Pandora : une plate-forme efficace pour la construction d'applications autonomes

Simon Patarin* — **Mesaac Makpangou****

* *Università di Bologna – Dipartimento di Scienze dell'Informazione*
Mura Anteo Zamboni, 7
I-40127 Bologne, Italie
patarin@cs.unibo.it

** *Projet REGAL – INRIA Rocquencourt*
Domaine de Voluceau, B.P. 105
F-78153 Le Chesnay Cedex, France
mesaac.makpangou@inria.fr

RÉSUMÉ. *L'informatique autonome a récemment été proposée comme une réponse à la difficulté de gérer au quotidien des applications dont la complexité ne cesse d'augmenter. Les applications autonomes devront être particulièrement flexibles et pouvoir se surveiller en permanence. Cette étude présente une plate-forme, Pandora, qui facilite la construction d'applications qui satisfont ce double objectif. Pandora s'appuie sur un mode de programmation original des applications — fondé sur la composition de couches et le passage de messages — pour aboutir à un modèle et une architecture minimalistes qui lui permettent de contrôler les surcoûts imposés par la complète réflexivité de la plate-forme. Un prototype fonctionnel de la plate-forme a par ailleurs été développé en C++. Une étude détaillée des performances, ainsi que des exemples d'utilisation, complètent cette présentation.*

ABSTRACT. *Autonomic computing has been proposed recently as a way to address the difficult management of applications whose complexity is constantly increasing. Autonomous applications will have to be especially flexible and be able to monitor themselves permanently. This work presents a framework, Pandora, which eases the construction of applications that satisfy this double goal. Pandora relies on an original application programming pattern — based on stackable layers and message passing — to obtain minimalist model and architecture that allows to control the overhead imposed by the full reflexivity of the framework. Besides, a prototype of the framework has been implemented in C++. A detailed performance study, together with examples of use, complement this presentation.*

MOTS-CLÉS : *Informatique autonome, modèle de composants, reconfiguration dynamique*

KEYWORDS: *Autonomic computing, component model, dynamic reconfiguration*

1. Introduction

Les applications distribuées à large échelle occupent une place grandissante. Les réseaux de distribution de contenu, les grilles de calcul, les systèmes d'échange de fichiers pair-à-pair, les tables de hachages distribuées, les systèmes omniprésents : les exemples sont nombreux et leur liste augmente constamment. L'environnement dans lequel ces applications sont déployées, Internet, dispose de caractéristiques originales que sont l'hétérogénéité, l'évolution rapide de l'ensemble des acteurs (matériels, logiciels, mais aussi humains) et un relatif manque de fiabilité. La diversité des plates-formes matérielles et logicielles rend alors particulièrement difficile la configuration de ces applications. Si l'on parvient malgré tout à franchir cette première étape, les évolutions et les possibles pannes peuvent remettre en question les choix faits au préalable et réduire à néant les efforts précédents. Il est donc impératif de faciliter ces opérations en les automatisant autant que possible. Ces constats sont à la base du développement de l'informatique autonome (*autonomic computing*) [KEP 03] dont le but est d'incorporer aux applications les moyens de diagnostiquer les difficultés auxquelles elles font face et d'y remédier elles-mêmes. Les problèmes à résoudre avant de parvenir à une solution satisfaisante sont évidemment très nombreux. Nous nous consacrons à l'étude de l'un d'entre eux : le support système nécessaire au développement de telles applications.

Il est possible d'identifier plusieurs fonctionnalités qui devront être proposées par une plate-forme de construction d'applications autonomes. La première et la plus importante concerne la flexibilité des applications : il est en effet inutile d'imaginer l'autonomie d'une application si celle-ci ne peut pas être modifiée et reconfigurée dynamiquement. De nombreux types de reconfiguration sont envisageables et tous devront être supportés : depuis la simple paramétrisation à l'ajout de propriétés non fonctionnelles susceptibles de modifier radicalement le comportement de l'application. Par ailleurs, l'application elle-même est souvent la mieux placée pour collecter les mesures qui lui permettront d'analyser son comportement : il faut donc que la plate-forme fournisse les mécanismes nécessaires à la diffusion de ces observations. Cette dernière, enfin, doit faciliter les interactions entre les différents éléments du système, en particulier donner accès à l'état courant des applications et aux mesures effectuées (ce qui exige donc que la plate-forme soit parfaitement réflexive [SMI 84, MAE 87]).

Cette flexibilité que nous requérons ne doit pas être obtenue au détriment des performances ; ce n'est cependant pas l'approche retenue dans les systèmes actuels où une seule de ces deux propriété est développée au détriment de l'autre. Ainsi, dans le cas des plates-formes spécifiquement conçues pour le développement d'applications autonomes (comme AutoPilot [RIB 98] ou AutoMate [AGA 03]), la flexibilité (et donc les possibilités de reconfiguration) des applications est mise à mal et est notamment insuffisante pour répondre à la diversité des besoins que nous anticipons. À l'inverse, dans le domaine de la programmation par aspects [KIC 97] ou celui des systèmes à composants [CLA 01], les applications construites peuvent faire preuve d'une très grande flexibilité, mais les surcoûts introduits réduisent considérablement le niveau de performance.

L'approche que nous défendons pour répondre à ce problème repose sur un compromis original. Plutôt que de transiger sur les possibilités de flexibilité offertes ou le langage utilisé, nous mettons en avant l'utilisation d'un mode de programmation alternatif : l'empilage de composants indépendants. Cette approche, même si elle n'a encore jamais été employée dans ce contexte à notre connaissance, n'est cependant pas nouvelle et plusieurs projets l'ont expérimentée, soulignant son expressivité. Parmi les premiers travaux dans ce domaine, on retrouve *x*-Kernel [HUT 91] et Ficus [HEI 91] (respectivement un système d'exploitation spécialisé dans les communications et une méthode de construction de systèmes de fichiers) ; plus récemment, deux nouvelles architectures ont été proposées : un routeur flexible, le *Click Modular Router* [KOH 00] et SEDA [WEL 01] qui permet la construction efficace de services Internet. Alors que la plupart des systèmes patrimoniaux doivent instrumenter les appels de procédure (ce qui se révèle coûteux et pousse à utiliser un langage interprété), ce mode de programmation, permet de définir le degré d'intervention voulu au travers du choix de la granularité des composants. La plate-forme que nous présentons, Pandora, s'appuie donc sur ces techniques et fournit une interface réflexive qui permet aux applications extérieures, ainsi qu'aux composants eux-mêmes, de reconfigurer dynamiquement le système entier.

Nous allons maintenant présenter (section 2) un rapide panorama des travaux sur lesquels s'appuie cette étude. Nous décrivons ensuite le modèle de composants de Pandora (section 3) et la grande flexibilité de la plate-forme qui en résulte (section 4). Nous précisons ensuite la mise en œuvre de cette plate-forme et donnons quelques exemples d'utilisation du prototype (section 5). Nous terminons cette étude par une évaluation des performances de Pandora (section 6) et quelques remarques de conclusion (section 7).

2. Travaux comparables

Comme nous l'avons mentionné, il existe quelques plates-formes qui poursuivent des objectifs comparables aux nôtres. Il s'agit par exemple d'AutoPilot [RIB 98] qui se focalise plus particulièrement sur la conception de moyens d'observation intégrés aux applications, ou d'AutoMate [AGA 03] qui cible pour sa part les applications déployées sur les grilles de calcul. La flexibilité de ces plates-formes est relativement limitée : le paramétrage dynamique autorise, au mieux, le choix entre différentes mises en œuvre pour des fonctionnalités prédéfinies et les possibilités d'extension et de modification de propriétés non fonctionnelles sont quasiment inexistantes. Ce manque de flexibilité nous a poussé à considérer les approches actuellement proposées pour répondre à ce problème : les systèmes à composants et la programmation orientée aspects.

Les systèmes à composants patrimoniaux — .NET, CCM (*Corba Component Model*) ou EJB (*Enterprise Java Beans*) — sont en réalité relativement peu adaptés à la conception d'architectures flexibles. La granularité importante des composants, les liaisons entre composants difficilement modifiables dynamiquement et l'ensemble limité et prédéfini de propriétés non fonctionnelles fournies par les conteneurs de composant y contribuent grandement. Ceci a favorisé le développement de systèmes plus légers (et plus performants). Ainsi, la plate-forme OpenCOM [CLA 01] autorise la reconfiguration dynamique des composants et des liaisons qui existent entre eux par le biais d’une interface réflexive. Un autre exemple est Gravity [HAL 03] qui adopte une démarche originale : le système établit dynamiquement les liaisons entre les composants selon les interfaces qu’il fournit et celles qu’il requiert. Pour notre application la principale limitation d’un tel système est l’absence de support pour les reconfigurations « simples » comme la modification de la valeur d’un paramètre.

La programmation orientée aspects [KIC 97] est une méthodologie qui promeut la séparation des préoccupations (*separation of concerns*) : les fonctionnalités transverses à plusieurs modules d’un programme sont isolées (ce sont les *aspects*) et seront « tissées » avec le reste de l’application au moment de la compilation ou de l’exécution. La flexibilité de ces architectures provient de la relative indépendance entre les différentes entités (modules et aspects) de l’application que cette approche procure : il est alors possible de modifier un module ou un aspect sans perturber le reste du programme. C’est la plate-forme JAC [PAW 02] (*Java Aspect Components*) qui se rapproche le plus de nos objectifs. Ici, en effet, les aspects sont encapsulés sous la forme de composants et tissés à l’exécution (ce qui en autorise donc la reconfiguration dynamique). La principale limitation de cette plate-forme est son manque de support pour la reconfiguration des modules eux-mêmes (ceux dont les fonctionnalités ne peuvent pas être vues comme des aspects).

3. Modèle de composants

Pandora s’appuie sur la notion de *composant* comme unité de travail élémentaire et indépendante ; ces composants peuvent être assemblés sous forme de piles pour exécuter des tâches de plus haut niveau. Nous allons décrire plus précisément le modèle de composant qu’utilise Pandora et nous présentons ensuite — brièvement — le langage de description d’architecture utilisé pour définir et paramétrer ces assemblages de composants.

Contrairement à la plupart des modèles de composants existants, les composants utilisés par Pandora communiquent exclusivement par passage de message et non par invocation. Cette contrainte contribue grandement à réduire la complexité des composants : un composant ne définit en effet qu’une seule interface, celle qui lui permet de recevoir un message. Dans la terminologie utilisée, les messages échangés entre les composants sont appelés événements, et c’est ce terme que nous emploierons par la suite. Les événements sont typés et un composant a la possibilité d’utiliser cette information pour déterminer les traitements qu’il aura à effectuer.

Chaque composant dispose, au maximum, d’un port d’entrée et d’un nombre quelconque de ports de sortie. Le transfert d’événements entre les composants est unidirectionnel, synchrone et se fait par flot continu. Ceci signifie, en particulier, que deux composants, une fois la communication établie, sont durablement associés l'un à l'autre et s'oppose à la transmission de messages par *boîte aux lettres* (nous verrons plus loin que ce mode de transmission existe néanmoins dans le système). L'ensemble des composants associés par de telles relations est appelé *pile*. Un nom est associé à chaque pile et il est possible de faire coexister plusieurs instances d'une même pile dans le système qu'on peut différencier au moyen de « poignées » (*handles*) attribuées par le système ou bien d'*alias* spécifiés explicitement.

Du fait du synchronisme de la transmission des événements au sein d'une pile, les traitements des composants sont directement associés à la réception d'un événement, lesquels peuvent à leur tour entraîner la transmission d'un ou plusieurs événements. Ainsi, une fois qu'un événement est « produit » l'ensemble des traitements générés (par transmission successive d'événements) est traité séquentiellement. La production initiale des événements est assurée dans chaque pile par un composant spécifique nommé *composant initial* qui est ordonnancé de manière indépendante. Ce composant — il n'en existe qu'un seul par pile — est donc indirectement responsable de l'activité de l'ensemble de la pile : s'il cesse de produire des événements, c'est toute la pile qui est au repos pendant ce temps.

Un autre caractéristique importante des composants est leur entière indépendance : un composant ignore totalement le contexte dans lequel il sera utilisé. Ainsi les ports d'entrée et de sortie sont parfaitement anonymes et un composant n'a pas la possibilité de choisir le type des composants avec lesquels il communique ; ceci est déterminé au moment de la configuration de la pile. Dans le cas général, un composant qui désire transmettre un événement le transmet donc à son *successeur*, sans connaître son identité. Lorsqu'un composant dispose de plusieurs ports de sortie, il existe deux possibilités (à l'exclusion l'une de l'autre) :

1) *alternative* : des ports alternatifs sont identifiés par un numéro d'ordre et il sera possible de configurer la pile pour que des composants de nature différente correspondent à chacun des ports ;

2) *démultiplexage* : des ports de démultiplexage permettent à un composant de classifier les événements : dès qu'une nouvelle catégorie est découverte, un nouveau port est créé dynamiquement, alors que les événements appartenant à une catégorie existante sont dirigés vers le port qui y avait été précédemment associé.

Nous voyons alors que chaque composant à sorties multiples doit être configuré à l'aide de « sous-piles » pour chaque port de sortie. Ces sous-piles sont nommées *branches* et correspondent aux traitements différenciés des événements au sortir d'un composant à sorties multiples. Ainsi, tous les événements qui sortent des branches associés à un composant à sortie multiple donné sont envoyés sur le port d'entrée d'un unique composant qui les multiplexe.

Chaque composant, outre les ports de sortie que nous avons mentionné, a la possibilité de communiquer directement avec une pile, prise dans son ensemble ; le port d'entrée d'une pile est alors défini par le port d'entrée de son composant initial. À la différence des communications entre composants qui sont anonymes, les communications entre piles sont nommées : un composant indique le nom de la pile (ou son *alias*) à laquelle il souhaite transmettre des événements. Ici, les communications peuvent être, au choix, synchrones ou asynchrones : ce modèle de communication peut donc être utilisé pour des transmissions d'événements selon le modèle des boîtes aux lettres. Les composant, enfin, ont la possibilité de déclarer explicitement des paramètres, que nous nommons *options*, identifiés par un nom et dont la valeur peut être configurée pour adapter leur comportement. Ces options peuvent être de tout type et il est possible d'associer un traitement particulier à l'affectation d'une valeur (par exemple transformer une chaîne de caractères représentant un nom de machine en adresse IP numérique).

La spécification de l'assemblage des composants au sein des piles, ainsi que leur paramétrisation initiale, se fait au moyen d'un langage spécifique. Afin de faciliter son utilisation et sa compréhension par les utilisateurs du système, une représentation compacte des piles nous a paru préférable. Il est cependant envisageable d'utiliser des langages de description de graphe (tel que `dot` [ELL 02]) ou des langages à balises (de type XML) pour aboutir à des résultats similaires. Nous nous limiterons ici à présenter très brièvement la grammaire de ce langage (tableau 1) et un exemple complet d'utilisation (figure 1), sa compréhension ne posant pas de problème particulier (voir [PAT 03a], section 3.4.4 pour une description détaillée).

```
pile        :: ‘%’id alias? ‘{’ composant* ‘}’
composant   :: simple | demux | alternative
simple      :: ‘@’id alias? options?
options     :: (‘[’ option (‘,’ option)* ‘]’)?
alias       :: ‘ :’id
demux       :: simple ‘<’ branche ‘>’
alternative :: simple ‘(’ branche (‘|’ branche)* ‘)’
branche     :: composant+
option      :: ‘$’id alias? (‘=’ valeur)?
id          :: [a-zA-Z]([a-zA-Z0-9_])*
valeur      :: entier | flottant | booléen | ‘"’caractères‘"’
```

Tableau 1 – Grammaire du langage de description d'architecture (BNF).

4. Plate-forme flexible

Nous avons dit à quel point les applications autonomes avaient besoin d'être flexibles. Ceci s'exprime à la fois dans la capacité à les paramétrer le plus finement possible, mais également dans la possibilité de revenir sur ces choix au fur et à mesure que le contexte dans lequel s'exécutent les applications évolue. Plutôt que de décrire l'architecture dans son ensemble (décrite avec précision dans des travaux précédents [PAT 03a]), nous allons nous concentrer sur ce point précis et détailler les mécanismes mis à disposition pour le développement d'applications autonomes.

```
%dns {
  @pcap [$filter = "udp and port 53"]
  @ipfragswitch ( @demux [$algo = "ipquad"] < @ipreass > )
  @demux [$algo = "ipcnx"] <
    @udpscan [$timeout = 10]
    @dnsscan
    @demux [$algo = "dnsreq"] < @dnsmatch >
  >
  @printer
}
```

Figure 1 – Exemple complet de la configuration d’une pile (utilisée pour la capture et la reconstruction du trafic DNS).

4.1. *Degrés de flexibilité*

Deux niveaux de flexibilité apparaissent dans la spécification des systèmes manipulés par Pandora : au niveau du paramétrage des composants et au niveau de l’assemblage des composants. La flexibilité liée au paramétrage passe par l’utilisation des options et est tout à fait classique. Elle peut cependant avoir une influence importante puisqu’il est par exemple envisageable d’utiliser des options pour spécifier un algorithme de démultiplexage ou bien un port de sortie au sein d’une alternative.

Pandora expose un second niveau de flexibilité dans la spécification des assemblages de composants. D’un côté, la possibilité d’avoir différents composants qui mettent en œuvre la même fonctionnalité permet de choisir la solution qui correspond le mieux à l’environnement dans lequel on se trouve (les algorithmes pour lesquels il faut trouver un compromis entre utilisation du processeur et occupation mémoire ne sont pas rares). D’un autre côté, le modèle autorise l’utilisation de composants non fonctionnels dont l’insertion dans une pile n’en modifie le comportement général mais peut altérer la façon dont les traitements sont exécutés ou y adjoindre certains effets de bord. Ainsi, on peut citer l’exemple de composants assurant la persistance des événements qu’ils reçoivent, la répartition des traitements sur différentes machines, la surveillance de l’application, la détection des pannes, la synchronisation des traitements, etc. Ce dernier point se rapproche des problématiques que l’on retrouve dans la programmation par aspects : sans modifier l’application (les composants orignaux), il est possible d’y tisser certains aspects (par l’adjonction de composants spécifiques dans la définition de la pile).

4.2. *Reconfiguration dynamique*

L’ensemble de la configuration de la plate-forme, ainsi que son état courant, sont exposés par son moteur d’exécution (qui prend la forme d’un micro-noyau) au travers d’une interface de réflection. Ceci concerne donc aussi bien les définitions de piles, que les valeurs des options ou les listes de ressources : absolument tout ce qui est configurable par un fichier de configuration l’est également au travers de cette interface. Pour chaque élément, il est en outre possible de préciser si l’on souhaite modifier les définitions stockées ou leur représentation active (modifiant ainsi directement le comportement de la plate-forme). La gestion des piles est également exposée, de telle sorte qu’il est possible de connaître les piles en cours d’exécution ou de demander le démarrage ou l’arrêt de l’une d’entre elles.

Parmi ces opérations, la reconfiguration dynamique des piles occupe une place particulière. En effet, contrairement à toutes les autres, cette manipulation conduit à modifier des instances de composant existantes ainsi que les liaisons qui existent entre elles. Les composants de Pandora sont considérés par défaut comme porteur d’un état et la plate-forme veille donc à réduire au strict nécessaire les destructions de composants lorsque l’on passe de l’ancienne description à la nouvelle.

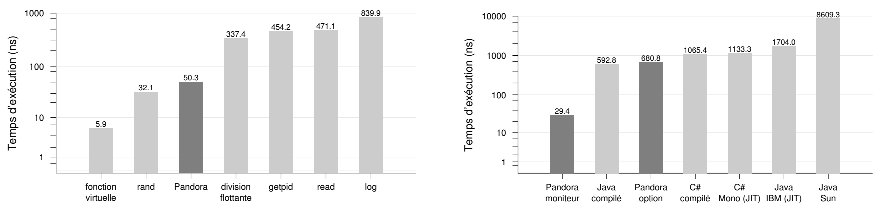
4.3. *Modes de contrôle*

Ces différentes opérations sont rendues accessibles pour les applications externes à la plate-forme au travers d’un protocole à meta-objets (*Meta-Object Protocol*). Pandora fournit une interface à ce protocole spécifique dans différents langages de programmation, notamment C, C++, Perl et Guile [JAF 02]. Il est ainsi possible d’écrire de véritables « scripts de contrôle », ce qui favorise le développement rapide d’applications autonomes.

Nous souhaitons également que les applications puissent s’analyser et se reconfigurer elles-mêmes. L’utilisation du protocole que nous venons de décrire n’est alors pas optimale en termes de performance, nous avons donc introduit un mécanisme spécifique de *capteurs* et de *moniteurs* pour répondre à ce problème. Cette technique permet à un objet (le moniteur) d’appliquer des traitements sur un ensemble de valeurs (exposées par les capteurs) de manière efficace. Un schéma de nommage simplifié permet en effet de réduire considérablement les surcoûts qui sont liés dans l’approche précédente à la localisation des paramètres au sein du système (à chaque accès il faut en effet identifier la pile, puis le composant dans la pile et enfin le paramètre dans le composant). De plus, les capteurs supportent plusieurs modes de fonctionnement : un mode *passif* où les moniteurs décident eux-mêmes de procéder aux traitements et un mode *actif* où la modification de la valeur d’un capteur déclenche les traitements.

5. Mise en œuvre

Nous avons développé une plate-forme logicielle qui met en œuvre cette architecture. Elle représente environ 50 000 lignes de code en C++ et comprend, outre le noyau, plus d’une centaine de composants. Environ le tiers de ces composants fait partie des composants « de base » : ce sont des composants qui mettent en œuvre des propriétés non fonctionnelles et peuvent être utilisés dans toute pile.



(a) Comparaison des temps d'exécution de diverses fonctions avec le temps de traversée d'un composant.

(b) Comparaison des mécanismes d'introspection de Pandora avec ceux fournis en standard avec Java et C#.

Figure 2 – Évaluation des performances

Pandora est utilisable sur un grand nombre de systèmes dont Linux, FreeBSD, NetBSD, Solaris, Digital Unix (Tru64) et, pour le noyau et les composants de base uniquement, Win32. La plate-forme est distribuée dans sa version la plus récente sous une licence *open-source* de l'INRIA (libre pour une utilisation non commerciale) à l'adresse suivante : `http://www-sor.inria.fr/projects/relais/pandora/`.

La plate-forme a déjà été utilisée dans plusieurs projets [PAT 03a, PAT 03b, FES 03, PAT 04]. L'application néanmoins qui met le mieux en valeur la flexibilité de Pandora et les possibilités qu'elle offre pour construire des applications autonomes est C/SPAN [OGE 03]. C/SPAN est un cache Web autonome qui s'appuie d'une part sur C/NN [PIU 01], un cache flexible, et sur une pile d'observation du trafic HTTP au-dessus de Pandora d'autre part. Dans ce système, C/NN et Pandora forment une boucle d'interaction rapprochée : C/NN, en fonction de son environnement (espace disque, taux de requêtes, etc.) ajuste le comportement de Pandora en utilisant son interface réflexive, tandis que Pandora reconfigure le cache en fonction de ses observations.

6. Évaluation des performances

Dans cette évaluation des performances de la plate-forme, nous nous sommes concentrés sur deux points particuliers : le coût du découpage en composants et celui des opérations d'introspection.

L'ensemble de ces tests a été réalisé sur une même machine munie d'un Pentium IV cadencé à 2,4 GHz qui exécute la version 2.6 du système d'exploitation Linux. Les mesures que nous présentons ici sont calculées à partir de la moyenne des valeurs obtenues pour 50 évaluations successives ; à aucun moment l'erreur standard associée à ces moyennes n'a dépassé 1 %. Les temps d'exécution des procédures ont été calculés à partir d'une boucle qui les répète un million de fois chacune : le temps total exprimé en millisecondes indique alors le coût d'une itération, exprimé en nanosecondes.

Pour évaluer le coût du découpage en composants, nous avons mesuré le temps de traversée d'un composant, c'est-à-dire le temps nécessaire à la transmission d'un événement d'un composant à son successeur. Les résultats présentés figure 2(a) montrent que ce temps est d'environ 50 ns. D'après les autres mesures que nous avons effectuées, nous voyons que ce temps est certes supérieur au temps nécessaire à l'appel de fonctions de bibliothèques simples mais est inférieur, par exemple, à celui utilisé pour effectuer une opération flottante ou un appel système. Ceci nous indique donc que pour des applications non triviales, le surcoût lié au découpage en composants reste très limité, voire négligeable.

Pour surveiller son propre comportement, une application autonome examine en permanence les capteurs dont elle dispose. Les modifications ne sont en effet supposées intervenir que lors de circonstances exceptionnelles. C'est donc l'opération de lecture d'un capteur qui est la plus critique en termes de performance, et c'est elle que nous avons donc choisi d'évaluer. À titre de comparaison, nous avons mesuré le temps nécessaire à la lecture d'une variable en utilisant les interfaces réflexives de deux langages couramment utilisés dans la construction d'applications autonomes : Java et C#. L'opération effectuée consiste ici à lire la valeur d'un champ entier d'un objet de la manière la plus simple (le code tient, dans les deux cas, en une seule ligne). Pour Java nous avons utilisé différentes machines virtuelles, avec et sans compilation dynamique (*Just In Time*) ; nous avons également compilé le programme en code natif en nous servant du compilateur GNU [BOT 04]. Pour C#, c'est l'environnement Mono [ICA 04] que nous avons employé, dont la machine virtuelle supporte à la fois la compilation dynamique et statique (avant l'exécution). Les résultats de ces différents tests sont présentés figure 2(b). Plutôt que de nous étendre sur les performances relatives des compilateurs ou des machines virtuelles les uns par rapport aux autres, ce sont les ordres de grandeurs qui nous intéressent ici. Il apparaît que l'utilisation des capteurs — en mode passif — de Pandora (*cf.* § **??**) est environ 20 fois plus rapide que le code Java compilé. Le temps d'exécution de ce dernier correspond en revanche à celui requis lorsque l'on utilise les options de Pandora au travers de l'interface externe. L'utilisation d'une machine virtuelle dégrade encore un peu les performances et l'absence de compilation dynamique les rend tout à fait catastrophiques. Ceci montre donc que l'utilisation de Pandora permet, à charge égale, de supporter simultanément un nombre bien supérieur (un ou deux ordres de grandeur) de capteurs et donc d'applications, comparés aux langages traditionnellement utilisés dans ce domaine. C'est la spécialisation de Pandora dans ces tâches (la plate-forme a été spécialement conçue et optimisée pour cela) par opposition à la nécessaire généricité d'un langage de programmation qui explique ces différences.

7. Conclusion

Nous avons présenté Pandora, une plate-forme pour la construction d'applications autonomes. Pandora s'appuie sur un mode de programmation original, l'empilage de composants, qui offre un bon compromis entre flexibilité et performance. En effet, le modèle de composants de Pandora qui en résulte est nettement plus simple que les approches les plus couramment utilisées qui procèdent par appel de procédure. Le langage de description d'architecture que nous avons conçu permet de définir les assemblages de composants de façon compacte et claire pour les utilisateurs de la plate-forme. L'architecture du système est organisée autour d'un micro-noyau qui expose une interface réflexive et propose les abstractions nécessaires au programmeur d'application pour permettre la configuration et reconfiguration du système tout entier. Cette architecture a été mise en œuvre au travers d'un prototype fonctionnel — disponible librement sur Internet — qui a déjà eu de nombreuses applications. Enfin, une évaluation des performances du système a montré que sa mise en œuvre soutenait notre objectif initial de concilier flexibilité et performance.

8. Bibliographie

[AGA 03] AGARWAL M., BHAT V., LI Z., LIU H., KHARGHARIA B., MATOSSIAN V., V.PUTTY, SCHMIDT C., ZHANG G., HARIRI S., PARASHAR M., « AutoMate : Enabling Autonomic Applications on the Grid », *Proceedings of the Autonomic Computing Workshop (AMS 2003)*, Seattle, WA, juin 2003.

[BOT 04] BOTHNER P., HALEY A., LEVY W. et al., « The GNU Compiler for the Java Programming Language », software, 2004, `http://gcc.gnu.org/java/`.

[CLA 01] CLARKE M., BLAIR G. S., COULSON G., PARLAVANTZAS N., « An Efficient Component Model for the Construction of Adaptive Middleware », *Lecture Notes in Computer Science*, vol. 2218, 2001.

[ELL 02] ELLSON J., GANSNER E., KOUTSOFIOS L., NORTH S. C., WOODHULL G., « Graphviz — Open Source Graph Drawing Tools », *Lecture Notes in Computer Science*, vol. 2265, 2002.

[FES 03] FESSANT F. L., PATARIN S., « MLdonkey, a Multi-Network Peer-to-Peer File-Sharing Program », Research Report n° RR-4797, avril 2003, INRIA.

[HAL 03] HALL R. S., CERVANTES H., « Gravity : supporting dynamically available services in client-side applications », *Proceedings of the 9th European software engineering conference held jointly with 10th ACM SIGSOFT international symposium on Foundations of software engineering*, ACM Press, 2003, p. 379–382.

[HEI 91] HEIDEMANN J. S., « Stackable Layers : An Architecture for File System Development », rapport n° UCLA-CSD 910056, juillet 1991, University of California, Los Angeles, CA (USA).

[HUT 91] HUTCHINSON N. C., PETERSON L. L., « The *x*-Kernel : An Architecture for Implementing Network Protocols », *IEEE Transactions on Software Engineering*, vol. 17, n° 1, 1991, p. 64–76, IEEE Computer Society.

[ICA 04] DE ICAZA M., MOLARO P., PRATAP R., PORTER D. et al., « The Mono project », software, 2004, http://www.go-mono.com/.

[JAF 02] JAFFER A., CARRETTE G., STACHOWIAK M. et al., « Guile, Project GNU's extension language », software, 2002, http://www.gnu.org/software/guile/.

[KEP 03] KEPHART J., CHESS D., « The Vision of Autonomic Computing », *Computer Magazine, IEEE*, , 2003.

[KIC 97] KICZALES G., LAMPING J., MENDHEKAR A., MAEDA C., LOPES C., LOINGTIER J.-M., IRWIN J., « Aspect-Oriented Programming », AKSIT M., MATSUOKA S., Eds., *ECOOP'97 — The 11th European Conference on Object-Oriented Programming*, vol. 1241 de *Lecture Notes in Computer Science*, Jyväskylä, Finland, juin 1997, Springer-Verlag, p. 220–242.

[KOH 00] KOHLER E., MORRIS R., CHEN B., JANNOTTI J., KAASHOEK M. F., « The Click modular router », *ACM Transactions on Computer Systems*, vol. 18, n° 3, 2000, p. 263–297.

[MAE 87] MAES P., « Concepts and experiments in computational reflection », *ACM SIGPLAN Notices*, vol. 22, n° 12, 1987, p. 147–155.

[OGE 03] OGEL F., PATARIN S., PIUMARTA I., FOLLIOT B., « C/SPAN : a Self-Adapting Web Proxy Cache », *Proceedings of the Autonomic Computing Workshop (AMS 2003)*, Seattle, WA, juin 2003.

[PAT 03a] PATARIN S., « Pandora : support pour des services de métrologie à l'échelle d'Internet », PhD thesis, Université Pierre et Marie Curie – Paris 6, juin 2003.

[PAT 03b] PATARIN S., MAKPANGOU M., « On-line Measurement of Web Proxy Cache Efficiency », Research Report n° RR-4782, mars 2003, INRIA.

[PAT 04] PATARIN S., SALAMATIAN K., FRIEDMAN T., « The Pandora Network Monitoring Platform », mai 2004, Submitted for publication.

[PAW 02] PAWLAK R., DUCHIEN L., FLORIN G., SEINTURIER L., « JAC : un framework pour la programmation orientée aspect en Java », *L'Objet*, vol. 8, n° 4, 2002, p. 135–158, Hermes.

[PIU 01] PIUMARTA I., OGEL F., BAILLARGUET C., FOLLIOT B., « Applying the VVM kernel to Flexible Web Caches », *Proceedings of the IEEE Workshop on Hot Topics in Operating Sy stems, HOTOS-VIII*, Schloss Elmau, Germany, mai 2001, page 155.

[RIB 98] RIBLER R. L., VETTER J. S., SIMITCI H., REED D. A., « Autopilot : Adaptive Control of Distributed Applications », *Proceedings of the The Seventh IEEE International Symposium on High Performance Distributed Computing*, IEEE Computer Society, 1998, page 172.

[SMI 84] SMITH B. C., « Reflection and semantics in LISP », *Proceedings of the 11th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, ACM Press, 1984, p. 23–35.

[WEL 01] WELSH M., CULLER D. E., BREWER E. A., « SEDA : An Architecture for Well-Conditioned, Scalable Internet Services », *18th Symposium on Operating Systems Principles*, Lake Louise, Canada, octobre 2001, p. 230–243.