait être suffisamment générique pour permettre la construction d'autre machines virtuelles objets. Le seul module qui ne peut pas être modifier à partir de cette structure est le module d'édition de lien pour lequel un autre système d'adaptation est proposé.

7.3. *Module Object*

La mémoire possède un ramasse-miettes. Ce ramasse-miettes est un marque-et-trace exact, coloré et incrémental inspiré de l'algorithme de Boehm [BDS91] et basé sur l'algorithme de Dijkstra/Lamport [DLM+78]. Le ramasse-miettes est écrit en C++ et interfacé *via* le module Object avec la JVM. Une référence est directement un pointeur vers la mémoire. Le ramasse-miettes a été écrit pour être le moins dépendant possible du système.

Chaque objet possède une interface (stockée à l'offset 0) qui spécifie le comportement de l'objet dans la mémoire. On trouve en particulier dans cette interface une fonction qui s'occupe de tracer un objet (*i.e.* de marquer les objets directement atteignable comme vivant). Cette fonction est compilée dynamiquement pour tracer sélectivement les objets Java. Par exemple, pour une classe Java qui ne contient qu'une référence et 10 entiers, la fonction trace compilée dynamiquement ne tracera que la référence ce qui évite de déréférencer inutilement les entiers. Cette technique évite aussi de confondre une référence avec un entier ayant par hasard pour valeur une référence.

L'allocateur et le ramasse-miettes peuvent être modifiés dynamiquement pour supprimer la gestion multitâche en supprimant les sémaphores et la barrière en écriture pendant une collection. Quelques paramètres du ramasse-miettes comme le nombre d'objets alloués entre deux collections peuvent être configurés dynamiquement.

Les objets Java possèdent aussi une classe stockée à l'offset 4 des objets. Une classe est un objet non typé décrivant les métadonnées de l'objet.

7.4. *Les classes, les méthodes et les champs*

Les classes, les champs et les méthodes sont des objets (donc collectés) non typés contenant les informations nécessaires au bon fonctionnement de la JVM. Chaque structure possède quelques octets réservés pour permettre une plus grande flexibilité à l'application active.

Ces structures sont forcément liées à une machine virtuelle, de cette façon, plusieurs machines virtuelles spécifiques peuvent cohabiter au sein de la même application active. Elles sont aussi liées à un symbole : le nom Java projeté dans l'espace de nommage *YNVM*.

7.5. *Module de chargement*

Ce module permet de charger un fichier de description de classe Java. Le fichier est lu à partir de l'entrée passée en argument par la fonction load de la structure de machine virtuelle.

La fonction prédéfinie permet à l'application d'ajouter dynamiquement le traitement d'attributs non connus. Certains attributs (comme l'attribut CODE qui donne le bytecode d'une méthode) doivent obligatoirement être gérés par la machine virtuelle. Ceux qui ne sont pas gérés doivent être ignorés silencieusement.

Les attributs sont lus et stockés dans les structures de classe, de méthode ou de champs. Une application peut ensuite gérer cet attribut plus tard (un exemple est donné avec la gestion d'un attribut d'analyse d'échappement en 7.8).

7.6. *Module d'édition de liens*

Le module d'édition de lien s'occupe de lier les symboles Java entre eux suivant la sémantique appropriée. Il existe 8 algorithme de liaison : appel de méthode virtuel, statique et spécial, accès au champs virtuel et statique, en lecture ou écriture et appel de la pseudo-méthode new. Comme ces algorithmes se ressemblent et comme l'interface d'adaptation de ces algorithmes est pratiquement la même, nous ne décrirons que l'algorithme d'appel virtuel.

L'interface du l'algorithme d'appel virtuel est constitué de deux fonctions : une fonction qui s'occupe de piloter directement le processeur virtuel de la *YNVM* pour implanter l'appel vpu-call-virtual et une fonction qui s'occupe de transformer un AST MVV d'appel en un AST qui implante l'algorithme d'appel : synt-call-virtual. Typiquement, l'appel (:Java.lang.Object.clone_sign obj) est réécrit par la syntaxe pour implanter de manière transparente l'algorithme d'appel virtuel. Ainsi, pour modifier l'algorithme, il suffit à une application active de modifier ces deux fonctions.

Les algorithmes utilisés par défaut sont assez classiques : l'édition de lien est effectuée au plus tard. Une méthode n'est compilée que lorsqu'on l'exécute, une classe

n'est chargée que quand on s'en sert, etc. Pour permettre une édition de lien tardive, un cache en ligne d'appel est utilisé. Ce cache contient les informations nécessaires à l'édition de lien (*i.e.* : le symbole VVM de la méthode), la classe du dernier objet qui a effectué l'appel (pour vérifier que la méthode virtuelle appelée correspond bien à la classe de l'objet transmis) et un pointeur vers une fonction. Ce pointeur est initialisé vers la fonction d'édition de lien virtuel et est remplacé par l'adresse de la méthode assemblée après l'édition de lien. Cette technique implique un appel indirect.

7.7. *Module de compilation*

Le module de compilation ne possède qu'un seul point d'entrée : la fonction compile qui s'occupe de compiler le bytecode d'une méthode Java. Pour compiler individuellement chaque bytecode, on dispose d'un vecteur de 256 fonctions. Pour modifier le compilateur de bytecode, une application active peut soit modifier globalement le pointeur compile dans la structure de machine virtuelle, soit modifier individuellement l'entrée d'un bytecode particulier.

Le bytecode est compilé en utilisant directement le processeur à pile de la *YNVM*, ce qui permet à la JVM d'utiliser un JIT.

7.8. *Quelques exemples*

Dans cette sous-section, nous allons présenter les premiers exemples d'applications actives.

La machine virtuelle de référence

Ce première exemple d'application active instancie une machine virtuelle standard respectant la spécification de Sun [GOS 96, LIN 96]. Les classes de base nécessaires au lancement de la JVM sont celles du projet *ClassPath* de GNU [GCP]. Cette application active permet de tester le bon fonctionnement des différents modules de Pomv. Elle permet aussi d'exécuter n'importe quelle application Java standard.

La machine virtuelle adaptable

Ce second exemple d'application active est une machine virtuelle Java adaptable à distance.

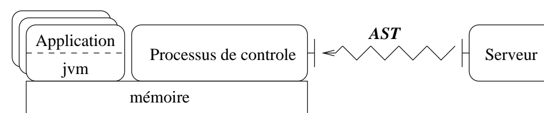

Figure 10. *Mise à jour à distance*

L'application active commence par lancer un processus de contrôle qui s'occupe de recevoir et d'exécuter des arbres syntaxiques sérialisés sur le réseau puis lance l'application Java (*via* l'application active précédente). Cette application active permet à un serveur de modifier à distance la machine virtuelle Java.

Modification de méthode

Cette application active modifie le fonctionnement des méthodes Java en cours d'exécution. Cette application active repose sur la précédente. L'AST envoyé par le serveur duplique la méthode finalize de la classe Java/lang/Object puis modifie la méthode originale. Cette nouvelle méthode écrit sur la sortie standard quel objet est détruit et appel l'ancienne méthode finalize.

Modifier la méthode finalize revient à modifier la description de la classe Java/lang/Object et à reinitialiser tous les caches d'appels de *.finalize() où * représente n'importe quelle classe. La synchronisation est effectuée grâce aux sémaphores de la classe Java/lang/Object et au sémaphore des caches en ligne.

Ce court exemple (42 lignes de code sont sérialisées par le serveur) illustre la possibilité de construire un tisseur d'aspect.

Analyse d'échappement

Cet exemple illustre la modification sémantique du bytecode `new` par le biais d'un attribut de code Java nommé EscapeMap. La nouvelle sémantique permet d'allouer un objet en pile ou dans le tas à la demande. Le bytecode allouant en pile respecte bien sûr la politique de sûreté des pointeurs de Java. L'attribut donne les allocations qui peuvent être gérées de manière sûre en pile, les autres gardant la sémantique de Java c.-à-d. allocation dans le tas. Cet attribut est le résultat d'une analyse statique de code qui s'appuie sur des techniques d'analyse d'échappement (*escape analysis*) [WHA 99, CHO 99, BLA 03]. Cette technique détermine la durée de vie des variables objets. Les objets qui peuvent être détruit à la sortie d'une méthode sont gérés par allocation en pile et non plus par le ramasse-miettes. Cela permet d'améliorer les performances globales de la machine virtuelle avec la même sûreté de fonctionnement : ces objets sont détruits plus rapidement (décalage de pointeur de pile) et le ramasse-miettes gère moins d'objet.

L'application active redéfinit les bytecodes d'allocation en vérifiant s'il faut allouer dans le tas ou en pile. Pour l'allocation en pile, il suffit de *(i)* décaler le pointeur de pile de la taille de l'objet à allouer *(ii)* initialiser la zone à zéro (voir code figure 11). Les objets ainsi en pile seront détruits à la sortie de la méthode par simple décalage de pointeur. La gestion des exceptions est elle aussi assez simple : lorsqu'une exception est attrapée, le pointeur de pile est déplacé et les objets sont bien libérés. Notre algorithme ne gère pas la finalisation : l'utilisation de l'analyse d'échappement est réservée aux machines virtuelles j2me.

Cet exemple montre comment une application active peut apprendre à la JVM à modifier dynamiquement la sémantique de ses bytecodes pour un coût de dévelop-

```
(rdef-opcode  :jnjvm.compile.new
 (lambda(no code-input curseur env comp vpu) (+ curseur 3))
  (lambda(no code-input curseur env comp vpu)
   (if (on-stack? env curseur)  ;; doit-on allouer l'objet dans la pile?
    (begin
     (let ([dcl ...]) ;; lit la classe à partir de l'entrée
    ... ;; injecte dans le vpu le chargement au + tard de la classe, pile = dcl
    (:compiler.vpu.ld-int vpu (+ dcl :jnjvm.class.virtual-pattern))
    (:compiler.vpu.rdw vpu) ;; charge sur la pile le modèle d'instance initialisé (vp)
    (:compiler.vpu.dup vpu 0)   ;; pile = vp vp
    (:compiler.vpu.dup vpu 0)   ;; pile = vp vp vp
    (:compiler.vpu.ld-int vpu :jnjvm.jobject.sizeof)
    (:compiler.vpu.icall vpu 1) ;; pile = vp vp (taille vp)
    (:compiler.vpu.put vpu 2)   ;; pile = (taille vp) vp (taille vp)
    (:compiler.vpu.alloc vpu)   ;; allocation en pile, pile = (taille vp) vp ptr
      ;; pile = (taille vp) vp ptr (addr memcmpy de glibc)
    (:compiler.vpu.ld-int vpu :system.memcpy)
    (:compiler.vpu.icall vpu 3) ;; pile = ptr
    (+ curseur 3)))
      ;; on ne doit pas allouer l'objet en pile
   (old-new no code-input curseur env comp vpu))))
```

Figure 11. *Nouvelle sémantique du bytecode* `new` *(extrait)*

pement assez faible (une centaine de lignes de code) et illustre quelques possibilités d'adaptation de la JVM.

7.9. *Quelques mesures de performances*

Les mesures de performances sont en cours de réalisation, c'est pourquoi nous n'indiquons que des ordres de grandeur. Dans le cas d'une JVM standard, le démarrage s'avère très lent : 3s sur un PowerPC G3 à 366 MHz sous linux contre 1s pour la machine virtuelle de Sun. Cet écart s'explique facilement : notre JVM est intégralement compilée au démarrage. Par contre, à l'exécution, les performances sont équivalentes à la JVM de Sun : le code compilé par le compilateur dynamique de la MVV est équivalent à du code C optimisé et le coût des quelques indirections dues à la structure de machine virtuelle s'avère négligeable.

8. Conclusion

Cet article a présenté notre approche pour la spécialisation et l'adaptation dynamique d'environnements d'exécution. Le projet *MVV* propose un cadre général pour la construction d'environnements d'exécution flexibles s'appuyant sur un principe de minimalité ainsi que sur la reconciliation des aspects système et langage. L'architecture résultante est composé d'une couche de réification du matériel supportant un compilateur dynamique réflexif, utilisé pour la construction dynamique d'environnement d'exécution spécifiques. L'absence de modèle de gestion de ressources, de sécurité ou de programmation défini à ce niveau, alliée à la réflexivité du compilateur dynamique, permet une flexibilité maximale.

L'implémentation de cette architecture, et plus particulièrement du compilateur dynamique réflexif *YNVM* a permis d'expérimenter l'intérêt de la flexibilité dynamique dans différents contexte applicatif : réseaux actifs, caches web adaptatifs ou encore applications actives. Les premières évaluations ont également permis de vérifier que cette flexibilité dynamique ne venait pas au prix d'une dégradation prohibitive des performances.

Nos travaux au sein du projet *MVV* continuent dans le but de fournir une approche systématique pour la construction d'environnement d'exécution flexible, adaptable et interopérable

9. Bibliographie

[ATK 96] ATKINSON M., DAYNES L., JORDAN M., PRINTEZIS T., SPENCE S., « An Orthogonally Persistent Java », *ACM Sigmod Record*, vol. 25, n° 4, 1996.

[AUB 99] AUBERT O., BEUGNARD A., « Towards a Fine-Grained Adaptivity in Web Caches », *the 4th International Web Caching Workshop*, San Diego, California, USA, April 1999.

[BAR 99] BARNES J. F., PANDEY R., « CacheL : Language Support for Customizable Caching Policies », *the 4th International Web Caching Workshop*, San Diego, California, USA, April 1999.

[BER 95] BERSHAD B., SAVAGE S., PARDYAK P., SIRER E. G., BECKER D., FIUCZYNSKI M., CHAMBERS C., EGGERS S., « Extensibility, Safety and Performance in the SPIN Operating System », *the 15th ACM Symposium on Operating Systems Principles (SOSP)*, Copper Mountain Resort, Colorado, USA, December 1995, p. 267–284.

[BLA 99] BLAIR G. S., COSTA F. M., COULSON G., DURAN H. A., PARLAVANTZAS N., DELPIANO F., DUMANT B., HORN F., STEFANI J.-B., « The Design of a Resource-Aware Reflective Middleware Architecture », *Reflection'99*, vol. 1616 de *LNCS*, Saint-Malo, France, July 1999, Springer-Verlag, p. 115–134.

[BLA 03] BLANCHET B., « Escape Analysis for Java. Theory and Practice », *ACM Transactions on Programming Languages and Systems*, , 2003, To appear.

[BOL 00] BOLLELLA G., GOSLING J., BROSGOL B., GOSLING J., DIBBLE P., FURR S., TURNBULL M., *The Real-Time Specification for Java*, The Java Series, Addison-Wesley, 2000.

[CHE 00] CHEN Z., *Java Card Technology for Smart Cards : Architecture and Programmer's Guide*, The Java Series, Addison-Wesley, 2000.

[CHO 99] CHOI J.-D., GUPTA M., SERRANO M., SREEDHAR V. C., MIDKIFF S., « Escape Analysis for Java », *ACM conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, Denver USA, 1999, p. 2–19.

[COL 00] COLAJANNI M., CASALICCHIO E., « Scalable Web Cluster with Static and Dynamic Contents », *the IEEE International Conference on Cluster Computing (CLUSTER 2000)*, Chemnitz, Germany, December 2000.

[DUM 98] DUMANT B., HORN F., TRAN F. D., STEFANI J.-B., « Jonathan : an open distributed processing environment in Java », *Proceedings of the IFIP International Conference*

on Distributed Systems Platforms and Open Distributed Processing (Middleware'98), Lake District, UK, September 1998.

[ENG 95] ENGLER D. R., KAASHOEK M. F., O'TOOLE J. W., « Exokernel : an operating system architecture for application-level ressource management », *the 15th ACM Symposium on Operating Systems Principles (SOSP)*, Copper Mountain Resort, Colorado, USA, December 1995.

[FAS 01] FASSINO J.-P., STEFANI J.-B., « THINK : un noyau d'infrastructure répartie adaptable », *the 2nd French Chapter of ACM-SIGOPS : CFSE'2*, Paris, France, Avril 2001.

[FOL 00] FOLLIOT B., « The Virtual Virtual Machine Project », *the 12th Symposium on Computer Architecture and High Performance Computing*, Sao Paulo, Brazil, October 2000.

[GAM 95] GAMMA E., HELM R., JOHNSON R., VLISSIDES J. M., *Design Patterns : Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995.

[Gem] GEMSTONE SYSTEMS, INC., `http://www.gemstone.com/`.

[GON 99] GONG L., *Inside Java 2 Platform Security : architecture, API design and implementation*, The Java Series, Addison-Wesley, June 1999.

[GOS 96] GOSLING J., JOY B., STEELE G. L., BRACHA G., *The Java Language Specification*, The Java Series, Addison-Wesley, 2nd édition, 1996.

[GUL 01] GULWANI S., TARACHANDANI A., GUPTA D., SANGHI D., BARRETO L. P., CONSEL C., MULLER G., « WebCaL : A Domain-Specific Language for Web Caching », *Computer Communications*, vol. 4, nº 2, 2001.

[HAR 99] HARRIS T. L., « An Extensible Virtual Machine Architecture », *OOPSLA'99 Workshop on Simplicity, Performance and Portability in Virtual Machine Design*, November 1999, page 135.

[HIC 98] HICKS M. W., KAKKAR P., MOORE J. T., GUNTER C. A., NETTLES S., « PLAN : A Packet Language for Active Networks », *the 3rd ACM SIGPLAN International Conference on Functional Programming Languages*, September 1998, p. 86–93.

[HOW 99] HOWELL J., MONTAGUE M., COLLEGE D., « Hey, You Got Your Compiler In My Operating System ! », *7th IEEE Workshop on Hot Topics in Operating Systems (HotOS-VII)*, Rio Rico, Arizona, USA, March 1999.

[Jav] JAVA 2 PLATFORM MICRO EDITION (J2ME), « Connected Limited Device Configuration (CLDC) Specification, Version 1.0 », `http://www.java.sun.com/products/cldc/`.

[KON 98] KON F., SINGHAI A., CAMPBELL R. H., CARVALHO D., MOORE R., BALLESTEROS F. J., « 2K : A Reflective, Component-Based Operating System for Rapidly Changing Environments », *Proceedings of the ECOOP'98 Workshop on Reflective Object-Oriented Programming and Systems*, Brussels, Belgium, July 1998.

[LAW 02] LAWALL J. L., MULLER G., BARRETO L. P., « Capturing OS expertise in an Event Type System : the Bossa experience », *ACM SIGOPS European Workshop 2002 (EW'2002)*, Saint-Emillion, France, September 2002, p. 54–61.

[LER 03] LEROY X., « Java bytecode verification : algorithms and formalizations », *Journal of Automated Reasoning — special issue on Bytecode Verification*, vol. 30, nº 3-4, 2003, p. 235–269, Kluwer Academic Publishers.

[LIN 96] LINDHOLM T., YELLIN F., *The Java Virtual Machine Specification*, The Java Series, Addison-Wesley, 2nd édition, September 1996.

[Mao] MAOSCO CONSORTIUM LTD., « Multos™ », http://www.multos.com.

[MER 00] MERILLON F., RÉVEILLÈRE L., CONSEL C., MARLET R., , MULLER G., « Devil : An IDL for Hardware Programming », *4th USENIX Symposium on Operating System Design & Implementation (OSDI)*, San Diego, California, USA, October 2000.

[Mic 97] MICHAEL GOLM, « Design and implementation of a meta architecture for Java », Master's thesis, University of Erlangen, January 1997.

[NEC 97] NECULA G. C., « Proof–Carrying Code », *the 24th ACM SIGPLAN-SIGACT symposium on principles of programming Languages*, Paris, France, January 1997.

[odp98] « Information Technology – Open Distributed Processing – Reference Model », 1996 — 1998.

[OMG99] « The Common Object Request Broker : Architecture and Specification », October 1999.

[PAT 00] PATARIN S., MAKPANGOU M., « Pandora : A Flexible Network Monitoring Platform », *USENIX Annual Technical Conference*, San Diego, USA, June 2000.

[ROM 99] ROMÁN M., KON F., CAMPBELL R. H., « Design and Implementation of Runtime Reflection in Communication Middleware : the dynamicTAO Case », *Proceedings International Conference on Distributed Computing Systems (ICDCS) – Workshop on Electronic Commerce and Web-Based Applications*, Austin, Texas, USA, June 1999, IEEE, p. 122–127.

[SC] « Squid Web Proxy Cache », http://www.squid-cache.org/.

[SEL 96a] SELTZER M., « The World Wide Web : Issues and Challenges », Presented at IBM Almaden, July 1996.

[SEL 96b] SELTZER M. I., ENDO Y., SMALL C., SMITH K. A., « Dealing with Disaster : Surviving Misbehaved Kernel Extensions », *Proceedings of the 2nd Symposium on Operating Systems Design and Implementation*, Seattle, Washington, 1996, p. 213–227.

[TEN 96] TENNENHOUSE D. L., WETHERALL D. J., « Towards an Active Network Architecture », *Computer Communication Review*, vol. 26, n° 2, 1996.

[THI 97] THIBAULT S., MARLET R., CONSEL C., « A Domain-Specific Language for Video Device Drivers : from Design to Implementation », *Conference on Domain Specific Languages*, Santa Barbara, California, USA, october 1997, USENIX Association, p. 11–26.

[THI 99] THIBAULT S., MARANT J., MULLER G., « Adapting Distributed Applications Using Extensible Networks », *Proceedings of the 19th International Conference on Distributed Computing Systems*, Austin, Texas, USA, May 1999, IEEE, p. 234–243.

[UCL] « Embedded Linux/Microcontroller Project : *μClinux* », http://www.uclinux.org/.

[WHA 99] WHALEY J., RINARD M., « Compositional pointer and escape analysis for Java programs », *ACM conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, 1999, p. 187–206.

Frédéric Ogel *était doctorant à l'INRIA, dans le cadre du projet Regal. Ses travaux portent sur la flexibilité dynamique dans les environnements d'exécution. Il est actuellement ingénieur de recherche à France Télécom - Division R&D.*

Gaël Thomas *est étudiant en thèse au laboratoire d'informatique de Paris 6 (LIP6), ses travaux portent principalement sur les mécanismes d'adaptation dynamique dans les intergiciels*

Antoine Galland *est ingénieur de recherche au sein du laboratoire de recherche de la société Gemplus et étudiant en thèse d'informatique au laboratoire LIP6 de l'Université Pierre & Marie Curie. Ses travaux portent sur le contrôle des ressources dans les cartes à puce.*

Bertil Folliot *est Professeur à l'Université Paris VI. Ses activités de recherche actuelle concernent la réalisation de systèmes répartis flexible, adaptable et interopérable. Il dirige le projet de « machines virtuelles virtuelles ».*