RENPAR'16 / CFSE'4 / SympAAA'2005 / Journées Composants
Le Croisic, France, 5 au 8 avril 2005

Distribution d'un composant hiérarchique dans un environnement partiellement connecté

Didier Hoareau – Yves Mahéo

Laboratoire Valoria – Université de Bretagne Sud
Campus de Tohannic, 56017 Vannes cedex
{Didier.Hoareau,Yves.Maheo}@univ-ubs.fr

Résumé
Cet article aborde l'utilisation de l'approche à composants pour la construction et l'exécution d'une application distribuée censée offrir ses services sur un ensemble d'équipements hétérogènes, potentiellement volatiles. Nous proposons d'exploiter un modèle de composant hiérarchique et présentons une méthode de distribution d'un composant rendant accessible a priori partout l'ensemble de ses interfaces. Le support d'exécution associé à ce modèle permet de répercuter les déconnexions des équipements sur l'architecture du composant en rendant inactives certaines de ses interfaces tout en autorisant un fonctionnement dégradé du composant.

Mots-clés : Composants hiérarchiques, composants distribués, adaptation au contexte, déconnexions

1. Introduction

La multiplication des équipements informatiques nomades comme les assistants personnels ou les téléphones mobiles, utilisés conjointement aux traditionnelles stations de travail, font émerger de nouveaux besoins applicatifs. Un des défis posés consiste à être capable de mettre en œuvre des applications distribuées qui puissent prendre en compte l'hétérogénéité et la dynamicité de ces nouveaux équipements. En effet, la construction et le déploiement d'applications pour ces plates-formes nous impose d'une part de considérer les spécificités logicielles et matérielles des différents équipements sur lesquels vont être déployées les applications et d'autre part de supporter la dynamicité (plus ou moins forte) de l'environnement d'exécution, dynamicité notamment induite par la volatilité des équipements et des connexions.
Les modèles de composants existants comme DCOM [13], EJB [14], CCM [15] et l'approche par composant de façon plus générale ont montré leur intérêt dans la conception, la construction et la maintenance d'applications distribuées. Ils permettent de concevoir une application comme une interconnexion de composants accessibles via des interfaces bien définies, le tout formant une architecture plus complexe. Mais la plupart des modèles et intergiciels associés se placent dans le cadre du développement d'applications visant traditionnellement des réseaux de stations de travail et font généralement des hypothèses relativement fortes sur la stabilité de la plate-forme d'exécution (disponibilité permanente d'un composant serveur par exemple) et supposent que chacun des équipements offre des ressources suffisantes. Les applications ainsi conçues ne peuvent pas dans la plupart des cas être installées ou exécutées sur des réseaux de machines à ressources parfois limitées et potentiellement volatiles.
Nous nous intéressons dans cet article à l'utilisation de l'approche à composants pour la construction et l'exécution d'une application censée offrir ses services pour un ensemble d'équipements mobiles présentant éventuellement de faibles ressources matérielles. Les différentes parties constituant une telle application ne sont pas installées sur tous les équipements mais sur chacun d'entre eux, cette application doit pouvoir être utilisée ; un fonctionnement en mode dégradé survient lorsque les différentes parties de l'application ne sont pas toutes accessibles.
Dans cette optique, nous proposons d'utiliser un modèle de composants hiérarchiques, dans lequel un composant peut être lui-même un assemblage de composants. L'approche communément utilisée pour distribuer une application à base de composants consiste à considérer que chaque instance de composant

est installée sur une machine, la distribution faisant référence au fait qu'un composant a la possibilité d'invoquer à distance les services mis en œuvre par un autre composant. Nous proposons pour notre part de tirer parti de l'aspect hiérarchique de l'architecture des composants pour permettre qu'un composant composite soit accessible sur plusieurs machines bien que ses sous-composants soit instanciés sur un seul hôte.
Dans ce contexte, la volatilité des équipements et des connexions se traduit par la rupture temporaire de certaines liaisons entre sous-composants. Pour éviter que l'application toute entière deviennent inutilisable, le support d'exécution met en œuvre des mécanismes permettant de détecter ces ruptures et de rendre inactives certaines interfaces d'un composant, les autres interfaces restant actives afin que le composant puisse continuer à assurer une partie de ses fonctions. Le support d'exécution que nous avons réalisé autorise l'introspection sur le fait qu'une interface soit active ou pas, permettant ainsi d'envisager le développement d'applications s'adaptant aux déconnexions des équipements.
Cet article est organisé de la manière suivante : nous présentons dans un premier temps le modèle de composants hiérarchiques que nous avons adopté et détaillons comment la distribution d'un composant hiérarchique peut être effectuée. Dans un deuxième temps, la notion d'interface active est abordée et nous précisons comment elle est mise en œuvre dans le support de notre modèle de composants hiérarchiques distribués. La section suivante donne quelques détails du prototype réalisé à partir du modèle Fractal. Nous concluons enfin en mentionnant les travaux connexes et les pistes que nous prévoyons de poursuivre pour compléter ce travail.

2. Vers un composant hiérarchique distribué

2.1. Composants hiérarchiques

Dans un modèle de composants hiérarchiques, les composants composites servent à avoir une vue uniforme d'une application suivant différents niveaux d'abstraction. Ainsi, un composant composite représente une structure plus ou moins complexe de composants interconnectés – décrite par une configuration stockée dans un descripteur d'architecture – et peut donc être utilisé comme un simple composant avec des interfaces requises et fournies bien définies. La récursivité s'arrête avec les composants primitifs, correspondant à des unités de traitement. Les composants sont interconnectés par des liaisons (*binding*) qui représentent chacune une référence (locale ou distante) entre une interface requise et une interface fournie.
La notion de composant composite est fréquemment utilisée au moment de la conception de l'application et se retrouve dans les langages de description d'architecture [12] qui permettent de préciser les composants constituant une application ainsi que leurs interactions. Dans le cadre applicatif que nous nous sommes fixé, il est cependant intéressant de pouvoir également manipuler un composite à l'exécution. Cela permet en effet de faciliter les mécanismes d'adaptation dynamique dont les objectifs sont de pouvoir ajouter, retirer, remplacer un sous-composant et redéfinir les liaisons entre composants, la structure hiérarchique permettant cette fois la prise en compte des différents niveaux d'abstraction à l'exécution.
Par rapport à d'autres modèles de composants hiérarchiques comme Darwin [9] ou Koala [16], Fractal [3] offre la plupart des caractéristiques que nous recherchons. Nous nous appuierons sur ce modèle de composants hiérarchiques dans la suite de cet article.
Le modèle Fractal définit un composant comme étant constitué de deux parties : une membrane (ou contrôleur) et un contenu. La membrane expose les interfaces (requises et fournies) des composants et intercepte toutes les invocations au niveau des interfaces. Un composant composite est défini comme étant un composant exposant son contenu, constitué d'un ensemble de sous-composants. À chaque interface fournie et requise d'un composant composite correspond une interface d'un sous-composant.
La composition dans Fractal se fait à travers les liaisons entre interfaces requises et fournies et par la présence de composants dans un composite. Le modèle est récursif : un composant composite peut lui-même apparaître dans le contenu d'un composant composite. La figure 1 présente l'architecture d'une application constituée d'un composant composite contenant deux sous-composants primitifs.
Fractal définit en outre une API précisant les moyens d'introspection et d'intercession supportés par le modèle ; par exemple, l'ajout et le retrait dynamique des liaisons entre composants. Ces mécanismes sont réalisés par des contrôleurs situés au niveau de la membrane et accessibles via des interfaces dites de

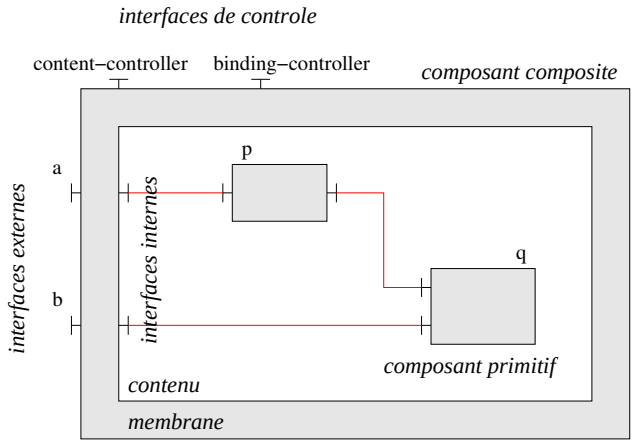

FIG. 1 – Vue interne d'un composant Fractal.

contrôle. Le modèle Fractal permet la redéfinition de ces contrôleurs et l'ajout de nouveaux contrôleurs.

2.2. Distribution d'un composant hiérarchique

Comme nous l'avons évoqué dans l'introduction, nous souhaitons déployer une hiérarchie de composants sur une plate-forme distribuée caractérisée notamment par son hétérogénéité et sa volatilité. Les composants de l'application seront répartis sur un ensemble de machines, le placement des composants étant guidé par une prise en compte des ressources matérielles et logicielles de chaque équipement. La manière dont ce placement est initialement choisi et comment il évolue est un des aspects de notre projet mais n'est pas traité dans cet article. Nous nous focalisons ici sur la description des mécanismes permettant une exécution distribuée des composants hiérarchiques.

Dans notre approche, l'architecture d'un composant est couplée à son placement et cette relation est traitée différemment suivant qu'il s'agit d'un composant composite ou d'un composant primitif. En termes de distribution, un primitif s'exécute sur un seul site alors qu'un composite est physiquement distribué sur un ensemble de machines. Ainsi, à chaque composant primitif correspond une et une seule instance. En revanche, une instance d'un composite peut être dupliquée. Dans un contexte de dynamicité et donc d'apparition de nouvelles machines, l'ensemble des sites impliqués dans la duplication des composants composites ne doit pas être considéré comme fixe. Cependant, pour simplifier, nous allons considérer dans ce papier que cet ensemble est défini explicitement dans le descripteur d'architecture. Ce dernier contient la définition des composants, leur interconnexion et les informations relatives à leur placement. À chaque composant primitif est associée une machine cible tandis que pour chaque composant composite est défini un ensemble de machines. Cet ensemble doit être un sous-ensemble de celui associé au composant composite englobant. Si pour un composite, l'ensemble des machines cibles n'est pas précisé, c'est celui du composant englobant qui est considéré.

Un composant composite c est distribué sur un ensemble de machine $\mathbb{M}$ s'il existe sur toutes les machines de $\mathbb{M}$ une instance de c. Toutes les instances de c sont créées à partir du descripteur d'architecture. À l'exécution, chaque instance de c maintient localement la configuration des sous-composants qu'elle contient. Ainsi, un composant composite c distribué sur $\mathbb{M}$ possède les propriétés suivantes :

– les interfaces fournies et requises de c sont accessibles sur toutes les machines m_i de $\mathbb{M}$. Ces interfaces sont celles définies dans le descripteur d'architecture.
– soit c un composite possédant un seul sous-composant *primitif* p. Il existe une seule machine m_i sur laquelle s'exécute p. Pour toute machine $m_j \in \mathbb{M}$ ($j \neq i$), il existe c_j une instance de c sur m_j. Chaque c_j possède une référence distance vers p (liaison distante).

Chaque instance qui représente le même composite est responsable localement de la configuration architecturale du composite. La figure 2 reprend l'exemple du composant composite de la figure 1 en présentant sa distribution sur trois sites. Si au cours de l'exécution, la machine m_1 devient inaccessible

depuis m_2 et m_3, le composant p ne peut plus être référencé. Par contre pour m_2 et m_3, la liaison vers q est toujours possible. Ainsi, sur m_2 et m_3, on peut toujours accéder aux services offerts par q par l'interface b. En effet, les interfaces du composite sont disponibles sur chaque site. Toutefois cela pose le problème d'invocation des méthodes sur ces interfaces dans le cas où les sous-composants ne sont plus accessibles. En effet, le fait d'instancier sur tous les sites le composite rend possible l'utilisation de ses interfaces fournies sur chacun de ces sites. Cependant du fait des déconnexions, pour un site donné, l'accès à un primitif distant peut être interrompu mais le composite continue d'exposer ses interfaces.

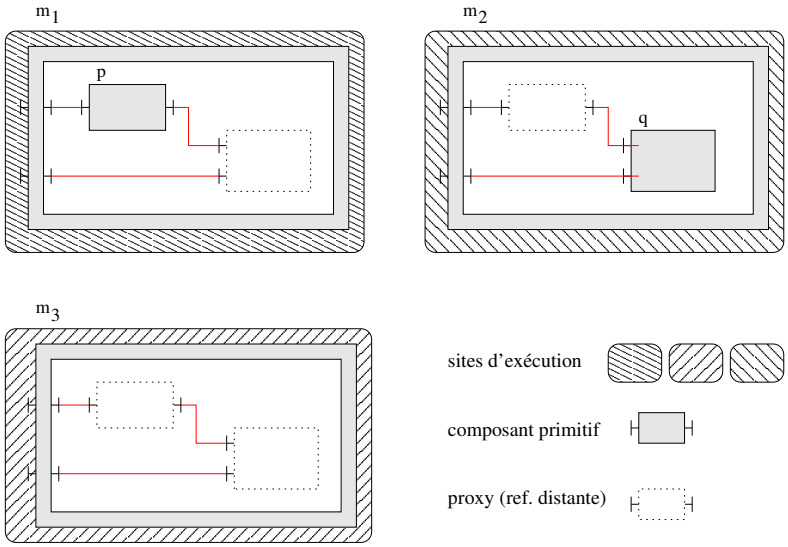

FIG. 2 – Distribution d'un composant composite sur 3 machines.

Ce problème n'est pas spécifique à notre approche mais apparaît dès que l'on utilise des références distantes qui peuvent à tout moment référencer un composant inaccessible. Afin de prévenir des invocations susceptibles de ne pas aboutir à cause d'une déconnexion, nous introduisons le concept d'état pour une interface qui peut être active ou inactive. Nous proposons par ailleurs l'ajout d'une interface de contrôle aux composants permettant l'introspection des états de leurs interfaces fournies et requises.

3. Interfaces actives

Le cycle de vie d'un composant est fortement couplé à celui des composants qu'il requiert : pour qu'un composant soit activé (*i.e.* pour que l'on puisse invoquer des méthodes de ses interfaces fournies), il est nécessaire que toutes les liaisons vers les interfaces requises soient réalisées. Dans un environnement dynamique où les déconnexions ne peuvent être écartées, cette approche peut se révéler extrêmement pénalisante. Il est souhaitable que l'ensemble d'un composant ne soit pas entièrement désactivé dans la mesure du possible même si certaines liaisons ne sont pas effectives, ceci afin que le composant continue à offrir un service (même s'il fonctionne en mode dégradé). La figure 3 montre un assemblage de composants mettant en évidence deux parties de l'architecture qui peuvent être isolées. Nous avons représenté sous forme de graphe les dépendances entre interfaces requises et fournies pour le composant composite **A**. Ainsi, si un des composants primitifs ayant ses interfaces appartenant au graphe *grA* n'est plus disponible, on peut toujours invoquer des méthodes sur l'interface a_2 sans que cela soit synonyme d'échec. Il est donc intéressant dans cette situation de laisser active l'interface a_2 : la topologie de l'architecture nous indique que les invocations sur cette interface aboutira. À l'inverse, les interfaces a_0 et a_1 doivent pouvoir être *désactivées* afin de rendre impossibles les invocations sur ces interfaces.
L'exemple précédent introduit la notion d'interface active. En effet, l'activité d'une interface représente l'état d'une interface vis-à-vis des invocations de méthodes. Une interface *active* délègue l'invocation de

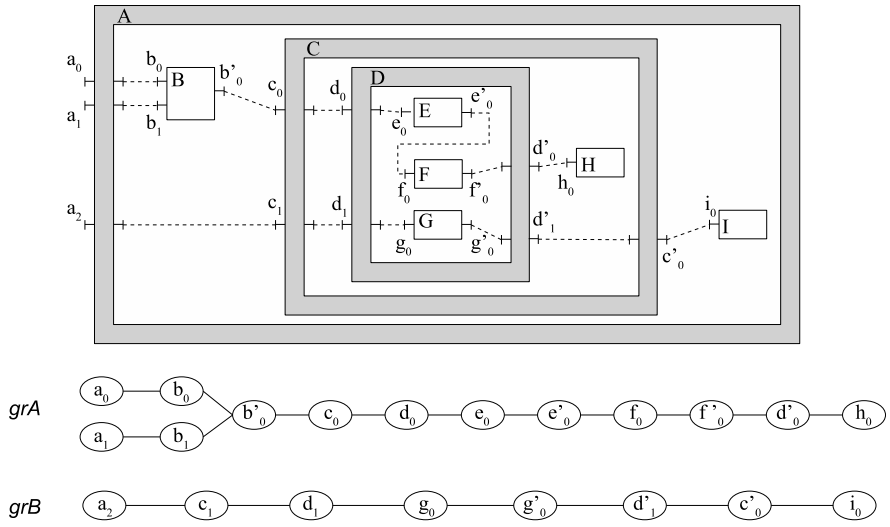

FIG. 3 – Dépendances entre interfaces fournies et requises au sein d'un composant composite. Les graphes *grA* et *grB* mettent en évidence la sous architecture du composant **A** qui peut être isolée.

la méthode au composant possédant cette interface. Une interface *inactive* ne fait aucune délégation (Il faut noter que notre définition d'une interface active diffère sensiblement de celle donnée dans [7, 6] où la notion d'activité d'une interface porte sur la possibilité de modifier son comportement).
Une interface requise est active si elle est liée à une interface fournie et que celle-ci est également active. Une interface fournie est active si :
– pour un composant primitif : toutes ses interfaces requises sont actives ;
– pour un composant composite : l'interface fournie du sous-composant correspondant est active.
Les interfaces inactives sont définies en prenant la négation des clauses précédentes. Il est intéressant de noter que pour un composant primitif, la désactivation d'une seule interface cliente a pour effet la désactivation de toutes ses interfaces fournies. Cela vient du fait qu'il n'existe a priori aucune information explicite précisant le lien entre les interfaces fournies et requises d'un composant primitif.
Il découle de notre caractérisation de l'état des interfaces fournies d'un composite que celui-ci n'est pas lié à l'état de ses interfaces requises. C'est en nous appuyant sur la structure hiérarchique du modèle que nous mettons en évidence les dépendances entre interfaces fournies et requises au sein d'un composant composite. En effet, l'activation ou la désactivation d'une interface va se propager au sein de l'architecture. Dans un environnement dynamique, les déconnexions vont se manifester par la rupture d'une liaison, ce qui va désactiver les interfaces. À l'inverse, l'apparition d'un composant de l'architecture (après une reconnexion) aura pour effet la création de liaison(s) et par conséquent l'activation d'interface(s).
Étant donné le modèle hiérarchique, nous pouvons nous appuyer sur cette structure pour *mettre à jour* l'état de l'architecture (état relatif à celui des interfaces). Lorsqu'un composant primitif active ou désactive ses interfaces fournies ou requises, cette information est propagée vers le composite englobant. Ce dernier possédant une vision de l'architecture plus large en terme d'interconnexion entre composants, va activer ou désactiver les interfaces des autres composants concernés.
Un autre exemple d'utilisation des interfaces actives (en dehors de celui lié à la prise en compte des déconnexions) est la reconfiguration d'une application par remplacement de l'implantation d'un composant par une autre. Afin de permettre cette substitution à l'exécution, il est nécessaire d'isoler la partie de l'architecture affectée par cette modification. La technique consistant à arrêter le composant composite qui contient le composant à remplacer peut être pénalisante car elle ne prend pas en compte les dépendances entre interfaces fournies et requises. Par exemple, sur la figure 3, la substitution du composant **H** demande l'inactivation des interfaces fournies de **H** (h_0) ce qui va désactiver de proche en

proche l'interface d_0 de **D** et c_0 de **C**. Ainsi l'interface c_1 de **C** reste active ce qui n'aurait pas été le cas si l'on devait *arrêter* **D**.

4. Implémentation

4.1. Extension du modèle Fractal

Nous avons réalisé un intergiciel pour le support de composants hiérarchiques distribués. Nous utilisons Julia [3], une implémentation en Java du modèle de composant Fractal. La prise en compte des interfaces actives dans Julia a été faite en intégrant un nouveau contrôleur aux composants primitifs et composites. Le rôle de ce contrôleur est de maintenir à jour l'état des interfaces requises et fournies. Il permet également d'activer ou désactiver chacune des interfaces du composant. La propagation au sein de l'architecture se fait en se basant sur les dépendances entre interfaces, dépendances explicitées par la clause `<binding>` du descripteur d'architecture de Fractal.

Notre contrôleur empêche l'invocation des méthodes sur les interfaces inactives. Nous utilisons les mécanismes d'intercepteurs (le `MetaCodeGenerator`) de Julia pour réifier tous les appels de méthodes. Si l'interface est inactive, la stratégie actuelle consiste à bloquer l'appel pendant un laps de temps paramétrable et de retourner une exception dans le cas où l'interface n'a pas été activée. Ce comportement peut être modifié, comme illustré dans la section 4.2.

L'intégration de la notion d'interfaces actives a été réalisée en utilisant les mécanismes de mixins offerts par Julia, mécanismes qui permettent la programmation des membranes des composants selon le principe de la programmation par aspect. Il est donc possible d'inclure notre contrôleur dans tous les composants qui sont développés sur Julia, indépendemment de notre plate-forme d'exécution. Nous offrons au programmeur une API permettant de connaître l'état des interfaces actives ainsi que les dépendances entre interfaces.

4.2. Prise en compte de l'environnement

La prise en compte des déconnexions a été réalisée à l'aide de l'intergiciel D-RAJE (*Distributed Resource-Aware Java Environment*) [10] développé dans notre équipe. Grâce à D-RAJE, un système distribué peut être modélisé à l'aide d'objets Java qui réifient les différentes ressources offertes par ce système. D-RAJE permet de modéliser à la fois des ressources systèmes (processeur, mémoire, disque, interface réseau...) et applicatives (processus, socket, thread, répertoire...). De manière similaire à ce qui a été fait dans un de nos précédents travaux sur la mise en œuvre de composants parallèles [11], nous avons modélisé les composants ainsi que leurs liaisons comme des ressources dans D-RAJE. Ainsi, il nous est possible de découvrir l'existence d'un composant spécifique, de rechercher un composant spécifique, et d'obtenir des informations sur l'état d'un composant (par observation directe ou notification) et de chacune de ses interfaces.

Les déconnexions et reconnexions liées au réseau sont répercutées sur les liaisons entre composants et ainsi sur les interfaces des composants. Nous utilisons pour cela la ressource D-RAJE `NetworkLink` qui modélise la liaison physique entre deux machines et qui maintient des informations sur l'état de la connexion. Nous avons ajouté à D-RAJE la ressource `RemoteBinding` afin de modéliser la liaison distante au niveau applicatif. L'utilisation de moniteurs sur des ressources `RemoteBinding` (qui observent des ressources `NetworkLink`) permet d'être notifié du changement d'état des différentes connexions réseaux et par conséquent d'activer et désactiver les interfaces associées.

Chaque instance d'un composant composite distribué est identifiée de manière unique sur chaque site (nom de la forme `componentName@hostA`). Ainsi, lors de la détection du rétablissement par D-RAJE d'une connexion réseau, le système peut découvrir les différentes instances du même composite et rétablir les liaisons entre composants si nécessaire.

La figure 4 illustre l'utilisation de notre API afin de contrôler l'action à effectuer vis-à-vis d'une interface inactive. Dans cet exemple, le programmeur décide d'attendre que la reconnexion se fasse avant de faire un appel de méthode. Il suffit pour cela d'attendre que l'interface possédant la méthode à invoquer redevienne active. À noter que nous aurions pu dans cet exemple utiliser uniquement l'état de l'interface sans passer par celui de la liaison distante.

```
// Obtention d'une reference de notre controleur
CubikController kcontroller =(CubikController)comp.getFcInterface("cubik-controller");

// Recuperation de l'etat de l'interface requise "cltItf"
if(kcontroller.getFcItfState("cltItf")==false) {

  // Gestion manuelle de la deconnexion
  RemoteBinding rb = kcontroller.getRemoteBinding("clientItf") ;

  while(!rb.getState()){
    // Attente de l'activation de l'interface
  }

  // L'invocation est maintenant possible
  clientItf.something() ;
}
```

FIG. 4 – Introspection sur l'état d'une interface.

5. Travaux connexes

Plusieurs travaux partagent les objectifs généraux de notre projet : fournir les moyens d'exploiter des composants logiciels distribués ayant des capacités d'adaptation. Toutefois, l'utilisation d'un modèle de composants hiérarchiques et la prise en compte des déconnexions sont rarement conjointement étudiés.
Notre démarche s'apparente à celles décrites dans [5, 1] dans lesquelles des mécanismes généraux d'adaptation au contexte sont proposés dans le cadre d'applications à base de composants hiérarchiques. Plus spécifiquement l'adaptation aux déconnexions est traitée dans [8], mais dans un modèle de composants à plat : les auteurs s'intéressent en effet à un service de gestion des déconnexions appliqué aux composants Corba [15]. Des « composants déconnectés » sont utilisés pour assurer la continuité de services. Ils permettent la journalisation des invocations et la réconciliation lors des reconnexions. Il n'est pas prévu que les services (mêmes dégradés) continuent à être rendus lors des déconnexions.
La notion de composants hiérarchiques physiquement distribués sur plusieurs sites se retrouve dans [2] qui présente une implémentation du modèle de composant abstrait Fractal en Proactive dans un contexte de Grid Computing. Bien que non explicitement traitées, les déconnexions peuvent être prises en compte dans une certaine mesure, l'utilisation de Proactive permettant notamment une exécution distribuée des composants communiquant de façon asynchrone.
Le projet Gravity [4] se base sur un *modèle de composant orienté service*. Dans leur approche, les liaisons entre composants sont vues comme des dépendances entre services (dans l'approche orientée service), ce qui permet de décrire des compositions entre composants s'adaptant automatiquement suivant la disponibilité des services. Au niveau des composants, la disponibilité d'un ou plusieurs services requis se manifeste par une instance valide ou invalide. L'environnement d'exécution actuel ne considère pas la distribution des composants.

6. Conclusion

Nous avons présenté dans cet article l'utilisation d'un modèle de composant hiérarchique permettant l'exécution d'une application distribuée sur un ensemble d'équipements hétérogènes et potentiellement volatiles. Nous exploitons la structure hiérarchique du modèle à la fois dans la gestion de la distribution et dans celle des déconnexions.
En nous appuyant sur le modèle de composants Fractal, nous avons défini un mode opératoire pour distribuer physiquement un composant composite sur plusieurs machines. Ce mode opératoire permet à un composant composite d'exhiber ses interfaces sur un ensemble de machines même si ses sous-composants sont localisés. La gestion décentralisée de l'architecture du composite offre un support pour

la prise en compte des déconnexions, déconnexions qui sont susceptibles d'entraîner des ruptures de liaisons entre sous-composants.
La notion d'interface active a été ajoutée au modèle hiérarchique afin d'isoler les dépendances entre interfaces requises et fournies. Les déconnexions induisent la désactivation de certaines interfaces, désactivation qui se propage dans la hiérarchie de composants. Les interfaces restant actives autorisent un composant à continuer de fonctionner, bien que dans un mode dégradé. Enfin une interface d'introspection est offerte au programmeur, lui permettant de connaître l'état de chaque interface afin de bâtir des stratégies d'adaptation aux déconnexions.
Un prototype a été réalisé et intégré à Julia, une implantation en Java du modèle Fractal qui nous a permis d'intégrer en tant que service non fonctionnel nos mécanismes gérant les déconnexions.
Les travaux présentés dans ce papier constituent une première étape dans le développement d'un intergiciel capable de supporter des composants adaptatifs, *i.e.* des composants capables de se reconfigurer en fonction des besoins et des changements de leur environnement d'exécution.
Nous envisageons d'intégrer des stratégies d'adaptation à partir des informations sur l'état de l'architecture de l'application à l'exécution. En particulier, nos travaux en cours portent sur l'adaptation du déploiement d'un composant hiérarchique qui pourrait être effectué de façon continue tout au long de l'exécution de l'application, en fonction de la disponibilité des ressources.

Bibliographie

1. Andersen (Anders). – *OOPP, A Reflective Middleware Platform including Quality of Service Management*. – Tromsø, Norway, Dr. sci. thesis, Department of Computer Science, University of Tromsø, février 2002.
2. Baude (Francoise), Caromel (Denis) et Morel (Matthieu). – From Distributed Objects to Hierarchical Grid Components. *In : Proceedings of the International Symposium on Distributed Objects and Applications (DOA)*. – Catania, Italie, novembre 2003.
3. Bruneton (Eric), Coupaye (Thierry), Leclercq (Matthieu), Quéma (Vivien) et Stefani (Jean-Bernard). – An Open Component Model and its Support in Java. *In : Proceedings of the International Symposium on Component-based Software Engineering (CBSE7)*. – Edinburgh, Scotland, mai 2004.
4. Cervantes (Humberto) et Hall (Richard S.). – Autonomous Adaptation to Dynamic Availability Using a Service-Oriented Component Model. *In : Proceedings of the International Conference on Software Engineering (ICSE)*, pp. 614–623. – Edinburgh, Scotland, mai 2004.
5. David (Pierre-Charles) et Ledoux (Thomas). – Towards a Framework for Self-Adaptive Component-Based Applications. *In : Proceedings of Distributed Applications and Interoperable Systems 2003, the 4th IFIP WG6.1 International Conference, DAIS 2003*, pp. 1–14. – Paris, France, novembre 2003.
6. Duclos (Frédéric), Estublier (Jacky) et Sanlaville (Rémy). – Architectures Ouvertes pour l'Adaptation des Logiciels. *Revue Génie Logiciel*, vol. 58, septembre 2001, pp. 19–25.
7. Heineman (George T.). – A Model for Designing Adaptable Software Components. *In : Proceedings of the 22nd International Computer Software and Applications Conference (COMPSAC)*, pp. 121–127. – Vienna, Austria, août 1998.
8. Kouici (Nabil), Conan (Denis) et Bernard (Guy). – Caching Components for Disconnection Management in Mobile Environments. *In : Proceedings of the International Symposium on Distributed Objects and Applications (DOA)*. – Agia Napa, Cyprus, octobre 2004.
9. Magee (Jeff), Dulay (Naranker), Eisenbach (Susan) et Kramer (Jeff). – Specifying Distributed Software Architectures. *In : Proceedings of the 5th European Software Engineering Conference (ESEC)*, pp. 137–153. – Sitges, Spain, septembre 1995.
10. Mahéo (Yves), Guidec (Frédéric) et Courtrai (Luc). – A Java Middleware Platform for Resource-Aware Distributed Applications. *In : Proceedings of the 2nd International Symposium on Parallel and Distributed Computing (ISPDC'2003)*. pp. 96–103. – Ljubljana, Slovénie, octobre 2003.
11. Mahéo (Yves), Guidec (Frédéric) et Courtrai (Luc). – Middleware Support for the Deployment of Resource-Aware Parallel Java Components on Heterogeneous Distributed Platforms. *In : Proceedings of the 30th Euromicro Conference - Component-Based Software Engineering Track*, pp. 144–151. – Rennes, France, septembre 2004.
12. Medvidovic (N.) et Taylor (N R.). – A Classification and Comparison Framework for Software Ar-

chitecture Description Languages. *IEEE Transactions on Software Engineering*, vol. 26, n1, 2000.
13. Microsoft. – *DCOM Technical Overview*. – Microsoft Windows NT Server white paper, Microsoft Corporation, 1996.
14. Monson-Haefel (Richard). – *Enterprise JavaBeans*. – O'Reilly, 1999.
15. Object Management Group. – *CORBA Components*. – Adopted Specification n formal/02-06-65, OMG, June 2002. Version 3.0.
16. van Ommering (Rob C.). – Koala, a component model for consumer electronics product software. *In : ESPRIT ARES Workshop*, pp. 76–86. – Las Palmas de Gran Canaria, Spain, février 1998.