Concerto : gestion de ressources pour composants parallèles adaptables

Luc Courtrai, Frédéric Guidec, Yves Mahéo

Chapitre 1

Concerto : gestion de ressources pour composants parallèles adaptables

Luc Courtrai, Frédéric Guidec et Yves Mahéo (Valoria, Université de Bretagne Sud)

1.1 Introduction

Le *Grid Computing* se propose de fournir un moyen d'exploiter au sein d'une même application des ressources de calcul et de stockage réparties, éventuellement disséminées sur de grandes distances. Parmi les ressources qui peuvent faire l'objet d'une telle exploitation figurent les grappes, que l'on retrouvent de plus en plus fréquemment dans les entreprises et les laboratoires. Ces architectures revêtent des formes variées. Outre les grappes dédiées qui font souvent office de super-calculateurs de faible coût, on peut aussi utiliser de simples groupes de stations de travail interconnectées par un réseau local performant. Dans un contexte de grande disponibilité de ces plates-formes, on peut envisager l'exécution d'applications parallèles tirant parti de plusieurs grappes. Mais cet objectif reste difficile à atteindre étant donné le manque d'infrastructure logicielle adéquate.

Diverses approches sont envisageables pour concevoir et déployer une application capable d'exploiter une ou plusieurs grappes de stations de travail. Parmi celles-ci, l'approche par composants [13] mérite d'être considérée avec attention. Elle permet en effet d'envisager le développement d'applications complexes par simple assemblage de composants pré-existants, chacun de ces composant étant conçu comme un «code parallèle» destiné à être déployé sur une grappe.

Même s'il est possible de concevoir un composant de manière *ad hoc* afin qu'il exploite au mieux une architecture spécifique, il est en général préférable de privilégier la portabilité des composants. Chaque composant devrait donc idéalement être déployable sur une grande variété de grappes. Pour ce faire on peut envisager d'étendre la notion de machine virtuelle à la grappe toute entière, afin de fournir en quelque sorte l'abstraction d'une « grappe virtuelle » présentant une interface homogène et masquant les spécificités des diverses architectures matérielles sous-jacentes. Avec une telle approche on peut espérer développer des composants parallèles pouvant être déployés sans que le programmeur du composant, l'administrateur de la grappe (s'il existe), ou l'utilisateur final aient à se soucier des spécificités présentées par la plate-forme matérielle visée. Une approche alternative, que nous étudions, consiste au contraire à renforcer l'adaptabilité du composant afin de lui donner les moyens de tenir compte des spécificités matérielles et logicielles de la grappe sur laquelle on le déploie.

L'adaptation du composant peut revêtir plusieurs formes (auto-adaptation du composant, adaptation dirigée par la plateforme d'accueil,...). Dans tous les cas, il est nécessaire de disposer de mécanismes de base qui permettent d'alimenter la décision d'adaptation en fonction de l'environement du composant.

S'il a la possibilité d'obtenir des informations concernant l'architecture cible, le composant adaptatif pourra choisir une configuration appropriée lors de son déploiement. L'état de la plate-forme peut être exprimé sous la forme d'informations qualitatives et quantitatives couvrant des aspects tels que le nombre de nœuds constituant la plate-forme, la puissance de ces nœuds, la bande passante théorique des liens de communication, la présence d'un équipement périphérique donné (scanner, imprimante, lecteur de bande, etc.), ou encore la disponibilité d'une bibliothèque logicielle spécifique (bibliothèque de communication, de calcul numérique, etc.). Mais cette configuration initiale du composant peut se révéler insuffisante. Dans la mesure où la plate-forme sur laquelle il s'exécute ne lui est pas dédiée, le composant est en effet susceptible de partager des ressources avec d'autres composants, voire avec d'autres applications. Il est donc possible que les conditions d'exécution identifiées lors du déploiement du composant ne soient plus les mêmes par la suite. Il faut donc fournir au composant le moyen d'obtenir tout au long de son exécution des informations relatives à son environnement du moment, informations sur lesquelles il pourra s'appuyer pour s'adapter, en redistribuant par exemple des données, en répartissant différemment la charge de travail, ou en choisissant un nouvel algorithme. Il s'agit de fournir au composant des informations dynamiques relatives, par exemple, à la charge observée au niveau du CPU d'un nœud particulier de la grappe, à la bande passante disponible sur un lien, etc.

Nous décrivons dans ce chapître la plate-forme logicielle Concerto, qui permet de déployer des composants parallèles sur une grappe tout en fournissant à ces composants les moyens de s'adapter. Cette plate-forme est dédiée au déploiement

et au support de composants logiciels parallèles écrits en Java. Les informations susceptibles d'alimenter les décisions d'adaptation sont issues de l'observation des ressources. Le terme de *ressource* doit ici être compris dans un sens large. Parmi les ressources envisagées, on va ainsi trouver des ressources offertes par le système (mémoire disponible, CPU, bibliothèques de calcul numérique ou de communication, etc.) et des ressources «conceptuelles », liées à l'application elle-même (sockets et threads utilisés dans l'application, etc.). L'objectif est de fournir des mécanismes permettant de réaliser la collecte d'informations relatives à un ensemble non limité de ces ressources. L'infrastructure que nous proposons est extensible dans la mesure où elle est conçue de manière à pouvoir incorporer facilement de nouveaux types de ressources au fur et à mesure que les besoins s'en font sentir. La plate-forme Concerto pourra ainsi évoluer au fil du temps de manière à prendre en compte les spécificités de nouvelles plates-formes matérielles.

Le reste de ce document est organisé comme suit. Le paragraphe 1.2 introduit le modèle basique de composant parallèle que nous proposons, et décrit les mécanismes mis en œuvre dans la plate-forme Concerto afin de supporter le déploiement de tels composants. La démarche adoptée pour modéliser les ressources au sein de la plate-forme et les mécanismes mis en œuvre pour observer l'état de ces ressources sont présentés dans le paragraphe 1.3. Le paragraphe 1.4 résume le travail réalisé jusqu'à ce jour et évoque quelques-unes des perspectives ouvertes.

1.2 Composants parallèles

La définition de composants pour le développement d'applications est possible à travers l'utilisation de modèles de composants issus de l'industrie tels que COM de Microsoft [10] ou les *Enterprise Java Beans* de Sun [5]. L'OMG propose pour sa part le *Corba Component Model* [11]. Ces modèles ne sont toutefois pas conçus pour supporter des composants parallèles, c'est-à-dire des composants mettant en jeu des activités parallèles. Quelques travaux préliminaires ont permis de proposer des pistes de modèles ou de plates-formes supportant des composants parallèles. Ceux-ci visent essentiellement la réutilisation de codes de calculs intensifs fondés sur le parallélisme de données. On peut citer l'initiative du *Common Component Architecture Forum* visant notamment la définition d'une API standard pour permettre la définition de ports garantissant l'interopérabilité de composants [1]. Par ailleurs, une approche étendant le CCM pour prendre en compte des collections de composants séquentiels identiques est présentée dans [12]. Dans ces travaux, l'accent n'est pas mis sur l'adaptabilité des composants, mais sur la performance des communications devant être effectuées en parallèle lorsque deux composants parallèles interopèrent.

La plate-forme Concerto est dédiée à l'accueil de composants parallèles adaptatifs. Bien que le concept de composant parallèle soit au cœur de notre projet, notre

objectif n'est pas de proposer un nouveau modèle de composant, mais de fournir une infrastructure favorisant l'adaptation des composants. Pour travailler dans cette optique nous nous contentons de proposer une définition minimale, développée ci-après, de ce qu'est un composant parallèle dans Concerto.

Le programmeur désirant développer un composant pour la plate-forme Concerto doit concevoir celui-ci comme un ensemble de threads Java coopérants. Il doit en outre définir lui-même la partie métier du composant (nom, interface, mise en œuvre). La plate-forme lui offre cependant des mécanismes utiles pour la gestion des aspects non fonctionnels du composant.

1.2.1 Interfaces du composant

Le composant parallèle accueilli par la plate-forme Concerto possède trois interfaces :

- Interface « métier » : Aucune hypothèse n'est formulée sur le type d'interface métier du composant. Le programmeur de composant peut par exemple proposer une interface métier construite sur les services RMI Java. Le composant est alors un objet implantant l'interface *Remote*, dont les méthodes pourront être invoquées à distance par les clients. Le composant peut aussi être un serveur à l'écoute d'un port de la machine sur lequel les clients doivent ouvrir une socket. En outre, il peut proposer une interface métier distribuée (*i.e.* associés à plusieurs objets implantant chacun une partie de l'interface) afin de pouvoir être interconnecté en parallèle avec un autre composant parallèle.

- Interface « cycle de vie » : À travers cette interface, on peut contrôler les différentes étapes de la vie du composant. À l'heure actuelle, ceci concerne essentiellement le déploiement du composant sur la grappe, et son arrêt. Il devrait être possible à l'avenir de distinguer plusieurs phases dans le déploiement, et de proposer des services de sauvegarde et de restauration de l'état du composant.

- Interface « ressource » : Le composant comporte une interface ressource à travers laquelle on accède aux informations relatives aux ressources utilisées par le composant. Plus précisement, le composant est considéré comme une ressource dans la plate-forme Concerto et, à ce titre il est réifié et possède une méthode *observe()* retournant un rapport d'observation le concernant (ces aspects sont abordés plus en détails dans le paragraphe 1.3). Le rapport d'observation généré par un composant est, par défaut, constitué par agrégation de rapports concernant toutes les ressources utilisées par ce composant. Le programmeur du composant peut cependant, s'il l'estime nécessaire (pour des raisons de sécurité par exemple), modifier la portée des informations divulguées aux

clients de son composant en définissant un type de rapport d'observation propre à son composant.

1.2.2 Structure interne d'un composant

Pour construire un composant parallèle, le programmeur développe un ensemble de threads Java (en réalité un ensemble de classes implantant l'interface *Runnable*). Ces threads coopèrent pour réaliser les méthodes de l'interface métier du composant.

Les threads sont regroupés en entités de placement (ou de distribution), appelées «fragments». Un fragment est un sous-ensemble des threads d'un même composant, destinés à être placés au sein d'une même JVM sur un même nœud de la grappe. Ces threads pourront ainsi se partager un espace d'objets. La communication et la synchronisation des threads d'un même fragment s'effectuent donc comme dans n'importe quel programme Java multithreadé. En revanche, les threads appartenant à des fragments distincts doivent s'appuyer sur des mécanismes de communication et de synchronisation tels que sockets, RMI, etc.

1.2.3 Déploiement d'un composant

Pour déployer un composant sur une grappe, on doit fournir un fichier de description de déploiement de son composant. Ce fichier décrit :

- la structure du composant en termes de fragments et de threads à déployer ;
- des directives de placement des fragments. On pourra ainsi par exemple dupliquer certains fragments sur tous les nœuds, ou placer un fragment donné sur un nœud spécifique.
- les contraintes imposées par le composant pour que son déploiement soit possible (présence d'une version précise de la JVM, d'un registre RMI...).

Nous avons développé un dialecte XML permettant d'exprimer de telles directives. La plate-forme Concerto est capable d'interpréter ce dialecte afin d'assurer le déploiement et le lancement des composants sur une grappe.

1.3 Modélisation et contrôle des ressources

1.3.1 Motivation et principes généraux

L'objectif principal du projet Concerto est de fournir aux composants logiciels les moyens de percevoir leur environnement d'exécution, afin qu'ils puissent adapter leur mode de fonctionnement à l'état de cet environnement, voire aux variations observées dans cet environnement au cours de leur exécution. Dans cette optique

nous avons entrepris de développer en Java une plate-forme logicielle dans laquelle l'environnement d'exécution d'un composant est modélisé sous la forme d'objets réifiant les différentes ressources offertes par cet environnement.

De manière générale, on qualifie ici de « ressource » toute entité (matérielle ou logicielle) qu'un composant logiciel pourra être amené à utiliser au cours de son exécution. Les ressources considérées à l'heure actuelle dans la plate-forme Concerto comprennent aussi bien des ressources dites « ressources système » (processeur, mémoire, disque, interface utilisateur, interface réseau, etc.) qui caractérisent essentiellement la plate-forme matérielle sous-jacente que des ressources dites « conceptuelles » (sockets, processus, threads, répertoires, fichiers, serveur RMI, etc.) qui ressortent plutôt de l'environnement applicatif considéré.

Un composant logiciel étant susceptible d'utiliser tout ou partie des ressources disponibles dans son environnement, la plate-forme Concerto doit mettre à sa disposition des mécanismes lui permettant de :

- vérifier la présence de telle ou telle ressource (ou de tel ou tel type de ressource) dans son environnement ;
- découvrir l'existence d'une ressource (ou d'un type de ressource) dans son environnement ;
- s'informer sur l'état d'une ressource particulière ;
- demander à la plate-forme de l'informer lorsqu'une certaine condition est vérifiée sur l'état d'une ressource donnée.

La plate-forme Concerto ayant pour vocation de permettre le déploiement de composants logiciels parallèles sur une grappe de machines, elle doit tenir compte de la dissémination des ressources au sein de la grappe. Les mécanismes évoqués ci-dessus doivent donc permettre aux composants de s'abstraire des contraintes posées par la répartition des ressources. En d'autres termes, un composant doit pouvoir s'informer sur l'état des ressources disponibles sur un nœud particulier de la grappe, mais il doit aussi pouvoir collecter des informations concernant les ressources disséminées à travers l'ensemble de la grappe.

1.3.2 Modélisation des ressources

Toutes les ressources susceptibles d'être utilisées par des composants déployés sur la plate-forme Concerto doivent être réifiées en Java sous la forme d'objets. Nous avons donc entrepris de bâtir une hiérarchie de classes extensible modélisant ces ressources. Une partie cette hiérarchie est reproduite dans la figure 1.1. On peut y voir :

- des classes modélisant des ressources caractéristiques du support matériel sous-jacent comme par exemple *CPU*, *Memory*, *NetworkInterface* ou *ClusterNode* ;
- des classes standard définies dans l'API du JDK *(Java Development Kit)* comme par exemple *Socket* et *Thread*. La mise en œuvre de ces classes a été révisée dans Concerto pour permettre l'observation de l'état des ressources qu'elles modélisent.
- des classes introduites afin de réifier des notions propres au projet Concerto (*Fragment* et *Component*).

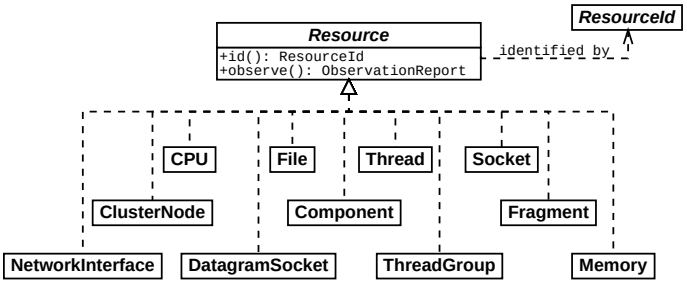

Figure 1.1 : Modélisation de quelques types de ressources

Pour que la collecte d'informations relatives à l'état des différentes ressources présentes au sein d'une grappe puisse s'effectuer de manière homogène, nous avons introduit la notion de «rapport d'observation», et développé une hiérarchie de classes Java permettant la génération, la collecte, et le traitement de tels rapports (voir figure 1.2).

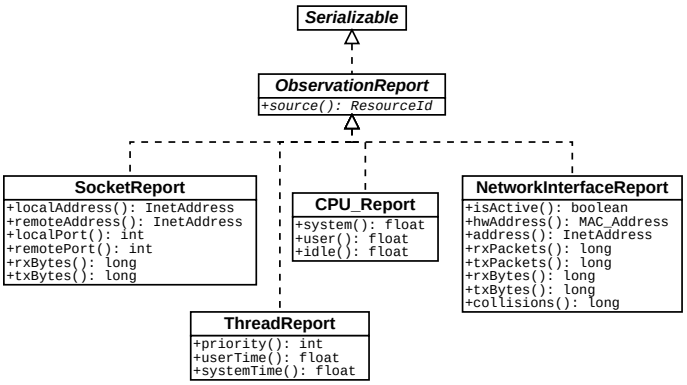

Figure 1.2 : Modélisation des rapports d'observation

Tout objet Java modélisant une ressource au sein de la plate-forme Concerto implémente la méthode *observe()*, qui permet d'obtenir de la ressource considérée

un rapport relatif à son état courant. Un rapport est modélisé sous la forme d'un objet Java implémentant l'interface *ObservationReport*. Le type exact du rapport dépend bien sûr du type de ressource considéré. Ainsi lorsqu'on invoque la méthode *observe()* sur un objet *Thread*, celui-ci retourne un rapport d'observation de type *ThreadReport*, la classe *ThreadReport* fournissant des informations caractéristiques de l'état de l'objet thread considéré (niveau de priorité courant, quantités de CPU et de mémoire consommées depuis le lancement du thread, etc.). L'invocation de la méthode *observe()* sur un objet *Memory* retournera de même un rapport de type *MemoryReport*, ce rapport indiquant l'état de la mémoire du système lors de l'appel.

1.3.3 Identification, localisation et classification des ressources

Dans la plate-forme Concerto, toutes les ressources sont modélisées sous la forme d'objets Java. Comme de tels objets peuvent être créés et détruits à tout instant et qu'il importe malgré tout de pouvoir identifier chaque ressource sans ambiguïté, la plate-forme met en œuvre un système d'identification et de localisation des ressources s'appuyant sur un schéma de nommage assurant l'unicité des identifiants attribués aux objets ressources. Dès la création d'un objet ressource au niveau d'un nœud quelconque de la grappe, cet objet se voit attribuer un identifiant unique (objet de type *ResourceId*, voir figure 1.1). En outre, l'objet ressource ainsi créé est immédiatement enregistré auprès d'un gestionnaire de ressources (objet de type *ResourceManager*), dont la fonction est d'identifier et de permettre le suivi des ressources existantes.

Une instance de la classe *ResourceManager* est créée sur chaque nœud de la grappe à chaque fois que l'on déploie un nouveau composant. La fonction de ce gestionnaire de ressources est de permettre l'identification, la localisation, et la collecte de rapports d'observation sur l'ensemble des ressources de la grappe, c'est-à-dire :

- des ressources conceptuelles utilisées par le composant auquel il est associé ;
- des ressources système de la grappe (considérées comme des ressources globales partagées entre tous les composants) ;
- des autres composants déployés sur la grappe (on rappelle que chaque composant est perçu comme une ressource, et peut donc produire sur demande un rapport d'observation le concernant).

La notion de «motif de recherche» a été introduite pour pouvoir cibler la recherche de ressources et la collecte de rapports d'observation. L'objectif est ici de pouvoir décrire sous la forme d'objets Java diverses stratégies de recherche, telles que par exemple la recherche localisée (ie limitée à un nœud précis de la grappe), ou bien encore la recherche globalisée (ie réalisée sur l'ensemble des nœuds de la grappe). L'interface *SearchPattern*, définie à cet effet, est destinée à servir de racine à une

arborescence de classes décrivant chacune une stratégie de recherche d'informations particulière (voir figure 1.3).

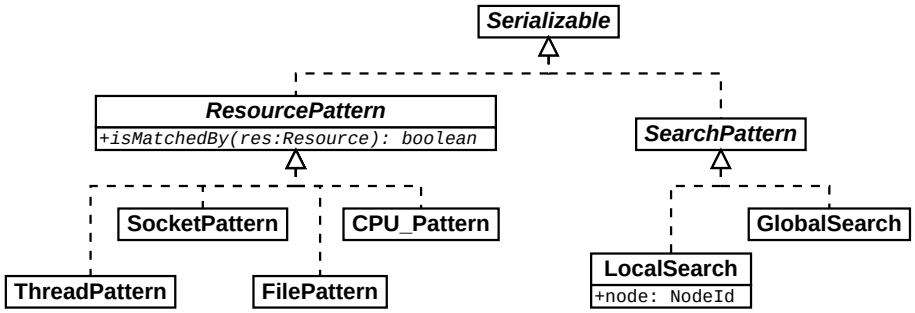

Figure 1.3 : Modélisation des « motifs » servant à la sélection des ressources (partie gauche) et à la description des stratégies de recherche (partie droite).

L'extrait de code suivant illustre le schéma de consultation d'un gestionnaire de ressources. On utilise ici des motifs de recherche afin de préciser que la recherche d'identifiants doit porter (1) sur les ressources locales exclusivement ; (2) sur les ressources recensées sur un nœud distant dont l'identité est passée en paramètre ; (3) sur l'ensemble de la grappe.

```
ResourceManager manager = ResourceManager.getManager();
Set localIds = manager.getResourceIds(new(LocalSearch()); // (1)
Set remoteIds = manager.getResourceIds(LocalSearch(remoteNodeId)); // (2)
Set allIds = manager.getResourceIds(GlobalSearch()); // (3)
```

Connaissant l'identifiant d'un objet ressource quelconque, on peut obtenir du gestionnaire de ressources qu'il collecte un rapport d'observation concernant cet objet précis (que celui-ci soit local ou distant) et nous retourne le rapport ainsi obtenu. Ainsi, l'extrait de code ci-dessous poursuit l'exemple précédent, en illustrant la manière selon laquelle on peut demander au gestionnaire de ressources de nous retourner un rapport d'observation concernant une ressource précise (on supposera ici que la valeur de l'identifiant *resId* a été extraite de l'un des trois ensembles d'identifiants collectés dans l'exemple précédent).

```
[...]
ObservationReport report = manager.observe(resId);
```

Les ressources enregistrées auprès des gestionnaires de ressources pouvant être de natures diverses (*CPU*, *Memory*, *Socket*, *Thread*, *File*, etc.), la plate-forme Concerto met en œuvre un mécanisme de classification et de sélection des ressources basé sur la notion de « motif de ressource ». L'interface *ResourcePattern* (voir figure 1.3) définit une fonction *isMatchedBy()*, qui prend en paramètre un objet ressource et retourne

une valeur booléenne indiquant si cet objet vérifie ou non le critère de sélection considéré. La sélection peut simplement porter sur le type effectif de l'objet ressource soumis au test. Ainsi, dans la classe *ComponentPattern* implémentant l'interface *ResourcePattern*, la méthode *isMatchedBy()* vérifiera simplement si l'objet considéré est ou non de type *Component*). La sélection peut également concerner des valeurs d'attributs des objets ressources. Le motif *socketPattern* présenté ci-dessous pourra servir à sélectionner les ressources de type *Socket* satisfaisant aux contraintes suivantes : l'adresse de l'hôte distant doit appartenir au réseau IP de classe C 195.83.160/24, et le numéro du port distant visé doit se situer entre 0 et 1023. En revanche l'adresse IP de l'interface locale, tout comme le numéro du port local servant à établir la connexion, peuvent ici être quelconques.

```
ResourcePattern componentPattern = new ComponentPattern();
ResourcePattern socketPattern =
    new SocketPattern(InetAddress.AnyAddress, 195.83.160/24,
        PortRange.AnyPort, new PortRange(0, 1023));
[...]
```

Parmi les diverses méthodes qui permettent de consulter le gestionnaire de ressources évoqué précédemment, on dispose de méthodes qui prennent en paramètre un objet de type *ResourcePattern*. On peut donc interroger le gestionnaire de ressources en lui demandant de retourner les identifiants des ressources dont les caractéristiques satisfont le « motif » de sélection indiqué. Si ce motif est, par exemple, l'objet de type *ComponentPattern* créé dans l'exemple précédent, alors le gestionnaire de ressources retournera exclusivement les identifiants des composants déployés sur la grappe. S'il s'agit au contraire du *SocketPattern* créé ci-dessus, alors le gestionnaire de ressources cherchera les sockets ouverts par le composant englobant et dont les caractéristiques (adresses IP, numéros de ports, etc.) satisfont ce motif.

```
[...]
ResourceManager manager = ResourceManager.getManager();
Set componentIds = manager.getResourceIds(componentPattern);
Set socketIds = manager.getResourceIds(socketPattern);
```

1.3.4 Mise en œuvre

Les divers mécanismes utilisés dans Concerto afin de modéliser les ressources et de permettre leur observation ont un champ applicatif qui dépasse celui de la programmation de composants parallèles adaptatifs. Ces mécanismes ont été regroupés sous l'appellation RAJE *(Resource-Aware Java Environment)*. L'environnement RAJE concentre les mécanismes permettant la réification et l'observation des ressources système. Il définit en outre les principaux schémas d'identification, de localisation, et d'observation des ressources. La plate-forme Concerto s'appuie sur cette infrastructure, et l'étend afin d'y intégrer des notions qui lui sont propres, comme par exemple les notions de composant parallèle et de fragment.

L'environnement RAJE, tout comme la plate-forme Concerto, sont actuellement mis en œuvre sous Linux, et s'appuient sur une variante de la JVM Kaffe 1.0.6. Des détails sur l'environnement RAJE peuvent être trouvés dans [7]. Cet article décrit en particulier les mécanismes mis en œuvre afin d'observer l'état des ressources du système et d'évaluer la part de ces ressources consommée par chaque thread Java. L'article [9] évoque également l'environnement RAJE (ainsi que la plate-forme JAMUS bâtie au dessus de cet environnement). Les services fournis par RAJE y sont notamment comparés à ceux offerts par d'autres outils tels que JRes [4], GVM [3], KaffeOS [2]. Naccio [6] Ariel [8], etc.

1.4 Conclusion

La plate-forme Concerto doit permettre le déploiement et le support de composants parallèles adaptatifs sur des grappes de stations de travail. Les travaux en cours visent à proposer un modèle basique de composant parallèle, et fournir les outils pour gérer le déploiement de ce type de composant. Notre objectif est d'adopter dans un premier temps un modèle aussi simple et aussi peu contraignant que possible, l'accent étant mis sur le développement de mécanismes favorisant l'adaptation des composants. La plate-forme Concerto doit en effet permettre le déploiement de composants parallèles sur des grappes non dédiées, constituées par exemple d'ensembles de stations de travail partagées entre plusieurs composants, voire avec d'autres applications et utilisateurs. L'environnement d'exécution offert à un composant parallèle étant par nature hétérogène et susceptible de varier au cours de l'exécution du composant, nous proposons un schéma permettant aux composants de percevoir leur environnement d'exécution, ainsi que les variations subies par cet environnement. L'environnement des composants est assimilé à un ensemble de ressources, dont chaque composant peut découvrir l'existence ou observer l'état par le biais des services fournis par la plate-forme.

Le développement de la plate-forme Concerto n'est pas achevé. Outre la finalisation de l'outil de déploiement, nous prévoyons de mettre en place des mécanismes d'interaction permettant aux composants de demander à la plate-forme de les informer des changements d'état observés sur certaines ressources. Ceci implique la définition d'un formalisme permettant aux composants de décrire les événements qui les intéressent, et le développement d'un schéma de notification d'événements tenant compte du caractère distribué des composants.

Bibliographie

[1] R. Amstrong, D. Gannon, A. Geist, K. Keahey, S. Kohn, L. McInnes, S. Parker, and B. Smolinski. Towards a Common Component Architecture for High-

Performance Scientific Computing. In *Proc. of the 8th International Symposium on High-Performance Computing*, Redondo Beach, Californie, August 1999.

[2] G. Back, W. C. Hsieh, and J. Lepreau. Processes in KaffeOS: Isolation, Resource Management, and Sharing in Java. In *4th Symposium on Operating Systems Design and Implementation*, October 2000.

[3] G. Back, P. Tullmann, L. Stoller, W. C. Hsieh, and J. Lepreau. Techniques for the Design of Java Operating Systems. In *USENIX Annual Technical Conference*, June 2000.

[4] G. Czajkowski and T. von Eicken. JRes: a Resource Accounting Interface for Java. In *ACM OOPSLA Conference*, 1998.

[5] L. DeMichiel. Enterprise JavaBeans Specification, Version 2.1. Rapport technique, Sun Microsystems, June 2002.

[6] D. Evans and A. Twyman. Flexible Policy-Directed Code Safety. In *IEEE Security and Privacy*, May 1999.

[7] F. Guidec and N. Le Sommer. Towards Resource Consumption Accounting and Control in Java: a Practical Experience. In *ECOOP'2002, Workshop on Resource Management for Safe Languages (Malaga, Spain)*, June 2002. URL: http://www.univ-ubs.fr/valoria/Orcade/RASC/Publications/ECOOP2002-WS01.pdf.

[8] M. B. Jones, P. J. Leach, R. P. Draves, and J. S. Barrera. Modular Real-Time Resource Management in the Rialto Operating System. In *5th Workshop on Hot Topic in Operating System (HotOS-V)*, May 1995.

[9] N. Le Sommer and F. Guidec. A Contract-Based Approach of Resource-Constrained Software Deployment. In *Proc. of the First International IFIP/ACM Working Conference on Component Deployment (CD'2002, Berlin, Germany)*, number 2370 in LNCS, pages 15–30. Springer, June 2002. ISBN 3-540-43847-5.

[10] Microsoft. The Component Object Model Specification. Rapport technique, Microsoft Corporation, October 1995.

[11] OMG. CORBA Components. Rapport technique OMG-orbos-99-07-01, OMG TC Documents, July 1999.

[12] A. Ribes. Vers l'utilisation de composants parallèle pour le couplage de codes de calcul scientifique. In *Actes des 14e Rencontres francophones sur le parallélisme, Renpar'14*, Hammamet, Tunisie, April 2002.

[13] C. Szyperski. *Component Software: Beyond Object-Oriented Programming.* ACM Press, Addison-Wesley, 1998.