Composants ubiquitaires pour réseaux dynamiques

Didier Hoareau

Composants ubiquitaires pour réseaux dynamiques

THÈSE

présentée et soutenue publiquement le 5 décembre 2007

pour l'obtention du

Doctorat de l'université de Bretagne Sud

Sciences et Technologies de l'Information et de la Communication

par

Didier Hoareau

Composition du jury

Président :	M. Flavio Oquendo	Professeur, Université de Bretagne Sud
Rapporteurs :	M^{me} Isabelle Demeure	Professeur, École Nationale Supérieure des Télécommunications
	M. Lionel Seinturier	Professeur, Université des Sciences et Technologies de Lille
Examinateurs :	M. Pierre Kuonen	Professeur, École d'ingénieurs et d'architectes de Fribourg
	M. Patrice Frison	Professeur, Université de Bretagne Sud (Directeur de thèse)
	M. Yves Mahéo	Maître de conférences, Université de Bretagne Sud

Laboratoire de Recherche en Informatique et ses Applications de Vannes et Lorient — EA 2593

Remerciements

La thèse... Une expérience unique, riche d'enseignements et de rencontres marquantes. C'est aussi de nombreux moments de questionnement, de doutes, de solitude intellectuelle. Cette thèse n'aurait donc pas pu aboutir sans l'aide et le support précieux de ces nombreuses personnes qui ont contribué, chacune à leur manière, à l'achèvement de ce long travail.

Mes premiers remerciements vont à Yves Mahéo qui a été mon encadrant, mon « Obiwan Kenobi ». Il est à l'origine de ce sujet passionnant et ce travail de thèse n'aurait pas pu voir le jour sans son implication, sa rigueur et sa disponibilité. Il a toujours été présent pour répondre à mes nombreuses sollicitations, a su m'écouter, me comprendre, me tempérer, me faire partager sa passion pour son métier... Je ne le remercierais jamais assez de la confiance qu'il m'a accordée ainsi que de la liberté qu'il m'a laissée dans cette aventure... Merci pour tout.

Je tiens à remercier chaleureusement celui qui m'a donné envie de faire une thèse, M. Éric Monfroy. L'initiation à la recherche à ses côtés avait commencé en maîtrise sur un sujet de recherche traitant de la simulation d'agents autonomes, puis en DEA sur la coopération dynamique de résolveurs de contraintes. Depuis je suis accro. Le monde de la programmation par contraintes et des agents influe toujours sur ma façon de penser. Merci de ta confiance, ton soutien et tes encouragements.

Je remercie vivement M^me^ Isabelle Demeure et M. Lionel Seinturier de l'intérêt qu'ils ont porté à mes travaux et de l'honneur qu'ils me font d'être rapporteurs de ce mémoire. Je remercie également M. Pierre Kuonen d'avoir accepté de juger mes travaux. Je voudrais exprimer ma reconnaissance à M. Flavio Oquendo d'avoir présidé mon jury de thèse.

En sus d'être membre de mon jury, M. Patrice Frison a été le directeur administratif de cette thèse. Je le remercie pour ses précieux conseils et je n'oublie pas nos discussions autour de l'*ubiquitisation* d'Ubiware.

J'ai eu la chance d'avoir un second maître *Jedi*, Frédéric Guidec. Toujours disponible, sa maîtrise de la maïeutique m'a aidé à affûter mes réflexions et convictions. J'espère toujours qu'on puisse faire le grand saut ensemble au dessus de l'île intense. Merci aussi de m'avoir initié à la poésie.

Merci à tous les membres de l'action CASA de leur accueil et de m'avoir autant impliqué dans la vie de cette action. Pour nos rougails futurs on peut toujours faire ça chez moi ;)

Je tiens à remercier tous les membres du laboratoire VALORIA pour ces quatre années. Sans aucun doute, le sérieux et la bonne humeur qui m'entouraient ont été sources de motivation et d'inspiration. Merci à tous. Nar' trouv ! Je remercie en particulier M. Pierre-François Marteau, directeur du VALORIA, de son soutien permanent et de ses conseils. Merci à Frédéric Raimbault pour ses encouragements et sa passion pour Linux (nar trouv' bientôt à Gillot ;). Merci à Salah Sadou et Régis Fleurquin de m'avoir fait découvrir SE et la bonne humeur qui y règne. Merci à Nicolas Courty pour les moments de franche rigolade (« kiléoulà ? ! »). Merci à Jean-François Kamp (mon Général) pour les rafraîchissements belges entre autres. Un énorme .E.I à SylvianeBoisadan de son soutien logistique et d'avoir résisté à ma gestion « à la 24 » des *deadlines*. Merci à Gersan Moguerou pour sa maîtrise des systèmes et ses dessins.

À l'heure où j'écris ces lignes, mes compagnons de bureau, Eugen Popovici et Chouki

Tibermacine ont atteint également la case 42 (42... tiens tiens !) du jeu de la thèse[1]. Encore félicitations à vous ! Je vous souhaite bonne chance et pleins de bonnes choses pour la suite. C'était un plaisir de vous avoir rencontré et d'avoir partagé cette aventure à vos côtés.

Je remercie également tous les thésards du VALORIA qui ont su me supporter ;). Je pense notamment à Hervé Roussain et Réda Kadri. Merci à Roméo Said, Julien Haillot et Mohammed Belatar pour leur aide dans l'organisation de la soutenance. Merci à Julien et Sandrine pour les photos du jour J. Bonne chance à tous pour la suite. Merci à Bart Georges de m'avoir fait partagé ses découvertes photographiques.

Je suis extrêmement reconnaissant envers M. Frédéric Mesnard, directeur de l'IREMIA, d'avoir permis que la fin de thèse se passe dans les meilleures conditions. Je le remercie notamment des moyens mis à ma disposition pour garantir le bon déroulement de la soutenance.

Merci à Jo, la patronne de la meilleure cafet' de l'UBS de toutes les marques d'affection qu'elle m'a témoignées. Encore mille mercis de ta précieuse aide quant au bon déroulement du pot de soutenance.

Merci à Alexis Troubnikoff et Corentin Durbach du PRISM de m'avoir fait découvrir Linux et LaTeX. Plus rien n'a été comme avant depuis !

Il me tient à cœur de remercier mes plus proches amis qui m'ont accompagné durant cette aventure. Chacun à leur manière, ils ont participé à mon équilibre. Merci à Grand Seb, Steph, Natacha, Mathieu, Aurélia, Line, Renaud et Corentin. Merci à Maëva d'être venue apaiser la fin de ma « saison 4 ». Merci à Seb et Flo d'avoir toujours été là. Merci encore à toi Seb, compagnon de l'abus, de m'avoir laissé gagné de temps à autre au squash ;).

Mes plus profonds remerciements vont à toute ma famille. Je remercie en particulier ma mère et mes frères. Plus qu'un réconfort, leurs soutiens inconditionnels et encouragements ont été ma force pour atteindre le sommet de cette incroyable randonnée. Merci à ma mère de m'avoir communiqué son goût du travail et sa détermination face aux problèmes à résoudre.

Je remercie enfin de tout mon cœur ma femme, Evelyne, pour sa patience, son écoute et son aide de chaque instant.

[1] http://www-valoria.univ-ubs.fr/Didider.Hoareau/pub/jeudelathese/

« — C'est une bonne situation ça docteur ?
— Je ne crois pas qu'il y ait de bonne ou de mauvaise situation. [...] Ma vie aujourd'hui, c'est d'abord des rencontres, des gens qui m'ont tendu la main, peut-être à un moment où je ne pouvais pas, où j'étais seul chez moi, et c'est assez curieux de se dire que les hasards, les rencontres forgent une destinée, parce que quand on a le goût de la chose, quand on a le goût de la chose bien faîte, le beau geste, parfois on ne trouve pas l'interlocuteur en face, je dirais le miroir qui vous aide à avancer ; mais ce n'est pas mon cas... »

À tous ceux portés par cette ineffable bigosité...

Table des matières

Table des figures

Liste des tableaux

Listings

Introduction générale

On assiste depuis quelques années à l'émergence de nouveaux réseaux de machines, qualifiés de dynamiques. Ces réseaux, de plus en plus répandus, sont caractérisés par l'hétérogénéité, la mobilité et la volatilité des équipements qui les composent. Par exemple, au sein d'un même foyer, les équipements suivants peuvent constituer un réseau dynamique : une ordinateur de bureau, un ordinateur portable, un assistant numérique personnel, un ordinateur multimédia ou encore une console de jeux. Les équipements mobiles et volatiles, comme l'ordinateur portable utilisé lors de nos déplacements et souvent mis en veille, imposent une disponibilité et des performances fluctuantes des liens de communication.

Devant cette panoplie d'équipements à notre disposition, de nouveaux besoins apparaissent. Lorsque plusieurs équipements sont en effet à notre portée, il y aurait un avantage certain de pouvoir profiter des fonctionnalités d'une même application indépendamment de l'équipement utilisé. En effet, la plupart des logiciels actuels ne peuvent être utilisés uniquement depuis les machines sur lesquelles ils sont installés. Les réseaux dynamiques commencent à être exploités pour fournir des applications censées être ubiquitaires, c'est-à-dire des applications dont les fonctionnalités sont accessibles depuis les différentes machines du réseau. Il est maintenant possible par exemple de consulter ses courriels depuis son ordinateur de bureau, de son assistant numérique personnel ou encore depuis son téléphone portable. Mais en réalité, la plupart de ces applications reposent sur un schéma classique « client-serveur ». L'aspect ubiquitaire caractérise alors moins les applications que les accès réseau offerts par la multitude d'équipements avoisinants.

La définition de véritables applications ubiquitaires devient nécessaire. Il ne s'agit plus d'avoir une version de l'application spécifique à chaque machine, avec des fonctionnalités différentes selon l'équipement et la version utilisée. La même application doit pouvoir être utilisée depuis différents équipements. Par exemple, si nous disposions d'un logiciel ubiquitaire de création de diaporamas il serait possible de préparer le diaporama depuis son assistant numérique personnel pour ensuite le visualiser sur n'importe quel ordinateur de la maison.

Concevoir et déployer de telles applications ubiquitaires dans des réseaux dynamiques nécessitent de prendre en compte l'hétérogénéité des équipements ainsi que les nombreuses variations de ressources qui peuvent survenir. Les applications ubiquitaires doivent pouvoir s'adapter à cette dynamicité.

Ces applications ubiquitaires, adaptables, sont par nature complexes. L'approche par composants permet d'envisager le développement de telles applications. Dans cette approche, une application est conçue comme un assemblage de briques logicielles réutilisables, les composants. Par ailleurs, le cadre de la programmation par composants logiciels offre des méthodes efficaces pour la gestion de l'intégralité du cycle de vie des applications, de leur conception à leur maintenance, en passant par leur déploiement. Cependant, les technologies à composants les plus répandues telles que CCM, .Net ou EJB n'ont pas été conçues initialement

pour des environnements dynamiques. Elles considèrent ainsi les différentes fluctuations de ces environnements, comme l'inaccessibilité d'une machine ou l'indisponibilité de certaines ressources, comme des exceptions. Le traitement des différents changements qui peuvent se produire au sein du réseau est réalisé généralement par l'ajout de codes d'adaptation accentuant la complexité des applications.

Nous proposons dans cette thèse des méthodes pour concevoir et déployer des applications ubiquitaires dans des réseaux dynamiques. Pour répondre d'une part aux difficultés posées par ces réseaux et d'autre part à la complexité des applications dans de tels réseaux, nous proposons le concept de composants logiciels ubiquitaires et nous définissons un support d'exécution pour ce type de composants.

En nous appuyant sur les entités des modèles de composants traditionnels, nous proposons une distribution particulière des composants afin de les rendre ubiquitaires. De cette manière leurs fonctionnalités deviennent accessibles depuis n'importe quelle machine du réseau. Ces composants, briques d'assemblage des applications ubiquitaires, intègrent dans leur modèle le support des déconnexions et reconnexions imposées par les réseaux dynamiques. Ainsi, notre première contribution porte sur la définition de mécanismes dans le modèle de composants Fractal permettant de rendre ces composants ubiquitaires.

Notre second objectif est de fournir un support intergiciel pour les composants ubiquitaires. Un tel support doit être capable d'instancier une architecture de composants et ceci malgré les conditions d'exécution changeantes imposées par le réseau. En particulier, les ressources qui sont présentes ne peuvent être connues à l'avance, ce qui empêche la désignation explicite des machines cibles devant héberger les composants.

Pour réaliser le déploiement des applications ubiquitaires, nous proposons un langage qui permet de spécifier le placement des composants à l'aide de contraintes exprimant le besoin des composants vis-à-vis des ressources nécessaires à leur exécution. Ainsi, il n'est plus nécessaire de désigner la localisation des composants par des noms ou adresses de machines. C'est au support intergiciel de déterminer une machine hôte pour chaque composant. Les ressources exigées par certains composants peuvent ne pas être disponibles et des machines peuvent ne pas être initialement accessibles. Le processus de déploiement prend avantage de la dynamicité du réseau en rendant possible l'instanciation des composants au fur et à mesure de la disponibilité des ressources et de l'arrivée des machines.

Lorsque des fluctuations de ressources interviennent dans le réseau, les applications ubiquitaires sont amenées à être reconfigurées. Cela est nécessaire pour augmenter la disponibilité des fonctionnalités de chacun des composants, mais aussi pour garantir la bonne exécution de l'application dans son ensemble. Ainsi, lorsque des composants sont sujets à des pannes ou que des ressources requises ne sont plus disponibles, un redéploiement s'impose. L'approche que nous développons dans ce mémoire consiste à rendre ces reconfigurations autonomes, ne nécessitant ainsi plus aucune intervention manuelle.

Organisation du manuscrit

La suite de ce mémoire est organisée en deux parties.

La première partie présente à travers quatre chapitres le contexte de notre étude ainsi que l'état des domaines considérés dans les travaux de cette thèse :

- Le chapitre 1 situe le domaine d'étude de notre travail. Nous présentons les visions développées par l'informatique ubiquitaire et autonome et nous caractérisons les réseaux dynamiques, plates-formes cibles de nos réflexions et expérimentations. Un aperçu de la programmation par composants logiciels ainsi que des supports intergiciels est ensuite donné. Enfin, nous présentons à partir de deux exemples d'application ubiquitaire les problématiques auxquelles répond les travaux présentés dans ce mémoire de thèse.
- Le chapitre 2 de ce mémoire porte sur l'étude des mécanismes gérant la distribution des composants. Après avoir défini les concepts communs aux différents modèles de composants et à leur support d'exécution, nous présentons un état de l'art des modèles de composants et de leur support intergiciel en nous intéressant à la gestion de la distribution.
- Dans le chapitre 3, nous nous intéressons plus spécifiquement aux modèles et technologies à composants qui prennent en compte le caractère dynamique de leur environnement d'exécution. Face aux réseaux dynamiques, des mécanismes qui permettent l'adaptation d'une architecture de composants sont alors nécessaires. Nous présentons d'abord les concepts permettant de définir l'adaptation dans les modèles de composants ainsi que la notion de cohérence architecturale. Puis, nous dressons un état de l'art des travaux centrés sur l'adaptation dans les modèles de composants et les approches intergicielles.
- Le chapitre 4 s'appuie sur les concepts présentés dans les chapitres précédents pour présenter le déploiement des applications à base de composants dans les réseaux dynamiques. Après avoir mis en avant les problèmes que posent ces réseaux pour le déploiement de composants logiciels et notamment le problème de la décision de placement des composants dans ces réseaux, nous décrivons une étude des solutions proposées dans les technologies CCM, Fractal et EJB. Dans un troisième temps, nous citons quelques travaux ayant pour objectif de rendre autonome le déploiement des composants logiciels.

La seconde partie de ce mémoire présente notre contribution autour des composants logiciels ubiquitaires et de leur support d'exécution. Elle est composée de cinq chapitres organisés comme suit :

- Le chapitre 5 présente notre proposition pour définir des composants logiciels ubiquitaires. En nous appuyant sur un modèle de composants hiérarchiques (à la Fractal), nous définissons un schéma de distribution des composants permettant de rendre leurs fonctionnalités accessibles sur l'ensemble des machines du réseau. Nous détaillons dans ce chapitre les mécanismes de composition de composants ubiquitaires. Nous proposons de prendre en compte, au sein même du modèle de composants, le caractère dynamique de la plate-forme à travers la notion d'interfaces actives et inactives ;
- Le chapitre 6 répond aux problèmes de la spécification du déploiement des composants logiciels ubiquitaires dans les réseaux dynamiques. Nous proposons un langage qui permet de façon déclarative d'indiquer, via des contraintes de ressources, les exigences

des composants vis-à-vis des machines censées les héberger. Au travers d'un deuxième type de contraintes, les contraintes de localisation, ce langage permet de contrôler le placement des composants sans pour autant désigner une machine par son nom ;
- la manipulation directe de la description des contraintes de ressources et de localisation ne permet pas facilement la mise en place d'un support devant réaliser le déploiement des composants logiciels ubiquitaires. Le chapitre 7 décrit la transformation de la description de ces contraintes en un ensemble de *problèmes de satisfaction de contraintes* (CSP pour *Constraint Satisfaction Problem* en anglais) tels que définis par le paradigme de la programmation par contraintes. Les éléments pour résoudre les CSP sont également décrits et permettent de trouver des solutions pour le problème de placement des composants dans des réseaux dynamiques ;
- le chapitre 8 décrit le processus de déploiement autonome de composants logiciels ubiquitaires sur des réseaux dynamiques. Ce processus est présenté en trois étapes. Nous présentons dans un premier temps un déploiement que nous qualifions de « propagatif » dans des réseaux connexes. Puis nous considérons toutes les caractéristiques des réseaux dynamiques (notamment la mobilité et la volatilité des équipements) ; un algorithme de consensus permettant de garantir les décisions d'instanciation des composants est alors présenté. Enfin la dernière section de ce chapitre décrit notre proposition pour rendre le déploiement autonome.
- le chapitre 9 conclut cette partie en présentant les éléments de mise en œuvre des concepts présentés dans ce mémoire de thèse. Nous détaillons comment l'aspect ubiquitaire des composants a été rajouté au support Fractal/Julia. Nous présentons l'architecture générale du support intergiciel que nous avons développé pour la réalisation du déploiement autonome de composants logiciels ubiquitaires.

Nous concluons ce mémoire en rappelant le contexte et les objectifs de ce travail de thèse. Nous évoquons finalement quelques perspectives de recherche qu'ouvrent nos travaux.

Première partie

Problématique et état de l'art

The most profound technologies are those that disapear

Mark Weiser

1 Domaine d'étude

Afin de faciliter la conception et le déploiement des applications ubiquitaires, nous avons été amenés à explorer différentes technologies ainsi que plusieurs axes de recherche qui constituent notre domaine d'étude. Ce premier chapitre identifie les thèmes couverts par notre domaine d'étude. Notre objectif est de définir les différents concepts et de présenter les technologies autour desquels se bâtit notre réflexion.

Nous présentons dans un premier temps les visions développées par l'informatique ubiquitaire et l'informatique autonome pour définir le concept d'application ubiquitaire. Nous présentons ensuite le paradigme de la programmation orientée composants que nous avons suivi pour concevoir les applications ubiquitaires. Nous présentons dans la section 1.3 l'approche intergicielle. Nous donnons dans la section 1.4 deux exemples d'application ubiquitaire qui nous permettront d'énoncer dans la section 1.5 les problèmes que tente de résoudre ce travail de thèse.

Sommaire

1.1 Vers des applications distribuées, ubiquitaires et autonomes

1.1.1 L'informatique ubiquitaire

L'informatique ubiquitaire ou pervasive [BD07] est un terme qui a été introduit pour la première fois par Mark Weiser [Wei99]. Dans sa vision, les équipements informatiques sont invisibles et omniprésents : contrairement à la réalité virtuelle qui définit un monde dans lequel nous pouvons évoluer, l'informatique ubiquitaire fait entrer les ordinateurs dans le nôtre. Cette vision a émergé principalement grâce à deux facteurs :

- *la prolifération du nombre d'équipements*. Dans les années 60, le terme « ordinateur » désignait un *ordinateur central* (*mainframe* en anglais), onéreux et utilisé par un grand nombre de personnes. Puis, le coût moindre et l'encombrement minimal des ordinateurs ont permis sa démocratisation : chaque utilisateur avait son propre ordinateur. Aujourd'hui, on voit apparaître une nouvelle évolution du rapport entre utilisateurs et ordinateurs avec le nombre croissant de petits équipements tels que les assistants numériques personnels, les téléphones portables, les ordinateurs portables : un utilisateur utilise ou possède plusieurs ordinateurs ;
- *la communication sans fil*. Le plupart des équipements cités précédemment utilisent des technologies de communication qui permettent la mise en place de réseaux spontanés (e.g. IEEE 802.11 Wireless Local Area Network, Bluetooth), ne nécessitant aucune infrastructure particulière. La mise en réseau des équipements est ainsi facilitée.

Ces facteurs ont transformé petit à petit le rapport entre l'utilisateur et les équipements informatiques. Dans notre environnement quotidien caractérisé par l'omniprésence des ordinateurs, il est maintenant possible d'avoir accès à divers réseaux informatiques (e.g. réseau personnel sans fil, réseau local, etc.) dont l'utilisation se limite pour le moment à l'accès à des services spécifiques. Il faut ajouter à cette intégration physique des ordinateurs dans notre monde, la capacité qu'ont de plus en plus ces équipements d'interagir entre eux de façon *spontanée* ([KF02]).

Considérons un réseau pervasif, c'est-à-dire un ensemble plus ou moins important d'équipements informatiques dotés de capacités de communication et qui, de par leur taille ou leur mode d'interaction, tendent à être *invisibles* pour l'utilisateur. Pour illustrer les objectifs que se donne l'informatique ubiquitaire dans ce genre de réseau, nous présentons l'un des scénarii du projet AURA [GSSS02], porté par l'université de Carnegie Mellon :

> Dans son bureau, Fred termine les derniers transparents qu'il doit présenter à une réunion où il fera également une démonstration de son dernier logiciel. Le lieu de la réunion est à une dizaine de minutes à pieds du campus. Il est temps de partir mais Fred n'a pas tout à fait terminé sa présentation. Il saisit son PalmXXII, assistant numérique personnel sans fil, et sort de son bureau. Le système Aura transfère l'état de son travail depuis son ordinateur de bureau à son assistant numérique et l'autorise à faire les dernières modifications à l'aide de commandes vocales tout au long du trajet. Aura déduit la destination de Fred à partir de son calendrier et du système de géolocalisation disponible dans le campus. À proximité de la salle de réunion, Aura télécharge la présentation ainsi que le logiciel de démonstration sur l'ordinateur que Fred va utiliser durant son intervention et allume le vidéo projecteur.

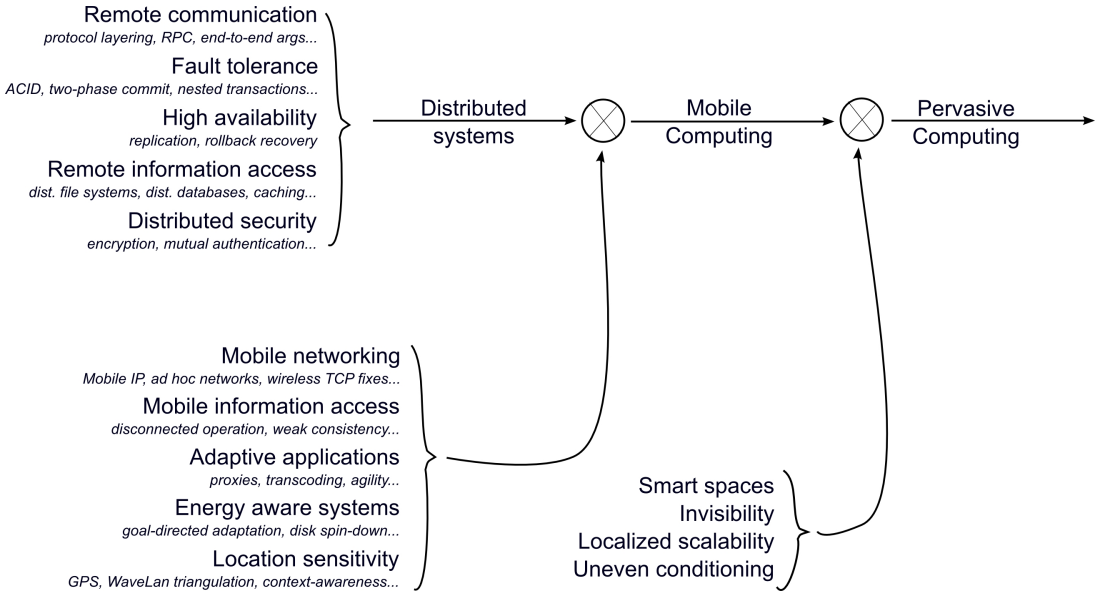


FIG. 1.1: Taxonomie des problèmes de recherche en informatique pervasive (extrait de [Sat01]).

Cet exemple illustre l'utilisation d'un même logiciel, l'éditeur de présentation, depuis différents équipements. Concevoir des systèmes capables de fournir une telle utilisation multi-postes constitue également un de nos objectifs. À travers cet exemple, nous pouvons constater que les technologies impliquées, que ce soit dans le transfert des données et de leur traitement d'un équipement à un autre, que dans l'interaction entre les équipements, sont aujourd'hui largement répandues (communication sans fil, géolocalisation, etc.) et font l'objet de nombreux travaux de recherche (cf. figure 1.1). Pour autant, un système comme Aura nous est moins familier. L'informatique ubiquitaire n'est pas seulement une simple mise en réseau d'un ensemble d'équipements, toujours plus nombreux.

L'informatique ubiquitaire définit un champ d'étude vaste, tant au niveau des équipements (et leur mise en réseau) que des logiciels devant s'y exécuter. Nous précisons ces deux aspects dans les sections suivantes.

1.1.2 Caractérisation des plates-formes cibles : les réseaux dynamiques

L'informatique ubiquitaire met en jeu une panoplie d'équipements qui, de part leur moyen de communication, peuvent constituer des réseaux pouvant être de nature très variée. Nous focalisons nos travaux sur des réseaux qualifiés de dynamiques. Nous préférons ce qualificatif à celui d'ubiquitaires ou de pervasifs car nous nous préoccupons moins des propriétés d'enfouissement des équipements — tendant à les rendre invisibles — que de celles liées à leur interaction.

1.1.2.1 Définition et caractérisation des réseaux dynamiques

Les réseaux dynamiques considérés dans ce mémoire sont caractérisés par :

– *leur hétérogénéité*. Les machines n'ont pas toutes les mêmes configurations matérielles et logicielles. Au sein d'un même réseau, des stations de travail aux capacités de calcul et

de stockage importantes peuvent coexister avec d'autres appareils à faibles ressources comme des assistants numériques personnels. Cette hétérogénéité des équipements ne disparaîtra pas. Au contraire, des capteurs et autres processeurs directement intégrés aux objets de notre vie quotidienne (e.g. dans les tasses à cafés, les vêtements) sont déjà commercialisés pour faciliter nos gestes de tous les jours.
- *leur mobilité*. De par leur nature, les équipements tels que les téléphones mobiles, les assistants numériques personnels ou les ordinateurs portables peuvent être transportés par leurs utilisateurs. Cette mobilité pose le problème de la connectivité du réseau. Un équipement peut donc ne pas toujours être accessible des autres équipements à tout moment.
- *leur volatilité*. Toutes les machines que nous emmenons lors de déplacements fonctionnent avec des batteries offrant une autonomie limitée (quelques heures pour un ordinateur portable, quelques jours pour les assistants numériques personnels). Il est donc fréquent que ces appareils soient mis en veille pour des raisons d'économie d'énergie. Dans un réseau sans fil, cela se manifeste pour les autres machines à portée radio par l'apparition ou la disparition d'un équipement.

L'hétérogénéité des équipements constituant un réseau dynamique nécessite d'accéder à des ressources distantes, non disponibles localement. Du fait de la faible capacité de stockage de certaines machines, il ne sera pas possible de disposer des mêmes ressources sur chacun des équipements. L'intérêt même de connecter une machine à un réseau est alors de pouvoir d'une part, accéder à des ressources externes, non disponibles localement (e.g. capacité de calcul, de mémoire, des ressources logicielles) et d'autre part, de mettre à disposition des autres machines du réseau, les ressources présentes localement. De plus, les équipements ont en général une portée limitée et peuvent ne pas être accessibles à tout moment. Avec l'arrivée ou le départ d'une machine, les ressources dont elles disposent peuvent apparaître et disparaître. Tous ces facteurs induisent une dynamicité de l'accès aux ressources. Cette dynamicité est accentuée par les fluctuations des ressources elles-mêmes : par exemple, la quantité de mémoire disponible sur un équipement peut ne plus être suffisante pour le bon fonctionnement d'une application.

Dans un réseau dynamique, les contraintes de communication entre équipements ainsi que la volatilité de ces derniers amènent le réseau à être fragmenté. Des zones entre lesquelles aucune communication n'est possible peuvent se former définissant ainsi des îlots. La présence de plusieurs îlots implique que deux machines se trouvant dans deux îlots distincts ne peuvent plus s'échanger de l'information. Par exemple, la figure 1.2 illustre un réseau dynamique dans lequel une partie des équipements (îlot B) communique par une liaison radio (e.g. en Wi-Fi). Un îlot A peut ainsi se former et est constitué des machines n'étant plus à portée radio des machines de B. La formation d'un îlot n'est pas seulement due aux communications sans fil. Sur l'exemple de de la figure 1.2, les machines présentes à la maison peuvent constituer un îlot lorsque la connexion internet est interrompue.

On peut noter que la prise en compte des îlots au sein du réseau — comme nous le faisons dans ce mémoire — ne correspond pas à la vision généralement admise d'un réseau dynamique tel que cela est présenté dans le cadre des environnements de type grilles, des réseaux pair-à-pair ou des MANET (*Mobile Ad-hoc NETworks*). Aucun îlot ou partition n'est généralement considéré : il n'y a qu'un seul réseau défini par la capacité de tout équipement du réseau à communiquer, directement ou indirectement, avec les autres. Dans un souci de

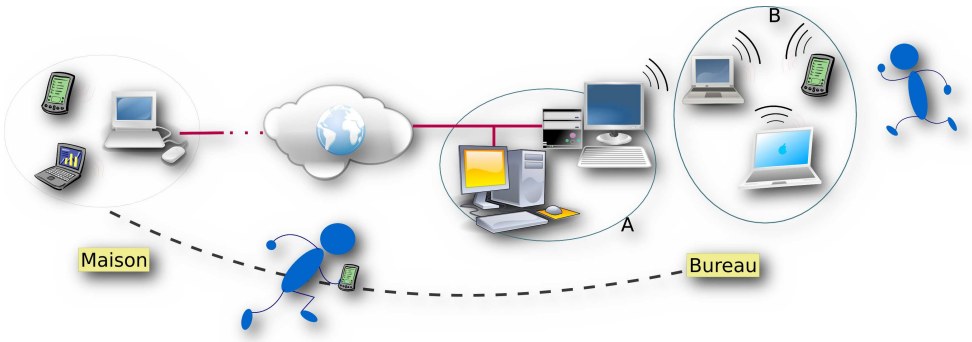


FIG. 1.2: Exemple d'un réseau dynamique. Un utilisateur a accès à un ensemble de machines comprenant des équipements mobiles et fixes.

Opération de mise à jour	*Correspondance sur le réseau*	
	Réseau filaire	*Réseau non filaire*
Ajout d'un nœud	Une machine est branchée sur le réseau	Arrivée d'une machine dans un îlot
	Une machine éteinte a été rallumée	
Retrait d'un nœud	Une machine est débranchée du réseau	Départ d'une machine d'un îlot
	Extinction d'une machine	
Ajout d'une arête		Une machine passe à portée radio d'une autre
	Routage entre deux machines établi	
Retrait d'une arête		Une machine sort de la portée d'une autre machine
	L'interface réseau d'une machine est désactivée	

TAB. 1.1: Correspondance entre les opérations de mise à jour sur un graphe modélisant un réseau dynamique et les situations réelles.

généralité, nous intégrons aux réseaux dynamiques le phénomène d'îlots étant donné que leur présence n'est pas une situation exceptionnelle mais courante.

Un réseau dynamique peut être modélisé par un graphe non orienté dans lequel les nœuds représentent les différents équipements. Une arête dans cette modélisation exprime la capacité pour les deux nœuds correspondants d'échanger des messages. Dans le cas d'un réseau radio sans routage (e.g. Bluetooth), cela signifie que les deux machines sont à portée radio l'une de l'autre (et bien évidemment, associées l'une à l'autre). Dans un réseau filaire routé, une arête sera présente entre deux nœuds s'il existe une route entre les deux équipements correspondants.

La mobilité des équipements se manifeste par la possibilité d'effectuer des opérations de mises à jour sur un graphe [DFI03]. Une opération de mise à jour peut être : l'ajout ou le retrait d'un nœud, d'une arête ou le changement d'une propriété sur un nœud ou une arête. Le tableau 1.1 résume les différentes opérations possibles sur un graphe et indique les situations qui correspondent à ces opérations.

Dans ce travail de thèse, nous définissons comme plates-formes cibles de nos travaux les réseaux qualifiés de dynamiques. Ces réseaux sont caractérisés par leur hétérogénéité, leur mobilité et leur volatilité provoquant une fluctuation des ressources disponibles et leur fragmentation en îlots.

1.1.3 L'informatique autonome

L'intervention de l'homme dans le contrôle et la maintenance des réseaux dynamiques tend à disparaître au profit de systèmes capables de s'autogérer. Ces systèmes sont capables d'agir sur leur propre fonctionnement afin de s'adapter aussi bien à des conditions d'exécution changeantes qu'à une erreur survenue en leur sein. L'objectif de définir de tels systèmes est d'alléger la tâche du concepteur, du déployeur et du mainteneur d'applications qui ne peuvent pour des systèmes de grande complexité, appréhender toutes les reconfigurations ou adaptations qui sont susceptibles d'intervenir dans le cycle de vie des logiciels. De nombreux travaux ont vu le jour quant à la définition de systèmes et logiciels *autonomes*. Nous pouvons citer l'initiative pour l'*autonomic computing*[2], lancée par IBM, qui définit des systèmes autonomes [BBC+03] comme des systèmes capables de :

1. **s'auto-configurer (*self-configuration*)** : cette propriété caractérise la capacité d'un système autonome à prendre en compte dans sa configuration actuelle et celle du réseau le supportant, de nouveaux services et l'arrivée d'un nouvel équipement ; aucune procédure d'installation manuelle n'étant nécessaire ;
2. **s'auto-réparer (*self-healing*)** : le système est capable de détecter, diagnostiquer et d'effectuer les réparations nécessaires suites à différentes pannes (e. g. crash d'une machine) ;
3. **s'auto-optimiser (*self-optimization*)** : le système est capable de redistribuer les différentes tâches sur les différentes machines composant le réseau afin de garantir une utilisation optimale des ressources ;
4. **s'auto-protéger (*self-protection*)** : le système est capable de détecter et de se protéger des attaques (e.g. virus, *hackers*) éventuelles.

Dans ce mémoire, nous allons nous intéresser aux deux premières propriétés, à savoir l'auto-configuration et l'auto-réparation. En effet, les travaux sur le déploiement dans les réseaux que nous visons apportent peu de réponses quant à l'automatisation du déploiement et de la configuration des applications distribuées. La gestion auto-régulée de l'allocation des ressources et les aspects liés à la sécurité peuvent être étudiés orthogonalement à notre proposition. Nous reviendrons sur ces aspects dans les exemples présentés à la section 1.4.

La conception et la maintenance des applications dans les réseaux dynamiques posent de nombreuses difficultés du fait de la nature même de ces réseaux et de la complexité croissante des applications. Nous avons pour objectif dans cet travail de thèse de fournir des mécanismes permettant de définir des applications autonomes c'est-à-dire des systèmes capables de s'autoréguler.

[2]Le terme anglais *autonomic* est inspiré du système nerveux humain. Ce système contrôle les fonctions importantes de notre corps (e.g. la respiration, les battements de notre coeur, la pression sanguine) sans aucune intervention consciente de notre part.

1.1.4 Vers des applications ubiquitaires

La prolifération des équipements informatiques dotés de moyens de communication toujours plus sophistiqués rend possible l'utilisation d'un même service logicielle indépendamment de l'équipement utilisé. Par exemple, à partir d'un assistant numérique personnel ou bien d'un téléphone portable, il est possible d'explorer la toile via un navigateur spécialement développé pour chacun de ces deux appareils. Il en est de même pour un logiciel de gestion de rendez-vous : des versions propres à chaque équipement permettent à l'utilisateur de consulter son agenda ou de noter ses rendez-vous depuis ses différentes machines.

Cependant, cette utilisation « multi-postes » n'est pas toujours possible : il peut en effet ne pas exister de version du logiciel pour chacune des plates-formes, ou bien, ce mode d'utilisation rend difficile la gestion des données manipulées par les différentes versions du logiciel, et donc le maintien de leur cohérence. Il est en effet fréquent, du fait de l'hétérogénéité et des capacités limitées de certains équipements, qu'un logiciel ne puisse se décliner sur ces équipements (e.g. un environnement de développement intégré). De plus, lorsque l'on est amené à utiliser plusieurs machines, par exemple dans le cas d'un logiciel de gestion de rendez-vous, les outils de synchronisation de données ne sont pas tous compatibles les uns avec les autres (e.g. ActiveSync de Microsoft n'est pas disponible sur un autre système d'exploitation). Dans ce cas, c'est à l'utilisateur de se rappeler de la localisation de ses données.

Par ailleurs, vouloir utiliser le *même* logiciel sur chacune des machines que l'on peut posséder présuppose une installation ainsi qu'une configuration sur chacune d'elles. Ces tâches, répétitives dès lors qu'un nouvel appareil doit être pris en compte, exigent dans la plupart des cas, une intervention de l'utilisateur.

Nous proposons dans ce travail de thèse de redéfinir cette utilisation multi-postes des fonctionnalités offertes par un logiciel. Dans notre approche, un même logiciel doit pouvoir être utilisé depuis n'importe quel équipement (ou du moins, sur tout équipement où cela a un sens). Une telle application est dite *ubiquitaire* : ses fonctionnalités sont mises à disposition sur plusieurs machines. Ainsi l'utilisation de la même application peut se faire sans que la nature de l'équipement ne soit un obstacle.

Le concept d'application ubiquitaire répond à l'évolution des réseaux dynamiques. La mobilité des utilisateurs d'une part et l'intégration physique des équipements et donc des moyens d'accès à l'information dans ces réseaux d'autre part, exigent que les logiciels qui y sont déployés soient omniprésents et que les accès aux services et aux données puissent se faire simplement, c'est-à-dire en masquant la complexité des interactions entre les équipements.

Nous utilisons volontairement l'adjectif *ubiquitaire* à la place d'*ubiquiste*. En prenant la définition du dictionnaire, la différence entre ces deux qualificatifs est qu'un objet ubiquiste *est* présent partout à la fois, alors qu'un autre, ubiquitaire, *semble* être présent partout à la fois. Si l'on transpose cette définition du dictionnaire aux logiciels, une application sera qualifiée d'ubiquiste lorsqu'elle sera (ainsi que les données qu'elle manipule) installée sur chacun des équipements alors qu'une application ubiquitaire permet seulement l'accès à ses fonctionnalités depuis différentes machines, sans que cela implique que ces dernières soient installées sur chacune d'entre elles.

Parmi les exemples d'applications ubiquitaires, nous pouvons citer celui de la télécommande universelle ubiquitaire. Avant d'être ubiquitaire, la télécommande universelle permet le contrôle d'une grande partie des appareils électriques que l'on peut trouver chez soi (e.g chaîne hifi, lecteur de dvd, télévision, lumières, etc.) et ce, depuis un seul équipement. Outre

le gain de place sur la table du salon, une telle télécommande offre la possibilité de regrouper en un seul endroit l'ensemble des actions nécessitant jusqu'alors plusieurs télécommandes. Cependant, l'utilisation de cette télécommande nécessite d'avoir un accès physique à cette dernière. Les fonctionnalités de la télécommande pourraient être rendues ubiquitaires s'il était possible de contrôler sa chaîne hifi depuis n'importe quel ordinateur de la maison (e.g. de son assistant numérique personnel), pour que l'on puisse jouer de la musique sur les enceintes présentes dans la pièce où l'on se trouve[3].

Dans ce travail thèse, nous nous donnons comme objectif de définir un cadre de développement pour la définition d'applications ubiquitaires fonctionnant dans un environnement dynamique et pervasif.

1.2 Les composants logiciels

Les applications ubiquitaires, répondant à l'évolution des réseaux dynamiques, définissent des schémas complexes de distribution et de déploiement. Le développement par composants est maintenant reconnu comme simplifiant la réutilisation, la réduction des coûts, l'interopérabilité ainsi que la gestion de la complexité des applications [Szy02]. Les composants logiciels sont donc utiles dans le contexte de développement d'applications ubiquitaires. Une application construite à base de composants logiciels est vue comme un assemblage, plus ou moins complexe, d'entités réutilisables : les composants. Un composant est défini par ses fonctionnalités, c'est-à-dire les services qu'il propose, ainsi que par ses dépendances vis-à-vis d'autres composants, i.e. les services qu'il requiert. Cette définition des composants logiciels a été formulée lors de la 10^{e} Conférence Européenne sur la Programmation Orientée Objets (ECOOP'96) :

> A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties.

Cette définition exprime les différents concepts que nous retrouvons dans la plupart des technologies de composants et des modèles associés que nous détaillerons dans la section 2.1 du chapitre 2. La figure 1.3 illustre la structure d'un composant logiciel. Ses fonctionnalités sont représentées par les interfaces fournies et ses dépendances par ses interfaces requises.

La réalisation d'un logiciel construit à l'aide de composants se fait par assemblage de ces derniers. Chaque composant étant défini par ses services (interfaces) fournis et requis, l'assemblage de deux composants consiste à créer une *liaison* entre une interface requise de l'un avec une interface fournie par l'autre. Cette mise en relation des interfaces représente la résolution de la dépendance d'un composant (celui qui possède l'interface requise) vis-à-vis du composant proposant l'interface fournie.

L'intérêt de l'approche par composants est qu'elle permet d'assembler des composants développés par différents fournisseurs de composants. Par exemple, supposons que l'on possède un composant DocumentSearch que l'on a acheté et qui offre des fonctions de recherche sur des documents de tout type : par exemple trouver des documents en fonction de leur date de

[3] La gamme Pronto développée par Philips est un exemple de télécommande universelle mais non ubiquitaire : `http://www.pronto.philips.com/`

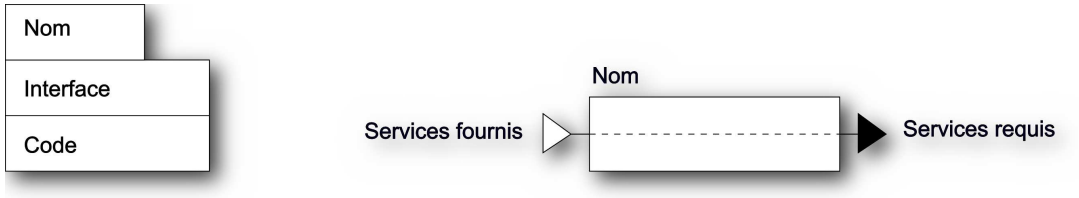


FIG. 1.3: Un composant logiciel est défini par un ensemble d'interfaces définissant chacune soit une fonctionnalité (un service fourni) soit une dépendance (un service requis d'un autre composant). Le code associé à un composant implante les fonctionnalités de ce composant.

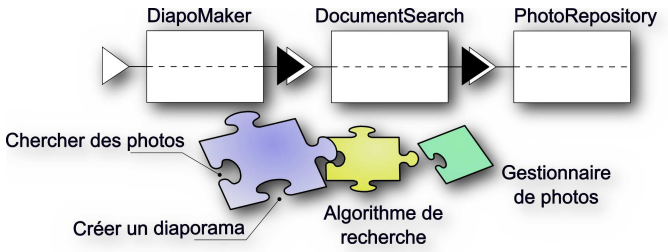


FIG. 1.4: DiapomaKer : une application à base de composants logiciels

création, de méta informations associées au document, etc. Pour fonctionner, ce composant requiert un autre composant offrant la possibilité de parcourir et récupérer des documents. Ainsi, si nous connectons DocumentSearch à un composant spécialisé dans le stockage de photos, il nous est alors possible de faire des recherches sur un ensemble de photos.

À son tour, DocumentSearch peut être utilisé par un autre composant qui, s'appuyant sur les fonctionnalités de recherche, offre la possibilité de créer un diaporama des photos trouvées. La figure 1.4 décrit l'assemblage de composants constituant un tel logiciel de photos, DiapoMaker.

Le développement à base de composants logiciels apporte des réponses quant à la spécification et la maintenance d'applications (distribuées). Les technologies ayant adopté cette approche, par exemple les Enterprise Java Beans de Sun ([MH99]), .Net de Microsoft ([dot]) sont utilisées, essentiellement dans des environnements stables, que l'on peut difficilement qualifier de dynamiques. Il est donc important dans un premier temps de déterminer si les composants logiciels sont adaptés aux environnements que nous visons. Plusieurs travaux de recherche ont apporté des éléments de réponses pour des environnements pervasifs. Nous avons étudié ces approches afin de définir les besoins et les concepts à ajouter aux modèles de composants traditionnels.

Nous étudions dans ce travail de thèse l'utilisation des composants logiciels comme brique de conception des applications ubiquitaires.

1.3 L'approche intergicielle

Les réseaux dynamiques rendent difficiles la conception et la gestion des applications ubiquitaires. Il est nécessaire de tenir compte notamment de l'hétérogénéité des équipements et

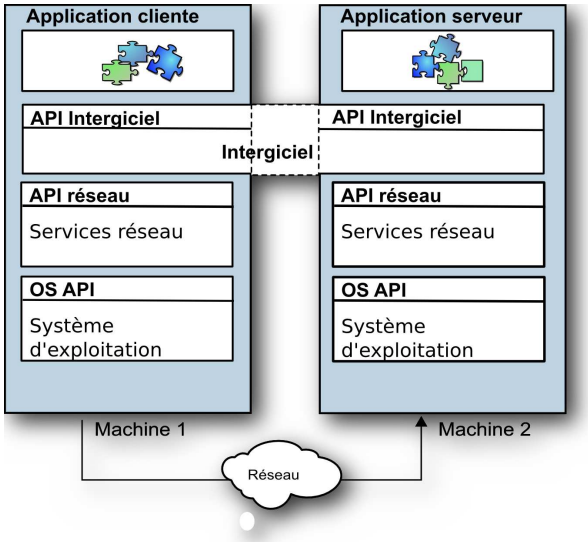


FIG. 1.5: Localisation de l'intergiciel.

des problèmes d'accès aux ressources. L'approche intergicielle offre un cadre de développement permettant aux développeurs d'applications de s'abstraire des spécificités de chaque équipement et de s'appuyer sur des interfaces bien définies dans le but de faire communiquer les différentes parties de leurs applications. Pour cela, les services intergiciels [Ber96] offrent des protocoles et des interfaces de programmation standardisés en se *logeant* entre les services de niveau système et les applications qui s'y exécutent.

La figure 1.5 illustre l'organisation générale d'une application distribuée utilisant un intergiciel : les composants de l'application utilisent les fonctions de l'intergiciel pour communiquer entre eux de façon transparente. C'est l'intergiciel qui utilise les services de base du système d'exploitation et l'environnement matériel sous-jacent pour fournir ces fonctions de communication de haut niveau. Chaque équipement possède donc une instance de l'intergiciel.

Étant donnée la dynamicité des réseaux sur lesquels les applications ubiquitaires doivent s'exécuter, la tâche pour le programmeur de telles applications s'avère difficile voire impossible sans la définition d'abstractions nécessaires pour la réalisation des logiciels ubiquitaires. L'approche intergicielle semble donc être une bonne approche pour la définition de supports d'exécution des applications ubiquitaires.

Nous proposons de suivre l'approche intergicielle pour définir un support d'exécution conférant une certaine autonomie aux applications ubiquitaires construites à base de composants.

1.4 Exemples

Nous présentons dans cette section deux scénarii qui décrivent l'utilisation de deux applications ubiquitaires. Ces applications *jouets* vont être reprises tout au long de ce mémoire, nous permettant d'illustrer notre contribution. Ces exemples introduisent chacun un aspect

que nous avons isolé pour présenter notre contribution. L'application DiapomaKer met l'accent sur les caractéristiques ubiquitaires des logiciels tandis que UbiKrisis se concentre sur les questions relatives au déploiement des applications ubiquitaires dans les réseaux dynamiques.

1.4.1 DiapomaKer

Au sein de sa maison, la famille Kaloubadja possède plusieurs équipements informatiques. Les membres de cette famille ont l'habitude de préparer des diaporamas pour leurs invités. Le logiciel DiapomaKer permet de créer un diaporama et offre comme avantage la possibilité de le faire depuis n'importe quel ordinateur que possède la famille Kaloubadja. Ainsi, il est tout à fait envisageable de préparer un diaporama depuis un PDA et de le visualiser ensuite sur l'ordinateur portable par exemple. Il n'est donc plus nécessaire d'utiliser l'unique machine sur laquelle se trouve le composant qui génère le diaporama.

Les composants constituant DiapomaKer sont les suivants :

- un composant, DocumentSearch, offrant des fonctions de recherche sur des documents de tout type. Ce composant permet de trouver des documents en fonction de leur date de création, de méta informations associées au document, etc. ;
- un composant, PhotoRepository, prenant en charge la gestion d'une bibliothèque de photos. Couplées au composant précédent, les fonctions de recherche sur les photos sont alors possibles ;
- un composant DocumentBuffer, stockant les photos répondant aux critères de recherche ;
- et un dernier composant réalisant la création du diaporama à partir des photos sélectionnées précédemment, DiapoMaker.

Chacun de ces composants nécessite des ressources particulières pour leur bon fonctionnement. Le composant DocumentSearch requiert un processeur cadencé au minimum à 1,3 GHz pour un confort d'utilisation. Le composant PhotoRepository quant à lui exige d'être installé sur une machine ayant au moins 200 Mo d'espace disque disponible afin de pouvoir stocker un minimum de photos.

Ces exigences en termes de ressources matérielles ne peuvent être satisfaites par toutes les machines que possèdent les Kaloubadja et empêchent donc l'installation de tous les composants constituant DiapomaKer sur chacune de leurs machines. L'ordinateur portable de la famille est le seul équipement répondant aux exigences du composant DocumentSearch mais il est souvent utilisé pour les déplacements de Mr Kaloubadja. Il serait dommage de ne plus pouvoir utiliser DiapomaKer malgré l'absence de DocumentSearch : les photos peuvent être toujours accessibles et il est encore possible de les visualiser.

DiapomaKer répond au besoin d'utilisation d'un logiciel, indépendamment de l'équipement utilisé. Du fait de contraintes empêchant l'installation de tous les composants de l'application sur chaque machine, DiapomaKer met en jeu des composants distribués. Par ailleurs, certaines fonctionnalités (comme la sélection de photos) de cette application peuvent ne pas être rendues lorsque des machines ne sont pas présentes. Cela ne doit cependant pas empêcher DiapomaKer de fonctionner.

1.4.2 UbiKrisis

Après un tremblement de terre plusieurs équipes dotées d'équipements mobiles (e.g. ordinateurs portables, PDA) sont envoyées sur le terrain afin d'évaluer l'état des différents sites.

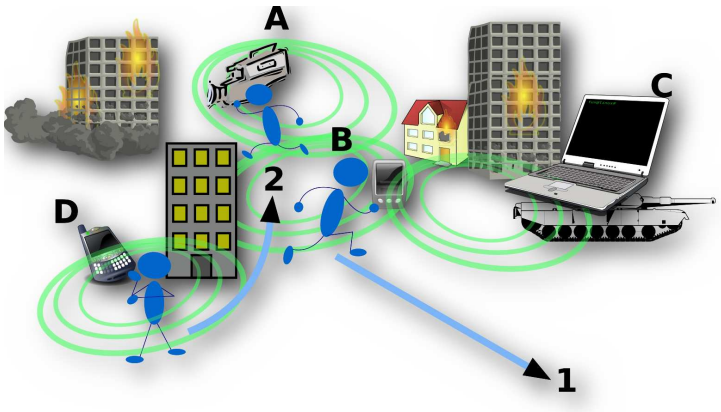


FIG. 1.6: UbiKrisis : une application de gestion de crise, ubiquitaire

Leur objectif est d'évaluer la gravité des dégâts. Sur place, les équipements constituent un réseau mobile ad-hoc (MANET, *Mobile Ad-hoc NETwork* en anglais). Dans un tel réseau, les machines communiquent entre elles via un canal radio (e.g. IEEE 80211b/g) sans le support d'une quelconque infrastructure. Chaque machine relaie de gré à gré les messages aux autres machines qui sont à sa portée. Les MANETs constituent des réseaux dynamiques et offrent une alternative adaptée aux situations où aucune infrastructure de communication n'est plus disponible.

UbiKrisis est un logiciel ubiquitaire permettant aux secouristes équipés d'un appareil mobile d'avoir une évaluation des dégats au fur et à mesure de la progression des équipes sur place. Il est constitué de plusieurs composants :

- un composant permettant d'enregistrer un flux vidéo obtenu à partir d'une caméra ;
- un composant qui convertit les vidéos en images qui ont été retravaillées afin d'éliminer les différents parasites (e.g. des fumées) ;
- un composant stockant les différentes images ;
- un composant comparant les images avec des images d'archives afin de mettre en évidence les défauts d'architecture suite à la catastrophe.

Chacun de ces composants exige des ressources spécifiques et ne peut être installé sur toutes les machines (e.g. sur la figure 1.6 ces composants sont respectivement installés sur les machines A, B, C et D). UbiKrisis permet d'avoir accès aux fonctionnalités de chacun de ces composants depuis les différents mobiles du MANET, pourvu que la fonctionnalité désirée soit accessible (i.e. mobile allumé et à portée radio).

Par ailleurs en tant qu'application ubiquitaire, UbiKrisis doit prendre en considération les déconnexions. Par exemple sur la figure 1.6 lorsque le le secouriste possédant le mobile B se déplace vers 1 (après avoir entendu des appels à l'aide), les flux vidéos ne peuvent être transformés en images (A et B ne sont plus à portée l'un de l'autre). Il est cependant intéressant de noter que D peut jouer le rôle de passerelle s'il se déplace en 2.

Un autre aspect lié aux applications ubiquitaires est le choix du placement des composants constituant l'application. Dans le cas d'UbiKrisis, le choix du placement des composants ne peut pas être décidé avant de connaître tous les équipements mis en jeu sur le terrain. Par exemple, des équipes de secouristes supplémentaires peuvent se rendre plus tard sur les lieux de la catastrophe, amenant avec eux des équipements plus performants que ceux déjà présents.

Ainsi, l'apport de ces nouvelles ressources peut remettre en question le placement initial des composants dans le but de rendre disponibles des fonctionnalités manquantes.

1.5 Énoncé du problème étudié

Nous avons décrit dans ce chapitre le contexte d'étude de notre travail de thèse. Dans le cadre d'un réseau dynamique, composé d'équipements hétérogènes, mobiles et volatiles, nous souhaitons développer des outils permettant de mettre à disposition des utilisateurs des applications ubiquitaires, c'est-à-dire des logiciels qui autorisent l'utilisation de leurs fonctionnalités via n'importe quelle machine du réseau.

Les difficultés majeures pour atteindre cet objectif sont d'une part liées à la nature des réseaux que nous visons, qui exigent par exemple la prise en compte des déconnexions et des caractéristiques des différents équipements. D'autre part, le concept d'application ubiquitaire exige la mise en place d'un support d'exécution prenant en compte la dynamicité du réseau.

Dans ce chapitre nous avons soulevé un certain nombre d'interrogations auxquelles nous devons répondre :

La distribution. Le problème de la distribution est inhérent à la nature même des applications ubiquitaires. Une application ubiquitaire est censée offrir ses services sur un ensemble de machines alors que l'ensemble des composants qui la constituent ne peuvent être installé sur chacune d'elles, pour des raisons de disponibilité de ressources notamment. Le problème de la distribution et de la gestion distribuée d'une application ubiquitaire soulève les questions suivantes :

- en quoi la conception d'une application ubiquitaire se différencie-t-elle de la conception d'une application répartie ?
- quels sont les mécanismes ou les schémas de distribution spécifiques aux applications ubiquitaires ?
- les modèles de composants logiciels existant sont-ils adaptés pour la conception de telles applications ?

La dynamicité des ressources. Les réseaux considérés dans ce travail de thèse sont caractérisés par leur dynamicité due aux fluctuations de ressources présentes dans l'environnement mais aussi à la mobilité et la volatilité des différents équipements mis en jeu. Les déconnexions réseau ne doivent plus être considérées comme des exceptions mais plutôt comme des phénomènes « naturels » et admis. Les applications ubiquitaires doivent donc supporter ces différents changements. Comment détecter ces variations et quelles sont les réactions possibles à ces changements ?

Le déploiement. Les questions précédentes font référence au problème du support des applications ubiquitaires dans les réseaux dynamiques. Avant de pouvoir utiliser ces applications, un prérequis est leur installation sur les différents équipements du réseau. Ce déploiement doit prendre en compte la dynamicité des ressources et les déconnexions réseau à la fois dans la spécification du déploiement de l'application et dans les processus de décision de placement des composants.

2

Composants logiciels distribués

Les applications ubiquitaires sont par nature distribuées, c'est-à-dire mettant en jeu des entités réparties sur les différents équipements constituant un réseau dynamique. Pour concevoir de telles applications, nous nous intéressons à l'approche par composants. Plusieurs modèles et technologies de composants ont été proposés pour la conception et le support d'applications distribuées.

Ce chapitre introduit dans un premier temps les concepts véhiculés par les modèles de composants et leur support d'exécution. Dans un second temps, un état de l'art des technologies de composants significatives est présenté, en mettant en avant les mécanismes de distribution.

Sommaire

2.1 Introduction générale sur les composants logiciels

2.1.1 Des objets aux composants

Dans cette thèse, nous nous intéressons à la conception et à la programmation d'applications ubiquitaires à base de composants. L'ingénierie du logiciel *orientée composant* a connu un essor considérable ces dix dernières années avec l'apparition de technologies industrielles telles que les Enterprise Java Bean de Sun ([MH99]), le modèle de composant Corba (CCM, Corba Component Model) de l'OMG ([Obj02]) ou encore .Net de Microsoft ([dot]). Les travaux sur les composants logiciels ont amené plusieurs modèles à être définis (e.g. Koala [vOvdLKM00], Sofa [PBJ98], Pecos [NAD+02], Fractal [BCL+04], Pin [ISW02], UML2.0) et les concepts sous-jacents sont maintenant clairement identifiés. En effet, depuis la première conférence sur l'ingénierie du logiciel organisée par l'OTAN en 1968, les efforts pour définir clairement ce qu'est un composant logiciel ont été nombreux (notamment WCOP 96 [Mue97], 2001 [Fro02]) et révèlent par ailleurs les lacunes de la programmation orientée objets (POO). En introduisant les concepts d'héritage — qui permet la définition d'une nouvelle classe en la faisant hériter des propriétés et comportements d'une autre classe — et de polymorphisme — qui rend l'utilisation d'une seule définition pour l'utilisation de plusieurs types de données —, la POO accroît la modularité et la réutilisabilité des systèmes (et de leurs parties) modélisés par des classes et des interfaces. Cependant, la vision du « tout objet » dans laquelle les fonctionnalités de l'application (la partie métier) et les services que cette dernière requiert (les services non fonctionnels, e.g. la persistance, la sécurité) seraient parfaitement modélisés par des objets, est utopiste : cette *séparation des préoccupations* est rendue difficile du fait de l'interaction forte entre le code métier et le code non fonctionnel et nécessite la plupart du temps de changer l'implantation d'un objet (rompant ainsi le principe de réutilisation). Par ailleurs, la POO souffre du manque de mécanismes et d'outils permettant la composition d'objets dès lors nécessaire lorsque des considérations de passage à l'échelle doivent être prises en compte. Cette limitation est accentuée par le couplage implicite qui existe entre les objets, couplage qui est introduit par l'existence au sein du code même des objets de références vers d'autres objets. Il est ainsi très difficile d'identifier, au sein d'une application mettant en jeu un grand nombre d'objets, les dépendances qu'un objet possède vis-à-vis des autres. Ces dépendances sont enfouies dans le code source, rendant abscons la possibilité d'identifier des objets précis en vue de leur réutilisation dans une autre application. Le paradigme objet ne fournit donc pas de réponses satisfaisantes au principe de substitution.

L'ingénierie logicielle basée composant (CBSE pour *Component Based Software Engineering* en anglais) apporte des éléments de réponses à ces lacunes en se fixant comme objectifs le développement de logiciels à partir de *reproducing parts*, la réutilisation de ces parties dans d'autres applications et de faciliter la maintenance et l'évolution des parties en vue de définir de nouvelles fonctionnalités ([HC01]). Ainsi, les modèles de composants se concentrent sur les notions d'assemblage et de substitution qui peuvent être réalisées de par la nature des composants. Ces aspects sont mis en avant par exemple par Jed Harris, président de CI Labs :

Un composant est un morceau de logiciel suffisamment petit pour que l'on puisse le créer et

le maintenir, et suffisamment gros pour que l'on puisse l'installer et en assurer le support. De plus il est doté d'interfaces standards pour pouvoir inter opérer

ou bien Bertrand Meyer [Mey03] :

Un composant est un élément logiciel (unité modulaire) qui satisfait les conditions suivantes :

1. *il peut être utilisé par d'autres éléments logiciels, ses clients.*
2. *il possède une description de son usage permettant aux concepteurs de* clients *de l'utiliser*
3. *il n'est lié à aucun ensemble fixe de clients*

Cependant, la plupart des définitions que l'on peut trouver dans la littérature ([HC01, Szy02]) ne spécifient pas aussi précisément que pour les modèles à objets les différentes entités d'un modèle de composants, ce qui peut expliquer la prolifération des différents modèles ou autres technologies mettant en jeu des composants logiciels. Par exemples, CCM et EJB, pourtant reconnus comme des standards, définissent différemment la notion de composants et ainsi l'assemblage de ces derniers. De ce fait, le principe de substitution n'est applicable qu'au sein d'un même modèle de composants[4]. Cependant, bien que chacun de ces modèles possède des spécificités propres, nous pouvons dégager un ensemble d'entités communes à ces modèles. Notre objectif ici est de pouvoir appliquer les différentes notions développées au travers de ce travail de thèse à différents modèles de composants[5]. Nous présentons donc maintenant les abstractions communes aux principaux modèles de composants. Le lecteur pourra se référer à [Lau06] pour une projection de ces différentes abstractions sur les différents modèles cités plus haut.

Composant Un composant désigne une entité logicielle qui est conforme à une spécification. La spécification d'un composant désigne tout document décrivant les fonctionnalités ainsi que les dépendances du composant. Cette spécification est nécessaire pour qu'une opération de composition puisse être définie.

Interfaces Les fonctionnalités d'un composant représentent ce que le composant « sait faire ». Par exemple un composant de chiffrement offre la possibilité de chiffrer une chaîne de caractères, un fichier etc. Les différentes fonctionnalités que propose un composant sont regroupées et spécifiées dans une ou plusieurs *interfaces* qui sont qualifiées d'interfaces *fournies* (appelées aussi interfaces *serveurs*[6]). Un composant peut utiliser les fonctionnalités d'autres composants. Cette dépendance entre composants est définie par l'existence d'interfaces *requises* (appelées aussi interfaces *clientes*). Les interfaces d'un composant représentent les spécifications que les composants doivent suivre.

Une interface est une liste d'opérations (appelées aussi méthodes) et de propriétés (appelées aussi attributs) regroupées pour former une fonctionnalité, un service offert par un

[4]Les travaux de [FM04] utilisant l'approche MDA [MM03] proposent de s'abstraire d'un modèle spécifique pour définir un assemblage de composants.

[5]Cette volonté de s'abstraire d'un modèle de composants particulier se retrouve notamment dans l'approche MDA [St00] où des méta-modèles de composants sont définis ainsi que les correspondances entre les différents modèles traités ([Tib06, Bar05]).

[6]Avec le développement des applications pair-à-pair et des approches orientées services, il n'est pas rare de trouver dans la littérature l'emploi de *services fournis* (ou tout simplement, services) pour désigner les fonctionnalités offertes par un composant.

composant. Un composant proposant une interface fournie implante donc l'ensemble des opérations et propriétés qui y est définie. Un composant mettant en jeu une interface requise, accède aux opérations et aux propriétés de cette interface dont l'implantation est fournie par un autre composant.

Les interfaces fournies et requises d'un composant définissent le contrat d'assemblage de ce dernier. Deux composants vont pouvoir être assemblés en mettant en relation une interface requise de l'un avec une interface fournie de l'autre. Les interfaces représentent donc les points de connexions des composants.

Type d'une interface Les interfaces spécifient les fonctionnalités et dépendances des composants. La liste des opérations et des propriétés d'une interface définit son type.

Type d'un composant Le type d'un composant est défini par le type de ses interfaces requises et fournies.

Implantation d'un composant Nous désignons par implantation d'un composant, l'ensemble des codes (sources ou binaires) ou tout autre document qui, en conformité avec le type d'un composant, réalisent les services fournis du composant.

Instance de composant À partir d'une implantation, un composant peut être instancié, i.e. être représenté à l'exécution par une entité possédant une identité unique : son instance. Une instance de composant est conforme à son type.

Liaisons Une liaison permet de modéliser l'interaction entre deux composants au niveau de leurs interfaces : lorsqu'un composant utilise les services d'un autre composant, cela se manifeste par l'existence d'une liaison entre une interface requise du premier et une interface fournie du second composant. Une liaison réalise donc la dépendance entre deux composants.

Une liaison entre une interface requise et fournie n'est possible que si ces dernières sont compatibles : l'interface fournie contient au moins l'ensemble des opérations et propriétés définies dans l'interface requise[7] .

Une liaison n'est pas forcément représentée par une entité à l'exécution mais exprime plus généralement l'association entre l'interface cliente d'un composant et l'identité du composant fournissant le service requis. Par exemple, une référence Java valide entre deux objets est une liaison. La figure 2.1 illustre la définition et la représentation graphique d'une liaison entre deux composants en UML 2.0.

Ports Lorsque l'on souhaite réaliser un assemblage de composant, les notions d'interfaces et de types d'interface ne sont pas toujours suffisantes pour décider des composants à interconnecter. On peut souhaiter par exemple relier des interfaces en fonction de leur propriétés en

[7]Par exemple, dans un modèle de composant mis en œuvre dans un langage à objets, une interface requise est compatible avec une interface fournie, si l'interface fournie est un sous-type de l'interface requise.

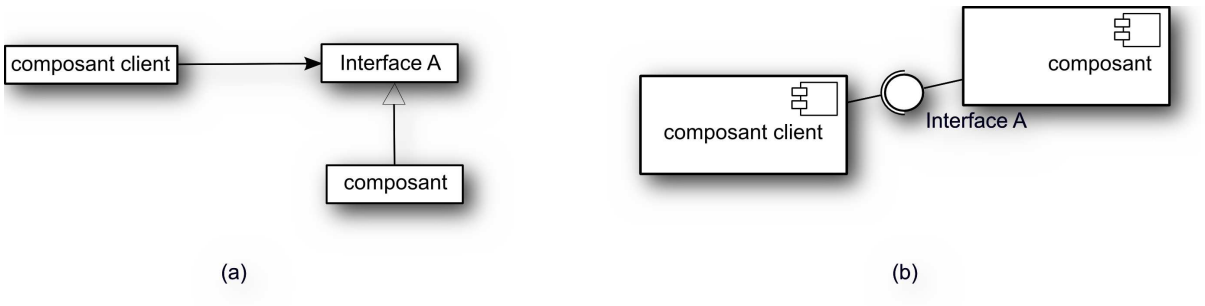


FIG. 2.1: Deux représentations en UML de deux composants liés. Une interface publique est définie par le composant qui fournit toutes les opérations souhaitées. Le composant client accède à ces opérations uniquement via cette interface. La définition de l'interface ne fait aucune hypothèse sur l'implantation faite par le composant.

plus de leurs types. La notion de port à été introduite à cet effet et permet de regrouper des interfaces.

Ces différents concepts apportent un réel avantage à concevoir une application à l'aide de composants en minimisant la relation de couplage entre les différentes entités de l'application. De cette manière, il est désormais possible de changer l'implantation d'un composant sans conséquence particulière sur ses (composants) clients.

2.1.2 Assemblage de composants

L'apport des modèles de composants dans la conception et le développement des logiciels réside dans la notion d'assemblage qui permet d'une part la définition des entités qui devront être assemblées mais aussi l'explicitation des interactions entre les différents composants, i.e. une meilleure compréhension du système. La maîtrise de la structure des applications fait l'objet de beaucoup de travaux car contrôler l'organisation des différents composants assure d'une part le bon fonctionnement de l'application toute entière mais également son développement, sa maintenance et son évolution. Ainsi en 1996, le comité de standardisation de l'ingénierie logicielle (SESC) de l'IEEE a été chargé de définir les termes et les principes concernant l'architecture logicielle [EIS+96]. L'architecture logicielle est une discipline qui s'intéresse aux structures d'un logiciel et a permis, grâce aux langages de description d'architecture (ADL pour *Architecture Description Language*), la formalisation du raisonnement et de l'analyse au niveau architectural.

Dans la plupart des ADL ([MT00]), une architecture consiste en un graphe de composants interconnectés. Les composants (les nœuds du graphe) encapsulent des unités de calculs ou bien d'autres composants. Les connexions entre les composants (les arcs) représentent les liaisons et modélisent ainsi leurs interactions. Dans les ADL, le terme *connecteur* est préféré à celui de liaison. En effet, un connecteur peut modéliser bien plus qu'une simple interaction entre deux composants : dans le but de faciliter la maintenance et la réutilisation des composants, un connecteur peut définir la sémantique de la communication (e.g. asynchrone), le format des messages échangés, etc.

Il est fréquent également de trouver au niveau des ADL la notion de composant hiérar-

chique. La composition hiérarchique consiste à définir un composant à partir de l'assemblage d'autres composants. Lorsque qu'un modèle de composants autorise la composition hiérarchique, il est qualifié de modèle hiérarchique, dans le cas contraire, on dira que c'est un modèle de composants « à plat ».

Les interfaces ou les ports sont utilisés comme points d'interactions entre composants. Ainsi à l'aide d'un ADL, on peut décrire une *configuration* à base de composants et de connecteurs. Une configuration doit être vue comme un assemblage statique de composants qui peut ensuite être instanciée.

La phase de conception d'une application à base de composants amène à définir et identifier les composants et aussi les relations entre ces derniers. Comme nous l'avons décrit précédemment, il en résulte une architecture de composants qui représente les différentes entités qui auront à coopérer à l'exécution. L'assemblage ainsi *constitué*, bien que descriptif, impose des contraintes au support d'exécution des composants : pour chaque composant de l'architecture il sera nécessaire de créer une instance de ce composant, et pour toutes les liaisons que met en jeu ce composant, la plate-forme devra lier les instances respectives des composants au niveau de leurs interfaces. Il est donc important de noter qu'une architecture de composants définit un objectif pour la plate-forme hébergeant l'application en terme d'instanciation de composants et de réalisation des liaisons entre composants.

Dans les technologies de composants actuelles, une configuration est décrite — via un ADL — dans un *descripteur d'architecture* qui liste l'ensemble des composants ainsi que les liaisons entre composants.

2.1.3 Support d'exécution d'applications à base de composants

L'utilisation des technologies à composants permet aux concepteurs et programmeurs de se focaliser sur le développement de composants métiers : chaque composant représente une fonctionnalité propre à un (ou plusieurs) domaine(s) d'application. Les différentes tâches consistant à intégrer ou considérer des propriétés externes à la logique applicative telles que la gestion du contrôle de la cohérence des données, la distribution des composants ou encore les aspects liés à la sécurité, sont prises en compte par une ou plusieurs entités extérieures qui correspondent à des services que l'on qualifie de services intergiciels.

Nous présentons dans cette section les fonctions des intergiciels, supports d'exécution des applications à base de composants, puis nous détaillons la notion de conteneur sur laquelle s'appuie les infrastructures de composants.

2.1.3.1 Intergiciels

Les travaux autour des intergiciels sont nombreux ([Ber96, Sch95, SS01]) et ont comme objectif commun de masquer l'hétérogénéité des plates-formes d'exécution ainsi que la répartition des différentes entités qui doivent interagir. Cependant, selon les paradigmes de programmation considérés (e.g. la programmation asynchrone, programmation par objets), plusieurs catégories d'intergiciels ont été étudiées : les intergiciels transactionnels (*distributed tuples*), l'appel de procédure à distance [Mic88] (RPC, *Remote Procedure Call*), les intergiciels orientés message (MOM, *Message-Oriented Middleware*), les intergiciels à objets (ou composants) distribués. Pour plus de détails sur les principes, les apports et les différences concernant les approches que nous venons de citer, le lecteur pourra se référer à [Emm00]. Les frontières

entre les paradigmes précédents ne sont pas opaques et les notions introduites par chacune d'entre elles peuvent s'entrelacer. Par exemple, certains intergiciels transactionnels s'appuient aussi bien sur RPC que sur l'envoi de messages asynchrones pour la réalisation des communications.

Pour bâtir notre réflexion autour des intergiciels, nous nous intéressons aux intergiciels à objets [8]. Cette catégorie d'intergiciels est une des plus répandues et a bénéficié des avantages de la programmation par objets de ces dix dernières années. Un intergiciel orienté objet propose les abstractions nécessaires pour que les paradigmes de la programmation par objets, tels que l'encapsulation, l'héritage et le polymorphisme, soient possibles dans un contexte distribué. Ainsi, un objet (i.e. une instance) situé dans un espace d'adressage différent (e.g. machine ou machine virtuelle distante), peut recevoir des appels de méthodes, de la même manière que s'il s'exécutait localement.

La plupart des intergiciels à objets possèdent une structure en couche illustrée par la figure 2.2. Ce découpage[9] a pour but de simplifier la construction de services intergiciels de niveaux d'abstraction différents ; chaque niveau s'appuyant sur celui qui lui est inférieur afin de proposer un service plus général :

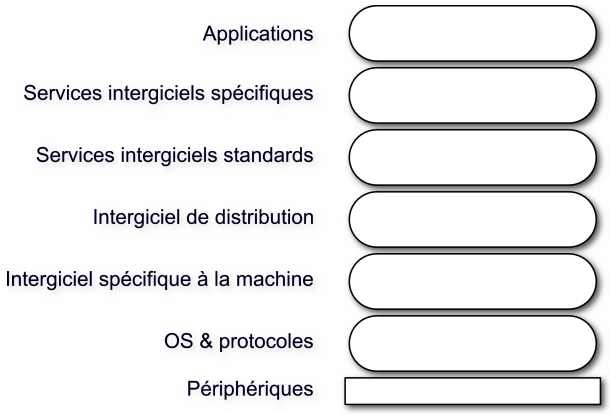


FIG. 2.2: Représentation en couches des services d'un intergiciel et des couches englobantes

Services intergiciels spécifiques à la plate-forme L'objectif des services de ce niveau est de masquer l'hétérogénéité des systèmes d'exploitation (OS) en offrant des abstractions aux primitives de communication et de cohérence spécifiques aux OS. Par exemple, la communication inter-processus basée sur les sockets est réalisée à l'aide de bibliothèques propres à chaque système d'exploitation, e.g. l'API Berckeley pour les systèmes Unix et Winsock pour Windows. Ainsi, deux versions d'un même programme doivent être écrites pour chacun de ces OS, ce qui peut engendrer des erreurs et nuire à la maintenabilité du programme. Un exemple d'intergiciel permettant de s'abstraire de ces spécificités est la machine virtuelle de Sun [LY99] qui permet l'écriture de programmes communicants aux travers d'une même API (i.e. `java.net.Socket`).

[8] L'étude des intergiciels à objets se transpose facilement aux intergiciels à composants. En effet, la très grande majorité des technologies à composants est programmée via un langage à objets. Ainsi les différents mécanismes (e.g. liés à la distribution) mis en jeu dans les intergiciels à objets se retrouvent dans ceux à composants.

[9] Le découpage présenté ici ainsi que la description qui en est faite sont extraits de l'étude menée par R.E. SCHANTZ et D.C SCHMIDT ([SS02]).

Services intergiciels de distribution Les services responsables de la distribution, utilisent l'API offerte par les services spécifiques de la plate-forme afin de rendre transparente la répartition des objets. L'API résultante permet la programmation d'une application distribuée de façon (quasi) transparente : les services de distribution prenant en charge les aspects liés aux protocoles de communication, à la localisation des objets, etc. JavaRMI [WRW96] permet l'invocation de méthodes entre deux objets s'exécutant dans des machines virtuelles distinctes sans modifier la syntaxe des appels sur les objets distants : tout se passe comme si ces derniers résidaient sur la machine virtuelle locale.

Services intergiciels communs Lors de la conception d'une application distribuée, plusieurs difficultés apparaissent du fait de la répartition des entités considérées. Par exemple, des aspects liés à la sécurité des communications ou encore l'ordonnancement des requêtes sur un serveur sont des mécanismes qui doivent être régulièrement pris en compte. Afin d'éviter l'écriture redondante de ces fonctionnalités, les services intergiciels communs, en se basant sur l'API de répartition offerte par les services de distribution, proposent des services réutilisables, i.e. indépendants de toute application.

Services intergiciels spécifiques les services intergiciels spécifiques sont décrits comme des fonctionnalités programmées explicitement pour des classes d'application (e.g. dans le secteur bancaire, l'aérospatiale, etc.) et qui s'appuient sur les différents niveaux décrits précédemment et non plus uniquement sur le niveau directement inférieur.

2.1.3.2 Les conteneurs de composants

En plus de fournir des services que l'on retrouve dans les intergiciels à objets (e.g. les services de transaction, de persistance, de journalisation, etc.), les intergiciels à composants réalisent des fonctions propres au support et à la manipulation des composants. Parmi les fonctions que doivent fournir les intergiciels à composants, nous pouvons citer toutes celles permettant la mise en place d'une configuration de composants à partir de la description de son assemblage, i.e. les fonctions de création des composants, de mise en place des liaisons, de recherche de composants, de nommage, etc.

La mise en œuvre de ces différentes fonctions s'appuie sur la notion de *conteneur* qui se retrouvent dans la plupart des technologies de composants. La figure 2.3 illustre l'architecture d'un intergiciel à composants définissant un conteneur. Le conteneur agit comme une « maison » de composants avec laquelle les autres composants interagissent. Les conteneurs abritent les composants métiers de l'application ainsi que tous les services non-fonctionnels requis par les composants (e.g. un service de transaction).

Nous détaillons maintenant quelques services intergiciels pris en compte par les conteneurs de composants.

Les services de nommage et courtage Le service de nommage permet de retrouver un composant grâce à son nom. Le service de courtage fournit un service équivalent en prenant comme paramètre les propriétés du composant. On peut faire une analogie entre le service de nommage et de courtage avec respectivement les pages blanches et les pages jaunes de l'annuaire téléphonique.

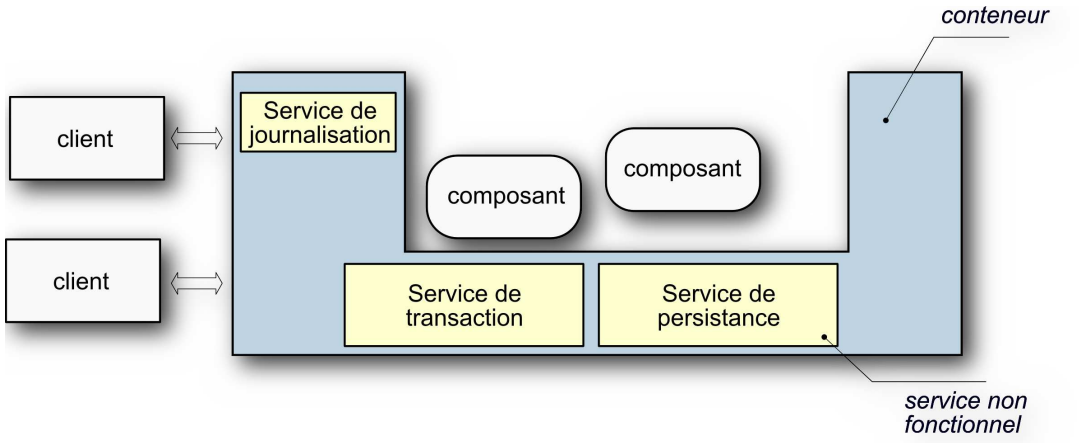


FIG. 2.3: Architecture à conteneur

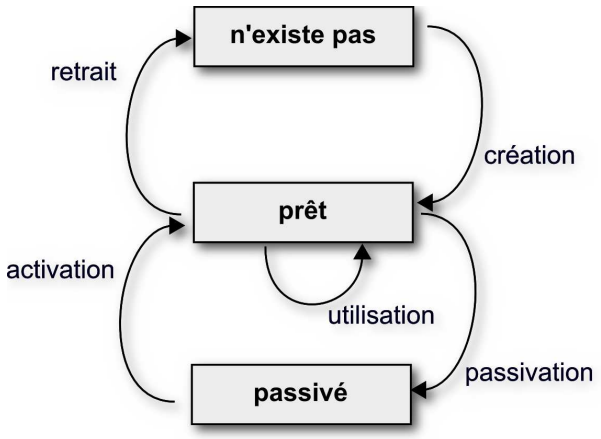


FIG. 2.4: Cycle de vie d'un composant

Les services de gestion du cycle de vie Les aspects liés à l'administration et à la maintenance des composants dépendent fortement de l'environnement d'exécution de ces derniers. Par exemple dans un serveur d'application J2EE, il est nécessaire de définir des politiques d'équilibrage de charge ou de *pooling* du fait du grand nombre de composants mis en jeu. Le développeur de l'application n'a pas à avoir connaissance de ces politiques qui définissent la gestion du cycle de vie des composants. Cette responsabilité est déléguée au conteneur. La figure 2.4 illustre le cycle de vie d'un composant EJB pour lequel les opérations d'activation et de passivation sont pris en charge par le conteneur. La plupart des technologies de composants permettent la redéfinition des états pris en compte dans le cycle de vie et les règles de passage d'un état à un autre.

Les services d'assemblage de l'architecture L'assemblage de composants constituant l'application est généralement décrit via un langage de description d'architecture (ADL) spécifiant les composants et leurs interconnexions. La création des composants doit donc être complétée par une phase de mise en place des liaisons (les phases de création et de liaison peuvent aussi bien être définies de façon séquentielle, parallèle ou ad-hoc). Cette dernière est également prise en charge par le conteneur : la programmation des liaisons au sein même des composants briserait le principe de réutilisation souhaité par ces derniers. Par ailleurs, dans le cas de deux composants distants, il est nécessaire de mettre en place des entités prenant en charge la distribution des composants. Cette tâche est dévolue au conteneur afin de ne pas *polluer* le code des composants de l'aspect « distribution ».

Les services personnalisés Le conteneur contrôle toutes les interactions des composants avec le monde extérieur. Ce contrôle est réalisé à travers des entités appelées les objets de contrôle. Dans la plupart des modèles de composants, le conteneur offre au minimum deux objets de contrôle, un intercepteur des appels entrant et un intercepteur des appels sortant. L'intercepteur des appels entrant est un objet de contrôle externe (exporté par le conteneur) ; il est visible de l'extérieur du composant et offre les mêmes interfaces que le composant. L'intercepteur des appels sortant est un objet de contrôle interne ; il n'est visible que de l'intérieur du conteneur (local). Le conteneur peut contenir d'autres objets de contrôle. Chaque objet de contrôle représente un accès vers un service non-fonctionnel de l'intergiciel ou bien réalise un service que le conteneur utilise dans la gestion des composants. Ces services peuvent être interrogés lorsque le composant reçoit ou émet des invocations.

Un exemple de services mis en œuvre par les objets de contrôle est le service de journalisation : en interceptant les requêtes des composants clients avant de les transmettre aux composants concernés, il est possible de sauvegarder des informations sur le nombre de requêtes, le temps de traitement, etc.

2.2 Distribution de composants

En séparant clairement les éléments logiciels d'une application, l'approche par composant permet de déléguer la gestion des interactions entre composants au niveau des services intergiciels. Dans le cas des applications distribuées, les modèles de composants n'intègrent pas toujours les aspects liés à la distribution des applications. C'est souvent grâce à des mécanismes intergiciels que l'on peut faire interagir des instances de composants distants. Ces

mécanismes définissent des services de distribution qui se retrouvent sous des formes plus ou moins similaires dans les différentes technologies de composants alors qualifiées de distribuées.

Nous présentons dans cette section les concepts manipulés au niveau des services intergiciels de distribution avant d'étudier les mécanismes mettant en œuvre les interactions entre composants distants. Enfin nous dresserons un état de l'art des technologies de composants distribués.

2.2.1 Composants distants, références distantes, liaisons distantes

Un application définie à l'aide de composants, est décrite par une configuration qui, une fois instanciée, est représentée à l'exécution par un ensemble d'instances de composants. Afin d'utiliser les services offerts par une instance particulière d'un composant, il est nécessaire d'avoir une information désignant parfaitement ce composant. Ainsi, une *référence d'un composant* est un identifiant qui se réfère à un composant particulier.

Une *référence d'un composant distant* doit en plus comporter les éléments nécessaires pour désigner la localisation du composant ainsi que les moyens d'accès à ce dernier (e.g. protocole de communication utilisé pour communiquer avec le composant).

L'interaction avec un composant se fait au travers de ses interfaces ce qui permet de découpler l'implantation d'un composant de ses dépendances. Les interfaces sont des entités qui ont une existence à l'exécution, et tout comme les composants, elles sont désignées par des références. Une *référence d'interface distante* désigne une interface d'un composant distant.

Comme nous l'avons présenté dans la section 2.1, l'interaction entre les composants est explicitée par les liaisons entre les interfaces requises et fournies des composants et représente l'utilisation d'un composant serveur par un composant client. Lorsque le composant serveur est distant (par rapport au composant client), la liaison reliant leurs interfaces est appelée *liaison distante*.

2.2.2 Mécanismes intergiciels pour la distribution

L'appel de méthode d'un composant client vers un composant distant, c'est-à-dire l'appel d'une méthode sur une de ses interfaces fournies, repose sur un mécanisme principal défini par le patron de conception Proxy [VCK96] dont l'objectif est de rendre transparente la distribution du côté du composant client. Le patron Proxy est généralement complété par le patron Broker qui facilite la mise en place des liaisons distantes entre composants. Nous détaillons ces deux patrons ci-dessous.

2.2.3 Patron de conception Proxy

L'appel de méthode d'un composant client vers un composant distant repose sur la présence du côté de l'appelant, d'un mandataire[10] client (*talon*). L'idée principale dans l'utilisation des mandataires ([Sha86]) est de faire réaliser l'appel du composant client sur une entité locale (le mandataire) au lieu du composant distant, rendant ainsi la distribution des composants transparente. De même, il existe sur le site du composant appelé, un mandataire serveur

[10]Le terme anglais *proxy* est traduit en français par *représentant* ou *mandataire*

(*squelette*) qui prend en charge les invocations entrantes des différents clients afin de les déléguer au composant serveur. Le talon serveur est donc un représentant de tous les composants clients potentiels. Le mandataire client et le mandataire serveur définissent un schéma d'interaction de type client-serveur et communiquent entre eux afin d'échanger les requêtes et réponses correspondant à l'invocation.

Dans un mode de communication synchrone, le composant client est bloqué pendant la phase de requête / réponse, i.e. il attend la réponse du serveur pour reprendre son exécution. À l'inverse, dans les modèles de communication asynchrone, le client n'est pas contraint par l'attente de la réponse du composant serveur, i.e. il peut effectuer d'autres opérations pendant le traitement de la requête par le composant serveur.

Dans les technologies à composants les mandataires client sont fournies généralement par le conteneur et implante les mêmes interfaces fournies que le composant qu'il représente. Les requêtes et réponses relatives à un appel de méthode distant, étant transmises sur le réseau, nécessitent l'utilisation d'un format adéquate généralement dépendant de l'infrastructure sous-jacente. Par exemple, la sérialisation en un flux d'octet est la technique la plus usité dans la plupart des intergiciels. De plus, le format des messages échangés doit correspondre à un standard afin de garantir une certaine portabilité entre les différents sites. La conversation des données dans un format transportable est appelé l'emballage (*marshalling* en anglais). L'opération inverse correspond au déballage (*unmarshalling*).

La structure interne d'un mandataire suit un schéma bien défini [Kra] qui facilite sa génération automatique :

- une phase de pré-traitement, qui consiste essentiellement à emballer les paramètres et à préparer le message de requête,
- l'appel effectif au composant serveur, utilisant le protocole de communication sous-jacent pour envoyer la requête et pour recevoir la réponse,
- une phase de post-traitement, qui consiste essentiellement à déballer les valeurs de retour.

2.2.4 Patron de conception Broker

La mise en œuvre du patron Proxy permet de faire communiquer des composants distants mais des services comme l'ajout, le retrait, l'échange, l'activation ou la localisation des composants sont aussi requis. Lorsque les applications utilisent ces services, elles ne devraient pas dépendre du système d'exécution sous-jacent afin de garantir une certaine portabilité ainsi que la réalisation de la distribution même dans le cas d'un réseau hétérogène.

Le patron de conception Broker est un patron architectural qui fournit une vue synthétique d'un intergiciel [Buschmann et al., 1996]. Son rôle est de « [..] structurer les applications réparties à l'aide de composants découplés qui interagissent par invocations de services distants. Un composant courtier ('Broker') prend en charge la coordination et la communication entre composants, tels que l'aiguillage de requêtes ou la transmission des résultats et des exceptions ».

Le patron Broker définit ainsi les composants Broker, Proxy, Client, Serveur, Référentiel et Passerelle. Ces composants interagissent de la façon suivante : les Serveurs s'enregistrent auprès du Broker grâce au Référentiel, et rendent des services accessibles par des Clients en publiant leurs interfaces. Les Clients accèdent aux Serveurs en échangeant des requêtes au travers du Broker, qui aiguille la requête jusqu'au Serveur destinataire, puis renvoie la réponse au Client.

Les modules Proxy et Passerelle gèrent les mécanismes de communication et assurent l'échange de données entre plates-formes hétérogènes.

2.2.5 État de l'art

Dans la section précédente, nous avons présenté les mécanismes qui régissent la distribution des composants dans les intergiciels. Nous allons maintenant étudier quelques technologies à composants, représentatives de l'ensemble de l'offre tant industrielle qu'académique, en mettant en avant les aspects liés à la distribution des composants.

2.2.5.1 Le modèle de composant Corba (CCM)

Le modèle de composants CORBA (CCM pour *Corba Component Model*) fait partie des spécifications de CORBA version 3.0 [Obj02] définies par l'OMG. Il représente le premier modèle industriel pour des composants métiers distribués, hétérogènes, indépendants des langages de programmation, des systèmes d'exploitation et des fournisseurs d'implantations. CCM spécifie l'intégralité du cycle de vie des composants (création, exécution, packaging, assemblage, et déploiement).

Modèle de composants. Les composants CCM s'exécutent au sein de conteneurs qui fournissent certains services système (transactions, sécurité, tolérance aux fautes, etc.) et qui utilisent un bus CORBA [Sie99] pour l'acheminement des communications entre les ports. Ainsi, conformément au modèle de composants logiciels, les composants CCM ne contiennent que du code métier, et les aspects non-fonctionnels de l'application sont définis par des descripteurs externes aux composants. Les conteneurs prennent en charge un seul type de composant. Ils contiennent une usine de composants qui se charge de la création et de la destruction d'instances de ce type de composant. Ils contiennent également des objets d'interposition pour les ports des composants. Les conteneurs permettent de gérer ces objets d'interposition à l'aide de deux interfaces : une interface d'introspection qui fournit des informations sur le type des composants, sur ses ports, ainsi que sur l'état de ses connexions avec les autres composants ; et une interface de gestion qui permet de créer des connexions entre les composants.

Un composant interagit avec les autres composants en utilisant des interfaces fonctionnelles (ou ports) dont il existe quatre types différents :

- une facette est une interface serveur qui peut être utilisée par des clients en mode synchrone,
- un réceptacle est une interface cliente en mode synchrone,
- un puits d'événements est une interface serveur qui peut être utilisée par des clients en mode asynchrone,
- une source d'événements est une interface cliente en mode asynchrone.

Par ailleurs, un composant possède des attributs qui permettent de le configurer lors de son déploiement ou pendant son exécution.

Prise en compte de la distribution CORBA fournit un modèle de communication client-serveur entre les composants, où les clients demandent le traitement de services à des serveurs distants en utilisant une interface bien définie qui est spécifiée via une extension du langage de définition d'interface (IDL, *Interface Definition Language*) de CORBA, CIDL (*Component IDL*).

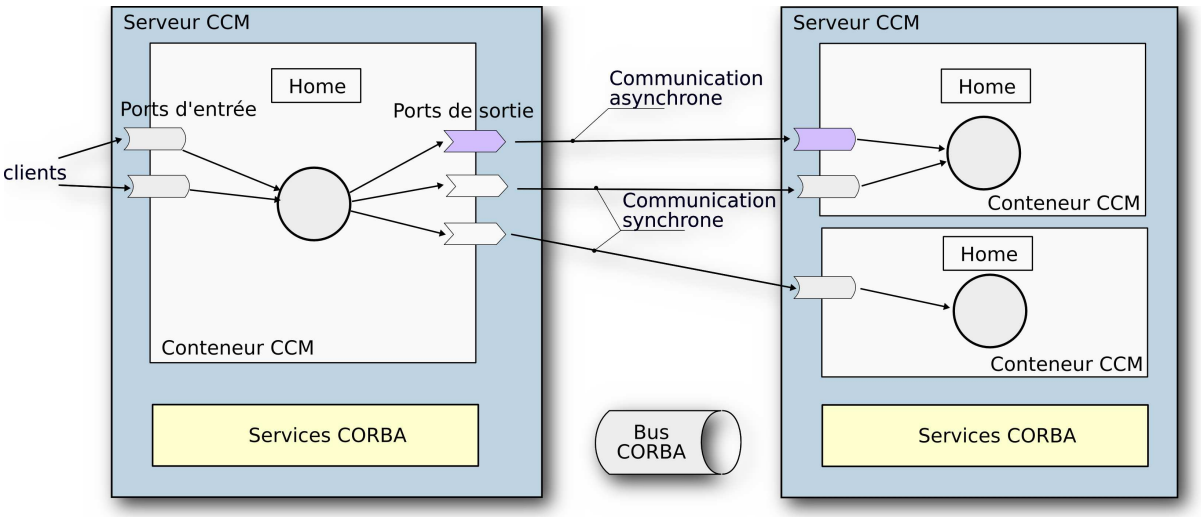


FIG. 2.5: Architecture des composants CCM

L'épine dorsale de CORBA est le courtier d'objet ORB (*Object Request Broker*) qui a pour fonction de localiser les objets et de transmettre les requêtes, événements et données aux objets distants. Les objets n'interagissent jamais directement les uns avec les autres mais passent par l'ORB.

Pour assurer la communication entre un composant client et un composant serveur, le composant serveur fournit (cf. figure 2.6) un fichier IDL qui définit l'interface que le composant client doit utiliser pour appeler ses services. Un compilateur IDL génère automatiquement les talons et squelettes. La gestion de la connexion entre ces entités est réalisée suivant le protocole GIOP (*General Inter-ORB Protocol*). GIOP peut être appliqué à tout type de protocole de transport orienté connexion. Le protocole s'appuyant directement sur TCP/IP définit le protocole IIOP (*Internet Inter-ORB Protocol*).

CORBA fournit les services de nommage et de courtage permettant ainsi d'assurer la transparence de localisation des composants.

2.2.5.2 EJB

Modèle de composants EJB (Enterprise Java Beans) est spécifié par Sun Microsystems[MH99]. Les EJB (appelés aussi *beans*) sont des objets Java dédiés à la construction d'applications J2EE. Ils implantent les interfaces nécessaires pour interagir avec leur environnement d'exécution, le conteneur des EJB. Le code des EJB représente la partie métier d'une application J2EE alors que le conteneur offre des services liés aux propriétés dites non-fonctionnelles, telles que la communication, les transactions et les aspects sécurité.

Un composant EJB se présente sous la forme d'un ensemble de classes Java regroupées dans un fichier de description. Chaque composant forme ainsi une unité réutilisable qui peut être combinée à d'autres pour former une application. La spécification EJB 2.x définit trois principaux types d'EJB : les beans sessions, les beans entités et les beans pilotés par messages. Chacun de ces composants est un programme de type serveur, destiné à traiter les requêtes d'autres programmes, comme des servlets ou des JSP ou toute autre application Java.

La figure 2.7 illustre l'interaction d'un client avec un composant EJB. Chaque compo-

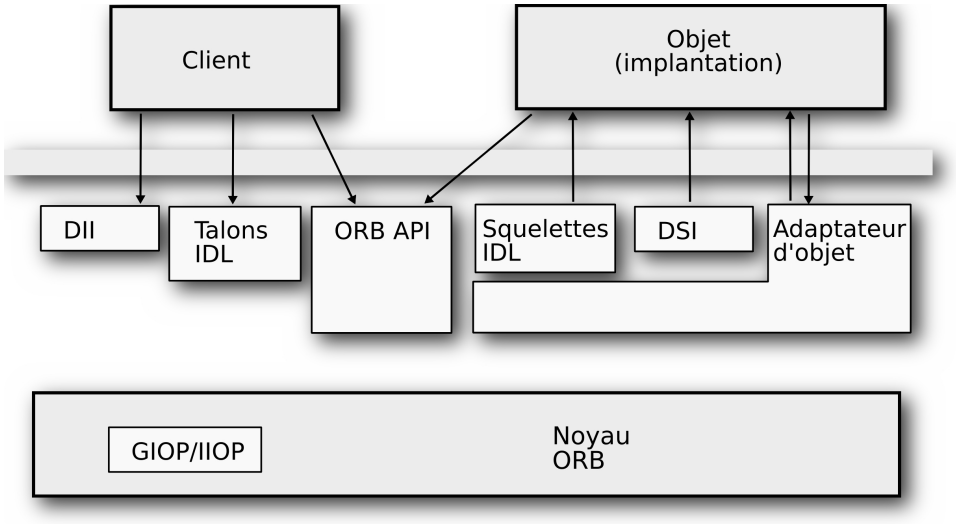


FIG. 2.6: L'architecture de l'Objet Request Broker dans CORBA

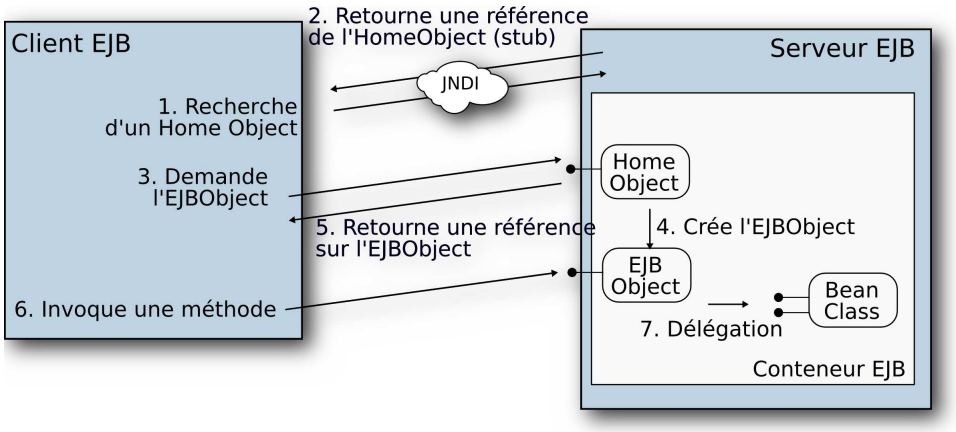


FIG. 2.7: Interaction avec un composant EJB

sant fournit deux types d'interfaces : l'interface `EJBHome` et l'interface `EJBObject`. L'interface `EJBHome` définit les méthodes de gestion du cycle de vie du composant. Les clients invoquent les opérations sur cette interface pour créer, supprimer les composants et trouver des instances particulières de composants. L'interface `EJBObject` spécifie les signatures des méthodes correspondant à la partie métier du composant.

Nous retrouvons ici l'utilisation du patron PROXY. L'`EJBObject` joue le rôle de mandataire. Il implante les interfaces (au sens Java) et transmet les invocations au composant (bean) fourni par le développeur.

Gestion de la distribution Les communications au sein de J2EE reposent soit sur Java RMI ([Mic]) soit sur GIOP ([KL00]). Les souches et squelettes permettant de rendre la distribution transparente sont générés à l'aide du compilateur `rmic`. J2EE fournit deux utilitaires de nommage : le registre RMI (*RMI-registry*) réalisant un espace de nommage plat et JNDI (*Java*

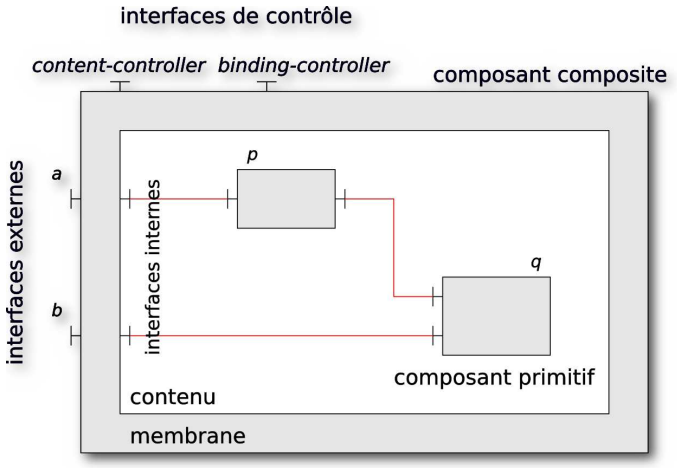


FIG. 2.8: Structure d'un composant Fractal

Naming Directory Interface), similaire au service de nommage de CORBA. L'avantage cependant de JNDI est de pouvoir s'abonner à des types d'événements ayant pour origine un changement de l'espace de nommage (e.g. l'ajout, le retrait, ou la mise à jour d'une liaison). Les serveurs d'application de composants J2EE ne fournissent pas encore un service de courtage.

2.2.5.3 Fractal

Modèle de composant 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 2.2.5.3 présente l'architecture d'une application constituée d'un composant composite contenant deux sous-composants primitifs.

La notion de composant partagé est également spécifiée dans Fractal. Un composant partagé est un sous-composant de plusieurs composants composites englobants et permet de modéliser des ressources et leur partage, tout en préservant l'encapsulation des composants.

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

Nous reviendrons plus en détail sur le modèle Fractal dans le chapitre 5.

Mécanisme de distribution Le modèle de composant Fractal ne définit pas la notion de composants distribués. Cependant des outils ou des implantations spécifiques du modèle

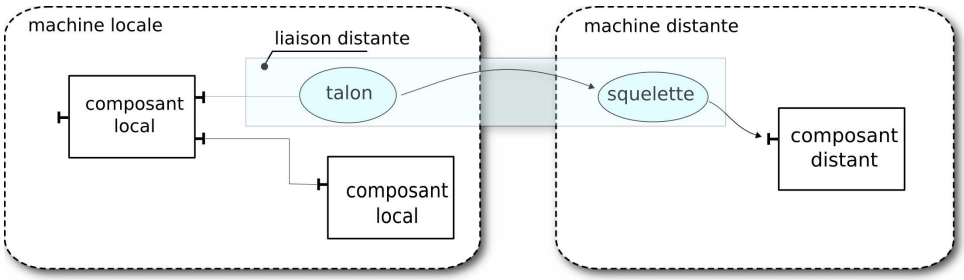


FIG. 2.9: Composants et liaisons distantes dans le modèle de composant Fractal

(e.g. Fractive, que nous décrivons dans le paragraphe suivant) permettent la programmation et le déploiement d'applications distribuées. FractalRMI[11] fournit un ensemble de composants pour créer des liaisons distribuées entre composants Fractal. FractalRMI s'appuie sur l'ORB Jonathan pour réaliser la communication entre composants distants. La figure 2.9 montre les mandataires clients et serveurs générés par FractalRMI. Nous détaillons plus en détail les mécanismes de distribution de FractalRMI dans la section 4.2.3 du chapitre 4.

2.2.5.4 Fractive

Fractive [BCM03] est une implantation du modèle Fractal utilisant l'intergiciel ProActive [CKV98]. Proactive est une bibliothèque écrite en Java qui fournit la notion d'objets actifs distribués. De tels objets peuvent être invoqués à distance et de façon asynchrone grâce aux références futures. Une application distribuée écrite à l'aide de ProActive est constituée de plusieurs activités, chacune possédant un point d'entrée bien identifié, l'objet actif, accessible de n'importe où. Tous les autres objets d'une activité (i.e. les objets passifs) ne peuvent être référencés directement en dehors de cette activité. Chaque activité possède son propre service de traitement des requêtes qui peut être redéfini par le programmeur. L'appel de méthode sur un objet actif se déroule de la manière suivante :

- l'appel de méthode sur un objet (e.g. `y=OB.m(x)`) est mise en file au niveau de l'objet appelé et une référence vers un objet futur `f` est créée et retournée (`y` référence maintenant `f`);
- au bout d'un certain temps, la requête est traitée sur l'activité appelée. La requête est enlevée de la file et la méthode est exécutée ;
- lorsque la méthode termine, la valeur du résultat est mise à jour : la référence sur l'objet futur `f` est remplacée par la valeur de retour de la méthode (la valeur de `y`). Lorsqu'une activité essaie d'accéder à une référence future avant que celle-ci soit mise à jour, elle reste bloquée jusqu'à ce que l'appel soit réalisé (attente par nécessité).

Mécanismes de distribution La manipulation d'objets actifs distants et donc de composants distants repose sur le concept de nœud virtuel. Un nœud virtuel représente une abstraction de la localisation d'un objet (composant) qui est manipulable au sein d'un programme. Pendant la conception et la programmation des composants, aucune URL, adresse IP n'est utilisée. Cela est fait au moment du déploiement en définissant, dans un descripteur, la correspondance

[11] `http://fractal.objectweb.org/fractalrmi/`

```
<virtualNodesDefinition>
 <virtualNode name='Renderer'
              property='multiple'/>
</virtualNodesDefinition>
 <mapping>
  <map virtualNode='Renderer'>
   <jvmSet>
    <currentJvm/>
    <vmName value='Jvm1'/>
    <vmName value='Jvm2'/>
   </jvmSet>
  </map> </mapping>
```

Listing 2.1: Distribution d'un même composant dans Fractive sur différentes machines virtuelles

entre nœuds virtuels et machines virtuelles Java (JVM). Les machines virtuelles définissent également une correspondance vers les machines du réseau. Plusieurs protocoles peuvent être spécifiés pour créer des JVMs sur les machines (ssh, Globus, LSF, rsh, services Web, etc.).

Il est possible dans Fractive de spécifier qu'un composant composite réside sur plusieurs nœuds virtuels, comme cela est illustré dans le listing 2.1. Ceci permet de répartir les sous-composants sur ces différents nœuds (e.g. placement en cycle des composants avec la propriété `multiple_cyclic`). Un composant composite peut donc résider sur plusieurs JVMs. Cela signifie que les services de ce composant sont accessibles depuis chacun des nœuds virtuels sur lesquels s'exécutent un mandataire (patron Proxy) du composant composite.

2.3 Discussion

La programmation par composants logiciels est maintenant reconnue pour faciliter la conception d'applications complexes mais aussi leur support et maintenance. Les technologies présentées dans ce chapitre étendent les travaux autour de la programmation par objets afin de permettre aux concepteurs et développeurs d'applications de se focaliser sur les aspects métiers de leurs applications.

En ce qui concerne les applications distribuées, les services intergiciels de distribution ont pour vocation de faciliter l'interaction des composants lorsque ces derniers résident sur des machines différentes. Les patrons de conceptions Proxy et Broker sont largement répandus dans les intergiciels du fait qu'ils rendent transparente la communication entre composants distribués. L'existence de tels services simplifie grandement la conception et la réalisation d'applications distribuées puisque les ressources distantes (e.g. les ressources systèmes, les autres composants) sont vues et utilisées comme des ressources locales.

Les mécanismes présentés dans ce chapitre partagent un même objectif : rendre les aspects liés à la distribution transparents, à la fois pour le programmeur d'application et pour les utilisateurs. Cependant, en cachant la distribution, ces mécanismes n'intègrent pas les aspects liés aux déconnexions et aux fautes en général : les applications distribuées sont conçus de la même manière qu'une application centralisée.

Nous donnons ci-dessous les défis auxquels les modèles de composants ainsi que leur support d'exécution doivent répondre :

- *Consommation en ressources*. Certains des équipements (e.g. les assistants personnels) qui peuvent constituer les réseaux dynamiques comme ceux décrits dans la section 1.1.2, sont limités en ressources alors que les technologies de composants actuelles, non prévues pour ce genre d'appareils, exigent une grande quantité de mémoire et des capacités de calculs relativement conséquentes pour offrir les services non fonctionnels. La totalité de ces services est généralement *intégré* dans les conteneurs mais peut ne pas être nécessaire pour toutes les applications.
- *Connectivité du réseau*. Les technologies de composants qualifiées de distribuées sont généralement conçues pour des environnements ayant une connectivité permanente, une grande bande passante et une faible latence pour l'acheminement des appels distants. Les réseaux dynamiques, du fait de leur dynamicité, introduisent des déconnexions réseau fréquentes. Par conséquent, les références vers les composants distants sont souvent invalidées. L'approche consistant à considérer ces déconnexions comme des erreurs nuit au développement et au maintien des applications dans de tels environnements.
- *Complexité*. Les modèles de composants actuels et les technologies associées reposent généralement sur une architecture client-serveur où le nombre et la localisation des clients ainsi que des serveurs sont parfaitement maîtrisés. Dans le cadre de nos travaux, la mobilité des équipements et la nature sporadique des connexions entres machines résultent en la formation d'un réseau à la topologie changeante et complexe. Les systèmes distribués considérés par les technologies tels que EJB, CCM et .Net, ne considèrent pas de telles structures.

The reasonable man adapts himself to the world ; the unreasonable one persists in trying to adapt the world to himself. Therefore all progress depends on the unreasonable man.

George Bernard Shaw, Maxims for Revolutionists : Reason, Man and Superman (1903)

3

Composants logiciels pour réseaux dynamiques

Les technologies à composants ainsi que leur support d'exécution décrits dans le chapitre précédent sont conçus pour des environnements stables. Il faut doter ces composants de facultés supplémentaires pour qu'ils soient utilisables dans des réseaux dynamiques. Ces capacités doivent servir essentiellement à permettre aux composants de s'adapter aux conditions fluctuantes.

Nous présentons dans ce chapitre un état de l'art des techniques autorisant ces adaptations. Nous définissons d'abord celles visant à adapter l'architecture même de l'application. Dans un second temps, nous détaillons des travaux portant sur des techniques d'adaptation centrées sur le comportement d'un composant individuel. Nous présentons ensuite quelques travaux significatifs dans le domaine des intergiciels quant à la prise en compte des adaptations dans des réseaux dynamiques.

Sommaire

3.1 Adaptation et cohérence architecturale

Pendant son exécution, une configuration de composants est sujette aux fluctuations imposées par les réseaux dynamiques pouvant amener au dysfonctionnement de l'application toute entière. Les composants logiciels doivent permettre l'adaptation de l'application ou, directement, pouvoir s'adapter aux différents changements pour continuer à offrir leurs services.

La mise en place d'un support d'adaptation pour les architectures de composants se décompose généralement selon trois temps :

- *l'observation*. Avant qu'une modification s'impose, il faut déterminer *quand* elle doit être effectuée. La phase d'observation a pour objectif de répondre à cette question ;
- *la décision*. Lorsqu'une adaptation doit être réalisée, il faut déterminer les éléments qui vont faire l'objet des changements. Il s'agit de répondre ici à la question du *quoi*, i.e. qu'est ce qui est susceptible de changer dans l'architecture ?
- *l'action*. Enfin, il reste à déterminer *comment* vont se réaliser les différentes modifications.

Le premier volet de ce triptyque consiste à observer les ressources (matérielles et logicielles) requises par les composants. La plupart des travaux (e.g. COSMOS [CRS07], DACAR [DM06], JADE [BdHT06], [GCS03], [GSC01], [VK02]) s'appuient sur l'utilisation de sondes et de jauges comme celles proposées dans les systèmes GANGLIA [MCC04], LeWys [CELQ05], Phœnix [SBF02] ou encore SAJE [CGLSM03]). Les sondes permettent l'observation d'une ressource spécifique sur une plate-forme donnée comme par exemple en calculant la mémoire libre disponible sur une machine, ou encore en estimant la taille d'un fichier, etc. Les jauges, en s'appuyant sur une ou plusieurs sondes, définissent des abstractions de plus haut niveau et peuvent être utilisées directement par l'application elle-même.

Concernant les phases de décision et d'action, les travaux traitant de l'adaptation se distinguent plus nettement en proposant des mécanismes spécifiques. Pour mieux comprendre ces travaux dont nous donnons un état de l'art dans la section suivante, nous présentons ci-dessous, les concepts et mécanismes qui régissent l'adaptation des architectures à composants.

3.1.1 Mécanismes d'adaptation architecturale

L'adaptation d'une architecture de composants consiste à apporter des modifications sur les éléments de cette architecture. Les opérations d'adaptation vont donc concerner les composants et leur liaisons.

Nous nous intéressons dans cette section aux différentes formes d'adaptation ou de reconfiguration qu'il est possible de réaliser sur une architecture à base de composants.

Substitution d'un composant En tant qu'unité de composition, un composant peut être remplacé par tout autre composant pourvu que ce dernier fournisse les mêmes fonctionnalités (voire plus) que le composant à remplacer. En effet le composant substituable, possédant au moins une interface, est utilisé par d'autres composants clients qui doivent, après le remplacement, pouvoir à nouveau bénéficier des services qu'ils requièrent. On peut noter que le nouveau composant peut ne pas avoir les mêmes interfaces que le composant à remplacer : s'il en a plus, les interfaces supplémentaires devront être liées. À l'inverse, s'il a moins d'interfaces, cela signifie que la nouvelle implantation intègre les anciennes dépendances.

La granularité de la substitution peut être plus ou moins fine : dans les modèles hiérarchiques par exemple, un composant composite peut être remplacé exactement de la même manière qu'un composant primitif.

Les raisons qui peuvent amener à substituer un composant peuvent être motivées par des questions de performances (e.g. une nouvelle *version* d'un composant implantant le même algorithme de façon plus efficace), d'utilisation (e.g. l'ajout d'un nouveau périphérique nécessitant un nouveau pilote) ou d'évolution (e.g. nouvelle implantation corrigeant des failles de sécurités, des bugs, etc.).

Modification de l'architecture Pour prendre en compte de nouvelles fonctionnalités ou en enlever certaines, devenues obsolètes, des composants peuvent être ajoutés ou supprimés d'une architecture de composants. Ces deux adaptations requièrent une reconfiguration des liaisons (e.g. ajouter des liaisons vers les nouveaux composants, comme il est détaillé ci-après).

- *L'ajout d'un nouveau composant* afin de faire bénéficier l'application du nouveau composant, de nouvelles liaisons doivent être rajoutées vers ses interfaces fournies. Si le composant ajouté comporte des interfaces requises, ces dernières doivent également être liées vers des interfaces offertes par les composants déjà présents ;
- *Le retrait d'un composant* nécessite d'enlever les liaisons *pendantes* qui existaient autour de ce composant. Les liaisons vers ses interfaces fournies doivent être satisfaites par un autre composant (déjà présent ou qui sera ajouté) et les liaisons que requérait le composant doivent être supprimées.

Ces deux opérations impliquent une gestion d'accès aux composants pendant les phases de reconfiguration. Il peu en effet être important d'empêcher l'utilisation d'un composant qui utilise à son tour le composant que l'on est en train de retirer.

L'ajout d'un nouveau composant nécessite d'ajouter des liaisons vers ce dernier. Par ailleurs, selon la sémantique donnée à la liaison entre deux composants (e.g. la référence — au sens de la programmation par objet — vers l'interface d'un composant requis, le nom d'un service requis qui peut être résolu par un système d'annuaire, le canal de communication entre deux composants défini par un protocole de communication spécifique, etc.), une modification de la liaison est parfois nécessaire. Par exemple lorsqu'un composant s'exécutant dans un autre espace d'adressage est utilisé à la place (après substitution) d'un composant local, la nature de la liaison doit être adaptée en changeant le protocole de communication entre les deux composants.

3.1.2 Cohérence des adaptations dynamiques

Les changements qui peuvent être réalisés sur une architecture de composants peuvent être mis en place aussi bien statiquement (i.e. lorsque l'application est arrêtée) que dynamiquement pendant son exécution. Les adaptations considérées dans cette thèse concernent des modifications qui ont lieu à l'exécution, i.e. des modifications qui ont lieu sur une configuration (de composants) qui a été instanciée.

Les principales techniques pour la modification d'une architecture à l'exécution sont souvent mises en œuvre grâce au concept de reflexivité. Cependant les mécanismes d'adaptation qui sont amenés à être définis ne peuvent ignorer les choix de conception et les propriétés de l'architecture sous peine de la complexifier, et ainsi la rendre encore plus difficile à adapter.

3.1.2.1 Approche réflexive

Un système réflexif offre une représentation de lui-même et permet ainsi de raisonner et d'agir sur cette représentation. En s'appuyant sur un niveau de base (le système) et un niveau méta (sa représentation), un système réflexif est capable d'*introspecter* : il peut découvrir, à partir du niveau méta, les fonctionnalités offertes par le niveau de base. L'action d'agir sur le niveau méta, qui a pour conséquence de modifier le niveau de base, s'appelle l'*intercession*.

Dans l'approche par composant, la réflexivité structurelle dénote la capacité pour le support des composants à fournir une réification complète de l'application en terme de structure : le niveau méta est représenté par un graphe de composants (les nœuds du graphe) et de liaisons (les arcs).

Ainsi lorsque l'architecture d'une application à base de composants est réifiée, la modification de l'assemblage des composants en cours d'exécution est répercutée sur le niveau méta, ce qui permet de prendre en compte l'état de l'assemblage de composants instancié. À l'inverse, pour réaliser des opérations d'adaptation sur l'application en cours d'exécution, il suffit de raisonner sur le niveau méta.

3.1.2.2 Cohérence architecturale

Les mécanismes d'adaptation qui sont amenés à être définis ne peuvent cependant ignorer l'architecture initiale de l'application sous peine d'ajouter de la complexité à cette dernière, et ainsi la rendre encore plus difficile à comprendre et à adapter. De plus, l'architecture d'une application est le résultat d'un ensemble de choix architecturaux [Tib06] censés garantir une certaine qualité du logiciel. Ne pas prendre en compte ces préoccupations lors de l'adaptation de l'architecture briserait les propriétés attendues.

Après avoir défini l'architecture de l'application, i.e. les composants et leurs interconnexions, il s'agit d'instancier cette configuration. Cette phase doit garantir que les composants instanciés sont exactement ceux décrits dans la configuration. Pour la majorité des ADL qui ne décrivent que des architectures statiques, cela revient à vérifier que chaque composant a été instancié et que les dépendances entre composants — explicitées par les liaisons — ont été résolues, i.e. que chaque interface requise possède une référence valide vers une interface fournie. L'application s'exécutant est ainsi *cohérente* avec l'architecture spécifiée à la conception.

Lorsque des adaptations s'imposent pendant l'exécution de l'application, on est amené à remplacer des composants et à reconfigurer des liaisons. Les modifications ainsi apportées peuvent aboutir à une configuration qui n'est pas la même que celle qui a été définie initialement. Cette nouvelle configuration peut refléter l'évolution de l'application, i.e. c'est cette nouvelle configuration qui doit être instanciée lors du redéploiement de l'application. Il est alors nécessaire que le descripteur d'architecture soit mis à jour.

De la même manière, l'évolution d'une configuration « sur le papier », i.e. la modification du descripteur d'architecture, doit être répercutée sur les configurations déjà instanciées.

Nous voyons donc que les modifications sur une architecture à composants peuvent se manifester sur des mises à jour du descripteur d'architecture mais aussi sur l'instance de la configuration correspondante. Il est important d'adopter une approche réflexive qui permet de maintenir une cohérence entre ces deux représentations.

3.2 Adaptation centrée sur l'architecture

Pour adapter une application à base de composants, des reconfigurations sont réalisées sur l'assemblage de cette dernière à l'exécution. Beaucoup de travaux se sont concentrées sur le maintien de certaines propriétés des assemblages de composants et définissent des adaptations centrées sur l'architecture ([OGT+99, SG02, GMK02]). Dans ces approches, l'application est modélisée par un graphe reconfigurable de composants interconnectés. Ainsi, à une configuration donnée — décrite via un ADL — correspond un graphe qui représente l'état de la configuration instanciée.

Nous présentons maintenant quelques travaux qui ont trait à l'étude des reconfigurations des architectures à composants. Ces travaux ont été regroupés en deux catégories : ceux qui définissent des schémas d'adaptation de façon programmatique et ceux mettent en jeu des adaptations autonomes, guidées par la simple connaissance de l'architecture des composants. Dans les deux cas, les approches proposées s'appuient sur la notion de *style architectural*. Un style architectural ([AAG93, MKMG97]) définit un ensemble de règles qui identifie 1) les types de composants et de connecteurs qui doivent être utilisés pour l'assemblage d'un système (ou d'un sous-système) et 2) des contraintes locales ou globales qui dictent comment la composition doit être faite.

3.2.1 Adaptation programmée

Plastik Les travaux de [BJC05, JBCG05] définissent l'architecture Plastik qui comprend trois niveaux :

- le niveau *style* comporte les différents types (i.e. composants, interfaces, connecteurs, etc.) présents dans la plupart des ADL ainsi que des invariants précisant comment ces types peuvent être arrangés. Les invariants correspondent aux contraintes définies dans les styles architecturaux ;
- le niveau *système* contient les instances des différents éléments définis dans un style et l'assemblage de ces instances vérifient les invariants imposés par le style :
- le niveau *exécution* correspond au support d'exécution nécessaire pour héberger les différentes instances. Ce niveau contrôle également les reconfigurations de ces instances.

La définition des assemblages de composants et des invariants est réalisée à l'aide d'une extension de l'ADL ACME [GMW00a] (qui permet de décrire un assemblage de composants) auquel ont été ajoutées les structures du langage ARMANI [Mon01] pour la définition des contraintes architecturales.

La prise en compte des fluctuations de l'environnement est traitée en ajoutant quatre extensions au langage ACME/Armani. La première extension permet de définir des opérations de modifications d'architecture lorsqu'un événement particulier se produit à l'aide de la structure (*On* (<predicat>) *do* <actions>). Par exemple, le listing 3.1 définit une stratégie consistant à reconfigurer l'architecture de l'application (suppression de la liaison du composant `MPEG-dec`; suppression du composant `MPEG-dec`; création du composant `H.263` et remise en place de la liaison) dès lors que la valeur de la bande passante transite par la valeur `low`.

Les autres extensions définissent les actions permettant de défaire des liaisons et de supprimer des éléments architecturaux à l'exécution (actions `detach` et `remove` respectivement); de créer des dépendances (dans la chaîne d'instanciation) entre composants. La définition des règles d'adaptation associées à une architecture de composants définit un script de reconfiguration.

```
On (net_bandwidth = low) do {
  detach MPEG-dec.req to conn-dec.p;
  remove MPEG-dec;
  Component H263-dec : decoder = new decoder extended with {
    Property decoder-type = "H26";
  };
  Attachments
    H263-dec.r to conn-dec.p;
}
```

Listing 3.1: Exemple de règle d'adaptation dans Plastik

Le modèle de composant utilisé dans Plastik est OpenCom [CBCP02] dont les propriétés réflexives permettent au support d'exécution de vérifier qu'un script de reconfiguration ne viole aucune contrainte architecturale. Le support d'exécution décrit par les auteurs ne prévoit pas la résolution des conflits de configuration à moins qu'un script de reconfiguration ait été défini spécifiquement pour les éventuelles erreurs.

AcmeStudio Le projet AcmeStudio[12] définit un environnement éponyme qui s'appuie sur l'ADL ACME pour réaliser l'assemblage des applications. Les auteurs de [SG02, GCS03] propose des extensions à AcmeStudio afin de pouvoir définir des styles architecturaux et disposer d'un support d'exécution réalisant lui-même les réparations nécessaires sur une architecture de composants.

La spécification des styles architecturaux, comme dans Plastik, se fait via le langage de contraintes Armani. Cependant les extensions apportées dans [SG02] ajoutent à Armani un langage impératif permettant de définir des « tactiques de réparation » plus évoluées.

Ainsi à partir d'une description d'un assemblage de composants et des contraintes architecturales associées, le support d'exécution met en place un ensemble de jauges responsables

[12] `http://www.cs.cmu.edu/~acme/AcmeStudio/index.html`

de l'observation des propriétés souhaitées de l'application. Lorsqu'une propriété dans l'architecture à l'exécution change, les contraintes sont réévaluées afin de déterminer si le système vérifie toujours toutes les contraintes. Si ce n'est pas le cas, un gestionnaire de réparation choisit une des tactiques de réparation parmi celles définies au préalable afin de rendre la configuration des composants à nouveau valide.

FORMAware Dans sa thèse [MBC02], R. Moreira propose un cadre de conception, FORMAware, qui a pour objectif l'adaptation des applications à base de composants à l'exécution. Pour ce faire, FORMAWare définit un intergiciel qui s'appuie sur les mécanismes de réflexivité pour manipuler d'une part les entités du modèle de composants à l'exécution (e.g. découvertes des interfaces, ajout de composants) et d'autre part, les styles architecturaux qui gouvernent l'assemblage d'un ensemble de composants. Ce dernier point consiste à rendre explicites les règles d'assemblage des composants. Pour ce faire, FORMAware propose un modèle de programmation dans lequel est définie une ontologie qui réifie sous forme d'objets les éléments architecturaux. Ainsi, il est possible pour le concepteur et le développeur de l'application, de spécifier ses propres styles en redéfinissant les méthodes (e.g. `add`, `remove`) qui agissent sur la structure du graphe des composants.

FORMAware ne fournit pas de support pour la description formelle des contraintes architecturales.

Aspects dynamiques dans les ADLs Les travaux cités plus haut (Plastik, FORMAware, AcmeStudio) définissent pour les besoins d'adaptation soit des extensions au niveau des langages de description d'architecture soit des mécanismes au niveau intergiciel qui permettent la programmation de stratégies d'adaptations. Plusieurs travaux se sont intéressés à définir des langages de description d'architecture qui prennent en compte à la fois les styles architecturaux et certains aspects dynamiques. Nous pouvons citer par exemple Acme [GMW00b], Aesop [GAO94], C2 [MORT96] et DynamicWright [ADG98]. Plus récemment, ArchWAre ADL [MKB+04] a été proposé. La particularité de cet ADL est le cadre formel sur lequel il repose (le π-calcul) et l'utilisation d'hyper-code pour la représentation du système à l'exécution (qui peut donc être utilisé pour l'introspection).

3.2.2 Adaptation autonome

Les travaux présentés dans la section précédente définissent programmatiquement des stratégies d'adaptation (ou de réparation) lorsque l'architecture de l'application n'est plus conforme à sa définition, spécifiée par des contraintes architecturales.

Nous décrivons dans cette section des solutions qui permettent l'adaptation de l'application de façon autonome, i.e. une adaptation qui ne met plus en jeu des schémas de réparation préconçus.

Composants-K Dans sa thèse [Dow04], J. Dowling propose un modèle de composants pour la construction de systèmes distribués adaptatifs. Un langage de description de contrats d'adaptation (ACDL, pour *Adaptation Contract Description Language*) est défini et permet de spécifier des règles d'adaptation selon le schéma Événement-Condition-Actions. Le support d'exécution, en fonction des événements observés, peut décider d'exécuter une ou plusieurs

règles d'adaptation. Mais il est possible dans ACDL d'utiliser également des opérateurs de coordination pour la définition d'agents collaboratifs. Ces opérateurs sont définis dans le cadre d'un modèle de coordination baptisé CRL (*Collaborative Reinforcement Learning*). Dans ce modèle, un ensemble dynamique d'agents coopère pour établir et maintenir les propriétés de l'application dans un contexte distribué. Ainsi, il est possible de construire une application où les opérations d'adaptation sont guidées via ACDL sans être programmées.

Unity Les travaux de [CSWW04] s'intéressent à la définition d'applications autonomes [BBC+03] par assemblage de composants, eux-mêmes autonomes. De tels composants sont responsables de la gestion de leurs propres comportements et des interactions avec les autres composants autonomes — afin de proposer ou d'utiliser des services. Ainsi, un composant est responsable de la gestion des ressources qu'il contrôle et des opérations d'auto-configuration qui le concerne. Au lieu de programmer le comportement des composants par des politiques d'adaptation, les auteurs se basent sur une spécification du comportement de plus haut niveau, l'auto-assemblage basé sur les buts (*goal-driven self-assembly* en anglais). Chaque composant, dès son activation ne connaît qu'une description de haut niveau de ce qu'il est censé faire, son but. Un but est défini par la spécification d'un état *souhaité* ou de plusieurs critères qui caractérisent un état souhaité. Chaque composant a alors pour charge de calculer une action, voire une série d'actions qui fera évoluer le système d'un état donné à l'état souhaité.

Les techniques d'intelligence artificielle sont utilisées dans la génération des actions à réaliser pour atteindre un but [KW04].

Programmation par contraintes Les auteurs de [DKM04a, DKM04b] modélisent une architecture de composants comme étant un problème de satisfaction de contraintes ([vHSD94]) à résoudre. Par exemple si une contrainte architecturale précise que le nombre d'instance d'un composant est bornée, une contrainte (au sens de la programmation par contrainte) est définie : une variable pour ce composant est créée et son domaine de définition s'étend (dans l'ensemble des entiers naturels) sur les bornes imposées par le style. Un résolveur de contraintes est utilisé pour attribuer à chaque variable une valeur, ce qui correspond à l'instanciation d'une configuration de composants où chaque contrainte architecturale est vérifiée.

Nous reviendrons plus en détail sur l'utilisation des résolveurs de contraintes et de la programmation par contrainte dans la seconde partie de ce mémoire.

3.2.3 Gestion distribuée des architectures dynamiques

Les plupart des travaux sur l'auto-reconfiguration des applications qui s'appuient sur les architectures logicielles dynamiques ([CGS+02, MRM04b, DvdHT02]) nécessitent un gestionnaire de reconfiguration responsable des prises de décisions quant à la réalisation d'une adaptation sur l'architecture de composants. L'utilisation d'un tel gestionnaire suppose l'existence d'une machine (i.e. un serveur) à tout moment accessible et sur laquelle repose la fiabilité et la cohérence des reconfigurations. Cette approche n'est pas viable dans les réseaux dynamiques que nous considérons étant donné que l'accès à un tel serveur n'est pas toujours possible du fait de la fragmentation du réseau en îlots. Il manque aux approches présentées précédemment une version distribuée du gestionnaire de reconfiguration ou une gestion répartie des

données manipulées par le gestionnaire.

I. Georgiadis propose dans sa thèse ([Geo02]) une gestion distribuée des reconfigurations : il y a autant de gestionnaires de reconfiguration que de machines. Chaque gestionnaire possède sa propre vue de l'état de l'architecture qui est représentée par un graphe de composants et de leurs interconnexions. Cette vue est modifiée par le gestionnaire de composants dès lors qu'une liaison est faite (défaite) ou qu'un composant est ajouté (supprimé). Lorsqu'une reconfiguration locale doit être réalisée (e.g. via l'ajout d'un nouveau composant), des actions d'adaptation atomiques sont introduites. Une action atomique réussit ou échoue. Dans le premier cas, la modification est réalisée localement et est propagée de façon *fiable* à tous les autres gestionnaires ; dans le second cas, aucune adaptation n'est réalisée. La diffusion fiable est réalisée à l'aide d'une diffusion totalement ordonnée afin de garantir qu'un gestionnaire n'exécute une modification uniquement sur une vue du système qui n'a pas été invalidée auparavant.

Même si cette dernière approche apporte une solution quant à la répartition du contrôle des adaptations, il n'est pas prévu que le réseau puisse être partitionné.

3.2.4 Synthèse

En centrant les décisions d'adaptation sur l'architecture des composants, les travaux présentés dans cette section garantissent certaines propriétés architecturales souhaitées au moment de la conception de l'application. Que ce soit les approches programmées ou celles définissant des adaptations autonomes, les solutions qui ont été décrites préservent ces propriétés durant toute l'exécution de l'application.

Les travaux qui s'appuient sur la définition de stratégies ou de plan de réparation ont été historiquement les premiers à tenter l'expérience de l'adaptation dynamique des applications à base de composants. Ainsi, l'approche par composant a pu montrer son intérêt pour les adaptations dynamiques et la plupart des modèles, du moins ceux les plus répandus actuellement, permettent ce genre d'adaptation. Les adaptations programmées ont pu faire leur preuve dans l'automatisation des adaptations, amenant les applications à être non seulement adaptables mais aussi adaptatives.

Cependant, la définition de plans de réparation devient rapidement absconse à mesure qu'augmente la taille de l'application. Si en plus, l'infrastructure de déploiement est un réseau dynamique, la complexité des stratégies d'adaptation à mettre en place exige une expertise certaine.

Nous nous sommes intéressés également à un ensemble de solutions mettant en jeu des composants autonomes dont les comportements ou les configurations peuvent être facilement définis : l'architecture de l'application devient un but à atteindre ; les adaptations sont générées automatiquement par l'application elle même. Ces solutions répondent aux besoins d'autonomie que nous avons détaillés dans le premier chapitre de cette thèse. Cependant, les algorithmes mis en jeu restent difficiles à écrire et sont dépendants du domaine applicatif.

Enfin, nous pouvons remarquer que la grande majorité des travaux présentés dans cette section, s'appuie sur une centralisation des décisions d'adaptation. Cette approche n'est pas viable dans un réseau dynamique où l'hypothèse d'une machine stable (responsable des décisions d'adaptation) ne peut être garantie.

3.3 Adaptation centrée sur le composant

Dans cette section, nous nous intéressons aux techniques d'adaptation qui ne font pas forcément intervenir la notion de style architectural et par conséquent où la notion de cohérence architecturale est délaissée. Les mécanismes mis en jeu considèrent alors le composant comme lieu privilégié pour la définition et le déclenchement des adaptations alors que dans la section précédente, un gestionnaire d'architecture décidait *à la place* des composants.

Nous étudions ces travaux dans la mesure où ils font intervenir un modèle de composants et qu'ils considèrent des environnements dynamiques. Nous présenterons ces travaux en mettant en avant le modèle de composants et les mécanismes intergiciels utilisés pour bâtir notre réflexion sur les propriétés et particularités des modèles de composants pour les réseaux dynamiques.

3.3.1 Le projet Gravity

Le projet Gravity [CH04] définit un modèle de composant *orienté services* dans le but de faciliter la construction et l'exécution de composants capables de s'adapter de façon autonome face à des situations de déconnexion, *i.e.* des situations où un service utilisé par un composant devient indisponible.

Composants L'approche orientée service partage l'objectif de la programmation par composant, à savoir l'idée qu'une application est assemblée à partir de briques réutilisables, mais dans le cas de l'orientation service, ces blocs sont dénommés service. Un service correspond à une fonctionnalité offerte. L'approche orientée service se focalise d'avantage sur l'aspect découverte de services à l'exécution.

Dans Gravity, un composant est défini par le fait qu'il réalise un contrat. Un contrat spécifie les caractéristiques d'un service et surtout ses dépendances envers d'autres services. Il n'est donc pas nécessaire d'avoir un descripteur d'architecture dans lequel est décrit l'assemblage des composants. La résolution des dépendances se fait automatiquement à l'exécution via un registre de services.

Intergiciel Le support d'exécution mis en œuvre dans Gravity permet de prendre en compte le caractère dynamique de la plate-forme d'exécution. Ainsi les points suivants sont abordés :

- l'arrivée d'un composant ;
- le départ d'un composant existant ;
- l'arrivée d'un nouveau service ;
- le retrait d'un service existant ;

La non disponibilité d'un service requis par un composant est réalisée à deux niveaux dans Gravity : au sein des instances des composants et par le support d'exécution.

La gestion des instances de composants est réalisée par des gestionnaires d'instances qui sont responsables de la création des liaisons entre les instances qu'ils gèrent et les autres instances fournissant le(s) service(s) requis ; de la gestion du cycle de vie des instances de composants et de la publication des services. Le gestionnaire d'instances a également pour rôle de contrôler la *validité* des instances : un composant est dans l'état *valide* lorsque toutes les dépendances de ce dernier sont résolues et donc que le composant est capable de fournir

ses services. Lorsqu'un composant est dans l'incapacité de rendre un de ses services, il sera mis dans l'état *invalide*. La prise en compte de la dynamicité de l'environnement est donc faite par ces gestionnaires d'instances qui sont capables de reconfigurer les liaisons en fonction de l'apparition ou la disparition d'un service (dans le registre de services).

L'autre niveau de réaction vis-à-vis de la dynamicité des services est dans la définition d'un gestionnaire de composition. Face à la disponibilité de deux composants fournissant le même service, le gestionnaire d'instance n'est pas programmé pour choisir un service plus qu'un autre. Ce rôle est laissé au gestionnaire de composition qui s'appuie sur des contrats dans lesquels des stratégies à adopter face à la création ou la mise en place de liaisons entre composants peuvent être définies.

3.3.2 MADA/Domint

Composants MADA (Mobile Application Development Approach) [Kou05] est une méthodologie de conception d'applications pouvant fonctionner en présence de déconnexions. L'architecte de l'application peut définir pour chaque composant de l'application trois métadonnées : la déconnectabilité, la nécessité et la priorité. La déconnectabilité indique si un composant de l'application peut avoir un mandataire sur le terminal mobile. La nécessité indique si la présence du mandataire, appelé composant ou service déconnecté dans le cache du terminal mobile est absolument nécessaire pour le fonctionnement de l'application en présence des déconnexions. La priorité permet d'ordonner les composants et les services de l'application dans le cache du terminal mobile. La déconnectabilité est uniquement attribuée par l'architecte de l'application tandis que la necessité et la priorité des services fixées initialement par l'architecte peuvent être redéfinis par l'utilisateur. Enfin, des règles pour le déploiement et le remplacement définissent la propagation de la nécessité dans le graphe des dépendances entre services et composants.

Intergiciel La plate-forme supportant MADA s'appelle DOMINT (*Disconnected Operation for Mobile INternetworking Terminals*), basé sur l'intergiciel OpenCCM. L'architecture de DOMINT est découpée en trois couches : composant, conteneur et intergicielle (cf. figure 3.1). La couche composant constitue la partie métier de l'application.

La couche intergicielle offre trois services : le gestionnaire du cache, le gestionnaire de réconciliation, et les détecteurs de connectivité, de déconnexions et de défaillances. Plus particulièrement pour notre étude, nous nous sommes intéressés au gestionnaire du cache. Ce dernier centralise la gestion de tous les composants déconnectés dans le cache du terminal mobile. Il fournit la mise en œuvre des stratégies de déploiement et de remplacement du cache basées sur le profil de l'application et le graphe de dépendances de MADA. La stratégie de déploiement se décline en trois étapes complémentaires : au déploiement initial (préchargement), à la demande (explicite) de l'utilisateur et à l'invocation (implicite). La stratégie de remplacement se décline elle aussi en trois étapes : à la demande de l'utilisateur, en cas de manque de place, et de façon périodique pour préserver une zone libre dite « critique » afin d'accélérer de possibles déploiements urgents.

Dans DOMINT, le service de gestion des déconnexions est contrôlé par le conteneur. Il coordonne la commutation transparente entre le composant serveur et le composant déconnecté suivant le mode de fonctionnement. Il est aussi responsable de l'orchestration des différents

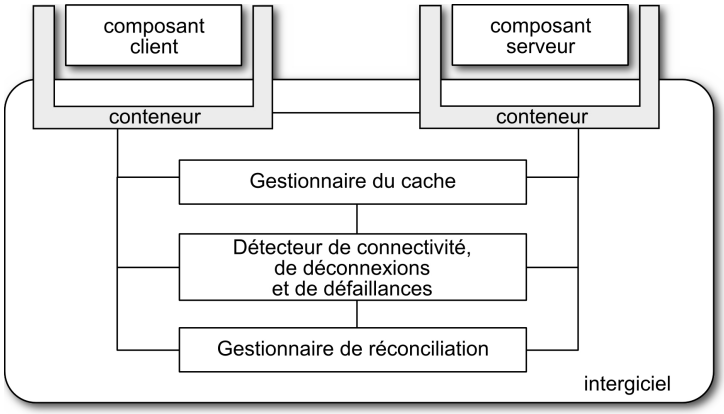


FIG. 3.1: Architecture de DoMINT (figure extraite de [Kou05])

services de la couche intergiciel de DOMINT dans le but de séparer les aspects fonctionnels de l'application de l'aspect gestion des déconnexions.

3.3.3 PCOM

PCOM [BHSR04] est un modèle de composants qui s'appuie sur l'intergiciel BASE [BHSR04, BS03] et qui permet la spécification d'applications distribuées à base de composants. PCOM se veut adapté aux environnements pervasifs en prenant en compte l'hétérogénéité et la dynamicité des équipements en définissant des scénarios pour de tels environnements et en proposant une implantation peu gourmande en mémoire. L'objectif de PCOM est de proposer une architecture à base de composants où les composants s'adaptent eux-mêmes aux différents changements qui surviennent dans l'environnement.

Les composants Nous retrouvons dans PCOM les abstractions présentes dans les modèles de composants : un composant est une unité de composition et de déploiement avec des interfaces bien définies et des dépendances envers les autres composants rendues explicites. PCOM introduit également des dépendances de contexte qui correspondent aux exigences des composants vis-à-vis des services non-fonctionnels que doit offrir la plate-forme hôte. La définition des interfaces ainsi que des dépendances est réalisée à l'aide de *contrats* dont un exemple est donnée dans le listing 3.2. Le contrat comprend, d'une part, la définition du composant en termes d'interfaces offertes, et d'autre part, la spécification de ses dépendances. Même si elles sont explicitées, les dépendances entre composants ne font que préciser les composants ou les types de composants requis par un composant particulier : la notion de *liaison* n'est présente qu'à l'exécution. C'est en effet lors de l'instanciation du composant qu'une référence valide vers un composant existant sera résolue lors d'une phase qualifiée de négociation.

```
<contrat>
  <offer>
  <interface type="pcom.ex.InputService" />
  <interface type="pcom.ex.BufferedInputService" />
  <event type="pcom.ex.NewLineEvent" />
  </offer>
```

```
    <requirement>
      <component Name="Input">
        <interface type=""/>
      </component>
      <platform>
        <profile>CLCD</profile>
        <memory>2048</memory>
      </platform>
    </requirement>

    <implementation />
  </contrat>
```

Listing 3.2: Contrat d'un composant PCOM. La première partie du contrat correspond aux fonctionnalités (interfaces) offertes par le composant. La deuxième partie précise les dépendances du composants vis-à-vis de la plate-forme censée l'héberger et des autres composants qu'il requiert pour offrir ses propres services

Afin qu'un composant puisse être instancié, son contrat doit être résolu, ce qui requiert que les composants dont il dépend soient également instanciés et que ses dépendances de contextes soient vérifiées. PCOM maintient à jour un *arbre de dépendances*[13] des composants instanciés : la racine de l'arbre correspond à l'application et les autre nœuds représentent les autres composants instanciés.

PCOM ne spécifie cependant pas comment est maintenue la cohérence globale de l'architecture, notamment dans la gestion de l'arbre de dépendance. De plus, dès lors qu'un composant devenu indisponible ne peut être remplacé, toute l'application est arrêtée (de proche en proche) alors que les autres services fournis par les autres composants pourraient encore être rendus.

Intergiciel Le support d'exécution des composants PCOM prend en compte la disponibilité (ou l'indisponibilité) des autres équipements (et donc les composants qui y résident) et les variations des ressources requises par les composants. PCOM s'appuie sur un service de découverte à la fois pour localiser les autres composants nécessaires à l'instanciation d'un composant mais aussi pour détecter l'inaccessibilité d'un composant (e.g. dans le cas où une machine n'est plus à portée radio des autres équipements). Ce service de découverte, basé sur le modèle publication / souscription, permet de remplacer un composant manquant par un autre dont l'existence a été détectée. De même, un composant peut être notifié dès lors qu'une ressource requise — spécifiée dans le contrat — change de valeur. La substitution est possible grâce à la comparaison des contrats des composants. Si un des composants requis n'est pas disponible, le composant peut décider de s'arrêter et informe alors les autres composants qui l'utilisent, propageant ainsi l'arrêt éventuel de chaque composant dans l'arbre de dépendances. Lorsqu'un composant n'est plus accessible, PCOM peut dynamiquement changer le protocole de communication utilisé jusqu'alors, en utilisant une autre interface réseau. De plus si un composant ne dépend pas d'un plug-in de transport particulier, il est possible

[13] PCOM interdit qu'un composant soit utilisé deux fois. Ainsi l'arbre de dépendance est vraiment un arbre et non pas un graphe.

de changer de protocole de transport. Ceci est mis en œuvre par un registre qui maintient la liste des plug-ins disponibles pour communiquer vers chaque équipement.

Que ce soit dans la phase de sélection des composants (i.e. la négociation des contrats), dans les réactions aux changements qui peuvent survenir au sein de l'environnement, PCOM permet au programmeur de définir ses propres stratégies.

3.3.4 RUNES

L'objectif du projet RUNES[14] est de fournir un support intergiciel et des outils de développement pour la construction d'applications distribuées prévues pour des environnements hétérogènes, large échelle et mettant en jeu des systèmes embarqués. L'intergiciel proposé est construit à l'aide d'un ensemble de composants dont nous décrivons ici le modèle.

Les composants Le modèle de composant RUNES [CCM+06, CCM+05] définit un composant comme une unité fonctionnelle en séparant implantation et fonctionnalités. Du fait de l'hétérogénéité des différents équipements qui exigent des implantations particulières des composants, RUNES modélise l'environnement d'exécution à travers une API. Cette API est rendue disponible aux composants par l'intermédiaire des *capsules* (i.e. des conteneurs), qui constituent l'espace d'exécution des composants. Des méta-données peuvent être associées aux composants afin de décrire les dépendances entre composants mais aussi les caractéristiques que doivent posséder les plates-formes cibles. RUNES utilise des attributs — des couples clé/valeur — pour décrire ces informations.

Les fonctionnalités des composants et leurs dépendances vis-à-vis des autres composants sont respectivement modélisées par les interfaces et les *réceptacles*. Les liaisons représentent les associations entre une interface et un réceptacle.

Par ailleurs, RUNES utilise le concept de *component framework* [Szy02] (désigné par CF dans la suite). Un CF représente un assemblage de composants spécifiant un ensemble de fonctionnalités qui peut être étendu ou modifié dynamiquement par l'ajout de composants. Un CF est un composant, ce qui permet de définir des structures hiérarchiques de CFs. Les composants constituant un CF et leur assemblage peuvent être contrôlés à l'aide de contraintes (e.g. définies à l'aide d'un langage tel que OCL [OMG05]).

Nous retrouvons dans le modèle la notion de liaison. La liaison entre une interface et un réceptacle est représentée par un composant particulier appelé *connecteurs*, dont le rôle est d'abstraire les appels d'un réceptacle vers l'interface associée. Les connecteurs peuvent jouer le rôle d'intercepteur et ajouter aux appels de méthodes du pré ou post-traitement. Un connecteur, en tant que composant, possède ses propres réceptacles pour réaliser (ou étendre) son rôle de « composant de communication ».

Intergiciel Le modèle de composants RUNES est un modèle de composants non distribué et utilise des primitives de mobilité pour fournir des services distribués. Le modèle ne s'appuie pas sur des invocations de services distants à travers le réseau mais sur le clonage et la migration de composants entre machines afin de garantir une certaine autonomie lorsque la connectivité est insuffisante ou non fiable. Ainsi une application modélisée par un ensemble de composants RUNES est représentée par un ensemble de composants instanciés localement. La

[14] Projet IST n° IST-004536-RUNES, `http://www.ist-runes.org/`

particularité du modèle réside principalement dans la manière d'offrir des services distribués aux composants locaux, en permettant aux instances d'envoyer et de recevoir des composants dynamiquement.

L'approche adoptée dans RUNES pour la prise en compte des changements dans l'environnement est de fournir un ensemble d'extensions réflexifs qui fournissent chacun un MOP (*Meta-Object Protocol*) et permettent aux différentes entités correspondantes d'être « inspectées » et manipulées. Chaque extension est réalisée par un composant ou un CF.

- L'extension réflexive *architecture* permet de réifier la topologie de l'ensemble des composants présents dans une capsule et de modifier la structure de cet l'assemblage en ajoutant, remplaçant ou supprimant des composants ou CFs. Les opérations utilisées pour ces adaptations sont celles fournies par l'API de la capsule.
- L'extension réflexive *interface* permet de connaître les fonctionnalités et dépendances des différents composants et d'invoquer les différents services offerts par un composant. Ainsi, de nouveaux services peuvent être découverts dynamiquement.
- Les connecteurs, comme évoqués précédemment, peuvent être le lieu d'adaptation diverses en interceptant les appels entre composants. Par exemple dans le cadre d'une application multimédia, un connecteur, en fonction de la bande passante disponible, peut décider quand remplacer un codec.

3.3.5 P2PComp

Le modèle de composants P2PComp[FHMO04] vise à faciliter le développement d'applications pair-à-pair fonctionnant sur des équipements à faibles ressources (PDA, téléphone mobile, *wearable devices*), dotés de moyens de communication. Les réseaux pris en considération sont qualifiés d'ad-hoc au sens où ils reposent sur aucune infrastructure particulière.

Les composants P2PComp met en jeu également des *conteneurs*[15] qui prennent en charge le cycle de vie des composants, à savoir : leur installation, activation, arrêt et retrait. La notion de *port* a été introduite pour les conteneurs et les composants. Le concept de port a été introduit comme un mécanisme de communication de haut niveau entre les composants et permet d'expliciter les dépendances entre ces derniers. Différents protocoles de communication peuvent être réalisés par un port. Comme pour les modèles de composants cités précédemment, des ports (interfaces) fournis et requis sont distingués :

- les ports *fournis* (*provides port*) qui correspondent aux services offerts par un composant et sont définis à travers des interfaces Java. À chaque interface proposée, correspond une entrée dans un descripteur de déploiement
- les ports d'utilisation (*uses ports*) correspondent aux services requis par un composant et auxquels le support d'exécution peut *connecter* les ports fournis d'autres composants.

Pour le conteneur, les ports *d'accès* représentent des points de connexions aux frontières des conteneurs et autorisent les communication entre ces derniers.

Intergiciel Au niveau de l'intergiciel, l'introduction du concept de ports a pour objectif de rendre transparente l'invocation d'un service requis, réalisé aussi bien par un composant situé

[15] La notion de conteneur est ici présente du fait que P2PComp est une extension de OSGi qui intègre cette terminologie.

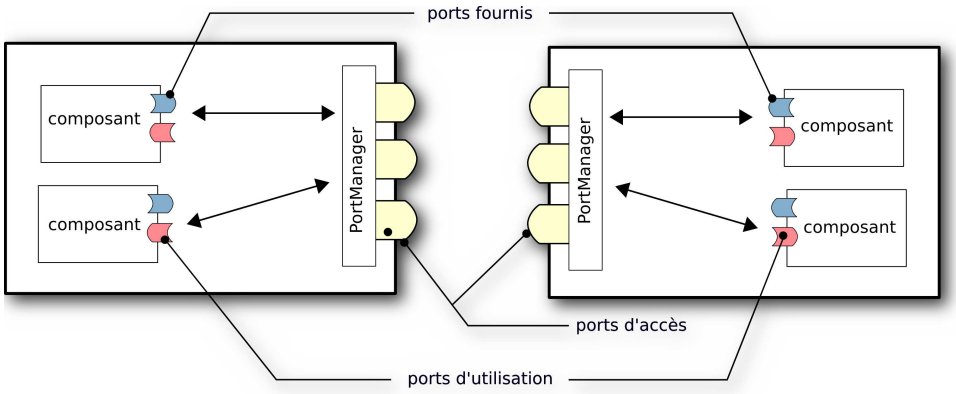


FIG. 3.2: Composant et conteneurs dans P2PComp (figure extraite de [FHMO04])

dans le même conteneur que par un composant hébergé dans un conteneur distant. Pour ce faire, les composants passent par l'intermédiaire du composant PortsManager, présent dans chaque conteneur, qui, lorsqu'un service requis localement n'est pas présent, interagit avec un de ses pairs (dans les autres conteneurs) afin de réaliser les invocations. La réalisation de la communication distante est assurée par le composant P2PService qui prend également en charge la réception des appels entrant et la détection de la disponibilité (ou non) des autres conteneurs.

La prise en compte de la dynamicité et la mobilité des équipements est essentiellement réalisée par le PortManager. Ce dernier peut découvrir les services fournis par les composants s'exécutant dans des conteneurs distants (et locaux) à l'aide de requêtes sur la description du service. Lorsqu'un composant n'est plus disponible (et donc les services qu'il rend non plus), le PortManager essaye de le substituer par un autre composant en recherchant d'autres services compatibles.

3.3.6 Synthèse

Toutes les travaux présentés dans cette section permettent l'adaptation de l'architecture de composants sans pour autant mettre l'architecture de l'application au centre des décisions d'adaptation. Ainsi, chaque composant, en fonction de ses besoins ou du contexte changeant, va pouvoir réaliser des opérations de création ou de modification de liaisons, voire même provoquer l'instanciation de composants requis.

Cependant, la notion d'architecture n'est pas totalement absente. Les éléments architecturaux, comme les interfaces ou liaisons, sont utilisés afin de détecter certains changements comme la non disponibilité d'un service requis. Toutefois, lorsque l'architecture des composants est prise en compte, comme dans PCOM [BHSR04] avec l'arbre de dépendances, le problème des déconnexions réseaux n'est pas très bien traité. En effet, la non disponibilité d'un seul composant peut rendre l'application toute entière inutilisable. Les seuls mécanismes automatisés consistent à sélectionner un nouveau composant fournissant les mêmes fonctionnalités lorsqu'un tel composant est accessible.

Le modèle de distribution des composants s'appuie sur un simple schéma client / serveur (synchrone ou asynchrone) qui implique la localisation unique d'un composant : pour les

autres composants (et l'utilisateur), un service ne se trouve qu'à un unique emplacement. Les solutions présentées ne définissent pas de fonctionnalités d'un composant comme étant ubiquitaires.

3.4 Services intergiciels pour l'adaptation

Les deux sections précédentes se sont essentiellement focalisées sur les approches où les notions d'architecture de composants ou de composants seulement étaient utilisées dans la définition de mécanismes d'adaptation exigés par les réseaux dynamiques.

Pour être complet, nous décrivons dans cette section les travaux traitant essentiellement de l'approche intergicielle. Bien qu'ils ne définissent aucun modèle de composants, ces travaux, principalement menés pour répondre aux problèmes des réseaux dynamiques, complètent notre étude sur les mécanismes intergiciels dédiés à l'adaptation des applications.

3.4.1 One.World

Les travaux présentés dans [GDL+04, Gri02] décrivent *one.world*, un intergiciel pour la construction d'applications pervasives. La définition de cet intergiciel a été guidée par trois principes :

- les changements se produisant dans l'environnement doivent être explicités afin de permettre aux applications de définir leurs propres stratégies en réaction à ces changements ;
- l'environnement d'exécution doit faciliter les compositions dynamiques de services puisque dans un environnement pervasif, les besoins des utilisateurs varient en fonction des services dont ils disposent ;
- enfin, les fonctionnalités doivent être séparées des données manipulées afin de permettre une meilleure gestion et évolution à la fois des services et des données.

L'approche adoptée dans *one.world* est de laisser aux concepteurs et programmeurs d'applications ubiquitaires la définition et la programmation des actions à réaliser lorsqu'un changement intervient au sein de l'environnement. Pour ce faire, *one.world* fournit un cadre générique basé sur la programmation événementielle aussi bien pour la l'assemblage des différentes parties constituant l'application (les composants) que pour la notification des changements aux entités concernées.

Afin de faciliter la programmation des applications, *one.world* propose un ensemble de services dédiés tels que : le passage de messages asynchrones (mode de communication par défaut des composants), la découverte de services, la migration, etc.

Sur le plan de la communication entre composants distants, *one.world* prône l'explicitation de la distribution afin de mettre en évidence la nature distribuée des applications ubiquitaires et la dynamicité de leur environnement d'exécution. Bien que cette approche permette aux applications de prendre en compte et de s'adapter aux conditions changeantes, il est laissé au programmeur la charge de définir ces adaptations : *one.world* n'émet pas d'événement particulier lorsqu'un « contexte » ou la localisation d'un composant est modifiée.

3.4.2 Gaia

Le projet Gaia [RC00, MRC02] vise à faciliter la construction d'applications ubiquitaires dans des environnements de type « environnements intelligents » (*smart spaces* en anglais). L'environnement d'exécution des applications est désigné comme un espace actif (*active space*) qui représente un espace physique coordonné par un logiciel sensible au contexte et qui permet à l'utilisateur d'interagir et de configurer son environnement (physique et applicatif) de façon homogène.

Gaia se présente avec un ensemble de services de base et une infrastructure facilitant la construction d'applications sensibles au contexte. Parmi ces services, nous pouvons citer :

- le *service de contexte* qui permet aux applications de récupérer des informations sur le contexte. Ces informations sont modélisées par une logique du premier ordre et une algèbre booléenne, ce qui permet de construire des `types` de contexte et des prédicats comme ceux définis dans le listing 3.3.
- le *service de présence* qui permet de détecter l'arrivée et le départ d'entités au sein d'un *smart space*. Les entités prises en compte sont : les applications, les services, les équipements et les personnes ;
- un *annuaire* qui contient une description des équipements et des composants présents dans un *smart space*.

```
Context(temperature, room3231, is, 98F)
Context(Application, Powerpoint, is, Running)
Context(Number of people, Room 2401, >, 4)
```

Listing 3.3: Prédicats décrivant un *smart space* dans Gaia

Les différentes notifications concernant des changements au sein de l'environnement se font par un gestionnaire d'événements. La réalisation des communications entre composants distants se fait via l'*Unified Object Bus* ([RC01]), un intergiciel caractérisé par sa capacité à faire interagir des composants provenant de différents modèles.

3.4.3 Aura

Le projet Aura [GSSS02], développé à l'université de Carnegie Mellon met l'accent sur la mobilité des utilisateurs. L'environnement considéré dans Aura entre dans notre définition de réseau dynamique et est similaire à la notion d'*active space* introduit dans Gaia.

Aura fournit également un ensemble de composants prenant en charge la mise à disposition des services et l'observation de l'environnement d'exécution ainsi que des informations de contexte. Ces composants, identifiés comme des tâches, ont pour objectif de maximiser l'utilisation des ressources disponibles et de minimiser la distraction de l'utilisateur. Les principaux composants d'Aura sont :

- *Coda*. Ce composant fournit un système de fichier permettant des opérations en mode déconnecté. La mobilité de l'utilisateur ainsi que les variations de la bande passante sont pris en compte pour la définition des stratégies d'accès et d'écriture à des données partagées ;
- *Odyssey*. Ce composant a pour rôle l'observation des ressources et la mise en place des stratégies d'adaptation ;
- *Spectra*. Des mécanismes évolué d'invocation à distance sont fournis pas ce composant ;

– *Prism*. Ce composant est responsable de l'obersvation des actions de l'utilisateur et sert à anticiper ses actions.

3.4.4 Easy living

Le projet Easy Living de Microsoft a pour objectif de définir une architecture pour les environnements intelligents. Ces environnements sont constitués d'un grand nombre d'équipements coopérant pour offrir à l'utilisateur l'accès à des services depuis toutes les machines présentes autour de lui.

L'intergiciel proposé dans Easy Leaving s'appuie sur des services de géolocalisation, d'annuaire et de communication asynchrone. Les auteurs de [BMK+00] suggèrent que pour l'intergiciel responsable de la coordination des différents équipements, l'approche centralisée soit délaissée au profit d'un système pair-à-pair.

3.4.5 Synthèse

Les travaux présentés dans cette section se veulent adaptés aux réseaux dynamiques et d'une manière plus générale ont pour objectif de proposer des infrastructures pour l'informatique ambiante. De ce fait, les applications considérées sont ubiquitaires : il est possible pour l'utilisateur de disposer d'une application indépendamment de l'équipement utilisé. L'accent est mis sur l'interaction naturelle de l'utilisateur avec les différents équipements. Pour ce faire, la définition de services comme la découverte de ressources, les services de présence, la gestion du contexte (i.e. les informations décrivant l'environnement d'exécution des applications), la composition automatique de services, se retrouvent dans les approches proposées.

3.5 Discussion

Pour compléter l'étude sur les technologies de composants qui a été faite dans le chapitre précédent, nous nous sommes focalisés dans ce chapitre sur les technologies à composants dédiés aux réseaux dynamiques. Si nous souhaitons déployer des applications à base de composants sur de tels réseaux, il devient nécessaire de définir des mécanismes servant à adapter les applications aux fluctuations de leur environnement. Après avoir défini les concepts régissant l'adaptation des architectures de composants, nous avons montré que ces adaptations ne pouvaient se faire en ignorant complètement la topologie de l'application.

De ce chapitre, nous pouvons tirer quelques leçons sur les modèles de composants et leur support d'exécution lorsque les réseaux visés sont dynamiques. En effet, les différents modèles de composants présentés se veulent adaptés à ces réseaux et leur étude a permis de constater que les entités communes à la majorité des modèles de composants, à savoir les notions d'interfaces et de liaisons, se retrouvent également dans un contexte de dynamicité. Cela montre que les réflexions actuelles sur la modélisation des composants et de leurs propriétés ont atteint une certaine maturité et que, notamment, les principes de modularité et de réutilisabilité sont maintenant bien cernés par l'approche par composants.

La présentation de ces différents travaux a permis de mettre en avant certains concepts et mécanismes sur lesquelles se basent les technologies de composants. Ces technologies s'appuient sur (1) une description plus ou moins de haut niveau de la plate-forme cible ; (2) des

mécanismes d'observation et de réaction aux différents changements ; (3) des mécanismes de communication entre composants supportant la dynamicité du réseau, en particulier, l'apparition et la non disponibilité des composants. Nous justifions ci-après ces trois aspects :

1. la description des besoins de l'application et de ses différents composants envers les ressources que doit posséder l'environnement d'exécution est un prérequis. Il est donc nécessaire de préciser les dépendances des composants envers la plate-forme censée les héberger ;
2. l'environnement d'exécution, à l'inverse, doit pouvoir détecter que les ressources requises par chaque composant sont effectivement disponibles. Cela implique la capacité de la plate-forme à observer et détecter les différents changements qui peuvent influer sur le bon fonctionnement de l'application (e.g. débranchement d'un périphérique, insuffisance de bande passante). La plate-forme cible ou les composants eux-mêmes doivent pouvoir déclencher des adaptations dès lors que leur environnement n'offre plus les ressources dont ils ont besoin. Dans les réseaux où la mobilité et la volatilité des équipements entrent en jeu, cela peut également être dû à la non accessibilité d'une ressource, et en particulier d'un composant ;
3. l'indisponibilité d'un composant dans le cas d'une connexion sans fil défaillante, et donc des services qu'il fournit, n'est pas un phénomène particulier dans les réseaux dynamiques. Les modèles de composants doivent donc supporter les déconnexions. Les mécanismes liés à la distribution des composants doivent également prendre en compte la connectivité du réseau.

La grande majorité des travaux mettant en jeu des composants distribués délèguent à l'intergiciel la gestion des distributions des composants même dans le cas où la plate-forme visée est un réseau dynamique. Cependant, en face des déconnexions, les opérations d'adaptation, lorsqu'elles sont centrées sur l'architecture, ne concernent que les éléments de l'application définis lors de sa conception. Ainsi les entités prenant en charge la distribution ne sont pas concernées par les stratégies ou plan de réparation. À l'inverse, les solutions centrées sur le composant, en ajoutant des entités spécifiques pour le traitement des déconnexions (e.g. les composants déconnectés dans DOMINT), rendent difficile une gestion autonome des reconfigurations qui serait basée sur une simple description de l'architecture.

Un des problèmes qui apparaît donc, étant donné l'état actuel des concepts proposés par les modèles de composants et les mécanismes définis au sein de leur support d'exécution, est que la gestion de la dynamicité des plates-formes cibles ne peut être totalement transparente à l'application, ni totalement prise en charge par cette dernière. Nous pouvons enfin noter que, dans les travaux sur l'informatique mobile — paradigme dans lequel l'utilisateur à l'aide d'équipements portables accède à des données et services indépendamment de sa localisation — l'étude réalisée dans [JHE99] liste les caractéristiques du modèle client-serveur dans des environnements mobiles et présente les nouveaux concepts permettant la réalisation du modèle client-serveur dans un contexte de mobilité. Nous pouvons souligner que dans cette étude, les deux approches, opposées, les adaptations « transparentes » à l'application et celles de type « laisser-faire » (i.e. c'est l'application qui est responsable de la gestion des déconnexions) ne sont pas appropriées dans un contexte de mobilité.

4

Déploiement de composants logiciels

Nous nous sommes attachés dans les chapitres précédents à présenter (1) les modèles de composants logiciels et les mécanismes mis en place au niveau des intergiciels à composants réalisant la distribution des composants ; (2) les différents mécanismes d'adaptation autorisant les applications à base de composants à être utilisées dans des réseaux dynamiques.

Dans ces deux chapitres, nous nous sommes focalisés sur les entités et opérations qui sont mises en jeu lors de l'exécution des composants (e.g. les composants mandataires, les adaptations d'une configuration de composants), en mettant de côté les questions relatives au déploiement des composants, i.e. aux différentes étapes ayant conduit à l'instanciation des composants sur les différentes machines du réseau.

Dans ce chapitre, nous nous intéressons au déploiement d'applications à base de composants. Nous présentons dans un premier temps la problématique générale du déploiement des logiciels. Puis la section 4.2 expose le déploiement des composants distribués à travers un état de l'art du déploiement dans les technologies les plus abouties (i.e. CCM, EJB et Fractal). Enfin, nous présentons quelques travaux qui s'attachent à rendre le déploiement plus autonome.

Sommaire

4.1 Déploiement de composants dans des environnements dynamiques

4.1.1 Problématique générale du déploiement de logiciel

Le déploiement logiciel est une activité complexe, qui couvre toutes les étapes depuis la validation du logiciel par le producteur jusqu'à son installation puis sa désinstallation sur les sites utilisateurs. Des travaux récents portant sur la problématique générale du déploiement d'applications à base de composants ont contribué à l'identification de phases élémentaires sur lesquelles repose ce déploiement. Il a ainsi été proposé de distinguer des phases relatives à la mise à disposition, l'acheminement, l'installation, la configuration, l'exécution, l'adaptation et la suppression des composants constitutifs d'une application ([Les03, Hal99]). Toutes ces étapes représentent le cycle de vie du déploiement logiciel représenté sur la figure 4.1. Une flèche entre deux étapes représente l'antériorité d'une étape par rapport à celle située à l'extrémité.

La description de l'application consiste à exprimer, dans un formalisme convenu, l'architecture de l'application : le ou les documents de description de l'application décrivent les parties (composants ou non) qui constituent l'application (version, localisation, etc.), leur assemblage, leur composition hiérarchique ou non, les interconnexions entre les constituants de l'application, la configuration de leurs paramètres, leurs dépendances vis-à-vis de fichiers de données, de bibliothèques, ainsi que leurs dépendances par rapport à d'autres composants et leurs versions. Parallèlement à la description de l'application, les programmes qui la constituent peuvent être compilés en des fichiers exécutables, si besoin. Ensuite, l'application peut être packagée pour être distribuée.

La phase de « publication » (*release* et *advertise* en anglais) de l'application consiste à la rendre disponible pour les utilisateurs, par exemple sur un site web ou par notification dans une liste de diffusion, et éventuellement à en assurer la maintenance. La phase de « retrait » de l'application consiste à la rendre obsolète, à ne plus la diffuser ni en assurer le support.

L'installation de l'application comprend le transfert des fichiers nécessaires à son exécution (données en entrée, exécutables, dépendances, etc.) ainsi que la configuration de son environnement (variables d'environnement, permissions sur les fichiers, etc.). L'opération contraire est la désinstallation qui doit éliminer les fichiers liés à l'application, et éventuellement ses fichiers de configuration. La mise à jour d'une application comprend les mêmes opérations que l'installation (transfert et configuration).

Le lancement de l'application comprend l'exécution des processus de l'application et l'activation de ses constituants (l'activation peut être précédée d'une étape de configuration dynamique de l'application et de connexion des composants le cas échéant). La terminaison de l'application consiste à arrêter ses processus pour libérer les ressources du système d'exploitation. L'adaptation dynamique de l'application consiste à modifier son comportement alors qu'elle est en cours d'exécution.

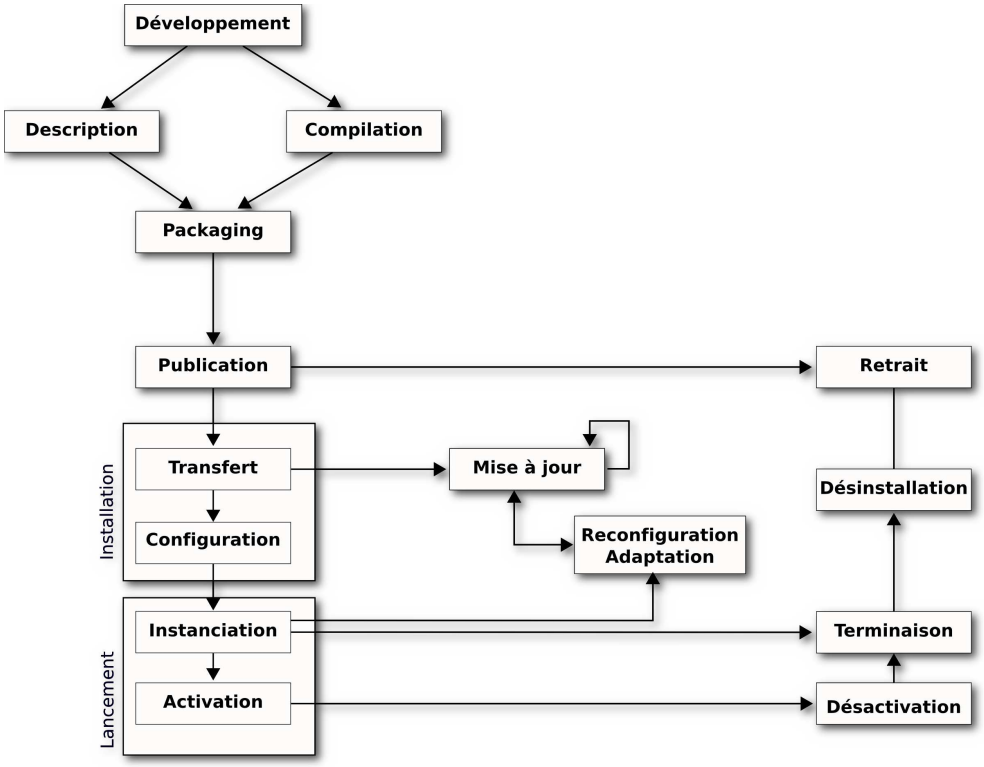


FIG. 4.1: Cycle vie apres développement

4.1.2 Déploiement de composants distribués

Une application distribuée met en jeu des composants répartis sur les machines du réseau. Ainsi avant la phase de lancement de l'application (ou du moins de ses composants), se pose le problème du choix des machines devant héberger les composants. Cette décision de placement est nécessaire en vue de l'instanciation de chaque composant. Cette étape peut aussi bien être réalisée en même temps que celles de description ou de packaging ou encore d'installation ou bien de (re)configuration. Dans la plupart des technologies à composants, la spécification d'une machine devant héberger un composant est faite avant la phase de packaging en indiquant dans un descripteur de déploiement son adresse IP ou son nom. Ceci suppose une connaissance assez précise de la plate-forme de déploiement : la désignation d'une machine ici signifie que les ressources nécessaires à l'exécution du composant seront disponibles sur cette machine.

Le processus de sélection des machines hôtes peut également se faire au moment de l'installation de l'application (i.e. avant ou en même temps que la phase de transfert). Dans ce cas, les ressources dont disposent les différentes machines sont découvertes pendant le déploiement et un processus, automatique ou non, est mis en place pour l'attribution d'une machine pour chaque composant.

La décision de placement d'un composant peut être remis en question après le lancement de l'application. Par exemple lorsque les ressources exigées par un composant deviennent insuffisantes sur la machine l'hébergeant, plusieurs stratégies peuvent être envisagées (e.g. remplacement du composant par un composant léger le temps de l'indisponibilité des res-

sources, arrêt du composant, mécanismes pour libérer des ressources sur la machine d'accueil etc.) et notamment le choix d'une nouvelle machine pour l'hébergement du composant.

4.1.3 Déploiement de composants distribués dans des réseaux dynamiques

La gestion du cycle de vie du déploiement est rendue complexe notamment du fait des interactions entre les différentes activités du déploiement. Traiter par exemple de la phase d'instanciation suppose au préalable la disponibilité du code et des différents fichiers nécessaire à la création des composants. Les travaux sur les workflows de déploiement d'applications à base de composants ([FM04]) permettent d'automatiser le calcul des dépendances entre certaines tâches du déploiement.

Lorsque l'on considère les réseaux dynamiques, cette complexité est d'autant plus accentuée du fait qu'il est difficile de contrôler les ressources nécessaires au déploiement. En effet, dans les réseaux dynamiques, aucune hypothèse ne peut être faite sur l'accessibilité d'une machine particulière et la disponibilité des ressources. Ainsi le problème du choix des machines devant héberger les composants ne peut se faire facilement, les machines connectées et les ressources disponibles ne pouvant être prédites. On ne peut plus indiquer pendant les étapes de description et de packaging le nom des machines hôtes sous peine de faire échouer le déploiement dès lors qu'une de ces machines n'est jamais connectée.

Pendant les phases de spécification, toutes sortes de contraintes peuvent être attachées aussi bien sur l'architecture de composants que sur les machines hôtes, par exemple, des contraintes architecturales sur le nombre d'instances maximales d'un composant ou encore des contraintes indiquant les ressources minimales que doivent posséder les machines censées héberger ces composants. Il souhaitable que le processus de déploiement calcule une configuration de placement des composants qui soit optimale vis-à-vis de ces contraintes.

Autant le problème d'optimisation du placement des composants se pose simplement dans les réseaux stables ([MRM04a, BTAB05]), sa formalisation n'est pas acquise dans les réseaux dynamiques. La présence des îlots dans ces réseaux rend difficile l'orchestration du déploiement, c'est-à-dire le contrôle sur l'exécution des différentes tâches. L'arrivée et le départ d'une machine pendant l'exécution du déploiement peut perturber les décisions prises par certaines activités du déploiement. Par exemple, une machine initialement connectée qui a été choisie pour héberger un composant peut ne plus être accessible.

Nos travaux autour du déploiement couvrent uniquement les problématiques liées aux décisions de placement des composants et de leur activation. Cette thèse ne traite pas de l'installation des composants et en particulier de la question du transfert des composants sur les machines du réseau qui est pourtant difficile dans les réseaux dynamiques ([RG05, Fré04]).

4.2 Déploiement de composants logiciels distribués

Nous présentons dans cette section les travaux sur le déploiement de composants logiciels distribués à travers les technologies les plus répandus que sont CCM [Obj02] et EJB [MH99]. Nous décrivons également les propositions autour du déploiement pour le modèle de composants Fractal. Nous nous focalisons dans la présentation de ces travaux aux aspects liés à la spécification du déploiement et aux supports intergiciels responsables de l'instanciation des composants.

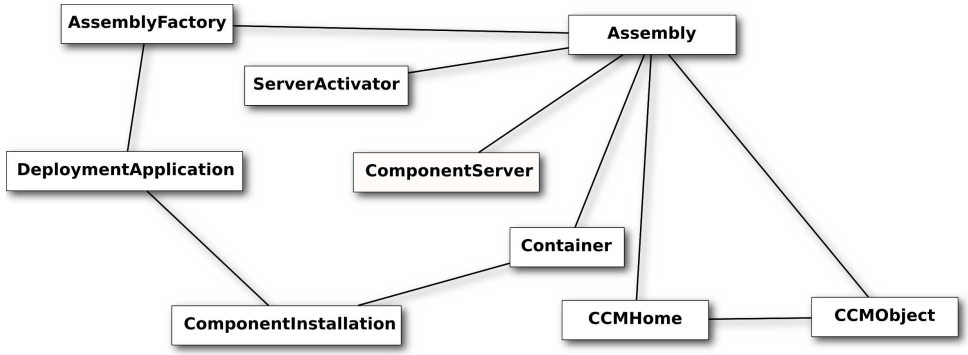


FIG. 4.2: Architecture de déploiement dans CCM

4.2.1 Déploiement de composants Corba

Avec la spécification CORBA 3.0, l'OMG (Object Management Group) ajoute à CORBA la notion de composants accompagnée d'un modèle de *packaging* et de déploiement.

4.2.1.1 Spécification du déploiement

Le modèle de packaging et de déploiement spécifié par CORBA 3.0 [omg03] définit un *descripteur de composant* via un fichier XML et associé au composant lui-même. Un tel descripteur contient la description du composant, i.e. son type en termes de ports d'entrée et de sortie, la catégorie à laquelle il appartient (processus, entité, session, service). Ces descripteurs sont générés lors de la compilation de la description des composants (faite à l'aide de CIDL), mais peuvent être modifiés pour paramétrer la gestion des aspects non-fonctionnels. Les composants sont ensuites empaquetés avec leurs descripteurs dans une archive (fichier) appelé *paquetage de composant*. Un paquetage contient une ou plusieurs implantation d'un composant. L'implantation d'un composant peut être une bibliothèque partagée, un fichier contenant du bytecode java, etc.

Les paquetages de composants peuvent ensuite être utilisés soit directement par un outil de déploiement soit pour définir conjointement avec d'autres paquetages de composants et un descripteur d'assemblage, un *paquetage d'assemblage* de composants. Un *descripteur d'assemblage* de composants permet de décrire l'architecture initiale d'une application. Il spécifie les instances de composants constituant l'application, définit des règles de placement sur des sites d'exécution, et indique les connexions à établir entre les différentes instances. Les règles de placement, appelées contraintes de colocalisation précisent si des composants doivent être instanciés sur la même machine ou au sein d'un même processus.

Le modèle de déploiement de CCM suppose l'existence d'un outil de déploiement mettant en jeu au minimum les entités ComponentInstallation, AssemblyFactory et Assembly. Les rôles de ces entités, représentées sur la figure 4.2 sont les suivants :

– l'interface ComponentInstallation modélise un gestionnaire d'archives de composants et fournit les opérations pour installer[16], retrouver et désinstaller les archives et implantations de composants sur les nœuds de l'infrastructure.

[16]Le terme « installation » dans la spécification de CCM représente l'action de récupérer le(s) paquetage(s) de l'application à déployer pour le placer dans un dépot contrôlé par l'outil de déploiement.

– l'interface AssemblyFactory modélise le gestionnaire d'instances d'assemblages et fournit les opérations pour créer des assemblages à partir de leurs archives, retrouver et détruire les instances d'assemblage.
– l'interface Assembly représente une instance d'assemblage déployée sur l'infrastructure. Cette interface permet de contrôler l'instanciation (ou déploiement) de l'assemblage ainsi que sa destruction (ou repliement).

Ainsi, en s'appuyant sur ces trois entités, nous pouvons résumer les étapes du déploiement telles que spécifiées dans CCM :

1. Dans un premier temps, l'attribution d'une machine à chaque composant de l'assemblage est déterminé. Cette étape se fait à l'aide de l'intervention de l'utilisateur et résulte en la création d'une copie du descripteur d'assemblage dans laquelle l'information sur la localisation des composants (tenant compte des contraintes de colocalisation) est renseignée. Cette copie sera ensuite utilisée par l'objet Assembly en vue de l'instanciation de l'assemblage.
2. Ensuite, l'ensemble des fichiers nécessaires à l'instanciation des composants est installé sur les machines où les composants seront hébergés.
3. L'instanciation des composants et des maisons de composants est enfin réalisée à partir des informations contenues dans le descripteur d'assemblage qui précise quels composants et quelles maisons de composants doivent être créés, quels sont leurs machines hôtes et quels sont les composants qui doivent être colocalisés. Finalement, la mise en place des connexions entre composants est réalisée.

4.2.1.2 L'infrastructure DCI

OpenCCM, une implémentation libre (*open source*) du modèle de composant Corba, s'appuie sur une infrastructure de déploiement baptisée DCI (*Distributed Computing Infrastructure*) qui a été réalisée dans le cadre du projet européen IST COACH ([HRR+03]).

La particularité de la plate-forme DCI, qui a pour vocation de déployer des composants Corba, est qu'elle est elle-même mise en œuvre par des composants Corba. L'infrastructure DCI est donc une application répartie, i.e. constituée d'un ensemble de composants distribués sur les machines participant au déploiement. Ces différents composants doivent interagir afin de garantir le bon déroulement du déploiement. Parmi les différents composants entrant dans la mise en œuvre de DCI, le composant DCIManager est le gestionnaire central de l'infrastructure DCI. Étant donné un domaine de déploiement, i.e. l'ensemble des machines et des interconnexions et des passerelles, le DCIManager se comporte comme un portail d'accès à l'ensemble des fonctions du domaine de déploiement. Ces fonctions sont :

– le maintien de la cohérence globale du domaine : le DCIManager assure l'utilisation des mêmes instances du service d'annuaire par toutes les machines d'accueil et tous les modèles d'assemblage installés ;
– l'accès aux méta-informations sur les caractéristiques matérielles et logicielles des nœuds connectés au domaine de déploiement ;
– le déploiement des assemblages.

En plus du DCIManager, une machine de déploiement répartie permet l'automatisation du déploiement (instanciation des composants d'un assemblage et la mise en place des liaisons entre composants).

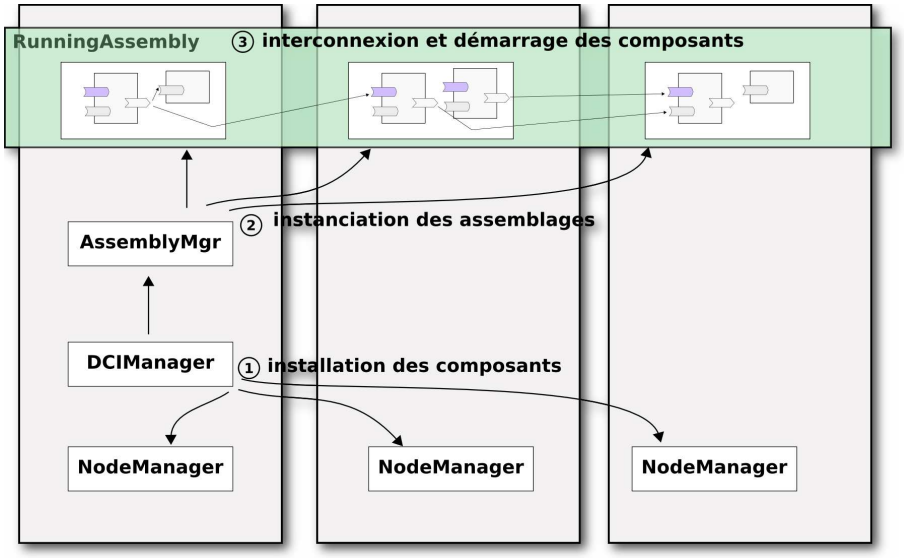


FIG. 4.3: L'infrastructure de déploiement DCI

4.2.1.3 Conclusion

Même si la spécification d'un modèle de déploiement de composants CORBA identifie les entités participant à la mis en œuvre d'une infrastructure de déploiement, les implantations actuelles souffrent de l'aspect centralisé du gestionnaire de déploiement. Dans le cadre de l'OPENCCM DCI, le DCIManager est un composant qu'il est nécessaire de distribuer dès lors que l'environnement de déploiement devient conséquent ou non fiable.

Par ailleurs, lors du déploiement de composants Corba, l'intervention de l'utilisateur est exigée afin de renseigner le serveur sur lequel chaque instance de composant devra être lancée.

4.2.2 Déploiement de composants EJB

4.2.2.1 Spécification du déploiement

Très récemment, depuis J2EE 1.4, le déploiement de composants EJB a été standardisé et est défini dans la spécification JSR88 [jsr]. Cette spécification différencie clairement les services que doivent fournir un intergiciel J2EE de ceux devant être réalisés par les outils servant à déployer des applications sur cet intergiciel. Se présentant sous la forme d'une API, la spécification définit une interface qui apparaît être au cœur du déploiement de composants EJB : le DeploymentManager. L'entité réalisant cette interface est responsable de l'installation et de l'instancition des unités de déploiement définies par une archive Java (e.g. les archives EJB, WAR, EAR et RAR). En ce qui concerne les composants EJB, ils sont archivés à l'aide d'un descripteur de déploiement. Ce descripteur se présente sous la forme d'un fichier XML qui indique aux serveurs d'applications comment déployer les beans contenus dans l'archive en définissant leurs caractéristiques (e.g. les noms des composants, les types de transaction, les méthodes d'accès sécurisées, etc.).

Le listing 4.1 illustre le descripteur de déploiement pour un composant EJB. Le type de bean (session, sans état), le nom de des interfaces métiers ainsi que le nom de la classe im-

plantant le bean sont renseignés.

```
<?xml version='1.0' encoding='UTF-8'?>
<ejb-jar
    version="2.1"
    xmlns="http://java.sun.com/xml/ns/j2ee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
                        http://java.sun.com/xml/ns/j2ee/ejb-jar_2_1.xsd"
    >
  <display-name>SimpleSessionJar</display-name>
  <enterprise-beans>
    <session>
      <ejb-name>SimpleSessionEjb</ejb-name>
      <home>beans.SimpleSessionHome</home>
      <remote>beans.SimpleSession</remote>
      <ejb-class>beans.SimpleSessionBean</ejb-class>
      <session-type>Stateless</session-type>
      <transaction-type>Bean</transaction-type>
      <security-identity>
        <use-caller-identity>
        </use-caller-identity>
      </security-identity>
    </session>
  </enterprise-beans>
</ejb-jar>
```

Listing 4.1: Exemple d'un descripteur de déploiement d'un composant EJB

4.2.2.2 L'infrastructure de déploiement J2EE

Le rôle central donné au DeployementManager par la spécification a pour but d'offrir un point d'entrée à n'importe quel produit certifié J2EE. La classe implantant Le DeploymentManager doit offrir les opérations permettant de configurer, distribuer, lancer, arrêter et replier une application J2EE. Le DeployementManager n'est pas utilisé pour la définition de l'assemblage de l'application, de la spécification des propriétés des composants et de la mise en paquetage des composants.

La définition des serveurs cibles pour les composants beans constituant l'application se fait manuellement en indiquant dans le descripteur de déploiement le nom des serveurs d'applications hôtes. Les descripteurs de déploiement sont mis à jour par le DeployementManager en y ajoutant les informations propres au serveur d'application.

4.2.2.3 Conclusion

L'objectif de la spécification JSR88 est de fournir une API permettant à différents vendeurs d'outils de configurer et de déployer des applications J2EE sur n'importe quel serveur d'application J2EE.

Nous pouvons regretter que cette spécification ne définisse pas comment un module, i.e. l'ensemble des classes et descripteur de déploiement servant à déployer un (ou plu-

sieurs) composants, peut être distribué sur des serveurs différents. En effet, comme le souligne [Exe04], même si le JSR88 indique que la cible de déploiement peut être soit un serveur soit un ensemble de serveurs, l'hypothèse qui est faite est que le module est dupliqué sur les différentes cibles.

Par ailleurs le modèle de déploiement J2EE n'offre pas le moyen d'exprimer les dépendances entre les modules.

4.2.3 Déploiement de composants Fractal

Le projet Fractal[17] définit le modèle de composants éponyme et propose actuellement plusieurs implantations de modèles (e.g. Julia, Koch, ProActive). Cependant, ces différents supports souffrent du manque d'outils proposant le déploiement de composants Fractal. Cela est dû au fait qu'aucune spécification n'a encore été validée à l'heure actuelle[18] en ce qui concernent le paquetage des composants. Cependant plusieurs projets intègrent en partie le déploiement de composants Fractal. Nous en détaillons quelques uns ci-après.

4.2.3.1 FractalADL

Le modèle de composants Fractal est accompagné d'un langage de description d'architecture permettant de spécifier une architecture de composants hiérarchiques. Ce langage est mis en œuvre par FractalADL qui est constitué d'un ensemble de modules, chacun représentant un « aspect » du langage de description d'architecture (e.g. le versionnement, l'architecture, etc.). FractalADL est extensible et permet donc l'ajout de nouvelles fonctionnalités comme cela sera détaillé dans la deuxième partie de ce mémoire (section 9.2.3 du chapitre 9).

Nous considérons dans la suite de ce mémoire que les descriptions d'une architecture à base de composants Fractal sont spécifiées via le langage XML même si FractalADL n'impose pas une syntaxe particulière.

FractalADL est définit par plusieurs composants Fractal qui peuvent être assemblés pour créer différents outils. Par exemple, étant donné un descripteur d'architecture de composants Fractal, il est possible d'instancier localement cette configuration en utilisant FractalADL qui définit une fabrique de composants. Cette dernière utilise un ensemble d'outils (des composants Fractal) pour analyser syntaxiquement une définition (composant loader), générer l'ensemble des tâches (composant compiler) nécessaires à la création des composants (réalisée par le composant backend).

```
Map context = new HashMap();
context.put("classloader",/* an home made class loader */); // optional
Factory f = FactoryFactory.getFactory(FactoryFactory.FRACTAL_BACKEND);
Component c = (Component)f.newComponent("myADLDefinition", context);
```

Listing 4.2: Instanciation d'une architecture de composants Fractal à partir de sa description XML

[17] http://fractal.objectweb.org/

[18] Une première version d'une spécification pour le packaging et déploiement de composants Fractal a fait l'objet de discussions sur la mailing-list en juin 2004.

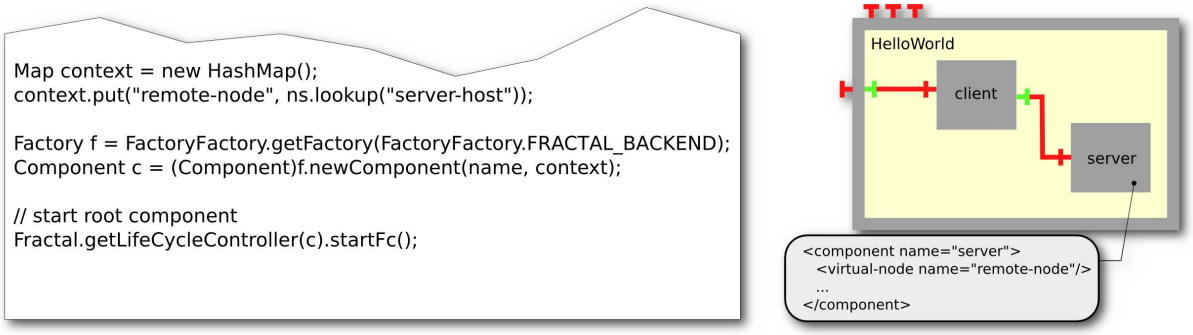


FIG. 4.4: Distribution de l'architecture client / serveur via FractalRMI

À partir d'une définition XML d'une architecture de composants, l'instanciation de cette configuration se programme directement en utilisant l'API Fractal comme illustré dans le listing 4.2.

Il est également possible de déployer une architecture de composants Fractal à l'aide d'un outil en ligne de commande.

4.2.3.2 FractalRMI

L'outil FRACTALADL ne permet pas de déployer (instancier) une architecture distribuée de composants, c'est-à-dire un ensemble de composants devant s'exécuter sur plusieurs machines (virtuelles) interconnectées. FRACTALRMI apporte une réponse à la distribution des composants Fractal en fournissant un ensemble de composants capables de réaliser des liaisons entre composants distants.

L'exemple de la figure 4.4 montre comment utiliser FRACTALRMI pour déployer une architecture client/serveur où les deux entités client et serveur résident sur deux machines distinctes. Il suffit de préciser dans un premier temps, au niveau de la description de l'architecture, la *localisation* des composants via la balise `<virtual-node>` ajoutée à FRACTALADL. Au moment de l'instanciation de l'architecture, réalisée comme dans le cas non distribué via FRACTALADL, il suffit de préciser le composant Fractal (le *bootstrap*) qui doit être utilisé pour instancier le composant distant. Ici le bootstrap qui instancie le composant `server` réside sur la machine `server-host`.

La génération des talons et squelettes est réalisée à l'exécution. Il n'est donc pas nécessaire de compiler les différentes entités prenant en charge la distribution des composants.

Nous reviendrons plus en détail sur ces aspects dans la deuxième partie de cette thèse où nous détaillons les extensions que nous avons apportées aux mécanismes de distribution proposés par Fractal.

4.2.3.3 FROGi

FROGi [DCD06] (pour Fractal over OSGi) est une proposition visant à faire profiter les composants Fractal des mécanismes de conditionnement et de déploiements des *bundles* — les unités de déploiement — définis dans OSGi [osg].

Spécification du déploiement Afin de spécifier le déploiement et le packaging de composants Fractal, FROGi étend le langage de description d'architecture FractalADL. Cette extension a pour but de définir une ou plusieurs unités de déploiement (bundles) pour chaque composant Fractal. Un bundle possède des informations comme le numéro de version ou des propriétés d'enregistrement des services dans des annuaires. La spécification des liaisons entre composants permet de décrire des instances à lier qui se trouvent soit au sein d'un même bundle, soit dans des bundles séparés ou encore des liaisons entre un composant et un service patrimonial OSGi. La spécification d'un service requis par un composant peut être guidé par une requête LDAP, permettant ainsi au support d'exécution de lier les composants en fonctions de propriétés contextuelles.

Infrastructure de déploiement Le support d'exécution fourni par FROGi définit quatre étapes quant à la gestion de l'activation des composants :

- dans un premier temps, FROGi se charge d'instancier le composant racine contenu dans le bundle à déployer. Les différentes interfaces (fonctionnelles et de contrôle de ce composant) sont publiées dans le registre de services d'OSGi. Ceci permet à d'autres bundles (en particulier celui d'administration) de contrôler l'instance du composant (e.g. gestion de son cycle de vie) ;
- lors de l'installation des bundles, le framework OSGi résout de façon automatique les dépendances explicitées par les interfaces requises et fournies. Ainsi les bundles contenant les composants requis sont également installés.
- ensuite, une fois les différents bundles déployés, grâce au mécanisme de courtage définit dans OSGi, les liaisons entre les instances de composants sont réalisées. L'annuaire contenant la description des services fournis permet de sélectionner un service par ses propriétés (e.g. `language=fr` ou `cron.pattern=***3***`) ;
- Une fois les dépendances résolues, le bundle peut être activé. La gestion du cycle de vie d'un composant racine peut être faite soit par son composite (livré dans un autre bundle) ou soit par le bundle lui-même.

4.2.3.4 Déploiement de JonasALaCarte

Les travaux de T. Abdellatif [Abd06] présentent la réingénierie du serveur d'application J2EE JOnAs en un serveur d'application configurable. Ce serveur, appelé JonasALaCarte est bâti à l'aide de composants Fractal. La configuration et le déploiement du serveur se basent sur FractalADL et permet ainsi d'obtenir des configurations à la carte selon l'environnement ciblé et d'automatiser le déploiement des composants dans cet environnement.

Comme dans le cas de FROGi, le déploiement proposé s'appuie sur OSGi pour l'implantation des paquetages sous forme de bundles. Le système de déploiement proposé, supposé être installé et lancé sur chaque machine participant au déploiement repose sur le composant DeploymentEngine qui étend l'usine de composants FractalADL pour la prise en compte de l'adresse des machines cibles, des identificateurs des paquetages, etc. Le DeploymentEngine orchestre le déploiement en indiquant à chaque machine cible le ou les composants à déployer. Mise à part les mécanismes de rapatriement du code des composants, l'instanciation des composants est réalisée grâce à FractalADL.

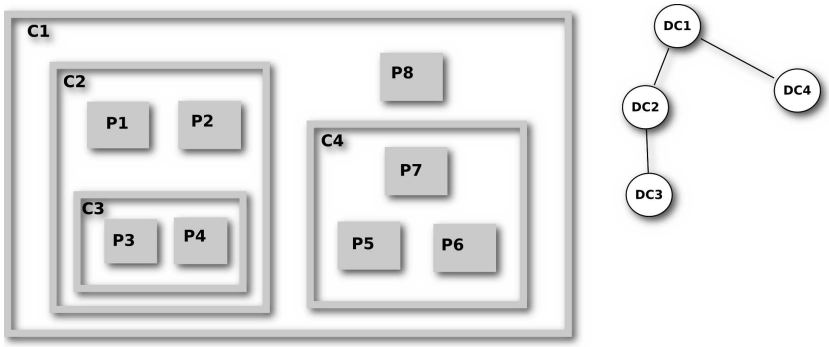


FIG. 4.5: Hiérarchie de contrôleurs

4.2.3.5 Déploiement asynchrone de composants Fractal

Les auteurs de [QBB+04] s'intéressent à la phase préalable à l'activation des composants d'une application. Cette phase consiste à installer le code des composants, l'instancier, effectuer les liaisons entre les différents composants, et enfin les activer. Contrairement aux approches décrites précédemment, l'approche adoptée ici est de considérer le processus de déploiement comme une application distribuée à part entière. Cette application s'appuie sur la hiérarchie de l'application pour répondre aux besoins de passage à l'échelle en répartissant au maximum les différentes tâches du déploiement.

Spécification du déploiement À l'aide d'une extension du langage de description d'architecture OLAN[19], il est possible de spécifier les composants, leur type (les interfaces fonctionnelles et non-fonctionnelles, primitif ou composite) ainsi que leurs liaisons. Par ailleurs, il est possible d'indiquer pour chaque composant sa localisation en indiquant son site de déploiement. Enfin, les ordres d'activation entre composants peuvent être précisés. Un composant n'ayant aucune dépendance vis-à-vis d'un autre composant est activable dès lors que toutes ses interfaces requises sont liées, même si le composant fournissant le service n'est pas encore activé.

Infrastructure de déploiement L'infrastructure de déploiement a été réalisée à l'aide de l'intergiciel ScalAgent qui offre au gestionnaire de déploiement un bus à message global qui a pour rôle de garantir la fiabilité des échanges de messages ainsi que leur ordonnancement.

Le gestionnaire de déploiement est constitué d'un ensemble de contrôleurs dont l'architecture est calquée sur celle de l'application à déployer. Sur la figure 4.5 est représentée à gauche la hiérarchie de l'application en termes de composants composites et primitifs (les liaisons ne sont pas représentées). À droite, nous voyons qu'un contrôleur est associé à chaque composant composite. L'arbre ainsi obtenu est appelé « hiérarchie applicative » par les auteurs. Un contrôleur est *initié* par son contrôleur parent.

Chaque contrôleur est responsable du déploiement du composant composite auquel il est associé et donc de tous ses sous-composants. Au sein d'un contrôleur s'exécute en parallèle ou en séquence un ensemble d'activités primitives. Les activités de création ont pour rôle

[19]Les travaux de [QBB+04] sont antérieur à à l'implantation de FractalADL et sont d'ailleurs à l'origine du modèle de composant Fractal.

d'instancier les sous-composants primitifs. Dans le cas d'un sous-composant composite, un contrôleur dédié sera créé et aura en charge le déploiement de la sous-hiérarchie. Les activités de liaisons prennent en charge la mise en place des connecteurs entre composants venant d'être instanciés et pour lesquels le descripteur d'architecture spécifiait une liaison. Enfin les activités d'activation propagent le démarrage de tous les composants à travers la hiérarchie des contrôleurs.

Chaque contrôleur possède un *référentiel* qui sert d'une part à stocker et à mettre à jour les résultats des activités (dans le cas d'une activité de création, la référence du nouveau composant y est ajouté) et d'autre part permet l'interrogation (par abonnement ou consultation directe) par d'autres activités du référentiel. Ce découplage entre activités permet une activation au plus tôt des différents composants.

4.2.3.6 Fractal Deployment Framework

Le cadre de conception FDF[20] (*Fractal Deployment Framework*) a pour but de faciliter le déploiement des applications distribuées de manière générale sans être limité aux applications Fractal. L'architecture de déploiement proposé par FDF est réalisée à l'aide de composants Fractal.

En proposant un langage dédié de description d'architecture, FDF permet aux utilisateurs de décrire à l'aide de ce langage les configurations à déployer (i.e. la liste des logiciels à déployer et des machines hôtes) au lieu de programmer ou *scripter* la manière dont le déploiement sera réalisé. Le langage définit les primitives nécessaires au déploiement : il est ainsi possible de préciser les protocoles de transfert (e.g. FTP ou SCP) ou encore les protocoles d'accès distant (e.g. telnet, ssh) pour décrire le réseau cible du déploiement. En ce qui concerne l'application à déployer, FDF définit des *personnalités* qui correspondent chacune à la description des logiciels, des bibliothèques, des paramètres nécessaires au déploiement de l'application. Ces descriptions sont également réalisées à l'aide du langage servant à décrire la plate-forme cible.

FDF est mise en œuvre par un ensemble de composants Fractal. Ces composants réifient d'une part les logiciels à déployer mais aussi toutes les entités de la plate-forme de déploiement, des machines aux protocoles de transfert.

4.2.3.7 Conclusion

Nous pouvons remarquer que les solutions proposées pour le déploiement de composants Fractal se distinguent nettement les unes des autres. La raison essentielle étant qu'il n'existe toujours pas à notre connaissance de spécification du déploiement de ces composants (seule un premier document de travail a été proposé sur la liste de diffusion du projet Fractal), même en ce qui concerne l'empaquetage des composants et des (méta) informations qui pourraient s'y trouver.

Les solutions que nous avons présentées, à l'exception de la proposition de [QBB+04], reposent toutes sur un processus de déploiement centralisé.

[20] `http://fdf.gforge.inria.fr/`

4.3 Vers un déploiement autonome de composants logiciels

Les sections précédentes ont permis de mettre en avant la spécification et les infrastructures de déploiement définies dans les modèles de composants CCM, EJB et Fractal. Les solutions proposées sont conçues pour des environnements très peu sujets aux pannes ou aux variations de ressources. Par ailleurs, la spécification du déploiement détermine les machines devant héberger les composants de façon définitive, généralement en précisant leur adresse IP, alors que le placement initial, conduit par des besoins en ressources (logicielles ou matérielles), peut être remis en question du fait de l'indisponibilité de certaines ressources. Ainsi, les reconfigurations qui s'imposent sont traitées de manière ad-hoc ou demandent l'intervention directement d'un administrateur voire même de l'utilisateur.

La discussion faite dans le chapitre précédent a évoqué la nécessité dans les technologies de composants d'avoir une description des besoins en ressources des composants du fait de l'incapacité avant la phase de déploiement d'avoir une vision globale des ressources disponibles dans des réseaux dynamiques. Les besoins des composants étant spécifiés, il est à la charge de la plate-forme d'exécution de déterminer la machine répondant (au mieux) aux exigences de chaque composant composant.

Nous présentons maintenant les travaux qui prennent en compte l'expression des besoins en ressources des composants en vue du déploiement de ces derniers. Dans ces travaux, le placement des composants est décidé à l'exécution en fonction des contraintes exprimées dans un descripteur de déploiement avec comme objectif principal, l'automatisation du déploiement.

4.3.1 Déploiement automatique de composants sur les grilles

Dans sa thèse, S. Lacour [Lac05] pointe la difficulté de déployer des applications dans des environnements de type grille. L'intervention manuelle de l'utilisateur dans la sélection des ressources et des implémentations de l'application pour chaque ressource sélectionnée, le transfert des fichiers à distance et la configuration de l'application imposent par ailleurs une certaine expertise de la part des utilisateurs. La proposition qui est faite est d'automatiser le déploiement d'applications et ceci de façon transparente.

Afin d'automatiser le déploiement des composants, l'outil de déploiement a besoin de la description de l'architecture à déployer ainsi que celle des ressources disponibles sur la grille, obtenue via un intergiciel d'accès aux ressources. L'utilisateur peut également indiquer des « paramètres de contrôle » précisant les contraintes imposées sur l'application comme le degré de parallélisme, le sous-réseau auquel les machines à sélectionner appartiennent, etc.

Intervient ensuite un planificateur, programme qui a pour rôle la sélection des ressources, la sélection des protocoles de lancement de tâches, le placement des différents composants de l'application et la sélection de l'implémentation adéquate pour chaque composant. Le planificateur de déploiement produit en sortie un « plan de déploiement » qui spécifie tous les choix qui ont été faits lors de la phase de planification. Une fois ce plan de déploiement exécuté et en cas de succès, l'application distribuée est entièrement déployée.

Nous voyons que l'automatisation du déploiement repose sur le planificateur. Ce dernier s'appuie sur un algorithme de planification. Nous pouvons mettre en avant l'algorithme *round-robin*, détaillé dans le listing 4.3, qui place les instances de programmes de l'application sur les ressources au fur et à mesure qu'elles sont découvertes, et en respectant les contraintes de compatibilité de système d'exploitation et d'architecture matérielle.

```
1.  Tant qu'il reste des instances de programme P à placer
2.   Découvrir la machine suivante disponible M (ou bien revenir à la
        première machine découverte)
3.   Pour chaque implémentation I du programme P
4.      Si le système d'exploitation et l'architecture matérielle de
           la machine M sont compatibles avec l'implémentation I,
5.         Alors ajouter au plan de déploiement l'association entre
              l'implémentation I et une méthode de soumission de
              la machine M, et continuer en 1.
6.         Sinon continuer en 3.
7.   Si aucune implémentation I ne peut être placée sur la machine M
        et que toutes les machines disponibles n'ont pas encore été
        envisagées pour le programme P
8.      Alors continuer en 2.
9.      Sinon il n'y a pas de solution : échec de la planification
10. Planification terminée avec succès
```

Listing 4.3: Algorithme round-robin de placement de composants (d'après [Lac05])

Un avantage de l'algorithme *round-robin* est qu'il n'a, en général, pas besoin de découvrir l'intégralité des ressources disponibles sur la grille, donc il passe bien à l'échelle.

Le redéploiement dans [Lac05] n'est pas pris en compte. L'algorithme de planification n'est exécuté qu'une seule fois déterminant le succès ou l'échec du placement des composants de l'application suivant les ressources disponibles de la grille au moment du déploiement.

4.3.2 Maximisation du placement

Dans [BTAB05], les auteurs proposent un algorithme pour le placement automatique de composants lors du déploiement. À chaque composant peuvent être associées des contraintes que la machine cible devra vérifier mais aussi des préférences (donc non obligatoires) sur les ressources offertes par un nœud. Une contrainte ou une préférence se caractérise par un intervalle de valeurs requises et un poids qui permet d'accorder plus d'importance à telle ou telle contrainte. Les liens réseaux entre composants peuvent également être caractérisés par des préférences ou contraintes mais cet aspect n'est pas pris en compte.

L'algorithme proposé décide pour chaque composant d'un nœud sur lequel il va être instancié, de telle sorte que l'ensemble des contraintes et préférences satisfaites soient maximales tout en minimisant les ressources consommées sur chaque nœud.

4.3.3 Redéploiement pour améliorer la disponibilité

Les travaux présentés dans le paragraphe précédent ne s'intéressent qu'au placement de composants à un moment donné, en fonction d'informations sur les ressources disponibles sur chaque machine et dont la disponibilité n'est pas remise en question une fois l'application déployée.

Les travaux de [MRM04b, MR04] formalisent le problème du placement des composants en vue d'améliorer la disponibilité de ces derniers lorsque se produisent des déconnexions en :

1. observant le système ;

2. calculant l'architecture du système à redéployer ;
3. en réalisant le redéploiement de l'architecture exhibée précédemment.

La mise en équation du problème de placement des composants inclut la quantité de mémoire disponible sur chaque machine et celle requise par chaque composant, la connectivité des nœuds, ainsi que la fréquence d'interaction entre composants. Sont formalisées également des contraintes de localisation et de colocalisation, i.e. les contraintes précisant les composants qui peuvent être hébergés sur la même machine et ceux devant résider sur des machines distinctes. Par exemple deux composants connus pour consommer intensivement du CPU devraient résider sur deux machines distinctes. Cependant les contraintes sur les ressources demandées par les composants, autre que celles portant sur la mémoire, ne sont pas prises en compte[21].

Les auteurs proposent un ensemble d'algorithmes ayant une complexité polynomiale qui améliore le placement des composants au regard d'une fonction à maximiser ou minimiser. La réalisation du redéploiement se fait en mettant en avant la différence entre l'architecture de composants actuellement déployée et celle calculée par l'algorithme de placement.

4.3.4 Programmation par contraintes

Les auteurs de [DKM04a, DKM04b] proposent la résolution du problème du placement de composants logiciels selon le paradigme de la programmation par contraintes [vHSD94]. Dans ce paradigme, au lieu d'écrire un algorithme visant la résolution d'un problème donné, on se limite à la génération d'un ensemble de besoins (les contraintes exprimant le problème à résoudre), leur résolution étant prise en charge automatiquement par les solveurs. Le cadre des problèmes de satisfaction de contraintes ou CSP (pour *Constraint Satisfaction Problems*) offre l'avantage :

- d'être extrêmement déclaratif : il suffit de déclarer les contraintes donc de modéliser le problème ;
- de proposer de nombreux algorithmes pour résoudre une grande variété de problèmes, de complexités variées.

Ainsi, les travaux de [DKM04a] présentent Deladas, un langage déclaratif pour la description des systèmes autonomes. Avec Deladas, il est possible de définir une architecture de composants (comme tout langage de description d'architecture) mais aussi des contraintes sur cette architecture. Par exemple, le listing 4.4 montre la spécification de deux contraintes avec Deladas. La première précise le nombre d'instances des composants Router et Client sur chaque machine. La deuxième indique que deux machines ne peuvent héberger chacune un composant Router uniquement si elles sont accessibles l'une de l'autre.

```
// une instance de composant Router ou Client par machine
forall host h in deployment {
  card(instanceof Router in h) = 1 or
  card(instanceof Client in h) = 1
}

forall Router r1,r2 in deployment {
```

[21] Les auteurs précisent que les contraintes relatives à la bande passante, à la charge du processeur, etc. peuvent être prises en considération via les contraintes de localisation

```
    reachable(r1,r2)
}
```

Listing 4.4: Exemple de contraintes avec Deladas

L'ensemble de ces contraintes définit l'objectif du déploiement. Toutes les contraintes sont ensuite utilisées par un résolveur de contraintes qui peut générer une voire plusieurs solutions. Chaque solution représente une configuration (interconnexions de composants et l'adresse d'une machine pour chacun d'entre eux) qui vérifie les contraintes exprimées via Deladas. Cette configuration est ensuite utilisée pour indiquer aux différentes machines les composants qu'elles sont « autorisées » à instancier. Selon les ressources disponibles, le résolveur de contraintes peut ne pas trouver de solution de placement des composants.

Sur chaque machine, les ressources nécessaires aux composants et l'état des composants sont observés de manière périodique afin de réagir aux variations pouvant rendre l'état de l'architecture déployée incohérent vis-à-vis des contraintes de déploiement (e.g. la panne d'un composant). Lorsqu'un tel changement intervient, l'entité responsable de l'orchestration du déploiement doit générer une nouvelle configuration. Un cycle « autonome » est ainsi mis en place : la détection de conflits par rapports aux contraintes définissant l'objectif du déploiement, le calcul d'une nouvelle configuration et le redéploiement des composants n'exigent aucune intervention de l'utilisateur.

4.4 Synthèse

Les travaux abordés dans ce chapitre ont permis de mettre en lumière les différents mécanismes et les techniques régissant le déploiement d'une architecture de composants. Ainsi, nous retrouvons au cœur de tout déploiement la description de l'architecture. Elle correspond à l'objectif à atteindre par un outil de déploiement et permet l'automatisation d'une partie des activités d'installation, à savoir l'instanciation et la mise en place des liaisons entre composants.

La section 4.2 de ce chapitre a présenté le déploiement dans les modèles de composants bien établis et pour lesquels des outils existent. Pour réaliser l'instanciation des composants, aucune information concernant le contexte d'exécution des composants n'est pris en compte : le placement des composants est indiqué par la présence de l'adresse de la machine devant les héberger. Dans les réseaux que nous visons, nous ne pouvons suivre cette approche car au moment du déploiement, nous ne pouvons garantir la présence d'une machine particulière.

Les solutions ensuite présentées dans la section 4.3 s'attachent à l'automatisation du placement des composants en prenant en compte les besoins de chaque composant et aussi des contraintes de colocalisation. Toutes ces propositions non conçues pour des environnements dynamiques considèrent que le déploiement a échoué dès lors qu'aucune solution de placement n'a été trouvée. Dans le cas des réseaux qui nous intéressent l'idée d'un déploiement qui est terminé (avec ou sans échec) n'a pas de sens dans la mesure où des ressources nécessaires et non présentes à un moment donné peuvent être disponibles plus tard. Par ailleurs, tous les travaux énumérés dans cette section centralisent l'intelligence de déploiement sur une machine dédiée, ce qui n'est pas viable dans le cadre des réseaux dynamiques.

Deuxième partie

Contribution

5

Modèle de composants hiérarchiques ubiquitaires pour les réseaux dynamiques

L'ingénierie des logiciels à base de composants a montré son intérêt dans la conception, la construction et la maintenance d'applications distribuées. Dans cette approche, une application est vue 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 sur la disponibilité des ressources requises par les composants. Les applications ainsi conçues ne peuvent pas dans la plupart des cas être installées ou exécutées sur des réseaux dynamiques, i.e des réseaux constitués de machines hétérogènes, mobiles et aux ressources parfois limitées et potentiellement volatiles.

Parmi les modèles de composants offerts par les technologies actuelles, les modèles hiérarchiques offrent un moyen naturel et expressif de concevoir des applications. Dans ces modèles, il est possible de créer des composants de haut niveau d'abstraction bâtis au-dessus de composants plus spécifiques. Cependant les difficultés posées par les réseaux dynamiques soulèvent le problème de la distribution des composants dans ce genre de réseaux : comment en effet autoriser des interactions entre composants dans un réseau partiellement connecté ?

Nous proposons dans ce chapitre un modèle de distribution permettant de rendre des composants logiciels ubiquitaires. Ces composants sont capables de supporter la dynamicité imposée par leur environnement d'exécution. Notre proposition s'appuie sur des concepts introduits dans le modèle de composants hiérarchiques Fractal.

La section 5.1 présente quelques modèles de composants hiérarchiques et se focalise sur le modèle de composants Fractal. Nous définissons dans la section 5.2 un modèle permettant de rendre des composants hiérarchiques ubiquitaires. Après avoir détaillé dans la section 5.3 les schémas de composition de composants ubiquitaires, nous présentons (section 5.4) le support

des déconnexions pour ces composants.

Sommaire

5.1 Modèles de composants hiérarchiques

Le modèle de composants sur lequel nous nous appuyons pour définir des composants ubiquitaires reprend les concepts spécifiés par le modèle de composants hiérarchiques Fractal [BCL+04] en y ajoutant un schéma de distribution des composants permettant de les rendre ubiquitaires. Nous justifions ci-après le choix d'un modèle de composants hiérarchiques dans le cadre de notre étude et en particulier celui de Fractal.

5.1.1 Intérêts des modèles de 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 modélisation d'un système complexe est facilitée par l'organisation hiérarchique des composants. Le concepteur de l'application peut mettre en place les différents composants en commençant par ceux de plus haut niveau et en raffinant par la suite chacun de ces composites par des éléments spécialisés. Par ailleurs, lorsqu'il s'agit de faire évoluer une application (e.g. par l'ajout de fonctionnalités, la reconfiguration d'une partie des composants, la redéfinition des liaisons, etc.), un modèle hiérarchique permet une meilleure compréhension de l'architecture.

5.1.2 Choix d'un modèle de composants hiérarchiques

Il existe un certain nombre de modèles de composants hiérarchiques. Par exemple, le langage de description d'architecture Darwin [MK96] fournit une notation simple pour l'expression de compositions hiérarchiques. Ce langage considère un élément de l'architecture (le composant) comme une entité pouvant être instanciée. Ceci a pour conséquence de spécifier de manière plus précise les interactions entre composants. En effet, il est par exemple utile de décrire qu'une instance de composant est connectée à plusieurs instances d'un autre composant pendant l'exécution ou de définir un paramètre dont la valeur sera déterminée pendant l'exécution et qui exprime la cardinalité des liens entre composants. Une architecture décrite à l'aide de Darwin est déployée et exécutée au sein de l'environnement Regis [MDK94]. Regis fournit un cadre écrit en C++ pour construire et exécuter des programmes distribués et spécifiés par Darwin. La possibilité de décrire des architectures dynamiques à l'aide de Darwin est d'un usage limité : il n'est pas possible de supprimer des composants dynamiquement ni de permettre à un composant primitif de communiquer avec des composants dynamiquement créés.

Le modèle de composant Koala [vOvdLKM00] a été proposé par Philips pour la programmation des pilotes de périphériques audio et vidéo des lecteurs CD/DVD ou téléviseurs haut de gamme. Les composants Koala sont décrits à l'aide d'un langage de description de composants spécifique appelé *Koala Language*. Ce langage permet de spécifier les interfaces requises et fournies d'un composant, mais aussi les sous-composants qu'il contient ainsi que les liaisons existantes entre ces sous-composants. Koala a pour avantage de clairement identifier les interactions entre un composant et son environnement car toutes les interactions sont réalisées au travers d'interfaces. L'environnement d'un composant peut aussi bien être logiciel (défini par une dépendance vers d'autres composants) que matériel (défini par une dépendance vers des services spécifiques du système d'exploitation sous-jacent). De plus, Koala permet de prendre en considération les aspects de consommation mémoire des composants. Cependant, il n'est pas possible d'étendre le modèle afin de considérer d'autres ressources.

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 [MT00] qui permettent de préciser les composants constituant une application ainsi que leurs interactions. Dans le cadre applicatif que nous nous sommes fixés, 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.

Parmi les modèles de composants existants, le modèle Fractal offre les abstractions nécessaires pour manipuler aussi bien les composants primitifs que les composants composites à l'exécution. Par ailleurs les supports d'exécution développés pour les composants Fractal offrent beaucoup de liberté quant à la définition de nouveaux services non-fonctionnels.

5.1.3 Le modèle de composants Fractal

Nos travaux autour du modèle de composants Fractal et les extensions que nous proposons au niveau du support d'exécution Julia — l'implantation de référence en Java du modèle Fractal — n'ont cependant pas pour objectif de se conformer aux spécifications Fractal. Nous

dressons toutefois ici un bref panorama des concepts caractérisant Fractal.

Le modèle de composants Fractal a pour objectif de simplifier le coût du développement, du déploiement et de la maintenance des systèmes complexes allant des systèmes d'exploitation aux intergiciels [CQSS07]. Le modèle est ainsi basé sur les principes suivants :

- *les composants composites*, i.e. des composants qui contiennent des sous-composants. Ceci permet d'avoir une vue uniforme des applications à différents niveaux d'abstraction.
- *les composants partagés*, i.e. des composants appartenant à plusieurs composites englobants. Il ainsi possible de modéliser les ressources et leur partage, tout en préservant l'encapsulation des composants.
- *les capacités d'introspection* qui permettent d'observer l'exécution d'un système.
- *les capacités de (re)configuration* pour permettre de déployer et de configurer dynamiquement un système.

Les capacités d'introspection et de (re)configuration dans Fractal sont rendues possibles du fait de la nature des composants. Un composant Fractal est une entité d'exécution possèdant une ou plusieurs interfaces.

Une interface est un point d'accès au composant. Elle implante un type d'interface qui spécifie les opérations supportées par l'interface. Un composant Fractal expose deux types d'interface : les interfaces serveurs — qui correspondent aux services qu'il fournit —, et les interfaces clientes qui correspondent aux services qu'il requiert.

La structure d'un composant Fractal met en jeu deux parties : une membrane — qui possède des interfaces fonctionnelles et des interfaces permettant l'introspection et la configuration (dynamique) du composant —, et un contenu qui est constitué d'un ensemble fini de sous-composants.

Les interfaces d'une membrane sont soit externes, soit internes. Les interfaces externes sont accessibles de l'extérieur du composant, alors que les interfaces internes sont accessibles par les sous-composants du composant. La membrane d'un composant, à rapprocher de la notion de conteneur, est constituée d'un ensemble de contrôleurs. Les contrôleurs peuvent être considérés comme des méta-objets ayant chacun un rôle particulier : par exemple, certains contrôleurs sont chargés de fournir une représentation causalement connectée de la structure d'un composant (en termes de sous-composants). D'autres contrôleurs permettent de contrôler le comportement d'un composant et/ou de ses sous-composants. Un contrôleur peut, par exemple, permettre de suspendre/reprendre l'exécution d'un composant.

Le modèle Fractal fournit deux mécanismes permettant de définir l'architecture d'une application : l'imbrication (à l'aide des composants composites) et la liaison. La liaison permet aux composants Fractal de communiquer. Fractal définit deux types de liaisons : primitives et composites. Les liaisons primitives associent une interface requise et une interface fournie deux composants résidant dans le même espace d'adressage. Par exemple, une liaison primitive dans le langage C (resp. Java) est implantée à l'aide d'un pointeur (resp. une référence). Les liaisons composites sont des chemins de communication arbitrairement complexes entre deux interfaces de composants. Les liaisons composites sont constituées d'un ensemble de composants de liaison (e.g. stub, skeleton) reliés par des liaisons primitives.

Le modèle de composant Fractal définit également la notion de composant partagé. Un tel composant appartient à plusieurs composants composites. Les composants partagés sont particulièrement adaptés à la modélisation des ressources.

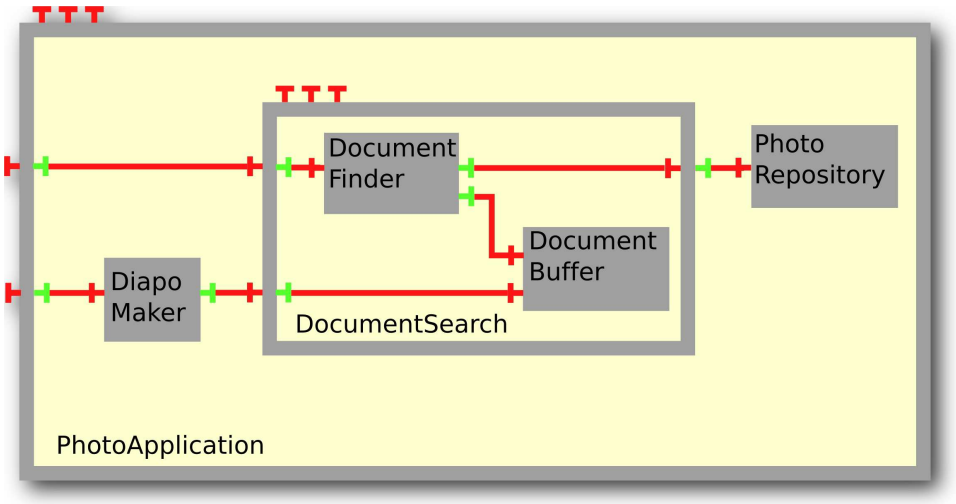


FIG. 5.1: Architecture Fractal de DiapomaKer

5.1.4 Exemple : DiapomaKer

Nous reprenons l'exemple de l'application DiapomaKer introduite au chapitre 1 (page 17) de ce mémoire. Cette application permet à un utilisateur d'effectuer des recherches sur une collection de photos et de réaliser un diaporama avec les photos sélectionnées depuis n'importe quelle machine qu'il possède. La figure 5.1 décrit l'architecture de l'application DiapomaKer en termes de composants Fractal. Le composant racine, le composant composite PhotoApplication, inclut un composant générique permettant de faire des recherches (DocumentSearch). Ce composant est aussi un composant composite, constitué des composants primitifs DocumentFinder et DocumentBuffer.

Le composant primitif DocumentFinder fournit une interface prenant en charge des requêtes plus ou moins complexes (e.g. sur le nom des photos, des méta-données associées aux photos, etc.) et la sélection de documents à partir d'un dépôt. Les documents (dans notre cas des photos) sélectionnés sont ensuite envoyés au composant DocumentBuffer. En plus d'une interface permettant l'ajout de nouveaux documents, le composant DocumentBuffer fournit une deuxième interface offrant la possibilité de trier les documents. Cette interface ainsi que celle de DocumentFinder sont rendus accessibles en tant qu'interfaces fournies pour le composant DocumentSearch. Le composant DocumentSearch est lié au composant PhotoRepository qui correspond à un dépôt de documents spécialisés (ici, des photos). Le composant DiapoMaker réalise les fonctions de création de diaporama selon des critères paramétrables (e.g. vitesse, type de transition etc.).

Pour des raisons de contraintes matérielles et logicielles, les différents composants constituant DiapomaKer ne peuvent être installés sur chacun des équipements définissant la plateforme cible. Nous supposons dans ce chapitre que le choix du placement des composants est déjà réalisé et que ces informations de placement sont disponibles dans un descripteur de déploiement qui accompagne le descripteur d'architecture de l'application.

5.1.5 Fractal pour la définition d'applications ubiquitaires

Parmi les concepts introduits dans le modèle de composants Fractal, nous avons retenu celui de membrane pour la définition des composants ubiquitaires. Dans notre travail, la membrane entre en jeu dans la structure (membrane/contenu) des composants ubiquitaires. Cependant, dans Fractal ou du moins dans ses implantations, la membrane joue le rôle de conteneur dans lequel sont définis les services non fonctionnels au travers de contrôleurs (e.g. la gestion du cycle de vie, la gestion des configurations, la configuration des propriétés des composants). Dans notre approche, la membrane joue, en plus, un rôle particulier dans la connaissance de l'architecture de l'application et du placement des différents composants la constituant. Ainsi dans le cas d'un composant composite, la membrane associée contient la description de l'architecture de ses sous-composants directs et pour chacun d'entre eux, la (ou les) machine(s) hôte(s).

Par ailleurs, nous n'avons pas gardé dans la définition des composants ubiquitaires la notion de composant partagé.

5.2 Modèle de distribution

Notre motivation première est de déployer une hiérarchie de composants sur un réseau dynamique. 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 la manière dont il évolue sont traités respectivement aux chapitres 6 et 8 de ce mémoire. Nous nous focalisons ici sur la description des mécanismes permettant une exécution distribuée des composants hiérarchiques.

Objectif L'objectif de notre modèle de distribution est de rendre les fonctionnalités d'un composant hiérarchique disponibles depuis un ensemble de machines. Ainsi ce composant pourra être qualifié d'ubiquitaire.

5.2.1 Modèle de distribution

Dans notre approche, la définition d'un composant est couplée à son placement et cette relation est traitée différemment suivant qu'il s'agisse d'un composant composite ou d'un composant primitif. En matière 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.

Notre modèle de distribution consiste à associer à chaque composant une ou plusieurs machines cibles. Les identités de ces machines définissent la localisation des instances des composants auxquelles elles sont attachées. À chaque 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.

Définition 1 (Composant composite ubiquitaire)
Un composant composite C est dit ubiquitaire par rapport à un ensemble de machines $\mathcal{M}$ s'il existe sur toutes les machines de $\mathcal{M}$ une instance de la membrane de C. Sur chaque machine de $\mathcal{M}$ la membrane de C contient la liste des machines constituant $\mathcal{M}$ ainsi que l'état de la configuration de ses sous-composants directs.

À l'exécution, chaque instance[22] de C maintient localement la configuration des sous-composants qu'elle contient.

Ainsi, un composant composite C distribué sur $\mathcal{M}$ possède les propriétés suivantes :
- les interfaces fournies et requises de C sont accessibles sur toutes les machines m_i de $\mathcal{M}$. 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 ($j \neq i$), il existe C_j une instance de C sur m_j. Chaque C_j possède une référence distance vers p (liaison distante).

Du fait de la duplication des membranes, il devient nécessaire de contrôler l'état des informations présentes dans la membrane des composants. D'une manière générale, l'état d'une membrane est manipulé à travers ses interfaces de contrôle, chacune ayant une sémantique particulière. Nous nous sommes intéressés dans notre travail plus particulièrement à la gestion de la cohérence des informations qui concernent la localisation des sous-composants. Cet aspect est traité dans le chapitre 8 de ce mémoire. Nous donnons dans la conclusion de ce mémoire quelques pistes de réflexions sur la mise en place de mécanismes de gestion de cohérence de l'état des membranes dupliqués dans les réseaux dynamiques.

Notation Soit C un composant ubiquitaire par rapport à l'ensemble de machines $\mathcal{M}$, on notera un tel composant : $C^{\mathcal{M}}$. Par ailleurs, nous adopterons la convention graphique donnée à la figure 5.2 pour représenter une architecture mettant en jeu des composants ubiquitaires : il suffit d'indiquer pour chaque composant composite la liste des machines sur lesquelles la membrane doit être dupliquée. Il est à noter qu'en UML, nous ne représentons pas à l'aide d'un diagramme de déploiement la relation entre les instances des composants et leurs machines hôtes. En effet, comme nous le verrons dans la suite de ce mémoire, la spécification des machines cibles ne se limite pas seulement à donner l'identité des machines participant au déploiement, mais peut aussi mettre en jeu des contraintes sur le choix de ces machines.

5.2.2 Exemple

La figure 5.3 montre comment les composants ubiquitaires PhotoApp et DocumentSearch sont distribués sur les machines m_2 et m_3. Les membranes des composants composites sont dupliquées sur ces machines. Sur une machine m où un composant primitif p n'est pas dupliqué (par exemple PhotoRepository sur m_2), la mise en œuvre de la liaison distante vers p est représentée par un composant mandataire sur m. Ce mandataire possède les mêmes interfaces (fournies et requises) que p.

[22]Comme cela a été avancé dans la section justifiant l'intérêt d'un modèle de composants hiérarchiques, l'instance d'un composant composite se manifeste par l'existence d'un objet membrane à l'exécution. Ainsi dans ce mémoire, lorsque nous utiliserons le terme *instance d'un composant composite*, cela désigne plus précisément l'*instance de la membrane du composant composite*.

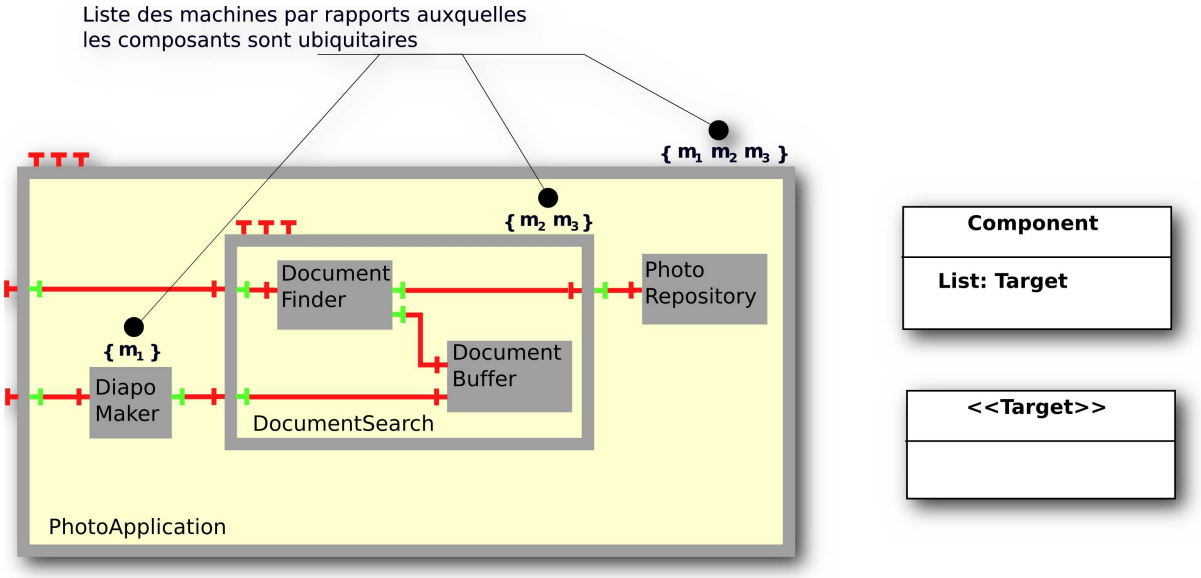


FIG. 5.2: Convention pour la représentation graphique d'une architecture à composants utilisant des composants ubiquitaires. Le schéma de gauche représente la convention utilisée dans le modèle de composant Fractal. Le schéma de droite précise les artefacts à utiliser dans UML pour la définition des machines cibles des composants ubiquitaires.

Remarques sur les composants mandataires Pour un composant primitif, nous avons fait le choix de représenter une liaison distante par la présence d'un composant mandataire. Le fait que le mandataire soit un composant facilite grandement la gestion de la distribution ainsi que la gestion de l'architecture de l'application. Par exemple, les opérations d'instanciation et d'activation d'un composant composite et de ses sous-composants sur chacune des machines participant au déploiement de l'application se font indifféremment de la localisation des composants. Il en est de même pour toutes les activités de reconfiguration de l'architecture notamment celles de reconfiguration des liaisons.

Convention graphique utilisée Sur une machine, l'instance d'un composant composite ubiquitaire sera représentée en indiquant la membrane d'un tel composant par un rectangle avec des tirets. Pour un composant primitif ubiquitaire, deux situations sont à distinguer : le code du composant primitif s'exécute sur la machine considérée ou bien un composant mandataire s'y exécute. Dans le premier cas, une boite pleine sera utilisée, dans le second cas, une boite claire (vide) avec des tirets est dessinée.

De l'exemple précédent, nous pouvons définir par construction un composant *primitif* ubiquitaire :

Propriété 1 (Composant primitif ubiquitaire) *Soit $C^{\mathcal{M}}$ un composant composite ubiquitaire et p un composant primitif, sous composant direct de C. Alors p est un composant ubiquitaire par rapport à $\mathcal{M}$: les fonctionnalités offertes par p sont disponibles sur toute machine de $\mathcal{M}$. Nous noterons un tel composant primitif $p^{\mathcal{M}}$.*

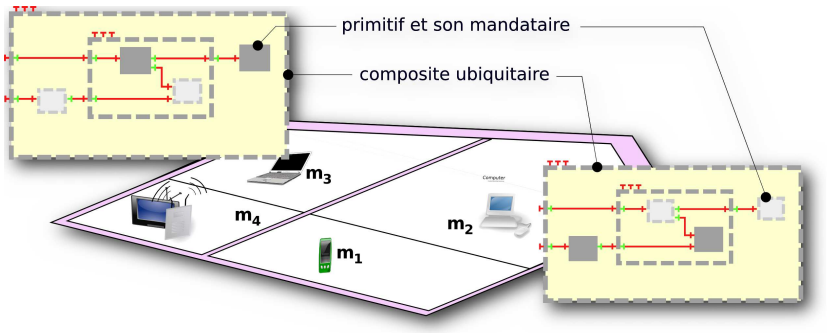


FIG. 5.3: Distribution des composants ubiquitaires dans DiapomaKer

Les fonctionnalités de p sont effectivement ubiquitaires par rapport à $\mathcal{M}$, au sein de son composant parent C : nous avons soit une instance de p ou soit un composant mandataire au sein des composites présents sur les machines de $\mathcal{M}$.

Ainsi nous pouvons dégager une règle simple pour rendre un composant ubiquitaire :

Propriété 2 (Rendre un composant ubiquitaire) *Soit un p un composant (primitif ou composite) non ubiquitaire, i.e. un composant dont les fonctionnalités ne sont accessibles que depuis une seule machine. Pour rendre p ubiquitaire par rapport à un ensemble de machines $\mathcal{M}$, il suffit d'encapsuler p dans un composant composite ubiquitaire $C^{\mathcal{M}}$.*

5.3 Composition de composants ubiquitaires

Nous nous intéressons dans cette section à détailler la distribution de composants ubiquitaires primitifs et composites à partir de la définition d'une architecture mettant en jeu de tels composants. Nous nous focalisons ici sur l'interaction entre composants ubiquitaires. Nous étudierons donc les différents scénarii de composition impliquant des composants ubiquitaires, à savoir :

- un primitif ubiquitaire lié à un autre primitif ubiquitaire ;
- un primitif ubiquitaire lié à un composite ubiquitaire ;
- un composite ubiquitaire lié à un composite ubiquitaire ;
- et enfin un composite ubiquitaire lié à un primitif ubiquitaire.

Après avoir explicité l'instanciation des composants ubiquitaires, prérequis avant toute composition, les différents cas énumérés ci-dessus seront détaillés en présentant les schémas de composition entre composants ubiquitaires, primitifs et composites.

5.3.1 Instanciation d'un composant ubiquitaire

Le choix du placement des composants étant réalisé, l'instanciation d'une architecture constituée de composants ubiquitaires ne diffère pas d'une configuration « classique ». Dans l'approche traditionnelle, une seule cible de déploiement est définie pour chaque composant et les différentes technologies étudiées dans le chapitre précédent s'appuient sur le descripteur d'architecture pour ordonner l'instanciation des composants sur chacune des machines

spécifiées et la mise en place des liaisons entre les composants ainsi créés. Dans notre approche, plusieurs machines peuvent instancier le « même composant » dès lors que ce dernier est défini comme étant ubiquitaire. Comme nous l'avons défini plus haut, l'instanciation d'un composant ubiquitaire $C^{\mathcal{M}}$ consiste à instancier sur chacune des machines de $\mathcal{M}$ la membrane de C si ce dernier est un composant composite. Si C est un composant primitif, une seule machine de $\mathcal{M}$ a instancié le code de C, les autres machines ayant déployé un composant mandataire, représentant de C.

5.3.2 Schémas de composition

5.3.2.1 Les composants primitifs

Un composant primitif n'étant pas dupliqué, son caractère ubiquitaire est rendu par l'existence de composants mandataires sur les machines où ses fonctionnalités doivent également être rendues. Nous nous sommes imposés cette restriction car la duplication d'un primitif induit la duplication des données qu'il contient. Ainsi en autorisant la duplication des données de l'application, la gestion de la cohérence de ces données (manipulées par un composant primitif) devient difficile du fait du caractère dynamique du réseau et de plus, est spécifique à chaque application. Il n'existe en effet aucune solution générique pour gérer la cohérence de données dupliquées ; cependant, selon le type de données manipulées, des solutions très abouties, même dans des environnements partiellement connectés, existent (e.g. le système de fichier CODA, les systèmes de contrôle de versions CVS, SVN, etc.)

Un primitif ubiquitaire s'exécute dans un composite ubiquitaire. De ce fait l'ensemble des machines sur lesquels les fonctionnalités du primitif seront disponibles est exactement l'ensemble des machines défini par le composant parent. La figure 5.6 illustre comment deux composants primitifs ubiquitaires p_1 et p_2 (le premier utilisant le second) sont distribués sur l'ensemble de machines $\{h_1, h_2\}$ défini par le composant composite ubiquitaire englobant. Conformément à notre modèle de distribution, nous voyons que le code de chaque composant ne s'exécute que sur une seule machine. Sur celles où le composant primitif ne s'exécute pas, nous trouvons un composant mandataire, possédant les mêmes interfaces serveurs et clientes que le composant primitif qu'il représente.

Ainsi la composition de deux composants primitifs ubiquitaires par rapport à $\mathcal{M}$ met en jeu une liaison entre deux composants primitifs ubiquitaires. Cette liaison est réalisée à l'exécution sur chaque machine de $\mathcal{M}$ entre deux composants hébergés localement. Ces composants peuvent prendre deux formes. Soit ils résultent de l'instanciation du code implantant le composant primitif. Cette instanciation, unique, ne se fait que sur une seule machine de $\mathcal{M}$. Soit ils correspondent à l'instanciation d'un composant mandataire.

Afin de mieux comprendre le rôle des différentes entités entrant en jeu dans la distribution des composants primitifs ubiquitaires, nous donnons à la figure 5.5 et 5.6 les diagrammes de séquences représentant l'appel d'une fonctionnalité d'un composant primitif ubiquitaire selon que l'appel est lancé depuis la machine hébergeant le code du composant ou depuis une machine où se trouve un de ses représentants, i.e. un composant mandataire.

Sur la figure 5.5 (a) l'appel de méthode est fait depuis la machine h_1, c'est-à-dire sur la machine où est instancié le code des composants primitifs p_1 et p_2. Aucun appel distant n'est donc requis dans ce cas.

Sur la figure 5.5 (b), l'appel de méthode sur l'interface fournie du composant p_1 est réa-

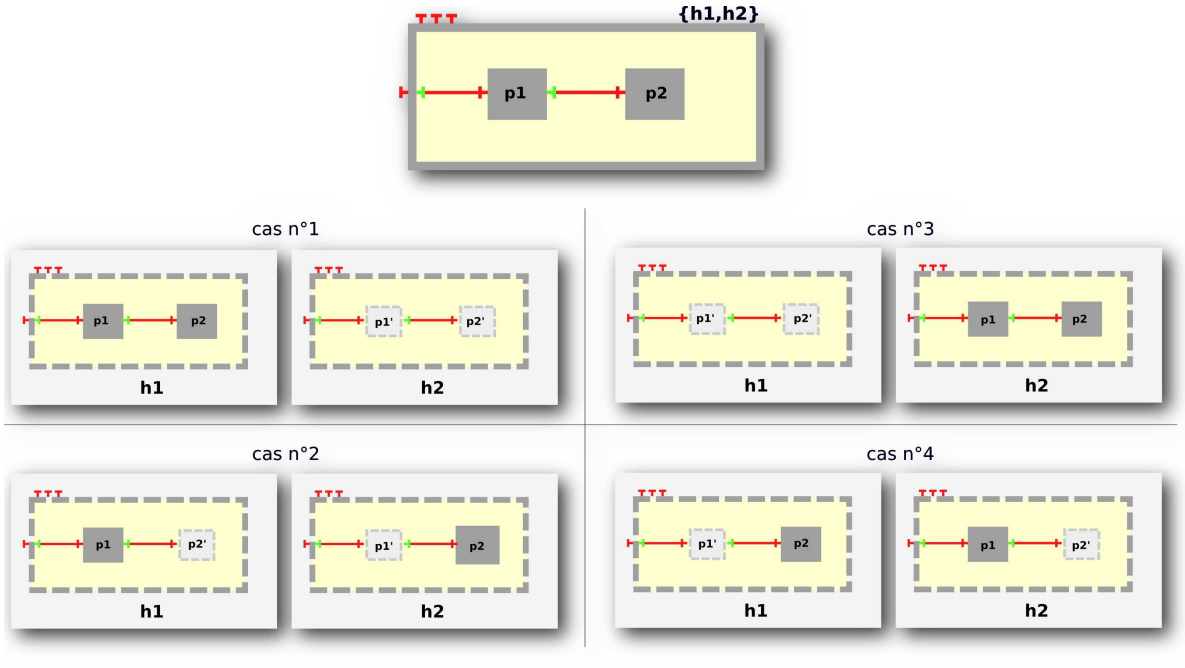


FIG. 5.4: Distribution de deux composants primitifs ubiquitaires.

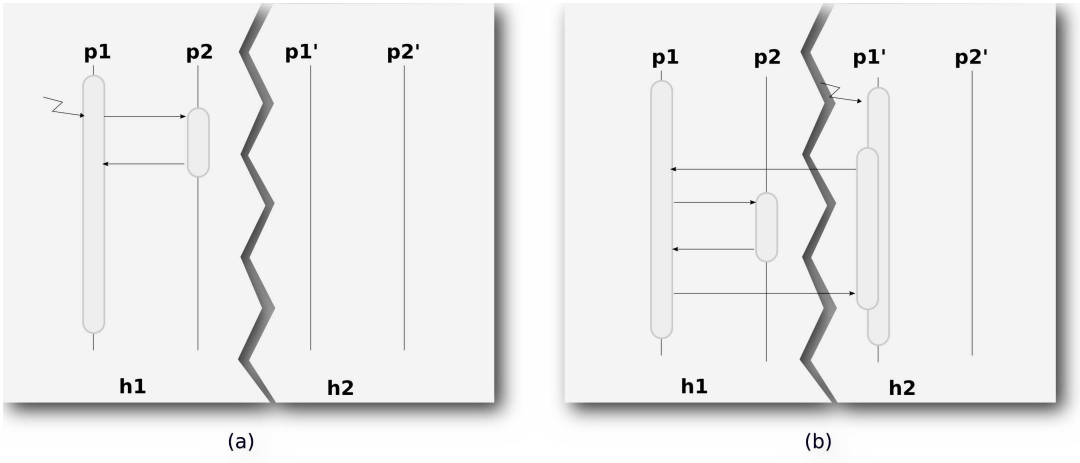


FIG. 5.5: Diagramme de séquence représentant l'appel de méthode sur un composant primitif ubiquitaire selon la machine où a été initié l'appel. Ce diagramme est à associer au cas 1 de la figure 5.6

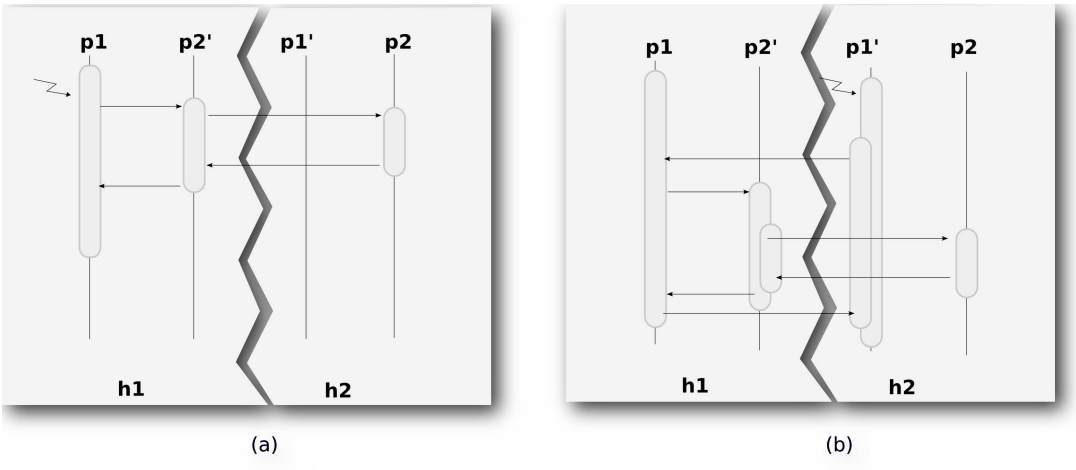


FIG. 5.6: Diagramme de séquence représentant l'appel de méthode sur un composant primitif ubiquitaire selon la machine où a été initié l'appel. Ce diagramme est à associer au cas 2 de la figure 5.4

lisée depuis la machine h_2 sur laquelle n'est instanciée que des composants mandataires. Le composant mandataire de p_1 transfert l'appel vers p_1, instancié sur h_1. Comme dans le cas précédent, les appels entre p_1 et p_2 se font localement sur h_1. Le retour de l'appel est finalement faite de p_1 vers son mandataire p'_1.

Sur la figure 5.6, p_1 et p_2 résident respectivement sur h_1 et h_2. Les interactions avec les composants mandataires sont donc plus fréquents.

Les diagrammes de séquences des figures 5.5 et 5.6 montrent que les interfaces clientes des composants mandataires ne sont jamais utilisées. Ce comportement, attendu, vient du fait que chaque composant mandataire ne fait que relayer les appels de méthodes vers la machine hébergeant le code du composant que représente ce mandataire. Cela n'aurait pas été le cas, si le composant mandataire contenait du code métier (en cas d'autorisation de duplication des composants primitifs par exemple).

De plus, ces diagrammes montrent que l'assignation de la référence distante à un composant mandataire est propre à chaque site : cela vient du fait que chacune des membranes dupliquées possède la connaissance de la distribution (et donc des machines cibles) de ses sous-composants.

5.3.2.2 Les composants composites

Un composant composite est rendu ubiquitaire par rapport à un ensemble de machines $\mathcal{M}$ en dupliquant sa membrane sur toutes les machines de $\mathcal{M}$. L'utilisation d'un composant composite ubiquitaire par un primitif ubiquitaire implique que ce primitif est également ubiquitaire par rapport à $\mathcal{M}$. Ainsi la liaison d'un primitif ubiquitaire vers un composite ubiquitaire correspond à une liaison :

– soit de l'instance (unique) du primitif vers la membrane locale du composite ;
– soit d'un composant mandataire vers la membrane locale du composite.

L'utilisation d'un composant composite ubiquitaire par un composant primitif (ubiquitaire) est illustrée à la figure 5.7. La liaison vers un composant ubiquitaire se manifeste à

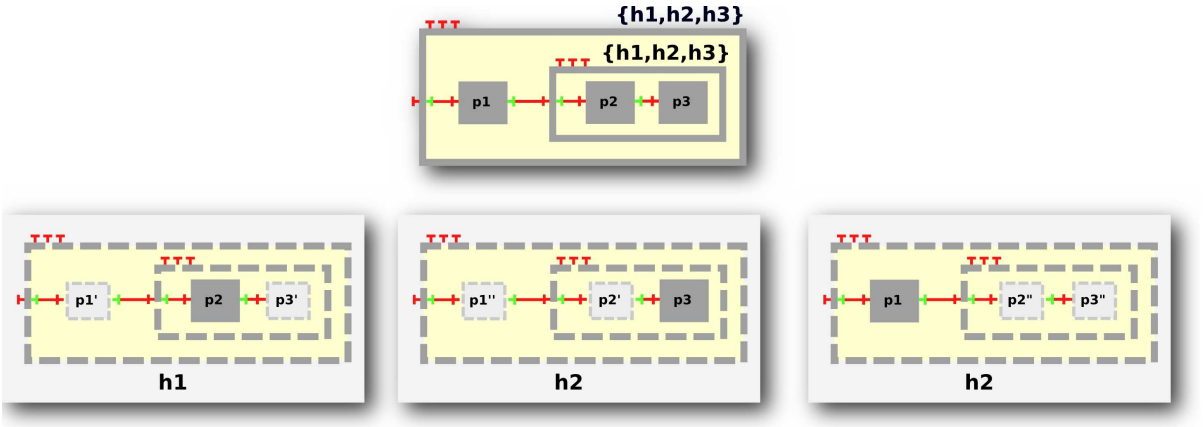


FIG. 5.7: Utilisation d'un composant composite ubiquitaire

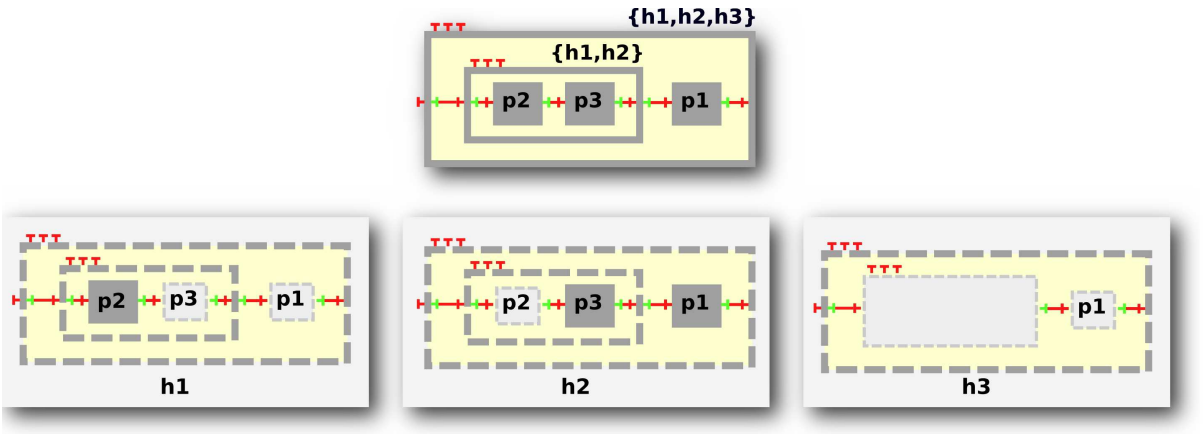


FIG. 5.8: Utilisation d'un composant primitif ubiquitaire

l'exécution par une liaison locale puisque la membrane d'un tel composant est dupliquée sur l'ensemble des machines où cette liaison doit être mise en place.

De la même manière, la liaison entre une interface requise d'un composant ubiquitaire et l'interface fournie d'un autre composant (ici primitif) est une liaison locale. La figure 5.8 illustre la composition entre un composant ubiquitaire C et un composant primitif ubiquitaire p. Sur toutes les machines (h_1, h_2 et h_3) la liaison entre les entités représentant ces composants à l'exécution sont des liaisons locales (e.g. sur la machine h_1, la membrane de C est reliée au composant mandataire représentant p).

5.3.3 Restriction de la distribution

Nous avons jusqu'à maintenant considéré qu'un composant ubiquitaire $p^{\mathcal{M}}$ fournissait ses fonctionnalités sur n'importe quelle machine de $\mathcal{M}$, ensemble directement hérité de son composant parent $Parent^{\mathcal{M}}$. Notre modèle de distribution prévoit en réalité que l'ensemble des machines cibles pour un composant ubiquitaire peut être un sous-ensemble de celui de son composant parent, i.e. que si $p^{\mathcal{M}'}$ est un sous composant de $Parent^{\mathcal{M}}$ alors $\mathcal{M}' \subseteq \mathcal{M}$. Les

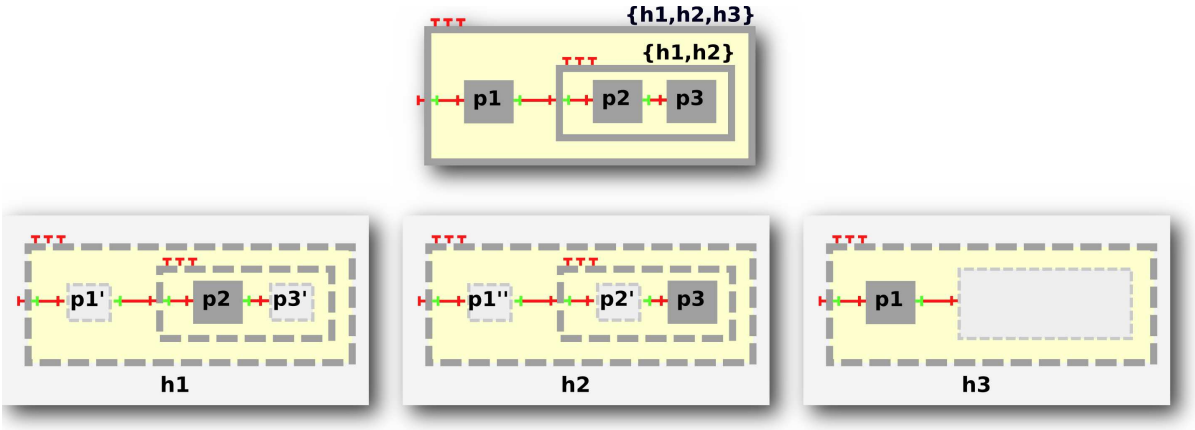


FIG. 5.9: Utilisation d'un composant composite partiellement ubiquitaire

différents schémas de distribution précédents considéraient $\mathcal{M}' = \mathcal{M}$.

5.3.3.1 Composants primitifs

Considérons un composant primitif ubiquitaire $p^{\mathcal{M}'}$, sous-composant direct du composite ubiquitaire $Parent^{\mathcal{M}}$ tel que $\mathcal{M}' \subset \mathcal{M}$ (relation stricte). Le placement d'un tel composant est défini comme suit :

- il existe une machine $m_i \in \mathcal{M}$ sur laquelle p a été instancié ;
- sur toutes les autres machines $m_j \in \mathcal{M}$ tel que $j \neq i$, un composant mandataire, représentant de p a été créé.

Sur toutes les autres machines n'appartenant pas à $\mathcal{M}'$, sur lesquels p ne doit pas être instancié, c'est un composant mandataire qui est créé. L'idée principale en définissant un p ubiquitaire sur $\mathcal{M}'$ est qu'une instance de p peut être créée sur n'importe quelle machine de $\mathcal{M}'$. Il reste encore à déterminer laquelle (ce qui sera discuté dans le chapitre 8).

5.3.3.2 Composants composites

Pour un composant composite $C^{\mathcal{M}'}$, si $\mathcal{M}' \subset \mathcal{M}$ (relation stricte), nous pouvons noter que nous retrouvons le schéma de distribution d'un composant primitif avec l'utilisation de composants mandataires : sur chaque machine $m_i \in \mathcal{M}'$ la membrane de C est dupliquée et pour chaque machine $m_j \in \mathcal{M}/i \neq j$ il existe un composant mandataire représentant une instance de C. La figure 5.9 illustre la distribution d'un composant composite ubiquitaire dont l'ensemble des machines cibles est restreint par rapport à celui de son composant parent. Ainsi, sur la machine h_3, le composant composite est représenté par un composant mandataire tandis que sur les autres machines, sa membrane est présente.

Ainsi, nous pouvons caractériser plus précisément un composant mandataire par :

Définition 2 (Composant mandataire d'un composite ubiquitaire)
Soit $Parent^{\mathcal{M}}$ un composant composite ubiquitaire et $C^{\mathcal{M}'}$ un sous-composant composite direct tel que $\mathcal{M}' \subseteq \mathcal{M}$. Sur chaque machine $m_i \in \mathcal{M}'$ il existe une membrane C_i de C et sur chaque machine $m_j \in \mathcal{M} \setminus \mathcal{M}'$ il existe un composant mandataire représentant d'un des C_i (n'importe lequel).

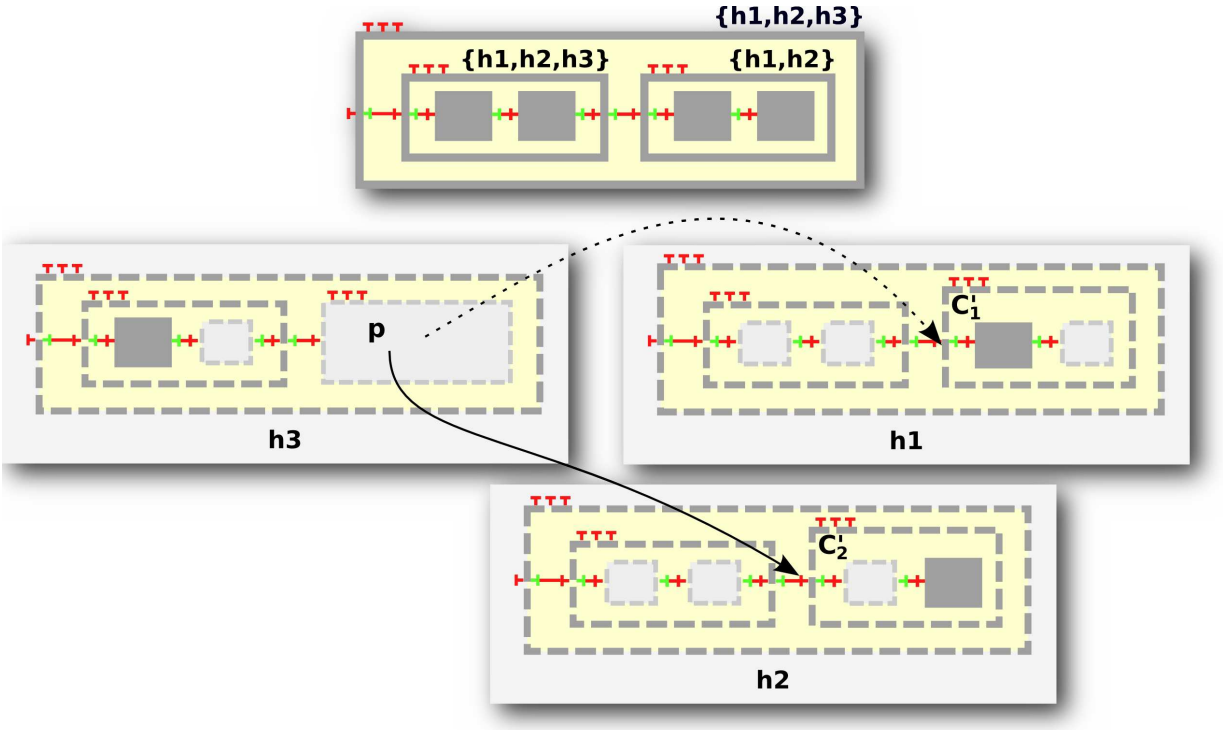


FIG. 5.10: Composant composite partiellement ubiquitaire. Le composant mandataire p peut être représentant aussi bien de C'_1 que de C'_2

5.3.3.3 Mise en place des liaisons

Lorsqu'un composite ubiquitaire est contraint dans son placement, on se retrouve avec des machines sur lesquelles sont dupliquées sa membrane et d'autres machines sur lesquelles a été créé un composant mandataire. Comme tout mandataire, ce dernier contient l'identifiant (i.e. le nom du composite et un nom de machine) de la membrane qu'il représente. Or, la membrane du composite étant dupliquée, il peut exister plusieurs possibilités d'affectation de cet identifiant. La figure 5.10 illustre un composant composite C ubiquitaire par rapport à $\{h_1, h_2\}$ tandis que son composant parent est ubiquitaire par rapport à $\{h_1, h_2, h_3\}$. La membrane de C est dupliquée sur h_1 et h_2 tandis que sur h_3, C est représenté par un mandataire. Ce mandataire peut aussi bien représenter la membrane présente sur h_1 que celle créée sur h_2.

Ainsi plusieurs choix sont possibles pour définir au niveau du composant mandataire la machine hébergeant la membrane qu'il représente. Plusieurs stratégies peuvent guider ce choix (e.g. choisir la machine vers laquelle la bande passante est la plus élevée, choisir une machine accessible). Nous donnons dans la section suivante quelques pistes utiles dans le cadre des réseaux dynamiques.

5.4 Gestion des déconnexions

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 l'éventualité de déconnexions ne peut être écartée, 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). Nous proposons donc d'expliciter les dépendances entre composants au niveau de leurs interfaces afin de permettre un fonctionnement de l'application même en face de déconnexions réseau.

5.4.1 Interfaces actives

Reprenons l'exemple de l'application DiapomaKer. Cette permet d'une part de faire des recherches parmi un ensemble de photos et d'autre part de réaliser un diaporama à partir des photos trouvées. Lorsque des photos ont été préalablement sélectionnées, il est tout à fait possible que la machine sur laquelle réside la base de données de photos ou bien celle hébergeant le composant prenant en charge les requêtes ne soit plus accessible. Bien que les fonctions de recherches ne soient alors plus disponibles il serait intéressant de pouvoir continuer à utiliser l'application DiapomaKer, ne serait-ce que pour visionner (ou construire) le diaporama.

La figure 5.11 montre l'assemblage de composants mettant en évidence deux parties de l'architecture de DiapomaKer qui peuvent être isolées. Nous avons représenté sous forme de graphe les dépendances entre interfaces requises et fournies pour DiapomaKer. Le graphe de dépendance est construit en définissant un sommet pour chaque interface (requise et fournie) d'un composant et une arêt entre deux sommets dès lors qu'il existe une liaison entre l'interface requise et fournie correspondantes. Cette arête définit la double dépendance de l'interface requise vis-à-vis de l'interface fournie et de l'interface fournie vers l'interface requise. Dans le cas d'un composant primitif, l'approche boîte noire — définissant le composant uniquement par ses interfaces externes et masquant son fonctionnement interne — et l'absence de formalisme permettant de préciser les relations entre interfaces fournies et requises nous ont conduit à exprimer une dépendance entre une interface fournie et *toutes* ses interfaces requises.

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 aboutiront. À l'inverse, l'interface a_1 doit pouvoir être *désactivée* afin de rendre impossibles les invocations sur cette interface.

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* indique qu'un appel de méthode va pouvoir aboutir. Les invocations de méthode sur une interface active vont pouvoir être déléguées au composant possédant cette interface. Une interface *inactive* à l'inverse indique qu'une invocation ne pourra être réalisée correctement. Ayant cette information, nous pouvons définir plusieurs stratégies à adopter face à une invocation sur une telle interface (nous donnons quelques exemples de stratégies dans la section 5.4.2).

L'état des interfaces est défini par les clauses suivantes :

Définition 3 (Interfaces actives)

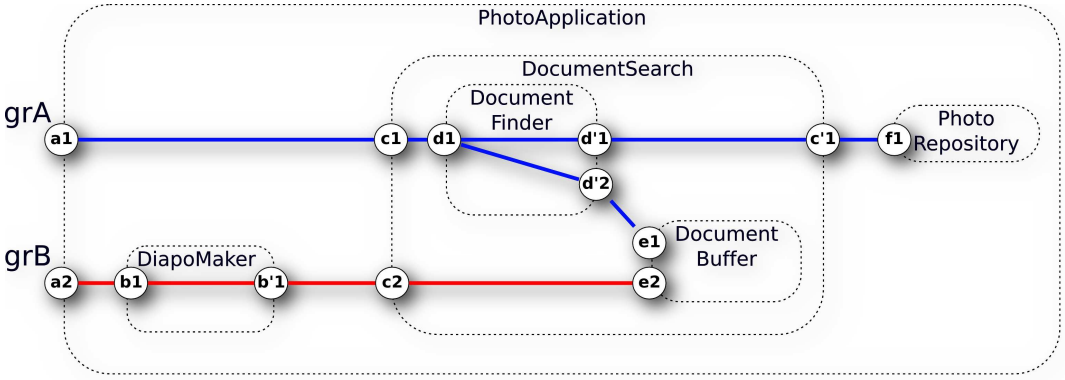


FIG. 5.11: Dépendances entre interfaces fournies et requises au sein de l'application DiapoMaker. Les graphes *grA* et *grB* mettent en évidence la sous architecture du composant **A** qui peut être isolée.

1. *une interface requise est active si elle est liée à une interface fournie et que celle-ci est également active.*
2. *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[23]. Du fait de notre choix concernant les dépendances entre interfaces d'un composant primitif, il en découle que la désactivation d'une seule interface requise a pour effet la désactivation de toutes ses interfaces fournies.

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 du composite selon l'assemblage de ses sous-composants.

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'interconnexions entre composants, va activer ou désactiver les interfaces des autres composants concernés.

[23]Notre définition d'une interface active diffère sensiblement de celle donnée dans [Hei98, DES01] où la notion d'activité d'une interface porte sur la possibilité de modifier son comportement.

5.4.2 Utilisation des interfaces actives

La notion d'interface active et inactive permet de répercuter les déconnexions (indisponibilité ou inaccessibilité d'un composant) sur l'architecture des composants pendant leur exécution. En explicitant les dépendances entre interfaces requises et fournies, il est ainsi possible d'isoler les parties de l'architecture qui ne peuvent rendre leurs services du fait de l'absence ou l'inaccessibilité d'un ou plusieurs composants.

En répercutant les déconnexions sur l'architecture, la définition d'un état actif/inactif sur les interfaces permet la mise en place de comportement spécifiques suivant l'état des interfaces. Nous donnons ci-dessous quelques exemples d'utilisation de la notion d'interfaces actives pour la gestion des déconnexions.

5.4.2.1 Reconfiguration de l'architecture

Un exemple d'utilisation des interfaces actives 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 5.11, la substitution du composant PhotoRepository demande l'inactivation de ses interfaces fournies (f_1) ce qui va désactiver de proche en proche l'interface c_1 de DocumentSearch. Ainsi l'interface c_2 de DocumentSearch reste active ce qui n'aurait pas été le cas si ce composant avait été *arrêté* entièrement.

5.4.2.2 Composants ubiquitaires

Nous avons vu que les interfaces requises des composants mandataires n'étaient pas utilisées dans la chaîne d'invocation des méthodes au sein de la structure hiérarchique d'une application ubiquitaire. L'intérêt d'avoir choisi des composants mandataires, i.e. des entités de même nature que les composants qu'ils représentent est qu'il est possible de définir les interfaces actives et inactives sur de telles entités. Ainsi, si un composant mandataire n'est plus en mesure d'assurer une liaison distante, ses interfaces fournies seront désactivées, évitant ainsi une invocation distante dont l'acheminement ne pourrait être assuré.

5.4.2.3 Sémantique des interfaces actives

Les raisons pour lesquelles une liaison distante est invalidée peuvent varier selon la nature de l'application. Cela peut aussi bien provenir d'une inaccessibilité au niveau réseau que d'une défaillance (logique ou physique) du composant distant. Selon les situations, des phénomènes d'hystérésis peuvent apparaître conduisant à des cycles de désactivation / activation pouvant entraîner un blocage de l'application. Afin d'éviter cela, nous pouvons implanter au niveau des composants mandataires des stratégies, propres à chaque application. On préférera par exemple laisser l'interface fournie d'un tel composant active et réaliser une mise en cache des appels entrant pendant toute la durée d'une phase d'instabilité du réseau.

6

Spécification du déploiement des applications ubiquitaires dans des réseaux dynamiques

Le déploiement d'une application distribuée reste une tâche fastidieuse du fait des nombreuses interventions manuelles pour la définition de scripts de configuration, l'installation et l'exécution de composants sur chacun des sites de déploiement. En particulier, la définition d'une machine cible pour chacun des composants impose une connaissance assez précise de la plate-forme de déploiement constituée des machines du réseau. En effet, le choix d'une machine parfaitement identifiée suppose que cette dernière propose l'ensemble des ressources nécessaires à l'exécution du composant. Dans des environnments constitués d'équipements mobiles et aux ressources fluctuantes, le choix du placement de chaque composant est rendu encore plus difficile voire impossible dans des phases antérieures au déploiement : dans ces réseaux dynamiques, aucune hypothèse ne peut être faite sur la présence d'une machine particulière ou sur la disponibilité de certaines ressources.

Ainsi, les réseaux dynamiques ne permettent pas de se fonder sur une simple mise en correspondance a priori entre les composants et des identités de machines. Chaque composant est susceptible de poser des contraintes sur les ressources de la machine qui l'héberge et l'état de ces ressources est susceptible d'évoluer dans le temps.

Dans ce chapitre, nous présentons CDL, *Constrained Deployment Language*, un langage minimaliste mais extensible permettant la spécification du déploiement d'applications à base de composants dans des réseaux dynamiques.

Sommaire

6.1 Besoins pour un déploiement sensible au contexte

6.1.1 Problématique

Comme nous l'avons énoncé dans l'introduction de ce chapitre, l'objectif du déploiement d'une application à base de composants est de déterminer une machine cible pour chacun des composants constituant l'application. Dans les approches actuelles (que nous avons présentées dans le chapitre 4), l'affectation d'une machine se fait avant le déploiement et consiste le plus souvent à indiquer dans un descripteur de déploiement l'adresse des machines devant héberger tel ou tel composant. Cette approche nécessite une connaissance a priori de l'environnement de déploiement et demande une intervention manuelle. L'affectation d'une machine pour un composant n'est d'ailleurs pas guidée seulement par la connaissance de l'existence de cette machine. En effet, le choix d'une machine (plus qu'une autre) se fait en connaissant les ressources offertes par cette dernière, ressources qui seront nécessaires au composant à installer. Ainsi, avant le déploiement d'une application, l'affectation des machines pour l'hébergement des composants implique non seulement d'avoir une connaissance sur l'identité des machines qui constitueront l'environnement d'exécution des composants, mais aussi de connaître les caractéristiques de chacunes d'entre elles.

Dans les environnements que nous avons considérés durant cette thèse, une machine peut à tout moment être en veille ou hors de la zone de portée des autres machines ; les équipements mis en jeu possèdent également des ressources susceptibles de fluctuer. Il n'est donc pas envisageable de demander aux concepteurs, aux développeurs, voire aux utilisateurs finaux de l'application, de connaître l'identité des machines devant héberger chacun des composants dans des phases antérieures au déploiement. Dans ces conditions, comment le déploiement des applications ubiquitaires peut-il être spécifié et contrôlé ?

6.1.2 Approche générale

Nous proposons de différer la détermination de la correspondance entre composants et machines au moment de l'exécution et de fonder sa spécification sur une extension d'un langage de description d'architecture existant. Cet extension prend la forme d'une description des propriétés des ressources que doivent satisfaire les machines pour pouvoir héberger des instances de composants. Il s'agit d'une approche purement déclarative telle que celle décrite dans [DKM04b]. Il n'est pas nécessaire de spécifier des noms ou adresses de machines explicitement dans la description du déploiement. Le placement des composants est décidé à l'exécution, conformément aux contraintes associées aux composants.

La description des propriétés que les machines cibles doivent satisfaire est donnée dans un descripteur de déploiement dans lequel il est possible de faire apparaître des références aux instances de composants (définis dans le descripteur d'architecture). Pour chaque composant,

un *contexte de déploiement* est ainsi défini. Un tel contexte liste toutes les contraintes qu'une machine hôte doit satisfaire.

6.2 CDL, un langage pour la spécification de déploiement contraint

Nous proposons dans le cadre de cette thèse, un langage extensible permettant la spécification du déploiement d'une architecture à base de composants sur des réseaux dynamiques.

Le langage que nous proposons, CDL (*Constrained Deployment Language*), rend possible la définition des machines cibles pour chaque composant de l'application sans pour autant connaître leur identité. CDL permet en effet de spécifier les caractéristiques d'une machine devant participer à l'hébergement d'un composant en fonction des caractéristiques exigées par ce composant. L'expression de ces besoins définit un ensemble de *contraintes* dont le respect devra être garanti par un outil de déploiement. Deux types de contraintes peuvent être définis : les *contraintes de ressources* et *les contraintes de localisation*. Les premières permettent de représenter les besoins en ressources matérielles et logicielles des composants. Les contraintes de localisation sont quant à elles utiles pour restreindre les choix de placement ou de co-placement de composants, indépendamment des ressources (on précisera par exemple que deux composants doivent résider sur des machines différentes).

Le language que nous proposons est volontairement minimaliste : les contraintes de ressources et de localisation que nous présentons ci-après sont limitées respectivement à un ensemble réduit de ressources et d'opérateurs. Nous avons souhaité en effet définir dans un premier temps les aspects directement implémentables et courramment utilisés dans les approches traditionnelles.

6.2.1 Contraintes de ressources

Même si, dans les réseaux dynamiques, il n'est pas concevable de connaître l'identité des machines qui hébergeront tel ou tel composant, il est cependant possible d'exprimer les besoins en ressources logicielles et matérielles de chaque composant, indépendamment de l'identités des machines. Ces exigences définissent les contraintes de ressources.

Nous allons détailler maintenant comment déclarer de telles contraintes pour les composants ainsi que pour leurs liaisons et nous identifierons les éléments pris en compte dans CDL.

6.2.1.1 Spécification des exigences matérielles et logicielles

Les composants Ces exigences vis-à-vis des ressources logicielles et matérielles sont exprimées dans CDL à l'aide des contraintes de ressources. L'utilisation des contraintes de ressources est illustrée dans le listing 6.1 qui montre au format XML les besoins du composant primitif PhotoRepository de l'application DiapoMaker. La contrainte de ressource présentée exprime le fait que le composant PhotoRepository doit être hébergé sur une machine qui dispose au minimum de 1 Go d'espace disque libre dans `/home`.

```
<component name="PhotoRepository">
  <deployment-context>
    <resource-constraint>
```

```
        <disk    free="1" unit="GB"
                 directory="/home/"
                 operator="min" />
      </resource-constraint>
    </deployment-context>
  </component>
```

Listing 6.1: Les besoins de chaque composant peuvent être spécifiés via les contraintes de ressources. Ces contraintes expriment les ressources logicielles et matérielles que la machine hébergeant le composant devra fournir.

Dans CDL, pour définir une contrainte de ressource pour un composant, il suffit d'associer à ce même composant un contexte (balise `deployment-context`) qui liste l'ensemble des ressources nécessaires au composant et donc que la machine l'hébergeant devra pouvoir fournir.

En plus des contraintes matérielles requises par un composant (e.g. quantité de mémoire, carte vidéo spécifique etc.), il est possible d'exprimer avec CDL des dépendances vis-à-vis d'éléments logiciels que devra fournir la plate-forme cible. Ceci se fait, comme l'illustre le listing 6.2 qui montre l'ensemble des contraintes de ressources du composant primitif DocumentBuffer.

```
<component name="DiapoMaker">
  <deployment-context>
    <resource-constraint>
      <cpu freq="1.5" unit="GHz"
           operator="min" />
      <disk     free="50"
                directory="/home/"
                unit="MB"
                operator="min"/>
      <software name="ImageMagick"
                version="6.2.4" />
    </resource-constraint>
  </deployment-context>
</component>
```

Listing 6.2: Contraintes de ressource sur un composant primitif dans CDL

Nous avons défini dans CDL un ensemble de ressources qui permettent de spécifier les exigences des composants. L'ensemble des ressources logicielles et matérielles pris en compte actuellement dans CDL est répertorié dans le diagramme de la figure 6.1. Pour la modélisation des ressources, nous avons repris les travaux développés dans la thèse de N. Le Sommer [LS03]. Chaque ressource définit un ensemble de propriétés qui la caractérise. Ce sont ces mêmes propriétés qui définissent les exigences des composants à travers un couple attribut/valeur. Ce diagramme peut facilement être étendu afin d'ajouter ou préciser de nouvelles ressources requises par un composant.

Les liaisons Au moment du déploiement d'une application distribuée, d'autres facteurs doivent être considérés pour le bon fonctionnement de l'application et notamment les ressources liées à la communication entre composants distants. Par exemple, il peut être exigé

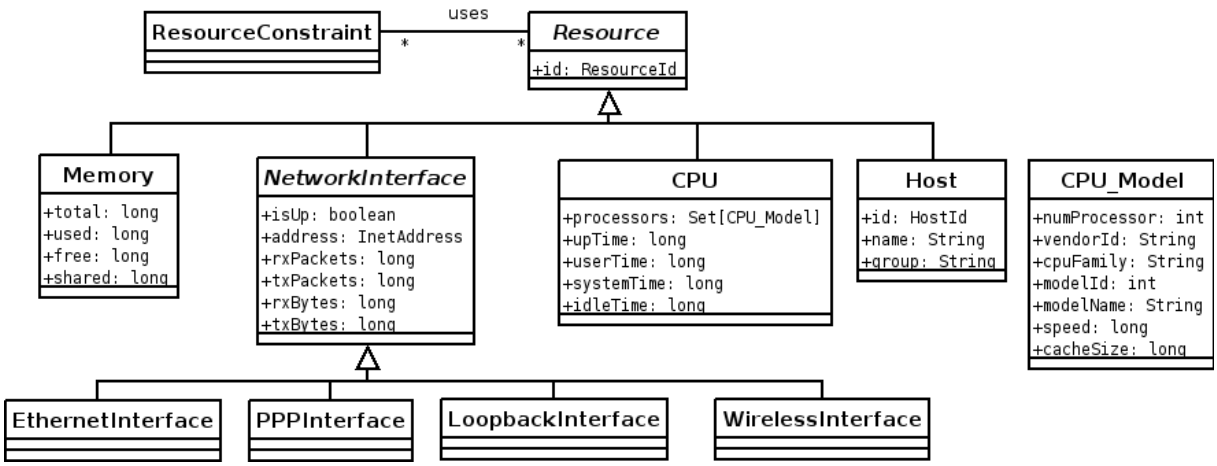


FIG. 6.1: Diagramme de classes des ressources pris en compte dans CDL. Ces ressources sont utilisées pour exprimer les exigences des composants.

une bande bassante minimale entre deux machines ou encore que les échanges de messages vers un composant passent par une communication sans fil. Ces aspects peuvent être spécifiés dans la description d'une architecture à composants au niveau de leurs liaisons. Les contraintes actuellement prises en compte dans CDL concernent les exigences sur la bande passante et les protocoles de communication. Sur la figure 6.2, nous avons représenté le diagramme de classes modélisant les contraintes actuellement prises en compte dans CDL sur les liaisons des composants. La contrainte `NetworkConstraint` est une classe abstraite servant à ajouter des contraintes supplémentaires sur les liaisons.

L'utilisation de contraintes de ressources sur les liaisons entre composants est illustrée dans le listing 6.3. Cet exemple impose que la liaison entre deux composants distants (`Client` et `Server`) soit mise en place uniquement si la bande passante[24] entre les deux machines hébergeant ces deux composants est au minimum de 150 Mbit/s.

```
<component name="ClientServer">
 <deployment-context>
    <binding from="Client" to="Server">
       <bandwidth="150" unit="Mb/s" />
    </binding>
</component>
```

Listing 6.3: Contrainte de ressources sur les liaisons entre composants : définition d'un minimum de bande passante entre le composant ClientServer et les composants dont il dépend

6.2.1.2 Composition et contraintes de ressources

Les contraintes de ressources spécifiées dans les deux listings précédents concernent des composants primitifs. Ces derniers représentant des unités de calculs, recquièrent des res-

[24]Dans cette thèse, nous nous limitons aux caractéristiques statiques des liaisons. Par conséquent, nous ne cherchons pas à obtenir des mesures instantanées de la bande passante.

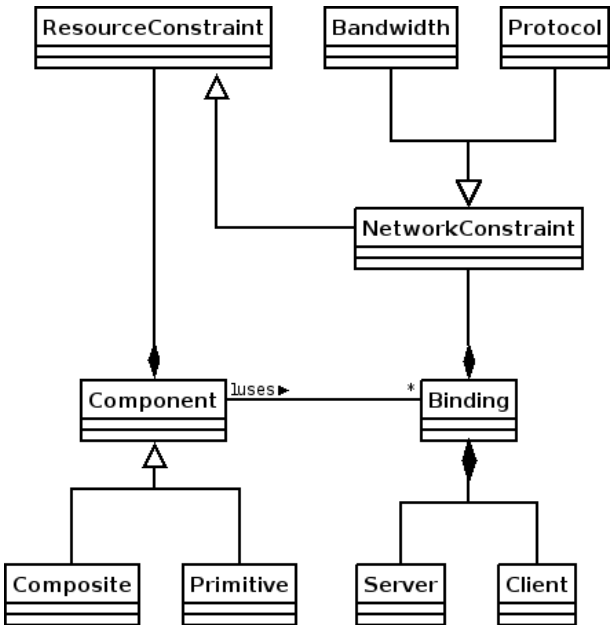


FIG. 6.2: Diagramme de classe des contraintes de ressources sur les liaisons

sources particulières assurant leur bon fonctionnement. Dans un modèle de composant hiérarchique, qu'en est-il des composants composites ?

Dans un modèle de composants hiérarchiques, les composants composites sont vus également comme des unités de composition et de déploiement. Il est donc tout à fait envisageable d'associer des contraintes de ressources à ces types de composants. Comme pour le cas des composants primitifs, une contrainte de ressources associée à un composite impose que la machine (ou les machines dans le cas des composants ubiquitaires) hébergeant ce composant dispose(nt) de la ressource exigée.

Un composite ne possédant aucune contrainte de ressources sera toujours instancié sur toutes les machines concernées par le déploiement. Par conséquent, ne pas définir de contraintes sur les composistes peut amener à un scénario de déploiement où uniquement les membranes des composites sont instanciées et où aucun primitif n'est créé (si toutes les contraintes de ressources des primitifs ne sont pas vérifiées). Dans un processus de déploiement tel que celui que nous proposons dans cette thèse, réalisant le placement d'abord des composites puis des sous-composants (i.e. du haut de la hiérarchie vers le bas), on peut éviter l'instanciation inutile de (sous) composants composites en répercutant au plus tôt dans la hiérarchie la ou les contraintes de ressources associées aux primitifs qui peuvent provoquer le « blocage » du déploiement. Il est donc utile de spécifier de telles contraintes sur les composants composites.

Cependant la définition des contraintes de ressources sur un composite pose quelques difficultés. Prenons l'exemple de deux composants primitifs qui exigent chacun 512 Mo de mémoire vive. Ces deux contraintes de ressources s'expriment-elles au niveau du composant parent par un besoin de 1024 Mo de mémoire vive ? De même pour les contraintes sur la capacité de calcul d'une machine et d'une manière générale pour toutes les ressources physiques partagées. On peut aussi spécifier une contrainte de ressources d'abord au niveau du composite (à l'inverse de l'exemple précédent) : par exemple un composite peut exiger un minimum d'espace disque. Dans ce cas, comment se partage cette ressource au niveau de ses

sous-composants ?

Beaucoup de travaux se sont intéressés à la définition d'ontologie sur les ressources et de language comportant des opérateurs de composition mais ne permettent pas l'automatisation systématique de la « propagation » des contraintes de ressources que ce soit des composants primitifs au composant composite que du composite à ses sous-composants ([SJSM04]). Les règles de composition dépendent en effet du domaine applicatif.

Nous pensons donc que la spécification des contraintes de ressources sur un composant composite doit se faire en faisant intervenir ou bien les concepteurs ou les déployeurs d'applications. C'est l'approche que nous avons adoptée pour notre étude. Il est donc possible avec CDL de définir des contraintes de ressources sur les composites. On peut envisager des outils d'aide au déployeur qui appliqueraient, dans un domaine donné, des règles de composition par défaut de ces contraintes. De cette manière, en tant qu'unité de déploiement, le composant composite reflèterait au mieux les exigences de ses sous-composants.

6.2.2 Contraintes de localisation

Au moment du déploiement, la présence de contraintes de ressources sur un composant va influer sur le placement de ces derniers. Il en est de même pour les besoins en matière de communication entre composants. En effet, une machine pourra héberger un composant si et seulement si tous les besoins exprimés à travers les contraintes de ressources sont vérifiés sur cette machine. Cependant, les contraintes de ressources ne constituent pas un moyen d'expression adapté au contrôle du placement des composants. Par exemple, il n'est pas possible d'imposer explicitement que deux composants doivent résider sur la même machine ou sur deux machines différentes[25].

Afin de pouvoir spécifier de telles contraintes, CDL introduit la notion de contrainte de localisation. Dans l'expression des contraintes de localisation, on peut désigner une machine soit par son nom (ou adresse IP) si elle est connue avant le déploiement soit par un nom de variable. Il est alors possible d'exprimer des contraintes sur l'identité des machines au niveau du composant composite englobant.

Par exemple, le listing 6.4 montre, dans le format XML, l'utilisation des contraintes de localisation pour le composant DocumentSearch, composite de DocumentFinder et DocumentBuffer. Pour chacun des composants, une contrainte de localisation est exprimée au sein de l'élément `location-constraint`. On désigne la machine hébergeant le composant DocumentFinder par x et celle hébergeant le composant DocumentBuffer par y. On indique ensuite dans les contraintes liées au composite DocumentSearch que les deux sous-composants doivent être placés sur deux machines distinctes via l'opérateur `alldiff`.

```
<component name="DocumentSearch">
 <component name="DocumentFinder">
   <deployment-context>
     <location-constraint>
        <target varname="x"/>
     </location-constraint>
```

[25] Le lecteur attentif aura remarqué que la définition d'une contrainte de ressources sur une liaison (e.g. la contrainte sur la bande passante) impose le placement des deux composants (à l'extrémité de la liaison) sur deux machines distinctes. Cependant la volonté de placer des composants sur des hôtes différents peut être guidée par d'autres considérations que les aspects liées aux performances du réseau.

```
    </deployment-context>
  </component>

  <component name="DocumentBuffer">
    <deployment-context>
      <location-constraint>
        <target varname="y"/>
      </location-constraint>
    </deployment-context>
  </component>

  <deployment-context>
    <location-constraint>
      <operator name="alldiff">
        <arg varname="this.DocumentFinder.x" />
        <arg varname="this.DocumentBuffer.y" />
      </operator>
    </location-constraint>
  <deployment-context>
</component>
```

Listing 6.4: Contraintes de localisation. Les composants DocumentFinder et DocumentBuffer doivent résider sur des machines distinctes

L'élément `target` permet de spécifier soit un nom de variable grâce à l'attribut `varname` soit le nom de la machine grâce à l'attribut `hostname` (ou `ip`). Les contraintes sur les variables sont spécifiées à l'aide d'opérateurs. Pour la définition des opérateurs, nous nous sommes inspirés de GNU-Prolog [DC01]. La liste des opérateurs actuellement supportés par CDL est donnée dans le tableau 6.1. Ces opérateurs prennent en argument des variables de placement.

Par ailleurs, chaque variable désignant une machine, il est possible d'accéder aux propriétés de cette dernière une fois la variable déclarée (e.g. au niveau du composant composite). Les propriétés correspondent à l'ensemble des exigences que l'on peut exprimer avec les contraintes de ressources comme la mémoire, la charge processeur etc. Par exemple, le listing 6.5 montre comment guider le placement des composants Client et `Server` de telle manière que le composant Server soit placé sur la machine ayant la puissance de calcul la plus élévée :

```
<component name="ClientServer">

  <component name="Client">
  <deployment-context>
    <location-constraint>
      <target varname="x"/>
    </location-constraint>
  </deployment-context>


  <component name="Server">
  <deployment-context>
    <location-constraint>
      <target varname="y"/>
```

Opérateurs	arité	description
alldiff	2+	Contraint toutes les variables passées en paramètre d'avoir des valeurs distinctes. Cette contrainte équivaut à exprimer que chaque paire de variable doit être différente.
equal	2+	Cette contrainte impose que les variables passées en argument soient égales.
greater	2	Impose que la valeur de la première variable soit strictement supérieure à la seconde
less	2	Impose que la valeur de la première variable soit strictment inférieure à la seconde
greater-eq	2	Impose que la valeur de la première variable soit supérieure ou égale à la seconde
less-eq	2	Impose que la valeur de la première variable soit inférieure ou égale à la seconde

TAB. 6.1: Liste des opérateurs permettant de contraindre les valeurs possibles des variables de placement

```
    </location-constraint>
  </deployment-context>

  <deployment-context>
    <location-constraint>
      <operator name="greater">
        <arg varname="this.Server.y.CPU"/>
        <arg varname="this.Client.x.CPU"/>
      </operator>
    </location-constraint>
  </deployment-context>
</component>
```

Listing 6.5: Contraintes de localisation et propriétés des machines cibles

6.3 Spécification du déploiement de composants ubiquitaires

L'utilisation de CDL permet de spécifier et contrôler le placement des composants d'une application à base de composants en précisant d'une part leurs besoins, à l'aide des contraintes de ressources, et d'autre part en imposant des contraintes sur le choix des machines à travers les contraintes de localisation. Cette spécification s'applique aussi bien aux composants ubiquitaires qu'aux applications distribuées « classiques ». L'avantage de CDL par rapport aux langages existant est de pouvoir s'affranchir d'une connaissance plus ou moins totale de l'environnement de déploiement. Nous allons détailler dans cette section les particularités de CDL lorsque l'application est ubiquitaire, i.e. constituée de composants ubiquitaires.

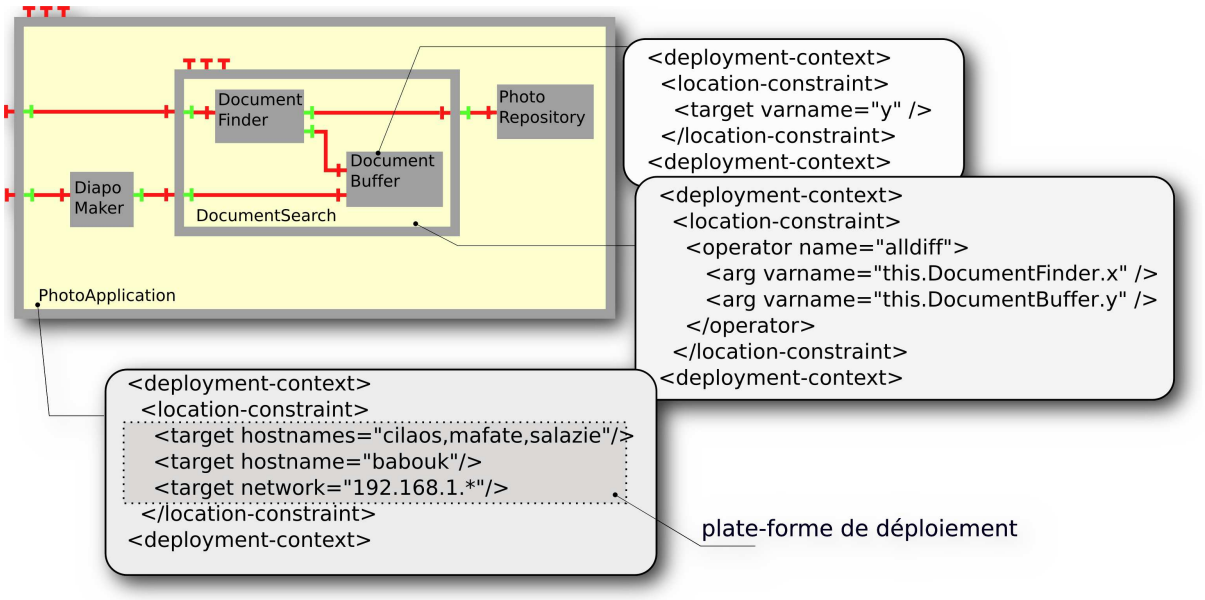


FIG. 6.3: Définition de la plate-forme de déploiement avec CDL

6.3.1 Spécification de la plate-forme de déploiement

La prise en compte dans CDL des composants ubiquitaires est réalisée en considérant le schéma de distribution de ces composants. Comme cela a été détaillé dans le chapitre précédent, un composant primitif se voit attribué une seule machine hôte tandis qu'un composant composite est réparti sur un ensemble de machines. Nous retrouverons donc cette spécificité dans la définition des machines cibles des composants ubiquitaires.

La définition d'une machine cible se fait dans CDL à l'aide de l'élément XML `target`. Pour un composant composite ubiquitaire $C^{\mathcal{M}}$, plusieurs clauses `target` pourront être définies, chacune d'elles représentant soit une machine de $\mathcal{M}$ ou soit une variable (prenant ses valeurs dans $\mathcal{M}$). Ainsi l'ensemble des clauses `target` définit l'ensemble des machines par rapport auxquelles C est ubiquitaire. La figure 6.3 illustre la spécication de la plate-forme de déploiement pour l'application DiapomaKer où les machines (et ensemble de machines — défini par l'attribut `network` de l'élément `target`) sont précisés au niveau du composite racine.

Pour un composant primitif p sous-composant de $C^{\mathcal{M}}$, une seule clause `target` peut être précisée et la machine hébergeant ce primitif devra être choisie parmi celles de son composant parent, i.e. également parmi $\mathcal{M}$.

6.3.2 Assemblage de composants COTS avec CDL

La programmation orientée composants a souvent vanté la possibilité de constituer une application par assemblage de COTS (*Commercial-Off-The-Shelf*), i.e. d'*entités* ayant été développées et packagées par divers revendeurs. La viabilité de cette approche réside dans le fait que les composants sont des unités de composition et de déploiement. Cependant l'intégration d'entités externes pose beaucoup de problèmes. Même si l'intégration de composants selon l'approche boîte noire offre de nombreux avantages, il n'est généralement pas possible d'accéder et de modifier le code du composant à intégrer, ce qui n'apporte pas de garantie totale

au développeur voire même à l'utilisateur. Par ailleurs, il n'est pas possible de contrôler les exigences d'un COTS et ce manque se retrouve à la fois du côté de l'utilisateur mais aussi de celui qui fournit le COTS : la spécification des besoins des composants est généralement délaissée, l'hypothèse que la plate-forme cible répondra aux exigences est généralement faite. Par ailleurs, lorsque des contraintes de placement doivent être considérées, il n'existe pas de formalisme ni d'outil permettant de garantir ces contraintes tout le long du cycle d'assemblage des COTS.

Nous pensons que CDL peut simplifier l'assemblage de composants provenant de sources différentes. Lors de la conception d'une application, un vendeur de composants peut spécifier les besoins de chaque composant et exposer sur le composant de plus haut niveau les contraintes de ressources qu'il jugera nécessaires pour tout déploiement (cf. notre discussion sur la composition des contraintes de ressources et de localisation). En ce qui concerne les contraintes de déploiement, un COTS peut être conçu avec des restrictions de placement sans que jamais l'identité d'une machine ne soit précisée. Le composant de plus haut niveau fera alors apparaître d'une part une liste de variables correspondant au nombre de machines impliqués par le déploiement, mais aussi les contraintes de localisation sur tout ou partie de ces variables.

Si nous prenons l'exemple de l'application DiapoMaker, nous pouvons imaginer que cette application ait été conçue à l'aide d'un composant COTS, DocumentSearch. En effet, les contraintes de ressources et de localisation de ce composant (listing 6.4) ne référencent l'adresse d'aucune machine laissant ainsi l'assembleur du composant composite DiapoMaker le choix des machines cibles. Au moment du déploiement, le respect des contraintes de placement du composant DocumentSearch sera garanti.

6.4 Synthèse

Nous avons présenté dans ce chapitre CDL, un langage permettant de spécifier et de contrôler le déploiement d'une application à base de composants dans des réseaux dynamiques qui constituent les cibles de déploiement des applications ubiquitaires. En précisant d'une part les besoins des composants, à l'aide des contraintes de ressources, et d'autre part en contrôlant le choix des machines censées les héberger à travers les contraintes de localisation, il n'est plus nécessaire de désigner une machine par son nom ou son adresse IP, ce que nous avions défini comme étant un prérequis pour les réseaux dynamiques.

Nous nous sommes attachés à définir un ensemble minimal de ressources et d'opérateurs pour la description des exigences des composants et la définition des contraintes. Cet ensemble peut facilement être étendu. Par exemple, nous avons proposés dans [THK07] la prise en compte des styles architecturaux dans CDL. Il est ainsi possible de définir des contraintes sur le nombre d'instances d'un composant ainsi que sur le nombre de leurs liaisons.

CDL est un langage purement déclaratif et par conséquent doit être utilisé conjointement avec un outil de déploiement qui a pour rôle de sélectionner une machine hôte pour chaque composant de l'application. La machine hôte d'un composant doit vérifier toutes les contraintes de ressources associées à ce dernier mais ceci ne suffit pas pour pouvoir héberger ce composant, les contraintes de localisation ne doivent pas être violées. Les chapitres suivants vont présenter la mise en place d'un outil de déploiement respectueux des contraintes de ressources et de localisation.

The big bang, the most cataclysmic event we can imagine, on closer inspection appears finely orchestrated.

7

Programmation par contraintes pour le déploiement autonome de composants

Le déploiement de logiciels dans les réseaux dynamiques pose le problème de la spécification du placement des composants dans des phases antérieures au déploiement. Comme cela a été mis en avant dans le chapitre précédent, il est en effet difficile de connaître a priori les machines qui seront présentes au moment du déploiement ainsi que l'ensemble des ressources qui seront disponibles. Le langage CDL que nous avons présenté permet seulement la spécification du déploiement de composants logiciels dans les réseaux dynamiques à travers les contraintes de ressources et de localisation.

Nous proposons dans ce chapitre la modélisation de la description des contraintes faites via CDL par des *problèmes de satisfaction de contraintes*, selon le paradigme de la programmation par contraintes. Notre approche permet la résolution incrémentale et non centralisée des contraintes de déploiement.

Sommaire

7.1 Vision générale de notre approche

Il est possible avec CDL de définir les besoins en ressources logicielles et matérielles des composants. Les besoins d'un composant expriment les ressources qu'une machine devra obligatoirement offrir pour pouvoir héberger ce composant. Par ailleurs, à l'aide de contraintes de localisation, il est possible avec CDL de spécifier comment contrôler le placement des composants.

Comme nous l'avons dit, CDL est un langage purement déclaratif. Il est donc nécessaire d'utiliser conjointement à CDL un outil de déploiement qui prendra en charge la résolution des contraintes de déploiement, contraintes définies dans un descripteur de déploiement. Bien qu'il soit possible pendant le déploiement et l'exécution des composants de vérifier si l'architecture déployée est conforme aux contraintes de déploiement, l'utilisation directe de la description de ces contraintes n'est pas adaptée à la définition de réaction vis-à-vis des changements qui peuvent intervenir au sein de l'environnement (e.g. l'arrivée d'une nouvelle machine, la disponibilité de nouvelles ressources, etc.). En effet, dès lors qu'un changement intervient, l'ensemble des contraintes listées dans le descripteur de déploiement doit être réévalué afin de vérifier que les ressources demandées par l'application sont toujours disponibles. Selon la complexité de l'application (le nombre de composants et de contraintes de déploiement mis en jeu), cette approche peut se révéler extrêmement coûteuse. La réaction aux différents changements est nécessaire car ces derniers peuvent entraîner une configuration de l'application qui n'est pas conforme aux contraintes de déploiement.

Notre approche consiste à réifier les contraintes de déploiement décrite via CDL afin de les manipuler pendant l'exécution de l'application. Ces contraintes que nous qualifions de « dynamiques » permettent de refléter l'état des contraintes architecturales et de déploiement pendant toute l'exécution de l'application. Conjointement à ces contraintes dynamiques, nous définissons des *résolveurs* qui prennent en charge la résolution de ces contraintes. Ces résolveurs rendent possible la détection des différents changements qui ont lieu dans les environnements dynamiques ainsi que la définition de réactions à ces changements. Ainsi, l'indisponibilité d'un composant provoque une incohérence architecturale qui nécessite par exemple soit la mise en place d'une nouvelle liaison vers un composant fournissant le même service, soit l'instanciation du composant indisponible et la mise en place d'une nouvelle liaison vers ce dernier.

Les contraintes dynamiques que nous introduisons dans ce chapitre sont générées automatiquement à partir de celles définies via CDL. Les contraintes dynamiques forment un problème de satisfaction de contraintes (CSP, pour *Constraint Satisfaction Problems* en anglais). La résolution de ce CSP par un résolveur permet le déploiement et la reconfiguration autonome de l'application du fait des changements dus aux fluctuations des ressources, à la mobilité et la volatilité des équipements.

Les sections suivantes présentent la formalisation des contraintes dynamiques ainsi qu'un ensemble de résolveurs permettant la résolution de ces dernières.

7.2 Programmation par contrainte

La programmation par contraintes est un paradigme de programmation dans lequel on se limite à la génération d'un ensemble de besoins (les contraintes), leur résolution étant

prise en charge automatiquement par les solveurs. Nous allons nous placer dans le cadre des problèmes de satisfaction de contraintes ou CSP. Ce dernier a pour but l'expression et la résolution de problèmes faisant intervenir des contraintes. Le cadre CSP offre un ensemble de caractéristiques intéressantes :

- les programmes par contraintes sont extrêmement déclaratifs : il suffit de déclarer les contraintes, donc de modéliser le problème ;
- de nombreux algorithmes existent pour résoudre une grande variété de problèmes.

Dans le cadre de cette thèse, nous avons modélisé le problème du déploiement de composants, i.e. l'affectation d'une machines cible à chaque composant de l'architecture, sous la forme d'un CSP. La résolution des contraintes de ressources et de localisation sera présentée dans le chapitre suivant. Nous présentons dans les deux sections suivantes la formalisation d'un problème de satisfaction de contraintes, ce qui nous permettra de comprendre comment ont été modélisées les contraintes de déploiement et celles sur l'architecture de l'application. Dans un second temps, nous allons mettre en avant comment la résolution de ces contraintes peut prendre en compte le caractère dynamique des environnements que nous avons considérés.

7.2.1 Les problèmes de satisfaction de contraintes

Considérons une séquence de variables $X = x_1, \ldots, x_n$ avec $n \geq 0$ associées aux domaines respectifs $D_1, \ldots, D_n$. Ainsi chaque variable x_i s'étend sur le domaine D_i.

Définition 4 (Contrainte)
Une contrainte sur X est un sous-ensemble du produit cartésien $=D_1 \times \ldots \times D_n$

Définition 5 (Problème de satisfaction de contraintes (CSP))
Un problème de satisfaction de contrainte (CSP) est donné par une séquence finie de contraintes $\mathcal{C} = \{c_1, \ldots, c_n\}$*, chaque* c_i *portant sur une sous-séquence de X et l'ensemble des domaines* $\{D_1, \ldots, D_n\}$.

Nous notons un tel CSP $\langle \mathcal{X}; x_1 \in D_1, \ldots, x_n \in D_n \rangle$ *ou encore* $\langle \mathcal{C}, \mathcal{D} \rangle$*, avec* $\mathcal{D} = D_1, \ldots, D_n$.

Les domaines des variables sont des éléments d'une structure algébrique appelée *ordre partiel conjonctif* qui est défini de la manière suivante :

Définition 6 (Ordre partiel conjonctif)
Un ordre partiel $\langle D, \sqsubseteq \rangle$ *est un* $\sqcup -$ *po (ordre partiel conjonctif) s'il vérifie :*

- $\langle D, \sqsubseteq \rangle$ *possède un plus petit élément* $e \in D$ *;*
- *pour toute suite croissante* $d_0, \sqsubseteq d_1, \sqsubseteq d_2 \ldots$ *d'éléments de D, le plus petit majorant* $\sqcup_{i \geq 0} \geq d_i$ *existe ;*
- *et pour tout* $a, b \in D$ *le plus petit majorant de* $a \sqcup b$ *existe.*

7.2.2 Résolveurs de contraintes

Pour un CSP donné, nous pouvons définir de manière générale un solveur comme étant un algorithme qui répond au problème de la *satisfaction*. Si nous reprenons les notations adoptées précédemment : soit un ensemble de contraintes $\mathcal{C} = \{c_1, \ldots, c_m\}$ portant sur $D = D_1, \ldots, D_n$. Un solveur de contrainte est caractérisé de la manière suivante :

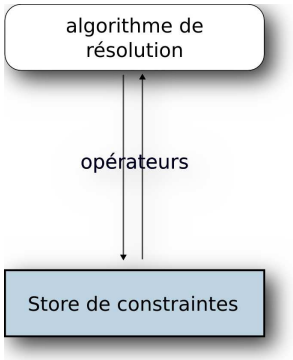


FIG. 7.1: Architecture générale d'un résolveur de contraintes

Le but d'un résolveur de contraintes est d'isoler les n-uplets de D satisfaisant la conjonction $c_1 \wedge \cdots \wedge c_m$.

Un solveur est généralement basé sur la notion de *store* de contraintes. Le store est une sorte de zone mémoire dans laquelle le solveur ajoute les contraintes et les variables associées (et donc leur domaine). Ainsi le solveur résout un ensemble de contraintes en fonction des informations présentes dans le store à un moment donné. Le store est qualifié de *monotone* lorsqu'on ne peut qu'y ajouter de l'information. Si au cours du temps au contraire, de l'information peut être retirée, le store sera qualifié de *non-monotone*. Dans le cadre de nos travaux, nous nous limitons aux stores monotone.

L'interaction avec le store se fait par l'intermédiaire d'opérateurs. Les opérateurs que nous avons utilisé sont les opérateurs TELL et ASK qui permettent respectivement d'ajouter des contraintes dans le store et de déduire si une contrainte (ou un fait) peut être déduite des informations présentes dans le store.

7.3 Résolution dynamique des contraintes architecturales et de déploiement.

Une configuration valide de placement de l'application, ne peut être calculée qu'au moment de son déploiement. Par configuration valide, nous entendons un ensemble d'instances de composants, interconnectées et pour laquelle, une machine cible a été choisie pour chacune de ces instances. Toutes les contraintes architecturales (c'est-à-dire la spécification de l'assemblage des composants) doivent être vérifiées et chaque machine hôte sélectionnée devra d'une part fournir les ressources exigées par chaque composant, spécifiées via les contraintes de ressources, et d'autre part à ne pas contredire les contraintes de localisation.

Notre approche consiste à manipuler toutes les contraintes architecturales, de ressource et de localisation à l'exécution, afin de refléter l'état du système déployé qui devra respecter ces contraintes. Nous présentons dans cette section comment ces contraintes sont transformées en CSP et comment ces CSP sont résolus.

7.3.1 Résolution des contraintes de déploiement

Les contraintes de ressources spécifiées via CDL définissent les besoins des composants en terme de ressources logicielles et matérielles. Du fait du caractère dynamique des plates-formes visées, les ressources exigées par un composant fluctuent de telle sorte qu'une ressource requise, alors non disponible au moment du déploiement, le devienne plus tard. De plus, du fait de la mobilité des équipements, l'arrivée d'une machine, jusqu'alors inaccessible (ou tout simplement mise en veille) peut provoquer l'apparition de nouvelles ressources utiles pour le déploiement de certains composants. Il est donc nécessaire de réagir à ces changements afin de proposer aux composants les ressources dont ils ont besoin. Pour cela, nous réifions les contraintes de ressources décrites dans le descripteur de déploiement sous la forme d'un CSP, noté $CSP(ResCst)$ constitué des contraintes $ResCst$.

Pour un composant C auquel est associée une contrainte de ressources décrite dans un descripteur de déploiement via CDL, la contrainte $C_{ResCst} = \{x \in D_x\}$ est définie — au sens de la programmation par contrainte. La variable x correspond à la ressource requise par le composant et s'étend sur D.

Par exemple, on associera à un composant requérant 100 Mo de mémoire vive la contrainte $C_{ram} = \{ram \in [\![100, \infty[\![\}$.

Condition pour l'hébergement d'un composant Soit C un composant et $\{C_{ResCst_i} = \{x_i \in D_{x_i}\}\}_{i \in \mathbb{N}_n}$ les $n+1$ contraintes de ressources associées à C. Une machine m pourra héberger le composant C si elle vérifie l'ensemble des contraintes $\{C_{ResCst_i}\}$, c'est-à-dire :

$$m(x_i) \text{ satisfait la conjonction } \bigwedge_{i \in \mathbb{N}_n} C_{ResCst_i} \tag{7.1}$$

où $m(x_i)$ représente l'évaluation de la variable x_i sur la machine m.

La condition 7.1 impose que tous les besoins exprimés par un composant soient offerts par une machine pour que cette dernière puisse héberger ce composant. Nous pouvons donc construire un résolveur pour les contraintes de ressources. Ce résolveur s'attache à récupérer l'état de chaque ressource exprimée par une contrainte C_{ResCst_i}. La disponibilité de la ressource exhibe une solution vis-à-vis de $CSP(ResCst)$. Nous noterons $Solveur(ResCst)$ un tel résolveur.

Le fait qu'une machine vérifie l'ensemble des contraintes de ressources n'est pas une condition suffisante pour que cette dernière puisse héberger un composant. En effet, il peut exister des contraintes de localisation associées aux composants. Ces contraintes peuvent conduire à écarter cette machine. Comme pour les contraintes de ressources, les contraintes de localisation sont transformées pour constituer le problème de satisfaction de contraintes $CSP(LocCst)$,

constitué des contraintes *LocCst*.

Pour un composant C auquel sont associées des contraintes de localisation décrites dans un descripteur de déploiement via CDL, les contraintes suivantes sont définies :

1. une contrainte $C_{LocCst} = \{hostname \in Target\}$. La variable *hostname* correspond aux identités des machines *pouvant* héberger C et s'étend sur *Target* ;
2. zéro ou plusieurs contraintes *LocCst* sur les domaines de la variable *hostname* des sous-composants de C, si C est un composant composite ;
3. zéro ou plusieurs contraintes de ressources associées aux sous-composants si C est un composant composite.

Le domaine de définition d'une variable *hostname* indique les machines pouvant héberger le composant, i.e. celles vérifiant les contraintes de ressources.

Comme nous l'avons vu dans le chapitre précédent, il est possible avec CDL de contrôler le placement des composants en permettant la sélection d'une machine suivant les ressources dont elles disposent. Par exemple choisir entre deux machines celle qui possède le plus grand espace de mémoire disponible. Les contraintes correspondant au troisième point réifient l'expression de ces exigences.

Par exemple, le composant DocumentSearch, qui doit être déployé sur les machines m_1, m_2 et m_3, impose que ses sous-composants DocumentBuffer et DocumentFinder résident sur deux machines distinctes. Le CSP correspondant à cette spécification du déploiement est le suivant :

$$
\begin{aligned}
DocumentSearch_{LocCst} &= \{DocumentSearch_{hostname} \in \{m_1, m_2, m_3\}\} \\
DocumentFinder_{LocCst} &= \{x \in \{m_1, m_2, m_3\}\} \\
DocumentBuffer_{LocCst} &= \{y \in \{m_1, m_2, m_3\}\} \\
\texttt{alldiff}(x, y) &= \{(x, y) | x, y \in \{m_1, m_2, m_3\}, x \neq y\}
\end{aligned}
$$

avec

$$
\begin{aligned}
x &= DocumentFinder_{hostname} \\
y &= DocumentBuffer_{hostname}
\end{aligned}
$$

Résolution des contraintes de localisation La recherche des solutions pour un $CSP(LocCst)$ détermine, lorsque cela est possible, une affectation pour chaque variable $p_{hostname}$ correspondant au composant primitif p. Ainsi, une solution d'un $CSP(LocCst)$ spécifie le placement de tous les composants impliqués par ce CSP. Les solutions pour le CSP explicité dans l'exemple précédent, par exemple $(x, y) = (m_1, m_3)$ ou $(x, y) = (m_2, m_1)$, peuvent être calculées du fait du nombre suffisant de valeurs possibles pour les variables x et y. La contrainte `alldiff` admet en effet une solution si et seulement si pour tout sous-ensemble K composé des variables passées en paramètre, la cardinalité de l'union de leur domaine est toujours plus grande que celle de K [vH01]. Dans notre situation de déploiement de composants, nous pouvons donc déclencher la résolution d'une contrainte `alldiff` dès lors que le nombre de machines pouvant héberger un composant est suffisant.

Comme nous l'avons précisé plus haut, une contrainte $C_{LocCst} = \{hostname \in Target\}$ spécifie les machines pouvant héberger le composant C. Ainsi, au moment du déploiement, *Target* peut être égale à l'ensemble vide. Au fur et à mesure de l'apparition de nouvelles machines et de l'apparition de nouvelles ressources, il est alors possible d'ajouter au domaine de définition de la variable *hostname* l'identité des nouveaux « candidats » à l'hébergement de C. Dans les réseaux dynamiques, il est alors nécessaire d'alimenter régulièrement le store de contraintes afin que le processus de résolution puisse dégager une solution au problème $CSP(LocCst)$.

Reconfiguration autonome et contraintes de déploiement Après que les composants d'une application aient été instanciés, il est parfois nécessaire de reconfigurer l'architecture déployée par exemple en créant de nouveaux composants ou en reconfigurant des liaisons (chapitre 3). La reconfiguration des applications comprend également la définition de nouvelles machines hôtes dès lors que le placement actuel des composants ne satisfait plus les contraintes de déploiement. Par exemple, les ressources exigées par un composants peuvent ne plus être disponibles sur la machine qui l'héberge. Dès lors, le placement du composant sur cette machine contredit les contraintes de ressources correspondantes. Il est nécessaire dans ce cas d'attribuer une nouvelle machine au composant, si une telle machine est disponible. La sélection d'une machine dépend par ailleurs des contraintes de ressources exigées par les composants.

Lorsque le nombre de composants devient important et que les ressources disponibles dans l'environnement fluctuent, il devient primordial de mettre en place des solutions permettant de rendre autonomes les reconfigurations liées au placement des composants. En suivant l'approche de la programmation par contraintes, ces reconfigurations autonomes peuvent être facilement réalisées du fait des mécanismes de résolution et de l'aspect déclaratif offerts par la programmation par contraintes. En effet, contrairement à une approche basée sur des règles de type « Événement-Condition-Action » dans laquelle la mise en place des reconfigurations nécessite d'expliciter d'une part les événements à l'origine d'une reconfiguration et d'autre part les actions à effectuer suite aux changements s'étant produits, notre proposition permet à partir de la description souhaitée du placement des composants de déduire automatiquement les actions à réaliser. Ainsi, le résolveur de contraintes $Solveur(ResCst)$ est utilisé pour détecter la violation des contraintes de ressources. Lorsque cela se produit, le résolveur prenant en charge la résolution du placement des composants $CSP(LocCst)$ est sollicité pour attribuer une nouvelle machine cible aux composants concernés.

Nous décrivons plus en détail la mise en place de ce « redéploiement » autonome des composants dans le chapitre 8. Les reconfigurations sont guidées par la spécification de l'assemblage des composants. Nous verrons dans la section suivante comment les reconfigurations autonomes intègrent ces aspects architecturaux.

7.3.2 Résolution des contraintes architecturales

Les contraintes *ResCst* et *LocCst* présentées précédemment permettent une réaction vis-à-vis des différents événements qui peuvent intervenir dans les environnements considérés dans le cadre de cette thèse. La résolution de ces contraintes a pour but d'une part de dégager les machines pouvant héberger un composant donné ($CSP(ResCst)$) et d'autre part à sélectionner parmi ces machines « candidates » celles qui respecteront les contraintes de localisation

($CSP(LocCst)$). La résolution de ces seules contraintes ne permet pas le déploiement d'une architecture de composants, i.e. l'instanciation de ces derniers et la mise en place de leurs liaisons. La spécification de l'assemblage des composants définit un ensemble de contraintes sur l'architecture de l'application en termes d'instances de composants et de leurs liaisons. Ces contraintes sont désignées par le terme contraintes architecturales. Nous présentons maintenant la réification de contraintes.

7.3.2.1 Instanciation des composants

La résolution des contraintes de déploiement (*ResCst* et *LocCst*) est conditionnée par les instances de composants à créer. Une machine sélectionnée pour l'hébergement d'un composant C ne pourra effectivement en créer une instance si et seulement si le composant C n'a pas déjà été instancié : à chaque définition d'un composant primitif dans le descripteur d'architecture doit correspondre une seule instance pour l'ensemble des machines du réseau.

La modélisation de cette contrainte architecturale se ramène dans un CSP à un problème de somme sur le nombre d'instance de chaque composant. Nous réifions donc ces contraintes de la manière suivante :

Pour chaque composant du descripteur d'architecture sont définies les contraintes :

$$\begin{cases} C_{instance} = 1 \\ C_{m_i} = 0 \; ou \; 1 \text{ pour chaque } m_i \in Candidats \\ \sum_{m_i \in Candidats} C_{m_i} = 1 \end{cases}$$

avec C_{m_i} la variable représentant le nombre d'instances de C sur la machine m_i . Ici on veut qu'à chaque composant défini dans le descripteur d'architecture corresponde une seule instance pour l'ensemble des machines du réseau. L'ensemble *Candidats* représente l'ensemble des machines pouvant héberger C, i.e. celles vérifiant les contraintes de ressources.
Ces contraintes constituent le problème de satisfaction de contraintes $CSP(Instances)$.

Ainsi, la contrainte $C_{m_i} = 1$ signifie qu'une instance de C est hébergée sur la machine m_i.

La résolution de $CSP(Instances)$ consiste à maximiser le nombre[26] d'instances de chaque composant afin de garantir la disponibilité et la continuité de services. L'information précisant qu'un composant a été instancié ou non ($C_{instance} = 0$ ou 1) doit être connue. Nous dénoterons $Solveur(Instances)$ le résolveur en charge de la résolution de $CSP(Instances)$.

Du fait de la présence des îlots au sein des réseaux dynamiques, l'information sur le nombre d'instances actuellement déployées sur chaque machine, manipulée par le résolveur $Solveur(Instances)$, est une information cruciale qui doit être maintenue cohérente dans chacun des îlots. En effet, lors d'une panne d'un composant, la stratégie consistant à maximiser le nombre d'instances d'un composant dès lors que ce dernier subit une panne, n'est pas possible si chaque îlot a sa propre « vision » du nombre d'instances total. Ceci aurait pour conséquence

[26]Nous avons proposé dans [THK07] un ensemble de contraintes permettant de modéliser les styles architecturaux dans lesquels la définition d'un composant ne se limite plus qu'à une seule instance à l'exécution. Dans ce cas, le nombre d'instance d'un composant peut être borné.

l'instanciation du même composant dans plusieurs îlots ; la configuration de l'assemblage de l'application ne serait alors plus conforme à sa spécification : plusieurs instances existeraient pour une seule définition du composant dans le descripteur d'architecture.

7.3.2.2 Mise en place des liaisons entre composants

Lorsqu'une instance de composant est créée ou enlevée, l'architecture de l'application, c'est-à-dire l'assemblage des composants, doit être configuré : en effet, la création d'une instance implique la mise en place de liaisons vers ce composant si ce dernier fournit un ou plusieurs services, ainsi que vers les composants dont dépend le nouveau composant. À l'inverse, le retrait d'un composant impose la suppression des liaisons vers ce dernier. L'ajout ainsi que la suppression de liaisons sur l'architecture doit être faite en conformité avec les contraintes architecturales définies dans le descripteur d'architecture.

Les contraintes *BindingCst* sont la réification de la spécification d'une liaison entre deux composants ou deux types de composants.

L'ensemble des contraintes $BindingCst = \{\texttt{binding}(C_{cltItf}, S_{serverItf})\}$ précisant une liaison entre l'interface cliente $cltItf$ du composant C et l'interface serveur $serverItf$ du composant S constituent le $CSP(BindingCst)$.

Reconfiguration autonome et contraintes architecturales La résolution des contraintes architecturales est couplée à celle des contraintes de déploiement : l'instanciation d'un composant est en effet envisageable dès lors qu'il existe une machine qui possède les ressources nécessaires à son exécution. Ainsi, la définition de plans de réparation servant à rendre l'architecture de l'application conforme à sa spécification doit prendre en compte les différents événements à l'origine des reconfigurations. Dans un réseau dynamique, on se heurte à un problème de combinatoire lorsqu'il s'agit de définir les actions à réaliser en face de telle ou telle situation.

Les résolveurs $Solveur(Instances)$ et $Solveur(Binding)$, à partir d'une description de l'assemblage de l'application, permettent d'automatiser les reconfigurations des composants et de leurs liaisons sans qu'il n'y ait besoin d'expliciter les actions à réaliser. Celles-ci sont déduites par les résolveurs pour qui l'architecture spécifiée est vue comme un but à atteindre. Ainsi, lorsqu'une nouvelle ressource requise par un composant devient disponible, l'ensemble *Candidats* des machines pouvant héberger ce composant augmente, ce qui permet de trouver une solution au problème $CSP(Instances)$. De la même manière, lorsqu'un composant subit une panne, il est possible de considérer son redéploiement : le nombre d'instance étant repassé à 0, il est permis de créer une nouvelle instance de ce composant.

En couplant le résolveur $Solveur(Instances)$ aux résolveurs sur les contraintes de déploiement ($Solveur(ResCst)$ et $Solveur(LocCst)$) il est alors possible d'instancier automatiquement les composants en fonction de l'évolution des ressources présentes sur le réseau.

De la même manière, lorsque deux composants distants ne sont plus accessibles l'un de l'autre, la mise en place de la liaison entre ces composants se fait automatiquement dès lors que la connectivité entre les composants est détectée : une simple unification sur les noms des composants et de leurs interfaces permet d'identifier les clauses du $CSP(Binding)$ correspondantes aux liaisons à mettre en place.

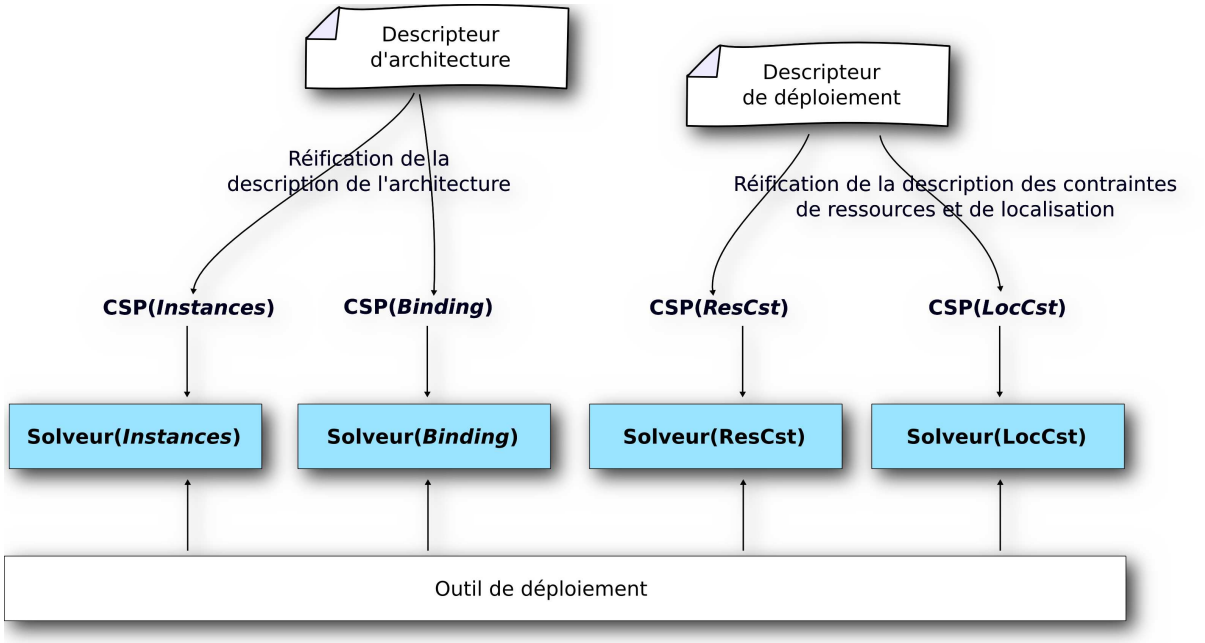


FIG. 7.2: Des descripteurs d'architecture et de déploiement aux résolutions des contraintes architecturales, de ressources et de localisation

7.4 Discussion

Les contraintes définies via CDL permettent la spécification de déploiement de composants dans des environnements dynamiques. Les outils de déploiement utilisant CDL doivent implanter les fonctions nécessaires garantissant les contraintes architecturales, les contraintes de ressources et de localisation. Il est en effet nécessaire que les choix concernant l'architecture de l'application mais aussi ceux définissant les besoins des composants et le contrôle de leur placement soient garantis tout au long de l'exécution de l'application.

Nous avons présenté dans ce chapitre la réification des contraintes de déploiement définies via CDL en vue, d'une part, de leur résolution permettant ainsi l'affectation d'une machine pour l'hébergement de chaque composant, et d'autre part, de maintenir les contraintes architecturales et de déploiement cohérentes après la phase de déploiement. La réification des contraintes consiste à générer un ensemble de problèmes de satisfaction de contraintes (CSP) à partir des descripteurs d'architecture et de déploiement, comme illustré sur la figure 7.2.

Nous avons mis en avant plusieurs résolveurs pour ces CSP. L'intérêt de notre approche réside dans la possibilité de réagir aux différents changements pouvant avoir lieu au sein de l'environnement en identifiant précisément les événements à l'origine de ces changements.

L'ensemble des CSP définis dans ce chapitre ($CSP(ResCst, LocCst, Instances, Binding)$) constitue le problème du déploiement des applications vis-à-vis de contraintes architecturales, de ressources et de localisation. La résolution de ces problèmes permet la génération d'une configuration valide de l'application. Le processus de déploiement que nous présentons dans le chapitre suivant s'appuie sur ces contraintes et ces résolveurs pour construire la correspondance entre les instances de composants et les machines définissant la plate-forme cible. En coordonnant l'activité de ces résolveurs, il est possible de rendre la reconfiguration

Solveur	Évaluation locale uniquement	Contrainte globale
$Solveur(ResCst)$	×	
$Solveur(LocCst)$		×
$Solveur(Instances)$		×
$Solveur(Binding)$	×	

TAB. 7.1: Caractéristiques des différents résolveurs permettant la résolution des contraintes architecturales, de ressources et de déploiement.

de l'application (redéfinition des machines cibles, création des instances manquantes, mise en place des liaisons) complètement autonome.

Par ailleurs, en découpant le problème du placement des composants en quatre sous problèmes, nous allons pouvoir plus facilement mettre en place des mécanismes de résolution distribués. Cette approche est en effet plus pertinente lorsque la plate-forme de déploiement est un réseau dynamique.

Le tableau 7.1 récapitule les résolveurs de contraintes mis en jeu dans la résolution des contraintes architectures, de ressources et de localisation. Suivant les variables mises en jeu par ces résolveurs, il est nécessaire d'alimenter leur store de contraintes avec des informations qui proviennent soit de la machine où se trouve le résolveur, soit des autres machines. Par exemple le $Solveur(ResCst)$ a besoin de connaître l'état des ressources sur une machine donnée pour savoir si cette dernière peut héberger un composant. Dans l'optique de distribuer le processus de résolution des contraintes, nous sommes amener à déterminer quels sont les résolveurs qui peuvent être exécutées par une ou plusieurs machines du réseau.

Nous voyons qu'il est facile de placer le résolveur $Solveur(ResCst)$ sur chaque machine du réseau : le processus de résolution étant complètement indépendant d'une machine à une autre. Pour les résolveurs $Solveur(LocCst)$ et $Solveur(Instances)$, il est plus difficile de les distribuer étant donné qu'ils mettent en jeu tous deux des contraintes globales, i.e. des contraintes qui manipulent des informations qu'il faut maintenir cohérentes. Ainsi deux résolveurs $Solveur(Instances)$ situés sur deux machines distinctes doivent se synchroniser afin de ne pas introduire d'instances supplémentaires dans l'architecture à l'exécution. Une organisation distribuée des différents solveurs permettant d'assurer une prise en compte cohérente des contraintes sera présentée dans le chapitre suivant.

8

Déploiement autonome de composants ubiquitaires dans des réseaux dynamiques

Sommaire

8.1 Aperçu général du processus de déploiement

8.1.1 Déploiement propagatif

Lorsque les descripteurs d'architecture et de déploiement ont été définis, la phase de déploiement que nous considérons dans ce travail de thèse consiste à choisir une machine cible

pour chaque composant de l'architecture. Cette sélection doit être faite en fonction du contexte de déploiement associé à chaque composant : la machine sélectionnée doit vérifier l'ensemble des contraintes de ressources et ne doit pas contredire les contraintes de localisation. Selon les ressources disponibles sur les différentes machines du réseau, plusieurs machines peuvent être choisies pour l'hébergement d'un composant : pour un composant primitif, une seule machine doit être sélectionnée alors que pour un composant composite, étant donné notre mode de distribution, plusieurs machines peuvent être choisies. Il est nécessaire de contrôler l'affectation des machines pour chaque composant. En effet, nous devons éviter, au sein d'un réseau dynamique dans lequel des îlots peuvent apparaître, que des décisions incohérentes soient prises. Par exemple, nous devons garantir que deux machines appartenant à deux îlots distincts ne sont pas sélectionnées pour l'hébergement d'un même composant.

En raison de la dynamicité qui caractérise les réseaux sur lesquels nos applications doivent être déployées, il n'est pas envisageable de définir un déploiement nécessitant une connectivité permanente des différentes machines composant le réseau. En effet, nous souhaitons disposer d'un déploiement qui instancierait les composants constituant l'application au fur et à mesure de la disponibilité des ressources qu'ils requièrent. Ainsi, les fonctionnalités de l'application pourront être mises à disposition même si les machines nécessaires au déploiement des composants qui ne sont pas encore installés ne sont pas disponibles. Dès lors que ces machines seront connectées, le déploiement se poursuivra. De plus, la progression du déploiement n'est pas seulement garantie par l'accessibilité d'une nouvelle machine mais aussi par les changements de l'état des ressources disponibles sur chaque machine. Nous qualifions ce déploiement de propagatif, l'arrivée de nouvelles ressources faisant avancer le déploiement.

8.1.2 Déploiement autonome

Cependant, étant donné la dynamicité du réseau et dès lors que des composants sont installés, les ressources requises par ces mêmes composants peuvent devenir insuffisantes. Un redéploiement doit avoir lieu. Le déploiement autonome consiste donc à remettre en question les choix de placement du déploiement propagatif afin de prendre en compte l'indisponibilité de ressources qui peut survenir.

La difficulté majeure d'un tel déploiement dans un réseau dynamique est de garantir l'unicité des instanciations, i.e. que le processus de décision de placement des composants affecte à chaque composant défini par le descripteur d'architecture une et une seule machine hôte. En effet, d'une part, puisqu'il n'est pas possible de prévoir quelles sont les machines qui seront connectées à tout moment, nous ne pouvons pas simplement en sélectionner une en particulier qui serait responsable des décisions d'instanciation pour l'ensemble de l'application ; d'autre part, si nous laissons plusieurs machines prendre des décisions d'instanciation, il faut pouvoir garantir que, au sein de deux îlots distincts, deux décisions contradictoires ne soient pas prises.

Nous présentons maintenant le déploiement autonome en deux temps. Tout d'abord nous détaillons le déploiement propagatif, puis nous présentons les mécanismes permettant de rendre autonome ce déploiement.

8.2 Déploiement propagatif sur réseau connexe

Le déploiement propagatif permet de mettre à disposition de l'utilisateur les fonctionnalités d'une application à base de composants au fur et à mesure que les ressources nécessaires à chacun des composants deviennent disponibles. Dans un souci de pédagogie, nous allons dans cette section poser des hypothèses simplificatrices pour présenter le processus régissant le déploiement propagatif. Ces hypothèses seront ensuite écartées dans la section 8.3.

Nous allons dans un premier temps supposer un réseau non partitionné dans lequel les communications entre les machines reposent sur un modèle synchrone.

Nous supposons également l'existence d'une machine stable que nous désignons dans la suite par le terme gestionnaire de déploiement (ou DeployManager). Dans cette section, ce gestionnaire est considéré unique. Une version distribuée du gestionnaire de déploiement est considérée dans la section 8.3. C'est depuis le DeployManager qu'est initié le déploiement. Finalement, nous considérerons dans toute cette section que les ressources du réseau ne subissent que des variations *positives*, i.e. que des ressources nouvelles peuvent apparaître, mais une fois qu'une ressource est présente, elle est toujours disponible.

8.2.1 Principe général

Le gestionnaire de déploiement (DeployManager) a pour rôle de maintenir à jour la liste des identités des autres machines qui sont connectées. Lorsque le déploiement est lancé (depuis le DeployManager), toutes les machines concernées par le déploiement sont accessibles et les ressources demandées par les composants sont disponibles sur les différentes machines du réseau (i.e. que pour chaque composant, il existe au moins une machine qui dispose des ressources demandées).

La phase initiale du déploiement consiste à attribuer au DeployManager les deux descripteurs concernant l'application à déployer. Le premier descripteur est le descripteur d'architecture qui contient la configuration de l'application en terme d'assemblage de composants. Le deuxième descripteur est le descripteur de déploiement qui liste pour chaque composant ses contraintes de ressources et de localisation. Chaque application est définie par un composant composite englobant tous les autres. La spécification de la plate-forme de déploiement (cf. chapitre 5, section 6.3.1) est réalisée en définissant pour ce composant composite la liste des machines concernées par le déploiement : il suffit de préciser l'identité des machines ou des réseaux concernés par un ou plusieurs éléments `target`.

Le premier rôle du DeployManager est de diffuser à toutes les machines connectées le descripteur d'architecture et le descripteur de déploiement. À la réception de ces descripteurs, chaque machine essaiera de faire avancer le déploiement en instanciant les composants définis dans le descripteur d'architecture. L'instanciation des composants est guidée par les contraintes de ressources et de localisation définis dans le descripteur de déploiement : une machine pourra effectivement héberger un composant si et seulement si toutes les contraintes de ressources associées au composant sont vérifiées et que le choix de cette machine n'est pas en contradiction avec les contraintes de localisation.

Nous avons choisi de déléguer à chaque machine la résolution des contraintes de ressources plutôt que de la confier au DeployManager. Ainsi chaque machine est capable de décider si elle peut ou non héberger un composant. Ce choix permet essentiellement de limiter le trafic réseau.

Le processus de déploiement met en jeu un ensemble d'activités qui sont réparties sur l'ensemble des machines participant au déploiement. Pour chaque machine m_i ayant reçu les descripteurs d'architecture et de déploiement nous détaillons ces activités pour le déploiement du composant composite C de plus haut niveau pour lequel l'ensemble des machines cibles est parfaitement identifié. Nous rappelons que C est un composant ubiquitaire et par conséquent, plusieurs machines cibles peuvent être définies pour l'hébergement de C.

8.2.1.1 Instanciation du composant composite racine

La première étape consiste à vérifier si m_i peut héberger C. Cette étape est nécessaire dès lors que des contraintes de ressources sont attachées à C. En effet, m_i doit vérifier que toutes les ressources exigées par le composant C sont disponibles localement. Si c'est le cas, i.e. le résolveur $Solveur(ResCst)$ retourne vrai, C peut être instancié localement, sinon c'est un composant mandataire qui doit être créé :

- toute machine m_i satisfaisant les contraintes de ressources pour le composant composite ubiquitaire C instancie la membrane de C localement et informe le DeployManager de cette instanciation.
- toute machine m_i ne possédant pas les ressources demandées par C, crée un composant mandataire pour C et envoie une requête au DeployManager pour connaître l'identité d'une machine ayant pu instancier la membrane de C. Lors de la réception de la réponse du DeployManager, m_i met à jour la référence distante nécessaire au composant mandataire.

Nous voyons ici que le DeployManager centralise les informations de placement des membranes de composants. Cette information est utilisée par les machines ne pouvant héberger le composant C afin de connaître l'identité de celles ayant instancié (la membrane de) C. Ainsi les liaisons distantes vers ces machines peuvent être mises en place.

Pour une machine m_i ayant instanciée un composant mandataire pour le composant composite C, le processus de déploiement est terminé. Pour les autres sur lesquels la membrane de C a été instancié, le processus de déploiement continue récursivement pour les sous-composants.

8.2.1.2 Instanciation des sous-composants

Le déploiement des sous-composants continue sur chaque machine m_i où la membrane de C a été créée. Sur m_i la première étape est similaire à celle présentée plus haut pour le composant C :

1. pour chaque sous-composant c_k, m_i vérifie si toutes les contraintes de ressources qui lui sont associées sont vérifiées localement ;
2. pour chaque sous-composant c_k pour lequel $Solveur(ResCst)$ a retourné vrai, m_i se déclare candidate pour l'instanciation de c_k auprès du DeployManager et attend une réponse ;

(a) si m_i reçoit une réponse positive, alors c_k peut être instancié sur m_i qui prend en charge cette tâche ;

(b) si m_i reçoit une réponse négative, i.e. si le DeployManager n'a pas choisi m_i pour l'hébergement de c_k, m_i est informé de l'identité d'une autre machine, m_k, sur laquelle c_k doit être instancié. La machine m_i crée alors un composant mandataire pour c_k avec pour référence distante la machine m_k.

Nous voyons ici que les machines souhaitant héberger un sous-composant doivent en demander l'autorisation au DeployManager. Ceci est nécessaire pour deux raisons :

- à chaque composant primitif défini dans le descripteur d'architecture doit correspondre une seule instance. Puisque plusieurs machines peuvent satisfaire les exigences de ressources d'un composant il faut pouvoir contrôler la décision d'instanciation d'un composant. Ce rôle revient au DeployManager.
- s'il existe des contraintes de localisation associées aux composants mettant en jeu plusieurs variables, il est nécessaire d'avoir une connaissance des valeurs possibles pour ces variables (le domaine de définition des variables peut ne pas être spécifié dans le descripteur de déploiement). En centralisant les candidatures au niveau du DeployManager, la résolution des contraintes de localisation est rendue possible.

Ainsi, pour le cas des sous-composants, le DeployManager a un double rôle. Comme pour le composant racine, il centralise les informations de placement précisant les machines autorisées à instancier les composants. D'autre part, le DeployManager, en fonction des *candidatures* pour l'instanciation d'un composant, décide quelle est la machine (ou les machines) devant effectivement héberger le composant en prenant en compte les contraintes de localisation.

8.2.1.3 Mise en place des liaisons

Pour chaque machine ayant instancié des sous-composants, le descripteur d'architecture spécifie les liaisons devant être mises en place entre ces composants.

Du côté de la machine m_i candidate à l'instanciation d'un sous-composant c_k, dès la réception de la réponse du DeployManager, m_i crée soit un composant mandataire pour c_k soit une instance de c_k. Dans ces deux situations, les entités créées représentent le même composant sur lesquels des liaisons doivent être effectuées. Ainsi, la mise en place des liaisons est faite indifféremment de la distribution des composants : tout se passe comme si l'architecture avait été instanciée localement.

8.2.1.4 Instanciation de l'ensemble de l'application

À ce stade du déploiement, les machines m_i ayant instancié C se voient attribuer l'autorisation d'instancier tout ou partie des sous-composants c_k de C. Pour tous ces sous-composants, si c_k est un composant primitif, le processus de déploiement s'achève, sinon, de manière récursive, le déploiement des sous-composants de c_k s'effectue de la même manière que pour le composant composite C.

Pour les sous composants c_k pour lesquels un composant mandataire a été créé sur m_i, le déploiement des éventuels sous-composants de c_k n'a pas lieu sur m_i.

8.2.1.5 Synthèse

Le processus de déploiement que nous venons de présenter montre comment, à partir des descripteurs d'architecture et de déploiement, l'instanciation d'une architecture de composants hiérarchiques est réalisée. L'approche proposée se base sur un gestionnaire de déploiement (DeployManager) qui centralise les candidatures des machines pouvant héberger un ou plusieurs composants. Cette centralisation permet de générer une configuration de placement conforme aux contraintes de localisation à partir des candidatures reçues.

À partir de la solution de placement, le DeployManager envoie ensuite les ordres de placement aux machines du réseau. Ces directives sont définies en mettant à jour le descripteur de déploiement et d'architecture : après le calcul d'une solution de placement, les variables présentes dans le descripteur sont remplacées par les identités des machines sélectionnées.

Le déploiement propagatif présenté dans cette section repose sur l'hypothèse de stabilité du réseau et de disponibilité des ressources exigées par chacun des composants de l'application. Ainsi, la mise en veille d'une machine, son inaccessibilité ou encore la non disponibilité de certaines ressources, qui caractérisent les réseaux dynamiques ne sont pas intégrées au processus de déploiement. Nous présentons dans la section suivante comment ces aspects dynamiques sont pris en compte dans le processus de déploiement précédent.

8.2.2 Prise en compte de l'arrivée des ressources

Nous levons maintenant l'hypothèse de disponibilité des ressources faites précédemment. Ainsi, lorsque le déploiement est lancé, certaines machines peuvent ne pas être allumées et des ressources exigées par certains composants, comme par exemple la quantité de mémoire disponible, peuvent ne pas être disponibles sur aucune des machines connectées au réseau. Dès lors qu'une machine devient connectée au réseau (qui rappelons le est ici supposé connexe), sa présence est détectée par le DeployManager qui maintient à jour la liste des identités des autres machines qui sont connectées.

8.2.2.1 Observation périodique des ressources

Dans les réseaux que nous considérons, même si des ressources sont manquantes à l'initiation du déploiement elles peuvent devenir disponibles ultérieurement. Afin de prendre en compte cette dynamicité, nous introduisons une observation périodique des ressources sur chaque machine participant au déploiement qui suit l'algorithme donné dans le listing 8.1 Pour chaque composant non encore placé (non instancié), i.e. pour chaque composant dont la cible `target` dans le descripteur de déploiement n'a pas été mis à jour, une observation périodique des ressources requises par ce composant est mise en place sur chaque machine. De cette manière, lorsque localement des ressources se libèrent, e.g. la quantité de mémoire requise par le composant c_k, sur la machine m_i est désormais disponible, m_i peut désormais envoyer sa candidature au DeployManager. Cette nouvelle candidature amenant éventuellement une solution de placement compatible avec les contraintes de localisation.

```
Pour chaque sous-composant C[k]
    Tant que C[k] n'a pas de machine ôhte
       Pour chaque contrainte de ressource R éassocie à C[k]
         1. Sondes[R][] = C[k] // ajout de C[k]
```

Listing 8.1: Algorithme permettant l'observation périodique des ressources requises par un composant. Pour chaque contrainte de ressource, une sonde correspondant à la ressource requise est créée

Sur chaque machine, l'observation périodique des ressources requises par un composant donné cesse dès lors que le composant a été instancié, localement ou non, i.e. lorsque le DeployManager informe qu'une machine hôte a été trouvée pour ce composant. Le listing 8.2 décrit la suite de la boucle d'observation périodique pour une machine donnée. À chaque tour de boucle, la machine teste si localement toutes les ressources demandées sont disponibles (résolution du $CSP(ResCst)$). Si c'est le cas, elle envoie sa candidature pour l'instanciation du composant (à condition de ne pas encore envoyé sa candidature) au DeployManager.

```
periode = getPeriod(R)
Tant que C[k] n'a pas de machine hote
   2. value = Sondes(R).getValue() ;
      ajouter (R,value) dans le store de contrainte
      Si pasEncoreCandidat && Resolveur(CSP(C[k],ResCst)) retourne vrai
         Candidater pour C[k] ;
      Sleep(periode)
```

Listing 8.2: Observation périodique et candidature

8.2.2.2 Arrivée d'une nouvelle machine

L'arrivée d'une machine au sein du réseau peut amener à faire évoluer le déploiement : soit du fait des nouvelles ressources qu'apporte cette machine, soit parce que le nombre de machines nécessaire à la définition d'un placement respectant les contraintes de localisation devient suffisant.

Le déploiement étant déjà « en cours », une machine qui n'était pas présente à l'initiation du déploiement est mise au courant par l'intermédiaire du DeployManager qui a d'une part détecté la présence de cette machine et d'autre part déterminé si cette dernière était concernée par le déploiement (i.e. qu'elle appartient à la liste des machines cibles définie par l'élément `target` du composant composite racine). Que l'instanciation de tous les composants de l'architecture ait été réalisée ou non, les descripteurs de déploiement et d'architecture sont envoyés à la machine nouvellement connectée, tout comme pour celles présentes initialement. Ainsi lorsqu'une machine m entre dans le réseau elle prend connaissance de l'application à déployer et lance le processus de déploiement (avec la mise en place de la vérification locale des contraintes de ressources) pour le composant composite racine. Ce processus est donc le même que pour n'importe quelle machine présente au moment où le déploiement a été initié. En effet, l'application à déployer, ubiquitaire également par rapport à m, doit offrir ses fonctionnalités depuis cette machine. Il est donc nécessaire d'instancier sur m des membranes de composants composites ou des composants mandataires si les ressources n'y sont pas suffisantes. Pour des composants primitifs pouvant encore être instanciés, m annonce au DeployManager sa candidature.

8.2.2.3 Activation progressive des composants

La prise en compte de la dynamicité des ressources ainsi que l'arrivée de nouvelles machines au sein du réseau définit un déploiement propagatif, i.e. une instanciation progressive des différentes composants de l'application. Puisque le déploiement de l'application se fait au fur et à mesure de la découverte des ressources présentes dans l'environnement, il est intéressant de disposer des fonctionnalités de l'application au plus tôt : les composants déjà instanciés, à partir du moment où leurs dépendances (services requis) sont résolues, peuvent être utilisés même si la totalité de l'application n'est pas déployée. Cette mise à disposition progressive de l'application est rendue possible du fait de la possibilité d'isoler les dépendances entre interfaces requises et fournies des composants grâce à leur activation et inactivation (cf. chapitre 5).

La distinction entre les interfaces actives et inactives que nous avons introduite dans le modèle de composants hiérarchiques permet en effet d'activer progressivement les fonctionnalités des composants en fonction du stade d'avancement de leurs instanciations. Dans les approches traditionnelles (e.g. Fractal, CCM, .Net), l'application n'est activée qu'à partir du moment où tous les composants sont instanciés et que les liaisons spécifiées dans le descripteur d'architecture sont mises en place. Avec la notion d'interface active, il est désormais possible d'isoler les fonctionnalités des composants qui peuvent être activées alors que tous les composants ne sont pas instanciés ou accessibles.

8.2.3 Discussion

Nous avons présenté dans cette section un processus de déploiement que nous avons qualifié de propagatif. En effet, le processus proposé permet l'instanciation progressive des composants de l'application au fur et à mesure que les ressources requises deviennent disponibles. En ayant intégré au modèle de composants la notion d'interface active, on a fait en sorte que certaines fonctionnalités de l'application puissent être activées alors que des composants ne sont pas encore instanciés.

Le processus de déploiement place les instances de composants en fonction des contraintes de ressources et de localisation. La résolution de ces dernières est faite par un gestionnaire de déploiement qui centralise les candidatures des machines possédant les ressources nécessaires à l'instanciation des composants.

8.3 Déploiement propagatif sur un réseau dynamique

Le processus de déploiement présenté dans la section précédente, bien que prenant en compte la dynamicité de l'environnement d'exécution, repose sur une entité particulière, le DeployManager, pour garantir la cohérence des décisions portant sur l'instanciation des composants. La solution jusqu'alors proposée donne au DeployManager un rôle privilégié du fait qu'il centralise les informations nécessaires à ce processus de décision. Dans les réseaux dynamiques, nous ne pouvons nous appuyer sur une telle entité, et d'une manière générale sur une architecture client / serveur. En effet, aucune hypothèse ne peut être faite sur la disponibilité d'une ou plusieurs machines particulières et donc de celle hébergeant le gestionnaire de déploiement. Il est donc plus réaliste de considérer que toutes les machines jouent le même rôle.

Nous allons voir dans cette section les difficultés que posent les réseaux dynamiques pour la mise en place d'outils permettant aux différentes machines de coordonner leurs actions. Dans un second temps, nous formaliserons le problème à résoudre.

8.3.1 Consensus dans les réseaux dynamiques

Ne pouvant nous appuyer sur un gestionnaire de déploiement, la distribution des informations nécessaires au déploiement sur les différentes machines du réseau est préférée à leur centralisation. Le problème du maintien de la cohérence de ces informations se pose alors et par conséquent des mécanismes de coordination entre les machines doivent être définis. En particulier, le décision de placer un composant sur une machine donnée ne peut se faire sans un accord des machines participant au déploiement. Parmi les travaux autour de la prise de décision collective, nous nous intéressons dans cette section aux problèmes de consensus. Avant de définir ces problèmes, nous caractérisons plus précisément les fautes qui peuvent se produire dans les réseaux dynamiques et qui influent sur les algorithmes de « résolution » de consensus.

8.3.1.1 Modèles de fautes

Dans les réseaux dynamiques la volatilité des équipements rendent l'échange de messages entre équipements non fiable : des messages peuvent être perdus. Par ailleurs, les délais de transmission peuvent difficilement être bornés : il n'est en effet pas toujours possible de déterminer si une machine mis en veille va à nouveau être accessible quelques secondes ou quelques heures plus tard.

Dans les réseaux dynamiques, les canaux de communications sont *non fiables* et *asynchrones*.

De plus, la mobilité des équipements et d'une manière générale la fragmentation du réseau en îlots peuvent être vues comme des fautes dès lors que les machines inaccessibles mettaient à disposition des ressources (logicielles ou matérielles) utilisées par une application distribuée.

Nous pouvons qualifier ce comportement comme faisant parti des fautes par omission [PT86] ou de performance [CASD95, AMP04]. Un équipement (processus) commet une faute par omission lorsque des actions qu'il devait réaliser ne sont pas faites. C'est le cas par exemple d'un processus ne recevant pas ou n'émettant qu'une partie des messages, e.g. une machine qui devient hors de portée des autres équipements. Un processus commettant des fautes par omission peut très bien se comporter normalement au bout d'un certain temps. Les fautes de performance résultent d'un non respect des délais de la part des processus fautifs, pouvant conduire à des délais de transmission infinis.

Pour des raisons logicielles ou matérielles, un processus peut subitement être stoppé, c'est-à-dire qu'il ne participe plus aux communications avec les autres entités du réseau. Ces fautes par arrêt [FLP85] sont définitives.

Un processus ne subissant aucune faute est qualifié de *correct*. Il est à noter cependant qu'un processus correct, dans un réseau comportant des îlots, sera qualifié comme tel pour les autres machines se trouvant dans le même îlot que lui. Pour les machines d'un autre îlot, ce processus est fautif.

Un réseau dynamique est qualifié par :
– le nombre maximum t de processus qui peuvent commettre des fautes ;
– et f le nombre effectif de processus fautifs.

L'asynchronisme et la présence de processus fautifs dans les réseaux dynamiques empêchent donc la centralisation des décisions qui portent sur l'affectation des machines cibles pour chaque composant de l'architecture d'une application (ubiquitaire). Il est difficile de définir a priori l'identité d'une machine responsable de ces décisions sachant qu'à tout moment elle peut être mise en veille ou hors de portée des autres machines. La distribution des décisions de placement impose par ailleurs une certaine coordination entre les entités prenant par à ce processus : il n'est pas autorisé qu'un composant se retrouve par exemple instancié sur deux machines différentes alors que le descripteur d'architecture et de déploiement ne définissait qu'une seule instance. Il est nécessaire qu'une « phase d'accord » ait lieu entre les différents processus afin de garantir la cohérence de l'application déployée vis-à-vis de ses spécifications. La résolution des problèmes faisant intervenir une prise de décision collective dans des environnements asynchrones avec fautes est... impossible, comme cela a été montré dans [FLP85]. Nous allons formaliser le problème du choix de placement des composants dans les réseaux dynamiques.

8.3.1.2 Problème du consensus

Dès lors que des collaborations entres processus répartis sont définis, qu'un processus responsable de la cohérence du système existe, la présence des fautes soulève le problème de la fiabilité et de la viabilité des informations nécessaires aux systèmes répartis. La mise en place des solutions pour tolérer les défaillances repose sur la résolution des problèmes d'accord. Parmi les problèmes d'accord, le problème du consensus [FLP85, DLS88, DRS90, PR04] fait figure d'exemple et s'énonce comme suit : « après avoir proposé une valeur, tous les processus doivent s'accorder sur une même valeur ». Différents autres problèmes d'accord (validation atomique, diffusion atomique etc.) peuvent être résolus en utilisant une solution au problème du consensus comme brique de base.

Un algorithme de consensus se déroule en trois étapes :

1. chaque processus propose une valeur ;
2. tous les processus se concertent pour le choix d'une valeur parmi celles proposées ;
3. les processus décident la valeur choisie.

L'action de décision est irrévocable : lorsqu'une entité décide d'une valeur, elle ne peut plus en changer (sauf en faisant un nouveau consensus). Nous reprenons ces étapes du consensus dans le processus de déploiement : la valeur proposée par les machines sera leur identité lorsqu'elles sont candidates à l'instanciation d'un composant. Étant donné la multiplicité des candidatures, l'algorithme de consensus devra garantir que toutes les machines du réseau éliront la même machine.

Trois propriétés permettent de spécifier un problème du consensus :

Terminaison : tous les processus corrects doivent finir par décider ;

Validité : toute valeur décidée a été proposée ;

Accord : si un processus correct décide une valeur v, alors tous les processus corrects décident v.

Dans un système asynchrone sujet à des défaillances, Fischer, Lynch et Paterson ont montré ([FLP85]) qu'il n'existe pas d'algorithme déterministe permettant de résoudre sur le problème du consensus. Ce résultat d'impossibilité (FLP) vient du fait que dans de tels systèmes, il n'est pas possible de détecter la panne d'un processus : il est en effet impossible de distinguer un processus lent d'un processus ayant *crashé* du fait des bornes infinies sur les délais de transmission des messages.

Il n'existe pas de solution déterministe au problème du consensus dans un environnement asynchrone même en présence d'une seule faute de processus.

L'impossibilité étant admise, il est cependant possible de la contourner : soit en affaiblissant le problème du consensus, soit en renforçant le modèle asynchrone. Dans le premier cas, cela consiste à modifier les exigences du problème de consensus (les propriétés de terminaison, de validité et d'accord) pour le rendre plus simple. Parmi ces simplifications, nous trouvons les problèmes de consensus probabiliste [BO83], et de consensus ensembliste. La seconde approche consiste à rajouter des hypothèses de synchronie afin de rendre le système partiellement synchrone [DLS88]. De cette manière, soit il existe une borne, non connue des processus, sur les temps de transfert et d'exécution, soit ces bornes existent mais seulement après un certain temps. D'autres travaux permettent de renforcer le modèle synchrone en offrant aux entités des informations sur l'état des processus, c'est le cas en particulier des travaux sur les détecteurs de défaillances proposés par Chandra et Toueg [CT96] qui enrichissent l'environnement par une liste d'identifiants de processus suspectés d'être fautifs.

Récemment, une nouvelle approche fondée sur des conditions a été proposée par Mostéfaoui, Rajsbaum et Raynal pour résoudre le consensus [MRR01]. Cette approche consiste en un affaiblissement du problème, plus exactement en une restriction de l'espace de définition du consensus. Nous avons choisi cette approche pour traiter le choix du placement des composants dans les réseaux dynamiques.

La sous-section suivante présente la proposition de [MRR01]. Puis nous détaillons les modifications que nous avons apportées à l'algorithme initialement conçu.

8.3.2 Consensus à entrées contraintes

8.3.2.1 Conditions et vecteurs d'entrées

Le problème du consensus considère l'ensemble $\mathcal{V}(|\mathcal{V}| \geq 2)$ des valeurs qui peuvent être proposées par les processus. À l'exécution, chaque processus correct p_i propose une valeur $v_i \in \mathcal{V}$ et tous les processus corrects doivent décider la même valeur v, qui doit faire partie des valeurs proposées. L'ensemble des valeurs proposées est représenté à l'exécution par un *vecteur d'entrées* tel que la i-ème entrée contient la valeur proposée par p_i, ou bien $\perp$ si p_i n'a participé à aucune étape de l'algorithme. Dans la suite, I désigne un vecteur d'entrées dont toutes les entrées sont dans $\mathcal{V}$, et J représente un vecteur d'entrées qui peut contenir des entrées valant $\perp$.

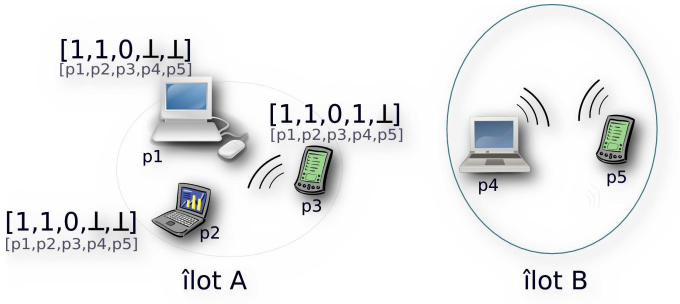


FIG. 8.1: Îlots et vues des machines

Les variables f et t représentent respectivement le nombre effectif de processus fautifs et le nombre maximal de processus pouvant être défaillants. Tous les vecteurs d'entrées J considérés dans la suite possèdent au maximum t valeurs égales à $\perp$. Un tel vecteur est appelé *vue*.

Soit $\mathcal{V}^n$ l'ensemble des vecteurs d'entrées possibles ayant toutes leurs entrées dans $\mathcal{V}$ et soit $\mathcal{V}_t^n$ l'ensemble de toutes les vues ayant au plus t valeurs égales à $\perp$.

Le problème du consensus peut être vu comme une fonction de $\mathcal{V}^n$ dans $\mathcal{V}$. Cependant une telle fonction ne peut exister, du fait du résultat d'impossibilité de FLP, dans les environnements dynamiques que nous considérons. Plus précisément, cette fonction n'est pas définie sur l'ensemble $\mathcal{V}^n$ tout entier. Les auteurs de [MRR01] proposent de trouver un sous-ensemble $C \in \mathcal{V}^n$ pour lequel la fonction sus-citée est définie malgré la présence de t processus fautifs au maximum.

Pour $I \in \mathcal{V}^n$, soit $\mathcal{I}_t$ l'ensemble de toutes ses vues possibles, i.e. l'ensemble des vecteurs J ayant au plus t valeurs $\perp$ et tel que les valeurs de J différentes de $\perp$ sont identiques à celles de I. Pour un ensemble C, $C \subseteq \mathcal{V}_t^n$, soit $\mathcal{C}_t$ l'union des ensembles $\mathcal{I}_t$ issus de tout $I \in C$.

Tout vecteur $J \in \mathcal{V}_t^n$ est un vecteur d'entrées possible du problème considéré. C est appelée *condition*.

8.3.2.2 Exemple

Soit un réseau dynamique constitué de cinq machines et dans lequel deux îlots se sont formés (Figure 8.1). Pour l'îlot A, les deux machines de l'îlot B sont inaccessibles. Pour p_1, p_2 et p_3 les machines de l'îlot B sont considérées comme défaillantes. De ce fait, nous avons dans l'îlot A :

- $\mathcal{V} = 0, 1$
- $\mathcal{V}^{n=5} = \left\{ \begin{array}{l} [0,0,0,0,0], [1,0,0,0,0], [0,1,0,0,0], [0,0,1,0,0], [0,0,0,1,0], [0,0,0,0,1], \\ [1,1,0,0,0], [1,0,1,0,0], [1,0,0,1,0], [1,0,0,0,1], [1,1,1,0,0], [1,1,0,1,0], \\ [1,1,0,0,1], [1,1,1,1,0], [1,1,1,0,1], [1,1,1,1,1] \end{array} \right\}$
- $t = 2$
- Soit $I = [1,1,0,1,0]$, nous avons

$$\mathcal{I}_t = \left\{ \begin{array}{l} [1,1,0,1,0], [\perp,1,0,1,0], [1,\perp,0,1,0], [1,1,\perp,1,0], [1,1,0,\perp,0], \\ [1,1,0,1,\perp], [\perp,\perp,0,1,0], [\perp,1,\perp,1,0], [\perp,1,0,\perp,0], [\perp,1,0,1,\perp], \\ [1,\perp,\perp,1,0], [1,\perp,0,\perp,0], [1,\perp,0,1,\perp], [1,1,\perp,\perp,0], [1,1,\perp,1,\perp], \\ [1,1,0,\perp,\perp] \end{array} \right\}$$

Une des vues possibles pour les machines (processus) p_1, p_2 et p_3 est représentée sur la figure 8.1. La machine p_3 s'étant déplacée — elle a par exemple rejoint l'îlot B pendant un moment —, p_4 n'a pas été considérée comme défaillante par p_3.

8.3.2.3 Condition acceptable et résolution du problème de consensus

Les auteurs de [MRR01] caractérisent un protocole résolvant la condition C tolérant t fautes si pour chaque exécution dont le vecteur d'entrées J appartient à $\mathcal{V}_t^n$, le protocole vérifie les propriétés d'accord, de validité et de terminaison.

D'autre part, des contraintes sont imposées sur les vecteurs d'entrées pour qu'à partir de ces derniers, une valeur de décision puisse être décidée par tous les processus corrects. Ainsi, le problème du consensus pour une condition C pourra être résolu par un algorithme si et seulement si il existe[27] fonctions P et S dont le rôle est de :

- pour P, tester si une décision est possible en fonction de la vue locale du processus ;
- et pour S, de calculer une valeur de décision à partir de la vue locale du processus. La valeur de décision appartient à l'ensemble des valeurs que le processus a vues.

Un ensemble de conditions contraintes ont été spécifiées par les auteurs ([MRRR01]) pour lesquelles les fonctions P et S ont été identifiées. Nous détaillons la condition $C2$ que nous avons choisie pour la résolution du problème du choix de placements des composants.

Intuitivement, un protocole utilisant la condition $C2$ essaie d'imposer comme valeur de décision la valeur extrémale vue par tous les processus. Ceci suppose une relation d'ordre totale sur les valeurs de $\mathcal{V}$ (par exemple, plus grand que ou plus petit que). Nous considérons (arbitrairement) la valeur la plus grande et incluons de ce fait dans C2 tout vecteur dont la plus grande valeur apparaît plus de t fois (les processus attendant $n - t$ propositions verront forcément la même valeur maximale). Nous notons $max(I)$ la plus grande valeur différente de $\perp$ contenue dans le vecteur I et $\#_{max(I)}$ le nombre d'occurrence du plus grand élément de I. La définition formelle est la suivante, pour t fixé :

$$(I \in C2) ssi [\#_{max(I)}(I) > t]$$

Les fonctions $P2$ et $S2$ pour $C2$ sont définies pour les vecteurs $J \in \mathcal{V}_t^n$ par :

$$\begin{cases} P2(J) = \#_{max(J)}(J) > t - \#_{\perp}(J) \\ S2(J) = max(J) \end{cases}$$

$\#_{\perp}(J)$ représente le nombre d'occurrence de $\perp$ dans J.

Un processus exécutant un protocole résolvant $C2$ pourra décider d'une valeur si dans un premier temps, $P2$ retourne vrai, c'est-à-dire que pour sa vue, le nombre de processus ayant proposé la plus grande valeur est majoritaire par rapport à ceux ayant proposé une valeur. Si tel est le cas, ce processus choisira ($S2$) cette plus grande valeur. Ce protocole étant distribué sur tous les processus, si le protocole du consensus termine, c'est la plus grande valeur qui sera choisie par tous les processus corrects.

Supposons que les vecteurs d'entrées possibles pour les machines de l'exemple précédent soit celui de la figure 8.1. Nous voyons que les machines m_1, m_2 et m_3 possèdent un vecteur d'entrées ayant une valeur majoritaire : ici 1 apparaît trois fois dans chaque vecteur. Pour

[27] Nous ne détaillons pas ici les différentes propriétés que doivent satisfaire les fonctions P et S. Le lecteur pourra se référer à [MRR01] ou encore à la thèse de R. Parvédy [Par04] pour plus de détails.

m_1, m_2 et m_3, l'évaluation de $P2$ retourne vraie : le nombre de 1 est supérieur à 0 $(2 - 2 \times \perp)$. Ainsi chacune des machines m_1, m_2 et m_3 choisira pour valeur de décision 1.

Dans l'îlot B, nous verrons que le protocole de consensus ne peut terminer du fait que le nombre de machines constituant l'îlot est insuffisant. Cela va être explicité dans la section suivante.

8.3.3 Consensus pour le placement des composants

La sous-section précédente a permis de présenter les concepts de vues et de conditions que nous allons maintenant utiliser pour décrire le protocole que nous avons défini pour résoudre le problème de consensus pour $C2$ dans les environnements dynamiques.

8.3.3.1 Fiabilisation des communications

Le protocole proposé initialement dans [MRR01] repose sur des communications fiables, c'est-à-dire que tout message émis à un destinataire sera effectivement reçu par ce dernier. Dans un réseau où des partitions sont possibles et dans lequel les communications radios sont potentiellement utilisées, cette caractéristique ne peut être assurée. Comme cela a été proposée dans la thèse de C. Marchand [Mar04] pour les détecteurs de défaillances, nous proposons de fiabiliser les communications en ajoutant un composant, sur chaque machine, responsable de la retransmission périodique des messages. Ainsi les entités à un moment isolées des autres machines du fait de pertes de communications ou de leur mise en veille, pourront de nouveau participer au protocole de consensus décrit ci-après. Des mécanismes d'accusé de réception sont par ailleurs ajoutés afin de garantir la bonne réception des messages. Les politiques de retransmission doivent cependant être maîtrisées pour éviter la surcharge du réseau par les messages réémis. Pour cela, nous avons défini sur chaque machine des composants responsables de la détection des équipements avoisinants. Ceci permet aux machines d'avoir une vision de la connectivité de l'îlot dans lequel elles se trouvent. Ces composants guident les retransmissions des messages en fonction de la présence des machines.

8.3.3.2 Le protocole de consensus

Lorsqu'une machine prend connaissance du descripteur d'architecture et de déploiement de l'application à installer, son objectif est d'instancier localement les composants pour lesquelles elle vérifie les contraintes de ressources. Cette condition n'est pas suffisante, il faut également que l'instanciation du composant sur cette machine ne contredise pas les contraintes de localisation qui lui sont associées.

Ne pouvant nous appuyer sur une machine particulière pour les prises de décisions, toutes les machines constituant le réseau joue le même rôle ; la décision de placer un composant sur telle machine plutôt qu'une autre sera validée par un consensus. Les grandes étapes du processus du déploiement pour une machine m_i et pour un composant composite C ubiquitaire sont décrites ci-après. Le nombre de machines constituant le réseau est de n et le nombre maximum de machines pouvant être fautives est t.

La première partie de l'algorithme ne diffère pas de la version centralisée, s'appuyant sur un DeployManager. Avant de pouvoir instancier un composant il est nécessaire de faire une

Algorithme 1 Algorithme distribué permettant de sélectionner la machine pouvant héberger un composant

1. pour chaque sous-composant c_k,
   - m_i définit un vecteur d'entrées V_{i,c_k}
   - m_i vérifie si toutes les contraintes de ressources qui lui sont associées sont vérifiées localement ;
2. pour chaque sous-composant c_k pour lequel $Solveur(ResCst)$ a retourné vrai, m_i se définit comme candidate à l'instanciation de c_k
   (a) cela consiste à définir comme valeur de proposition pour le protocole de consensus, le couple $v_{i,k} = (c_k, m_i)$
   (b) si aucune contrainte de localisation n'est associée à c_k, m_i diffuse $PROP(v_{i,k})$ à toutes les machines connectées (le placement peut se faire sur n'importe quelle machine)
   (c) sinon m_i diffuse $CANDIDAT(v_{i,k})$ à toutes les machines connectées
3. m_i reçoit également des candidatures $CANDIDAT(V_{j,k})$ de la part d'autres machines
   - dès que m_i possède suffisamment de candidatures pour calculer une solution de placement pour c_k, elle diffuse $PROP(v_{i,k})$ aux machines connectées.
4. m_i attend au moins $(n - t)$ messages $PROP$
5. m_i construit ensuite sa vue V_{i,c_k} à partir des propositions reçues
   - $V_{i,c_k}[j] = PROP(v_{i,k})$ si la machine j a délivré un tel message ;
   - $V_{i,c_k}[j] = \perp$ sinon
6. Si $P2(V_{i,k})$ retourne vrai, la machine ayant proposée une solution de placement et choisie par m_i est $w_i = S2(V_{i,k})$; sinon, m_i ne peut décider : $w_i = \top$
7. m_i diffuse de façon fiable sa valeur de décision en envoyant le message $DECISION(v_{i,k}, w_i)$ à toutes les machines du réseau. Les machines non connectées ou non accessibles au moment de l'émission du message finiront par le recevoir du fait de la fiabilisation des communications mise en place
8. m_i attend qu'une majorité de machines décide la valeur w avant de décider w ; Si m_i a reçu un w_j de la part de chaque machine m_j alors m_i décide de choisir celle ayant le plus grand identifiant.

évaluation locale des ressources demandées par chaque composant. Lorsqu'une machine satisfait les exigences des composants en termes de ressources et s'il n'y pas de contrainte de localisation associée aux composants, elle peut directement *exiger* leurs instanciations (messages $PROP(-)$). Dans le cas contraire, ne pouvant s'appuyer sur une machine particulière qui centraliserait les candidatures, chaque machine joue le rôle du DeployManager : en fonction des candidatures (messages $CANDIDAT(-)$), toutes les machines du réseau essayent de calculer une solution de placement des composants mettant en jeu des contraintes de localisation. Dès qu'une machine (il peut en avoir plus d'une) trouve une configuration de placement, elle essaye de la faire adopter par les autres machines (tout comme c'était le cas pour les composants dénués de contrainte de localisation).

Le protocole de consensus que nous proposons consiste donc à choisir une machine parmi celles proposant une solution de placement. Ce protocole se déroule en trois phases : dans un premier temps, chaque machine m_i diffuse sa proposition $v_{i,k}$ de placement pour le composant c_k aux autres machines et construit sa vue V_{i,c_k} à partir d'au moins $n - t$ propositions qu'elle a reçues. Dans un deuxième temps, la machine m_i essaie de calculer sa valeur de décision w_i, à partir de la condition ; si elle n'y arrive pas, elle fixe w_i à $\top$. Elle diffuse ensuite sa valeur de décision de façon fiable. Dans un dernier temps, elle attend la confirmation d'une majorité de machine voulant décider une même valeur w.

L'algorithme proposé met en jeu la variable t représentant le nombre maximum de machines sujettes à des défaillances (une machine m_i défaillante est perçue comme telle par les autres machines qui ne sont pas dans le même îlot que m_i). Une des conditions pour que l'algorithme termine est que le nombre de *participants* au consensus est suffisant. Nous fixons donc $t = \lfloor \frac{n}{2} \rfloor$, i.e. au moins une majorité de machines sont non défaillantes. Ainsi, dans un réseau dans lequel un îlot constitué d'une majorité de machines existe, il est possible de prendre des décisions de placement des composants et ainsi « faire avancer » le déploiement.

Le protocole de consensus et donc la progression du déploiement de l'application est possible au sein d'un îlot majoritaire.

Un consensus est lancé (toutes les machines non fautives y participent) dès lors qu'une machine a calculé une solution de placement pour un ou plusieurs composants (dans le cas des contraintes de localisation) et ce processus nécessite une majorité de participants. Cette majorité dépend de chaque composant composite de l'application : des cibles de déploiement étant définies à leur niveau, seules les machines concernées par le déploiement du composant composite vont constituer l'ensemble des participants pour toutes les décisions relatives à ses sous-composants directs. Par exemple, si le composant composite DiapomaKer est ubiquitaire par rapport à m_1, m_2, m_3 et m_4 alors le consensus peut terminer dès lors que trois de ces machines sont dans le même îlot. Pour le composite DocumentSearch, si des contraintes de localisation restreignent les machines cibles à m_1, m_2 et m_3, le consensus est résolu lorsque que deux machines parmi m_1, m_2 et m_3 se trouvent dans le même îlot.

Le nombre de participants à un consensus est fonction de la hiérarchie de l'application.

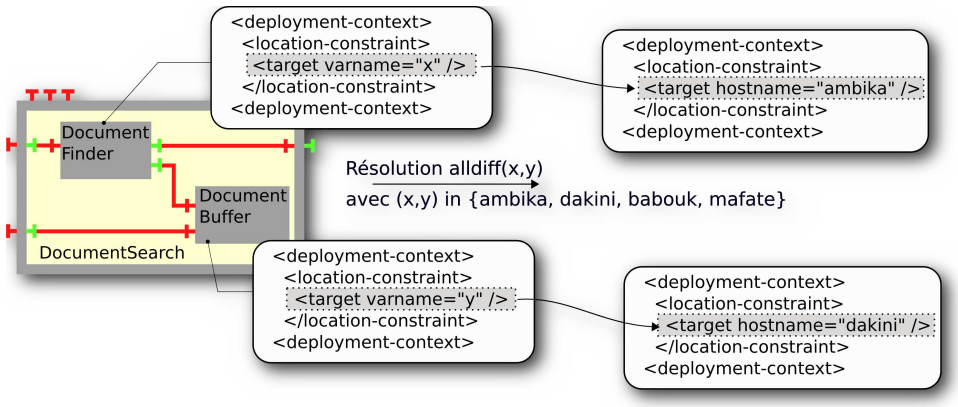


FIG. 8.2: Mise à jour du descripteur de déploiement après la résolution d'une contrainte de localisation

8.3.4 Propagation des informations de déploiement

Les phases du déploiement propagatif (diffusion des descripteurs de déploiement et d'architecture, évaluation locale des contraintes de ressources, candidatures à l'instanciation des composants, calcul d'une solution de placement) amène à une décision collective du placement des composants par les machines du réseau. Ainsi, une fois l'algorithme du consensus terminé, toutes les machines ayant pu décider (une majorité d'entre elles) ont validé une solution de placement :

- pour les composants n'ayant aucune contrainte de localisation, la solution de placement choisie est celle proposée par la machine qui a été *élue*. C'est cette dernière qui doit donc instancier ces composants : cette même machine a été *élue* par l'algorithme de consensus ;
- pour les composants pour lesquels des contraintes de localisation existent, la solution de placement choisie contient l'identité des machines devant instancier les composants.

Puisque les machines peuvent être mises en veille ces décisions de placement doivent être persistantes. Par ailleurs, certaines machines peuvent ne pas avoir connaissance de ces décisions de placement : certaines peuvent en effet être hors de portée voire mises en veille. Lorsque ces machines seront à nouveau accessibles, elles doivent être informées de ces décisions car dès lors qu'un îlot majoritaire se formera et dans lequel elles seront connectées, il faut empêcher l'instanciation de composants pour lesquels un placement avait déjà été décidé.

Pour diffuser l'information de placement aux autres machines, chaque machine ayant décidé à l'aide de l'algorithme de consensus met à jour le descripteur de déploiement. Le descripteur avec les nouvelles informations de placement, i. e. le nom de la machine sélectionnée pour chaque composant est ajoutée. Avant le déploiement, aucune identité n'est nécessaire pour définir la localisation d'un composant grâce à l'utilisation des variables. Par exemple, si les machines `ambika` et `dakini` sont choisies respectivement pour l'hébergement des composants DocumentFinder et DocumentBuffer, les lignes du descripteur de déploiement faisant références au noms des variables associées à ces composants sont remplacées sont modifiées comme illustrée sur la figure 8.2.

Le composant de retransmission des messages (assurant la fiabilisation des communications) est ensuite utilisé pour que les machines n'ayant pas participé au déploiement puissent recevoir la nouvelle version du descripteur. Ces machines prennent donc connaissance du

placement des composants et peuvent ainsi mettre à jour les références vers les composants distants. Grâce au mécanisme d'accusé de réception, nous évitons la diffusion inutile de messages informatifs.

Chaque composant composite ayant fait l'objet d'un ou plusieurs consensus, la mise à jour du descripteur contient également un numéro permettant d'identifier les décisions les plus récentes. Ceci est nécessaire car des machines non au fait de décisions antérieures, prises à propos de l'instanciation d'un composant, peuvent initier un consensus pour cette instanciation. Le numéro permet de faire échouer ce consensus dans la mesure où les décisions se faisant par majorité, ont garanti qu'au sein de tout îlot majoritaire il existe au moins une machine ayant participé aux décisions les plus récentes.

8.4 Du déploiement propagatif au déploiement autonome

Le déploiement propagatif permet à une application à base de composants d'être déployée au fur et à mesure que les ressources nécessaires à l'application deviennent disponibles. Mais, d'une manière générale et en particulier dans les réseaux que nous considérons, les ressources peuvent également disparaître (e. g. la quantité de mémoire libre requise devient insuffisante) et des pannes peuvent intervenir. Dans ces situations, un ou plusieurs composants doivent être redéployés.

Nous détaillons dans cette section quelles sont les fautes que nous prenons en compte (ou qu'il est possible de prendre en compte) et comment elles sont traitées. Nous présentons ensuite la gestion autonome du déploiement qui complète la phase propagative que nous avons décrit jusqu'ici.

8.4.1 Détection et gestion des fautes

Nous avons présenté dans la section 8.3.1 de ce chapitre les différentes fautes qui peuvent intervenir dans les réseaux dynamiques. Nous faisons comme hypothèse dans cette thèse que les pannes franches concernent uniquement les composants de l'application que l'on déploie. Les pannes des machines et celles de notre propre gestionnaire distribué de déploiement sont écartées. Cependant les machines peuvent à tout moment être isolées des autres machines et également mises en veille de façon volontaire ou involontaire.

Après que l'instanciation d'un composant a eu lieu, nous proposons de prendre en compte les deux fautes suivantes qui nécessitent un redéploiement du composant :

- la panne du composant ;
- la variation d'une ressource exigée par le composant de telle sorte qu'un contrainte de ressources associée au composant n'est plus vérifiée sur la machine l'hébergeant.

Nous détaillons comment nous avons pris en compte ces deux points ci-après.

8.4.1.1 Panne d'un composant

Une fois instanciés et activés, les composants de l'application peuvent *crasher*. La détection de ce type de faute ne peut être faite que par la machine hébergeant le composant. En effet, les autres machines suspectant la panne d'un composant distant peuvent se tromper : le composant distant peut toujours être en état de marche mais tout simplement hors de portée. Lorsque la panne d'un composant est détectée par la machine l'hébergeant (m_i), cette dernière

met a jour son descripteur de déploiement en enlevant la localisation actuelle du composant. Cette mise a jour consiste a remettre les contraintes définies dans le descripteur original. Si aucune contrainte de localisation n'est attachée au composant fautif, m_i teste si localement le redéploiement du composant est possible (évaluation locale des contraintes de ressources), si ce n'est pas le cas, ou s'il existe des contraintes de localisation, m_i propage l'information indiquant que le composant fautif doit être de nouveau instancier. Cette information est complétée par le numéro du dernier consensus qui a servi a instancier le composant.

Les machines qui prennent connaissance qu'un composant doit être instancié de nouveau mettent a jour leur descripteur de déploiement. Ces machines se trouvent donc a nouveau dans la phase du déploiement propagatif : il reste encore des composants à instancier. Les phases d'évaluation locale des ressources et de l'annonce des candidatures ont lieu pour le composant manquant.

8.4.1.2 Violation des contraintes portant sur les ressources

Lorsque les ressources demandées par un composant via ses contraintes de ressources ne sont plus disponibles sur la machine l'hébergeant, il est nécessaire de redéployer le composant sur une autre machine possédant les ressources requises. Comme pour le cas de la panne d'un composant, la détection du manque de ressources est réalisées par la machine hôte. Cependant, il est possible ici de sauvegarder l'état du composant proprement avant de l'arrêter.

Chaque contrainte de ressources associée à un composant et spécifiée à l'aide de CDL fait intervenir une sonde qui a pour rôle d'observer périodiquement l'état de la ressource correspondante. Cette observation prend place sur toutes les machines du réseau concernées par le déploiement du composant auquel est attachée la contrainte de ressources. Dans le processus de déploiement que nous avons présenté jusqu'à présent, cette observation périodique cessait dès lors que le composant exigeant la ressource avait été instancié sur l'une des machines du réseau[28]. Pour détecter que la ressource demandée par le composant n'est plus disponible, l'observation périodique ne doit pas être interrompue sur la machine hébergeant le composant (notons la m_i). Il est donc nécessaire de mettre en place une nouvelle observation périodique sur m_i. Cette observation met en jeu de nouvelles sondes (comme précédemment, une pour chaque contrainte de ressources) qui permettent cette fois de détecter la valeur à partir de laquelle une contrainte de ressources n'est plus vérifiée. Cette valeur peut être précisée explicitement à l'aide de CDL par l'utilisation de l'attribut XML `runtime` :

```
<disk free="1" unit="GB" directory="/home/" operator="min">
   <runtime min="50" unit="MB" />
</disk>
```

Cette contrainte indique que la machine pouvant héberger le composant PhotoRepository doit disposer (avant l'instanciation de PhotoRepository) d'un gigaoctet d'espace disque dans le répertoire `/home` et, que pendant l'exécution du composant PhotoRepository, la quantité d'espace disque (sur m_i) doit être supérieure à 50 mégaoctets. Ceci permet de tenir compte de la consommation en ressource des composants. En l'absence de l'attribut `runtime` le redé-

[28]Plus précisément, l'observation périodique cesse sur une machine lorsque l'information concernant l'instanciation d'un composant lui parvient.

ploiement du composant PhotoRepository serait initié dès lors que la quantité d'espace disque devient inférieure à un gigaoctet.

8.4.2 Automatisation du redéploiement

Le traitement des fautes pouvant amener au redéploiement des composants consiste a ramener le processus de déploiement dans sa phase propagative. Pour cela, nous avons vu que la simple injection dans le réseau d'un message demandant le redéploiement d'un composant suffit. Ainsi, à partir du descripteur de déploiement et d'architecture, chaque machine a pour rôle d'instancier les composants de l'application. L'installation progressive des composants se fait sans aucune intervention manuelle. Il en est de même lorsqu'un composant doit être redéployé du fait d'un crash ou du manque de ressources sur la machine hôte. Les phases du redéploiement autonome se décomposent en trois étapes :

1. Le ou les composants fautifs ou pour lesquels des ressources ne sont plus suffisantes sont stoppés. L'arrêt d'un composant provoque la désactivation de ses interfaces fournies. Toutes les interfaces requises (distantes ou non) liées à ces dernières deviennent inactives, ce qui, par ricochet, désactivera toutes les interfaces menant à ces composants. L'application fonctionne alors en mode dégradé.
2. L'état des composants est alors sauvegardé sous une forme sérialisable (on suppose que le programmeur à prévu cette situation).
3. Un message informant de l'identité des composants à redéployer est diffusé. Ce message contient également l'endroit à partir duquel l'état sérialisé des composants peut être récupéré. Chaque machine recevant ce type de message met à jour leur descripteur de déploiement en supprimant la localisation des composants.

Dans notre approche, le descripteur d'architecture de l'application à déployer est considérée par chaque machine comme un but à atteindre en termes d'instanciation des composants vis-à-vis de contraintes à respecter.

8.5 Discussion

Au cœur du déploiement d'applications à base de composants, le processus d'affectation d'une machine hôte à chacun des composants de l'application constitue une activité difficile à mettre en œuvre lorsque les plates-formes cibles sont des réseaux dynamiques. Dans de tels réseaux, ce processus ne peut être centralisé : il n'est pas possible de faire d'hypothèse sur la présence d'une machine particulière, responsable de la décision de placement des composants du fait de la volatilité des équipements. Par ailleurs, lorsque le déploiement est initié, des machines peuvent ne pas être accessibles et des ressources requises par certains composants ne sont pas nécessairement disponibles.

Nous avons proposé dans ce chapitre un algorithme distribué permettant de décider du placement des composants tout en tenant compte du caractère dynamique de la plate-forme cible. Ainsi, l'instanciation des composants se fait au fur et à mesure de l'arrivée des ressources. Les fonctionnalités de l'application peuvent alors être mises à disposition même si les machines nécessaires au déploiement des composants qui ne sont pas encore installés ne sont pas disponibles. Ce déploiement, qualifié de propagatif, est rendu autonome : dès lors que

des ressources requises par un composant ne sont plus disponibles ou bien qu'un composant subit une panne, le redéploiement du composant est considéré sans qu'aucune intervention manuelle ne soit exigée.

Le déploiement autonome s'appuie sur un algorithme de consensus permettant de garantir l'unicité de l'instanciation des composants malgré la présence d'îlots. Le processus de déploiement étant complètement distribué, il faut en effet empêcher qu'un composant ne soit instancié dans deux îlots différents.

Les décisions sur le placement des composants se font en conformité des contraintes de ressources et de localisation exprimées dans le descripteur de déploiement via le langage CDL.

The first step in innovation is to know that a thing can be created. After that, the rest is a matter of detail.

DUNE : House of Corrine.
Brian Herbert and Kevin J. Anderson

9

Éléments de mise en œuvre : le projet CUBIK

LES CONCEPTS ET MÉCANISMES DÉCRITS dans la seconde partie de ce mémoire ont été mis en œuvre au sein du projet CUBIK. Le prototype que nous avons implanté permet le déploiement et l'exécution dans des réseaux dynamiques d'applications ubiquitaires conçues à l'aide de composants logiciels. Les objectifs du projet Cubik sont en particulier de définir un mode de distribution d'un composant dans un tel contexte ainsi que de concevoir un support pour le déploiement et l'éxécution distribuée des composants. Le modèle de composants sur lequel nous nous appuyons est le modèle de composant Fractal, associé à son implantation de référence en Java, Julia.

Le projet Cubik comporte deux volets :

- CUBIKADL correspond à l'implantation des outils nécessaires à la compilation et la génération des contraintes exprimées à l'aide du langage CDL (*Constrained Deployment Language*) que nous avons présenté dans le chapitre 6 ;
- Le deuxième aspect du projet Cubik est la mise en œuvre d'un intergiciel extensible pour le support et le déploiement de composants Fractal ubiquitaires. Cet intergiciel, baptisé UBIWARE a été implanté en composants Fractal/Julia.

Nous présentons dans ce chapitre la suite d'outils CUBIKADL et l'intergiciel UBIWARE. Nous donnons dans la section 9.1 un aperçu de la programmation d'applications ubiquitaires à l'aide des outils et concepts implantés dans le projet CUBIK. Puis nous décrivons dans la section 9.3 l'architecture de l'intergiciel UBIWARE qui fournit une extension à Fractal/Julia en mettant en œuvre le concept d'applications ubiquitaires.

Sommaire

9.1 Programmer et déployer des composants ubiquitaires Fractal : vision de l'utilisateur

Nous présentons dans cette section comment développer des applications ubiquitaires du point de vue utilisateur (concepteur / deployeur / utilisateur final) à l'aide des outils mis en place dans CUBIK et comment déployer de telles applications dans des réseaux dynamiques.

9.1.1 Conception et implémentation

Les phases de conception d'une application ubiquitaire ne diffèrent pas de celles définies par la programmation orientée composant et en particulier de la méthodologie décrite par le projet Fractal. Il est nécessaire d'identifier dans un premier temps les composants qui doivent être implantés puis les dépendances entre ces composants[29]. L'organisation de l'application en composants composites permet également l'identification des composants et de leurs dépendances (par exemple en définissant les fonctionnalités de l'application de niveaux d'abstraction les plus hauts aux plus bas).

La programmation des composants peut se faire en deux étapes : définition et création des interfaces d'abord, puis l'implantation de ces interfaces dans les classes des composants. Cette séparation entre interfaces et implantations correspond bien à la caractérisation des composants donnée en section 2.1 du chapitre 2. Dans Cubik, l'utilisation du patron interface / classe va permettre la substitution des composants et la mise en place d'objets d'interposition. La définition des composants et de leurs dépendances peuvent être réalisées graphiquement, à l'aide d'un outil comme FractalGUI dont une capture d'écran est donnée à la figure 9.1. FractalGUI est fournie dans la distribution standard de Fractal.

Dans le cas du composant composite DocumentSearch, les interfaces fournies qui doivent être implantées sont :

```
public interface Query { ArrayList<Document> query(String query) ; }
public interface Cache { ArrayList<Document> getDocuments() ; }
```

Listing 9.1: Interface du composant DocumentSearch

[29] La définition des dépendances entre composants se fait la plupart du temps en même temps que l'identification des fonctionnalités offertes par les composants.

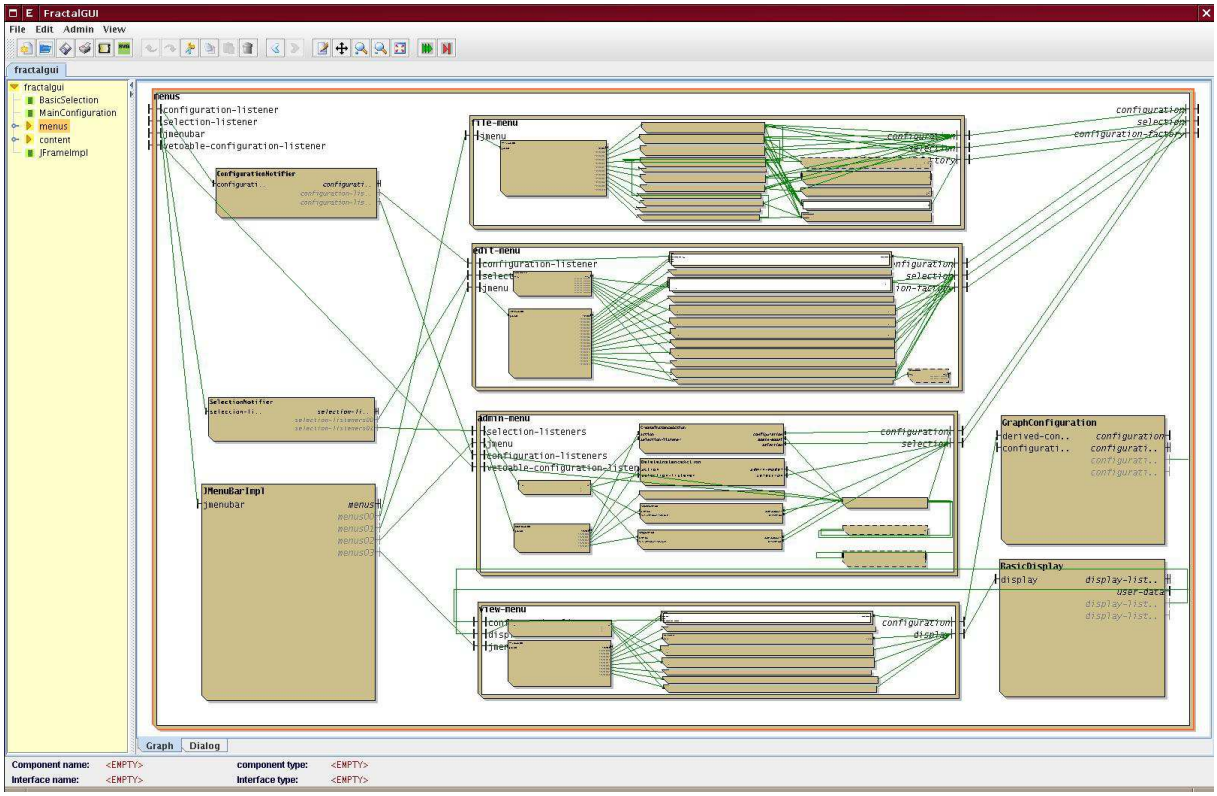


FIG. 9.1: Utilisation de FractalGUI pour définir l'architecture de l'application, ses composants et leur interconnexion

Ces interfaces seront implantées respectivement par les classes `DocumentFinderImpl` et `DocumentBufferImpl`, correspondant aux composants primitifs DocumentFinder et DocumentBuffer.

Le descripteur d'architecture du composant composite DocumentBuffer contient la définition de chaque composant, le type de leurs interfaces requises et fournies, les liaisons ainsi que, pour les composants primitifs, le nom de la classe implantant leurs interfaces fournies :

```
<definition name="DocumentSearch">
  <component name="DocumentFinder">
    <interface  name="query"      type="server" signature="standard.Query" />
    <interface  name="repository" type="client" signature="standard.Document" />
    <interface  name="cache"      type="client" signature="standard.Cache" />
    <controller desc="cubik-primitive"/>
  </component>

  <component name="DocumentBuffer">
    <interface  name="cache"          type="server" signature="standard.Cache"  />
    <interface  name="buffer"         type="server" signature="standard.Buffer" />
    <controller desc="cubik-primitive"/>
  </component>

  <controller desc="cubik-composite"/>
</definition>
```

Listing 9.2: Descripteur d'architecture du composant DocumentSearch de l'application DiapomaKer

Le caractère ubiquitaire des composants est précisé par la balise `<controller/>` qu'offre le langage FractalADL, précisant si le composant ubiquitaire est primitif (`cubik-primitive`) ou composite (`cubik-composite`). Les valeurs `cubik-primitive` et `cubik-composite` correspondent à une extension que nous proposons dans le cadre du projet Cubik.

9.1.2 Spécification du déploiement

Pendant la phase d'implantation des composants décrite précédemment, l'hypothèse généralement faite est que les ressources utiles à chacun des composants seront effectivement présentes sur les machines hôtes. Il reste donc à définir un placement des composants qui tient compte de ces exigences. Ceci est possible à l'aide des contraintes de ressources définies dans un descripteur de déploiement écrit via le langage CDL que nous avons présenté dans le chapitre 6.

```
<component name="DocumentFinder">
  <deployment-context>
    <resource-constraint>
      <cpu freq="1.3" unit="GHz"
           operator="min" />
    </resource-constraint>
  </deployment-context>
</component>
```

Listing 9.3: Contrainte de ressource du composant DocumentFinder

```
<component name="DocumentBuffer>
  <deployment-context>
    <resource-constraint>
      <disk free="15" unit="MB"
        operator="min"  dir="/tmp">
    </resource-constraint>
  </deployment-context>
</component>
```

Listing 9.4: Contrainte de ressource du composant DocumentBuffer

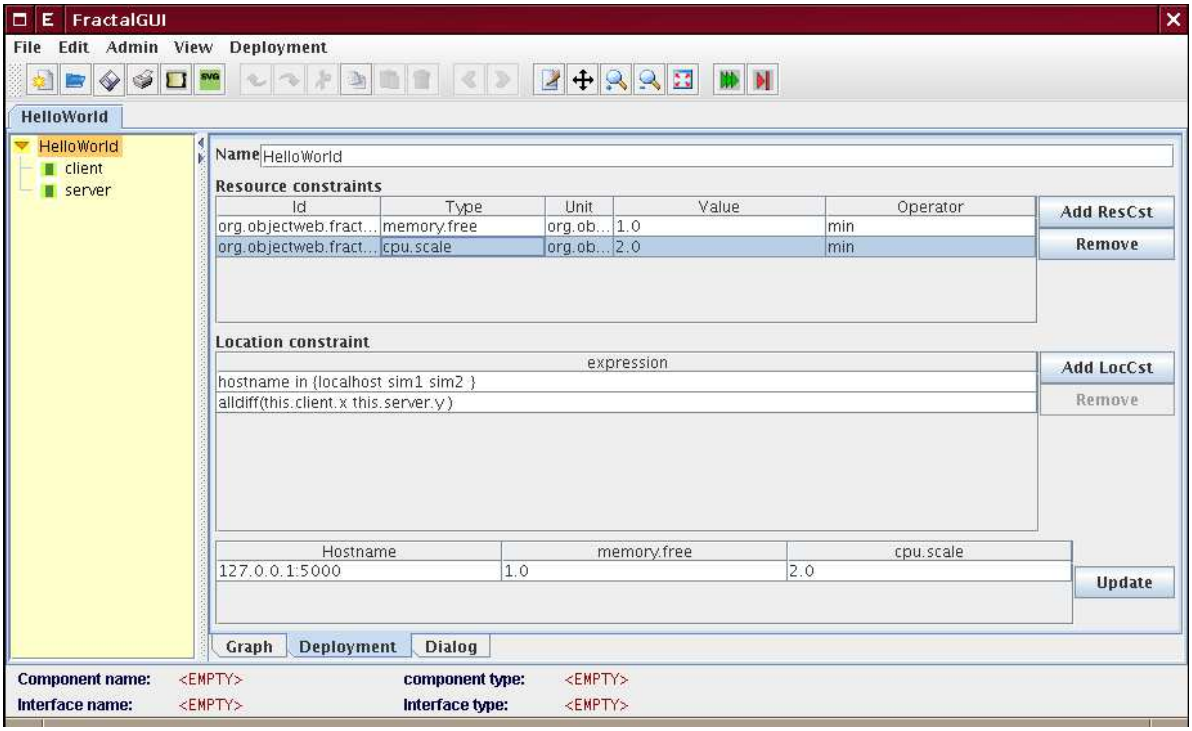


FIG. 9.2: Définition des contraintes de ressources via FractalGUI

La définition des contraintes de ressources peut également se faire graphiquement à l'aide de FractalGUI que nous avons étendu. Nous avons en effet ajouter à FractalGUI un ensemble de composants (FractalGUI est développé à l'aide de composants Fractal) permettant la spécification des contraintes de ressources. Sur la figure 9.2, nous pouvons voir que pour chaque composant de l'architecture, il est possible de spécifier ses besoins en ressources. À partir de cette même interface, le descripteur de déploiement contenant les contraintes de ressources et de localisation peut être généré au format XML.

9.1.3 Déploiement

Pendant la conception de l'application, le caractère ubiquitaire des différents composants est défini dans le descripteur d'architecture (en précisant la valeur `cubik-primitive` ou `cubik-composite` au niveau de l'élément `controller`). Il reste à spécifier la plate-forme de déploiement, c'est-à-dire l'ensemble des machines par rapport auxquelles l'application sera ubiquitaire. Avec CDL, il suffit de préciser l'ensemble des machines concernées par le déploiement au niveau du composant composite racine, soit en renseignant la balise `deployment-context` de l'identité des machines, soit en utilisant l'interface que nous avons ajouté à FractalGUI.

Le listing 9.5 précise une plate-forme de déploiement pour l'application DiapomaKer.

```
<definition name="DiapoMaker">
  [...]
 <target hostnames="dakini,ambika,mafate,babouk,deschanel"/>
</definition>
```

Listing 9.5: Définition de la plate-forme de déploiement pour l'application DiapomaKer

Une fois la cible de déploiement précisée, le déploiement de l'application ubiquitaire DiapoMaker se fait en utilisant un outil de déploiement tel que KONSOLE ou notre extension de FRACTALGUI. KONSOLE est un outil que nous avons développé et qui permet de déployer et administrer une application ubiquitaire (ou non) depuis un terminal. La figure 9.3(a) correspond à une capture d'écran de l'utilisation de KONSOLE depuis le terminal Xterm et la figure 9.3(b) illustre l'architecture de KONSOLE en termes d'assemblage de composants. KONSOLE a en effet été développé à l'aide de composants Fractal.

Ensuite, le placement des composants de l'application est initié à l'aide de KONSOLE ou de FRACTALGUI à partir desquels le descripteur d'architecture ainsi que le descripteur de déploiement peuvent être chargés. La machine à partir de laquelle se fait le déploiement est une des machines faisant partie de la plate-forme cible, i.e. une des machines listée dans l'élément `target` du composant racine. Au moment du déploiement, les autres machines peuvent être déconnectées (e.g. mise en veille ou hors de portée).

9.2 CubikADL

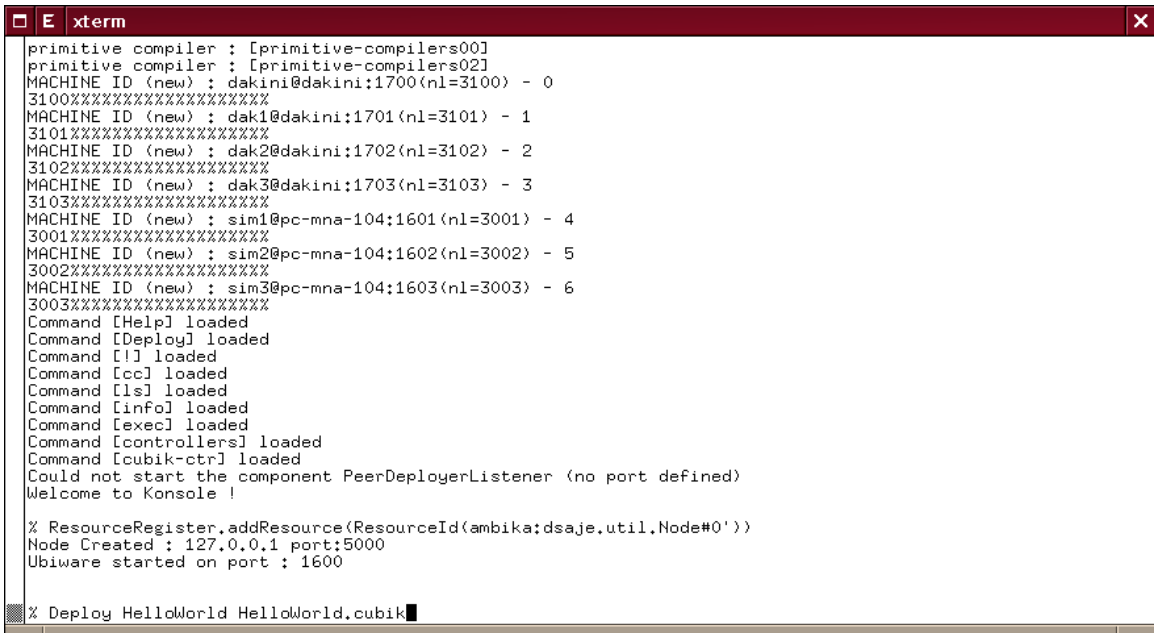
CUBIKADL est une extension du langage de description d'architecture FractalADL. Il fournit une DTD XML pour décrire les spécifications de déploiement pour des composants Fractal ubiquitaires.

Les descripteurs d'architecture et de déploiement sont utilisés à la fois pendant les phases de conception et de déploiement. Ces descripteurs utilisent une syntaxe XML. Nous avons utilisé pour la définition de ces descripteurs le langage ADL de FRACTAL. Fractal ADL est constitué de deux parties : un langage basé sur XML et une usine qui permet de traiter les définitions faites à l'aide du langage. Nous décrivons dans les sous sections suivantes le langage et l'usine définis dans Fractal ADL, puis nous détaillerons comment nous avons étendu ces éléments pour la définition et l'implantation du langage CDL.

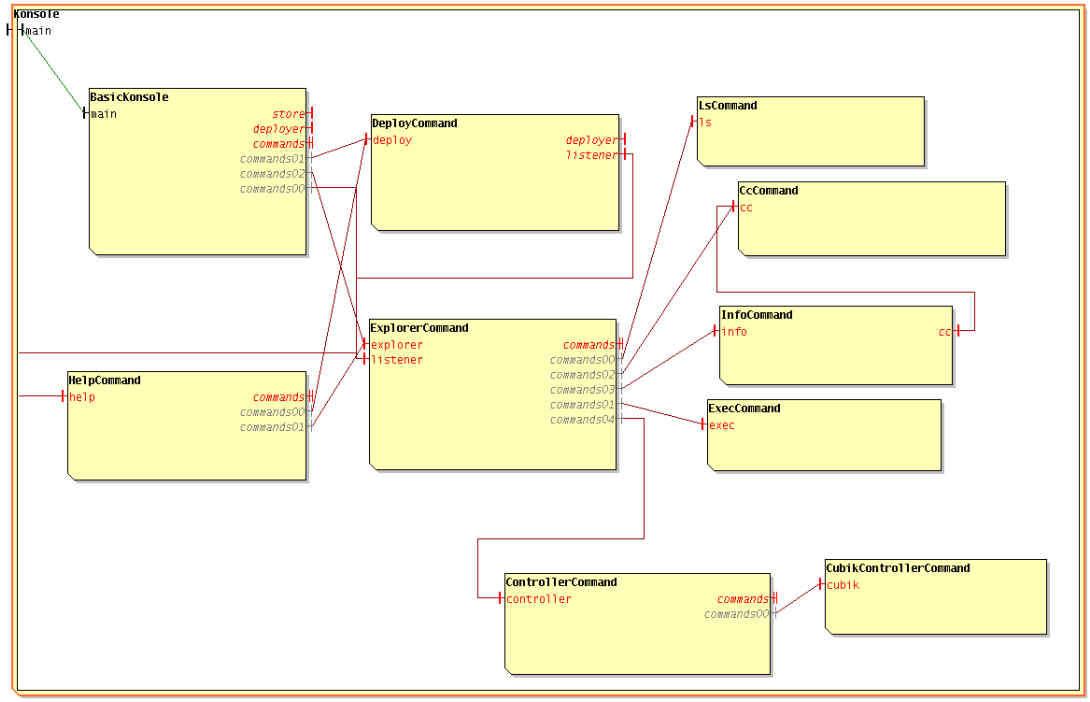
9.2.1 Le langage FractalADL

Contrairement aux autres ADL qui fixent l'ensemble des propriétés (implantation, liaisons, attributs, localisation, etc.) qui doivent être décrites pour chaque composant, l'ADL Fractal n'impose rien. Il est constitué d'un ensemble (extensible) de modules permettant la description de divers aspects de l'application.

Chaque module — à l'exception du module de base — s'applique à un ou plusieurs autres modules, c'est-à-dire rajoute un ensemble d'éléments et d'attributs XML à ces modules. Le module de base définit l'élément XML qui doit être utilisé pour démarrer la description de tout composant. Cet élément, appelé `definition`, a un attribut obligatoire, appelé `name`, qui spécifie le nom du composant décrit. Différents types de modules peuvent être définis. Un exemple typique de module est le module `containment` qui s'applique au module de base en permettant d'exprimer des relations de contenance entre composants. Ce module définit un élément XML `component` qui peut être ajouté en sous-élément d'un élément `definition` ou de lui-même pour spécifier les sous-composants d'un composant. Notons que l'élément `component` a un attribut obligatoire `name` qui permet de spécifier le nom du sous-composant. Fractal ADL définit actuellement trois autres modules qui s'appliquent soit au module de base, soit au module `containment` pour spécifier l'architecture de l'application :



(a) Konsole à l'exécution



(b) Architecture de Konsole

– le module `interface` permet de décrire les interfaces d'un composant ;
– le module `implementation` permet de décrire l'implantation des composant primitifs (i.e. une classe Java) ;
– le module `controller` permet la description de la partie contrôle des composants.

9.2.2 L'usine FractalADL

L'usine FractalADL permet de traiter les définitions écrites à l'aide du langage extensible Fractal ADL. L'usine est constituée d'un ensemble de composants Fractal qui peuvent être assemblés pour traiter les différents modules de Fractal ADL décrits précédemment :

– le composant loader analyse les définitions ADL et construit un arbre abstrait correspondant (AST pour *Abstract Syntax Tree*). L'AST implante deux API distinctes : une API générique similaire à celle de DOM qui permet de naviguer dans l'arbre ; une API typée qui varie suivant les modules qui sont utilisés dans Fractal ADL. Par exemple, si le module `interface` est utilisé, l'API typée contient des méthodes permettant de recupérer les informations sur les interfaces de composants (nom, signature, rôle, etc.).
– le composant compiler a pour rôle de générer un ensemble de tâches à exécuter à partir d'un AST. Des exemples typiques de tâches sont la création d'un composant, l'établissement d'une liaison, etc. Notons que le compiler définit également les dépendances entre les tâches qu'il crée.
– le composant builder définit un comportement concret pour les tâches créées par le compiler. Par exemple, un comportement concret d'une tâche de création de composant peut être d'instancier le composant à partir d'une classe Java.

9.2.3 Extension du langage et de l'usine FractalADL

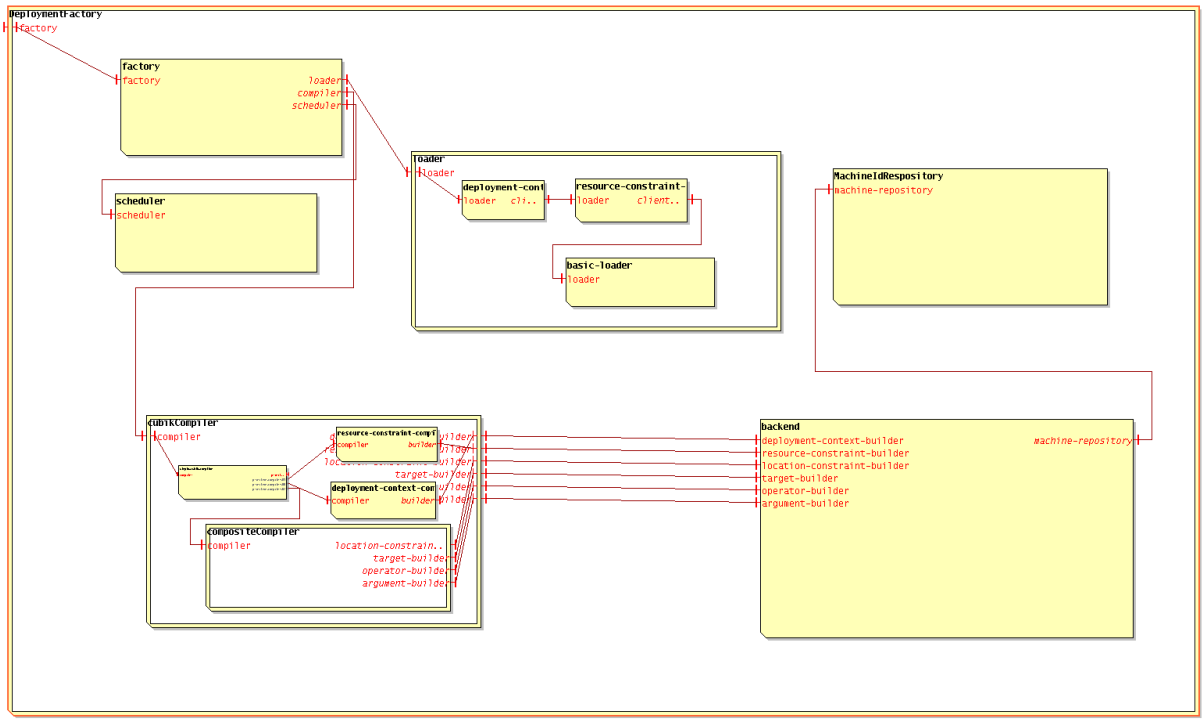
Nous décrivons maintenant comment nous avons étendu FractalADL pour l'implantation d'un compilateur pour le langage CDL.

La figure 9.3(c) représente l'architecture générale des composants responsables de la compilation et de la génération des contraintes de déploiement à partir d'un descripteur écrit avec CDL.

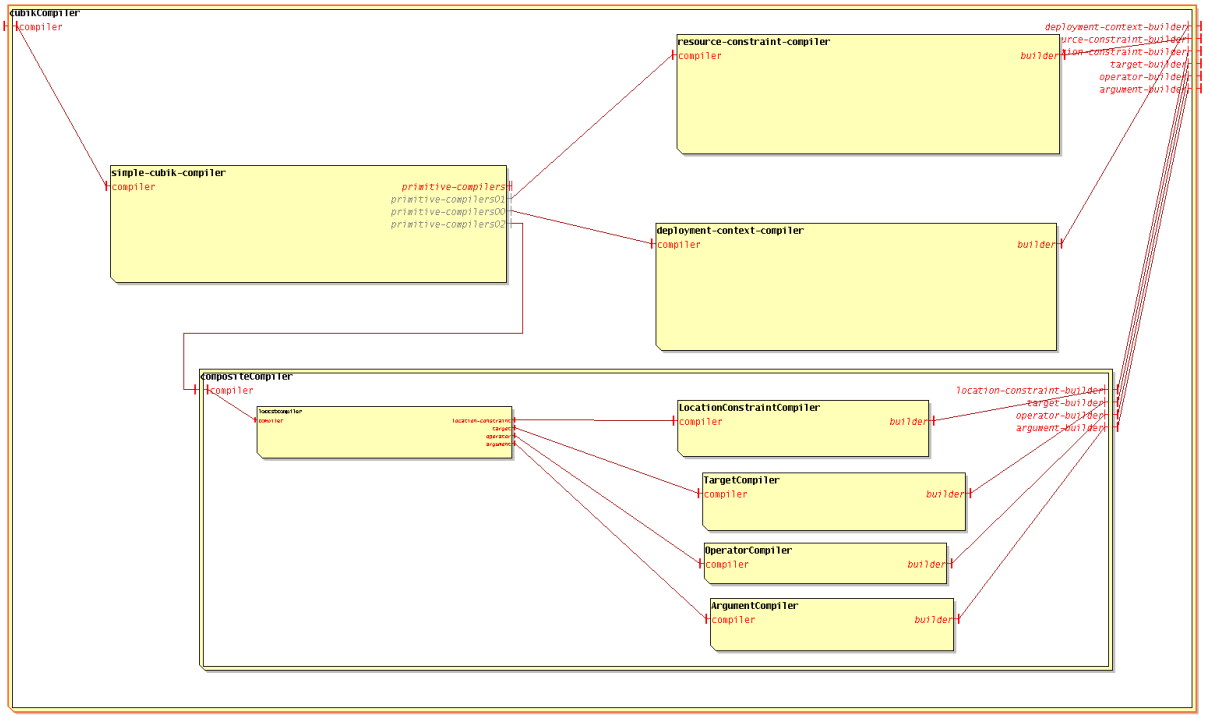
Pour chaque élément XML définit par le langage CDL (e.g. `deployment-context`, `location--constraint`, `target`, etc.) nous définissons un nouveau module. Pour chaque module nous avons rajouté un composant primitif pour la construction de l'arbre AST, i.e. un composant loader et un autre pour la génération des tâches de création, i.e. un composant compiler.

Le composant cubik-loader encapsule une chaîne de composants primitifs. Le composant le plus à droite dans la chaîne (basic-loader) est responsable de la création des AST à partir de définitions ADL. Les autres composants exécutent des vérifications et des transformations sur les AST. Chaque composant correspond à un module de l'ADL. Par exemple, le composant target-loader (prenant en charge la balise XML `target`) vérifie que l'identité de la machine (un nom de machine ou une adresse IP ou encore un nom de variable) a été spécifiée dans l'AST.

Le composant cubik-compiler (figure 9.3(d)) est un composite encapsulant un ensemble de composants compiler primitifs. Chaque compiler primitif produit des tâches correspondant à un ou plusieurs modules ADL. Par exemple, le composant resource-constraint-compiler produit des tâches de création des contraintes de ressources. Chaque compiler est associé à un composant builder qui définit un comportement pour les tâches générées par le compiler. Un builder est



(c) Composants en charge de la compilation des descripteurs de déploiement écrit avec le langage CDL et de la génération des contraintes de déploiement



(d) Zoom sur les composants responsables de la compilation des éléments du langage CDL

un composant composite qui encapsule plusieurs builders primitifs chargés des différentes tâches créées par les compilers primitifs.

Ainsi, afin de rajouter l'élément XML `<target...>` au langage, il faut définir un module target. L'écriture d'un tel module consiste à :

- définir une interface spécifiant le *nœud* XML `target` qui peut posséder l'attribut `varname` ou `hostname`. Ceci se fait en définissant des méthodes `get` et `set` correspondantes ;
- écrire des fragments de DTD spécifiant que le module `target` permet d'ajouter un élément XML à l'élément `location-constraint`.

Les différents modules que nous avons ajoutés à FractalADL sont décrits dans l'annexe A qui présente également la grammaire XML du langage CDL.

9.3 Ubiware : architecture et mise en œuvre

Le projet CUBIK fournit avec CubikADL un ensemble d'outils permettant la compilation et l'interprétation des descripteurs de déploiement écrit avec le langage CDL. Ces outils ont été intégrés à un intergiciel, baptisé UBIWARE, que nous avons développé. UBIWARE est dédié au déploiement et à la gestion des reconfigurations des applications ubiquitaires à base de composants. À partir d'une description d'architecture de composants Fractal et d'un descripteur de déploiement écrit via CDL, UBIWARE réalise :

- l'instanciation des composants définis dans le descripteur d'architecture ;
- la sélection automatique, parmi celles connectées, des machines devant héberger chaque composant, conformément aux contraintes de ressources et de localisation spécifiées dans le descripteur de déploiement ;
- l'activation (resp. la désactivation) des interfaces des composants dès lors que les composants requis sont disponibles (resp. non accessibles), i.e. la gestion automatique des déconnexions ;
- la distribution des fonctionnalités d'un composant sur un ensemble de machines si ce dernier a été défini comme étant ubiquitaire ;
- le redéploiement des composants en cas de pannes ou de violation de ressources.
- l'observation périodique des ressources requises par les composants ainsi que de la connectivité du réseau.

UBIWARE a été conçu à l'aide de composants Fractal. Nous avons regroupé les différents services nécessaires au déploiement et reconfiguration autonome dans des composants composites représentés sur la figure 9.3. Les éléments en gras correspondent aux éléments que nous avons développés, à savoir :

- le composant Saje prend l'observation périodique des ressources logicielles présentes sur une machine ;
- le composant P2P implante les primitives de communication asynchrone entre les différentes instances d'UBIWARE ;
- le composant ConstraintResolver s'appuie sur une bibliothèque de résolution de contraintes sur les entiers (Cream) pour résoudre les contraintes de ressources et de localisaton ;
- le composant Consensus implante l'algorithme de décision permettant de choisir collectivement une machine qui réalisera l'instanciation d'un composant parmi plusieurs machines candidates ;
- enfin, le composant composite Deployer coordonne le déploiement d'un composant, i.e.

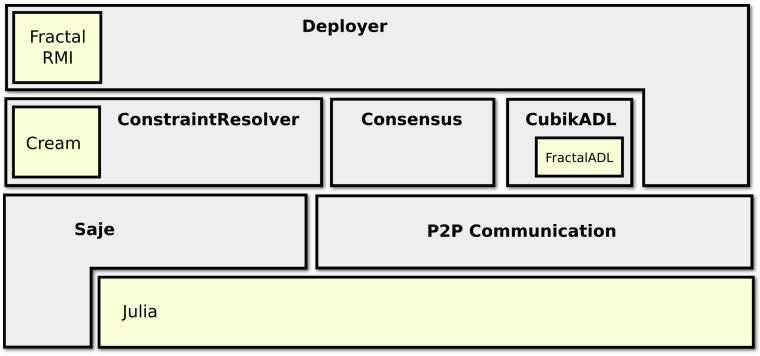


FIG. 9.3: L'architecture en couche d'UBIWARE

le chargement de ses contraintes de ressources et de localisation, l'attente du résultat du consensus, le création des composants mandataires, etc.

9.3.1 Observation des ressources

L'observation périodique des ressources locales à une machine est prise en charge par le composant Saje. L'implantation de Saje correspond à la componentisation de l'extension Java éponyme développée au sein de l'action CASA du laboratoire VALORIA. Les références dans ce mémoire à Saje correspond à la version compentisée. Saje permet de modéliser les différentes ressources offertes par le système. Saje permet de modéliser à la fois des ressources système (processeur, mémoire, disque, interface réseau...) et aussi des ressources applicatives (processus, socket, thread, répertoire...).

Pour chaque type de ressources (e.g. mémoire disponible, CPU, logiciel) que requièrent les composants, une ressource Saje est créée et une observation périodique est mise en place. Par rapport à sa version initiale, nous avons avons ajouté à Saje de nouvelles ressources afin de prendre en compte les déconnexions réseau. Nous avons défini la ressource `NetworkLink` qui modélise la liaison physique entre deux machines et qui maintient des informations sur l'état de la connexion. Nous avons également ajouté 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 réseau et par conséquent d'activer et désactiver les interfaces associées.

La figure 9.4 illustre l'architecture des composants constituant Saje qui est organisé de la manière suivante :

- toutes les ressources systèmes et logicielles sont réifiées sous forme d'objets Java afin d'être observées. Les références à ces différents objets sont prises en charge par le composant ResourceRegister ;
- le composant LocalResourceExplorer gère les sondes qui doivent être lancées. À chaque sonde correspond un type de resource.
- Le composant ResourceManager est le composant façade de Saje. C'est ce composant qui est utilisé pour la consultation de l'état d'une ressource particulière ;

Les autres composants (NodeManager et SearchFactory) correspondent à des fonctionnalités de Saje qui ne sont pas utilisées pour la mise en œuvre des concepts décrits dans cette thèse.

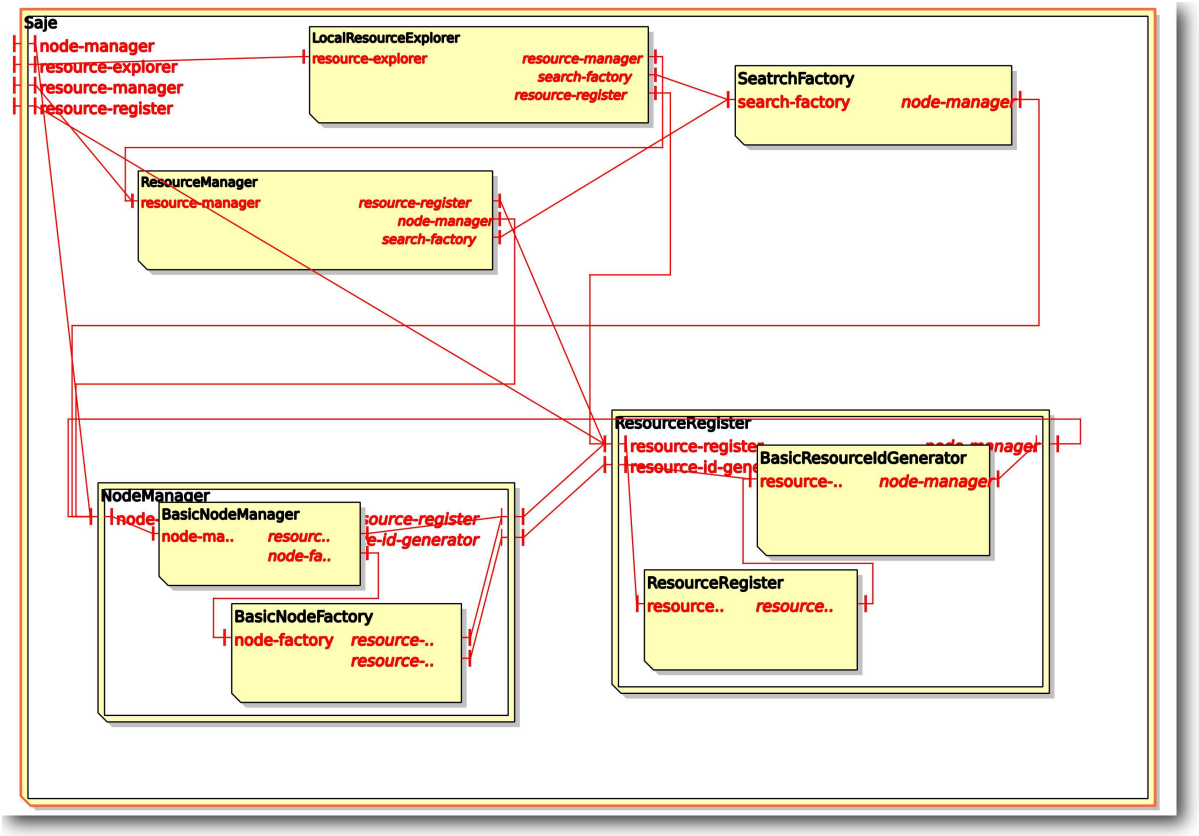


FIG. 9.4: Composant Saje, en charge de l'observation des ressources.

Sondes à lancer	MemoryProbe	Memory+CPUProbe
temps d'observation avec Saje	2ms	3,3ms

TAB. 9.1: Évaluation du temps pris pour l'obersavation des ressources avec Saje.

Nous avons mesuré le temps mis par Saje pour récupérer l'état des ressources demandées par les composants DocumentFinder et DocumentBuffer de l'application DiapomaKer. Chaque machine concernée par le déploiement de cette application réalise cette observation périodique afin de détecter si les composants peuvent être instanciés localement ou au contraire, dans le cas où le composant est déjà instancié si les ressources demandées sont toujours présentes. Les résultats fournis dans le tableau 9.1 montrent, bien que les performances dépendent des composants — plus précisément du nombre de sondes nécessaires à lancer —, que le coût associée à chaque observation demeure raisonnable.

9.3.2 Interfaces actives et contrôle de l'état des interfaces

Lorsqu'un composant distant *C* devient inaccessible, du fait par exemple d'une déconnexion réseau, les composants clients de *C* ne peuvent plus fournir correctement leurs services. Il faut alors empêcher les invocations de méthodes sur les interfaces fournies de ces composants. L'approche que nous avons adoptée est de définir un état pour les interfaces (requises et fournies) des composants (cf. chapitre 5, section 5.4). Sur une interface active, l'appel de méthode est délégué au composant implantant le service tandis qu'une invocation sera bloquante dans le cas d'une interface inactive (d'autres stratégies peuvent être définies. La section 9.3.6 décrit comment spécifier sa propre stratégie).

L'ajout de l'état actif / inactif au sein des composants Fractal / Julia a été réalisé en définissant un nouveau contrôleur pour les composants : le `cubik-controller`. À l'aide de ce contrôleur, il est possible de définir et récupérer l'état de chacune des interfaces requises et fournies d'un composant.

La mise en œuvre de contrôleurs en Fractal / Julia consiste à définir des classes *mixin* et des intercepteurs. Une classe mixin est une classe dont la super-classe est spécifiée de manière abstraite en indiquant les champs et méthodes que cette super-classe doit posséder. La classe mixin peut s'appliquer (c'est-à-dire surcharger et ajouter des méthodes) à toute classe qui possède les caractéristiques de cette super-classe.

Les classes mixin peuvent être mixées pour obtenir des classes *normales*. Ainsi la classe mixin `CubikControllerMixin` que nous avons définie s'appuie sur d'autres mixins (e.g. le mixin `UseComponentMixin` fourni par Julia). L'ensemble de ces mixins définissent le `cubik-controller`.

Le `cubik-controller` gère l'état de chaque interface du composant auquel il est attaché. Dans le cas des composants composites, le changement d'état d'une interface fournie (resp. requise) est propagé vers le `cubik-controller` des sous-composants (resp. des composants parents) liés à cette interface. En ce qui concerne les composants primitifs, cette propagation intervient uniquement vers les composants parents dès lors qu'une interface fournie change d'état.

La propagation du changement de l'état des interfaces requises et fournies passent par une gestion des dépendances entre interfaces requises et fournies. Les méthodes implantées par le

contrôleur `cubik-controller` sont spécifiées par l'interface `CubikController` (Listing 9.6).

- La méthode `getSubComponents` permet de récupérer la liste des sous-composants concernés par le changement de l'état d'une interface (cliente ou serveur).
- La méthode `getItfDependencies` permet de récupérer pour un composant donné les interfaces à activer (désactiver) sans passer par la propagation. Pour cela, la méthode `link` permet de définir une dépendance d'une interface fournie à une interface requise (et inversement). De cette manière, l'activation ou la désactivation de cette interface sera répercutée directement sur la ou les interfaces clientes dépendantes.

```
public interface CubikController {

  boolean getFcItfState(String itf, boolean server);
  void setFcItfState(String clientItfName, boolean state, boolean server);

  List getItfDependencies(String serverItf, boolean server);
  List getSubComponents(String itfName, boolean server);

  boolean dependsOn(String serverItf, String clientItf);
  void link (String clientInterface, String serverInterface, boolean serverToClient);


}
```

Listing 9.6: Interface du CUBIK-CONTROLLER

Pour intercepter les appels de méthodes, nous utilisons conjointement aux mixins, un *intercepteur*. Cet intercepteur est généré à partir du générateur d'intercepteur fourni par JULIA (la classe `InterceptorClassGenerator`). De cette manière, tous les appels de méthodes sont réifiés et contrôlé par le `cubik-controller`.

9.3.3 Résolution des contraintes de déploiement

9.3.4 Mise en œuvre

L'instanciation d'un composant sur une machine impose que d'une part les ressources qu'il requiert soient disponibles sur cette machine et d'autre part que les contraintes de localisation soient vérifiées. Notre approche a consisté à modéliser toutes ces contraintes sous la forme de CSP (chapitre 7). Pour mettre en oeuvre cette approche, nous avons utilisé CREAM, une bibliothèque Java offrant des algorithmes de résolution de problème de satisfaction de contraintes (CSP) [Tam].

CREAM permet la définition de CSP dont les variables mises en jeu sont définis sur les entiers. CREAM fournit également plusieurs stratégies de résolutions de CSP. Nous avons dû définir une bibliothèque de fonctions pour utiliser CREAM : en effet, pour générer les différents CSP définissant le problème du placement des composants, il est nécessaire de transcrire les différentes entités définies dans les descripteurs d'architecture et de déploiement en des variables prenant leur valeurs dans l'ensemble des entiers naturels (e.g. la liste des machines définissant la plate-forme cible définit le domaine de définition D de la variable *hostname*).

Lors de l'analyse du descripteur de déploiement par CUBIKADL, les objets représentant les différentes contraintes de ressources et de localisations sont créés, initialisés et ajoutés aux stores de contraintes correspondant à l'aide de l'opérateur `tell`.

```
public interface Store {
  boolean tell(DeploymentContext dc, Constraint c) ;
  boolean ask (DeploymentContext dc, Constraint c) ;
  Variable getVariable(DeploymentContext dc, String varName) ;
  boolean canLocallyBeDeployed(DeploymentContext dc, Map ctxt) ;
  void init(DeploymentContext rootDeploymentContext);
}
```

Listing 9.7: Opérations définissant les fonctionnalités d'un store de contrainte

Les résolveurs de contraintes *ResCstSolveur* et *LocCstsolveur* associés respectivement aux stores de contraintes de ressources et de localisations implantent l'opérateur `ask` indiquant si la contrainte passée en paramètre possède une solution en fonction des contraintes déjà présentes dans le store. Par exemple, pour une contrainte portant sur la quantité de mémoire disponible, le composant responsable de l'observation périodique de la mémoire teste si la quantité observée est conforme avec celle exigée de la manière suivante :

```
this.store.ask(dc,x.equalTo((int)mem.getFreeMemory())) ;
```

La méthode ASK ajoute au store l'état actuel de la contrainte et teste si le store est toujours cohérent : si la nouvelle contrainte ajoutée viole une des contraintes du store (celles spécifiées par le descripteur de déploiement), la valeur retournée indiquera que la contrainte observée ne vérifie plus les exigences de l'application. Le listing 9.8 décrit l'implantation de l'opérateur ASK à l'aide de la bibliothèque CREAM.

```
public boolean ask(DeploymentContext dc, Constraint c) {
  ...
  jp.ac.kobe_u.cs.cream.Store s = (jp.ac.kobe_u.cs.cream.Store)storeMap.get(dc) ;
  int choice = s.makeChoicePoint() ;
  boolean isConsistent = tell(dc,c) ;
  s.backtrackTo(choice) ;
  return isConsistent ;
}
```

Listing 9.8: Opérateur ASK

9.3.5 Performance

Les réseaux que nous considérons pour déployer les composants ubiquitaires rendent difficile la mise en place d'une évaluation complète du déploiement et de redéploiement de ces composants. Il serait nécessaire pour cela de contrôler parfaitement la dynamicité des différentes ressources et machines mises en jeu dans ce type de réseaux. Cependant, nous pouvons mesurer précisément la faisabilité ainsi que les performances de nos résolveurs de contraintes lorsque toutes les ressources sont disponibles. Ainsi nous avons évalué le temps pris par une machine du réseau pour calculer une solution de placement pour un assemblage de composants possédant des contraintes de localisation. La figure 9.5 montre le temps pris pour résoudre la contrainte `alldiff` imposant le placement des composants sur des machines distinctes. Nous avons considéré un réseau constitué de cent machines dans lequel le nombre de composants à instancier varie de un à cent. L'évaluation a été réalisée sur un ordinateur portable (Pentium Centrino, 1,7GHz).

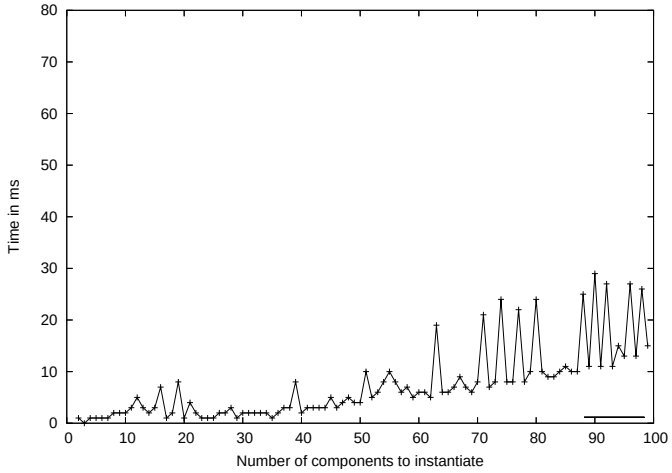


FIG. 9.5: Temps pris par une machine pour décider du placement des composants en fonction du nombre de candidatures

Cette expérimentation a permis de vérifier que le temps pris pour calculer une solution de placement, à l'aide de Cream, reste acceptable lorsque l'on met en balance le coût de communication entre les machines. Nous avons volontairement choisi dans cette évaluation la contrainte `alldiff`. C'est en effet cette dernière qui, en terme de complexité ($O(n^2)$), a le plus d'impact sur les performances de notre support de déploiement.

Dans notre approche, nous ne cherchons pas à trouver un placement optimal des différents composants. En effet, à cause de la dynamicité de l'environnement, il est difficilement concevable de définir un état stable pendant lequel toutes les ressources nécessaires seraient disponibles.

Lorsqu'une solution de placement a été exhibée par une machine, cette dernière essaye de la faire adopter par une majorité via un algorithme de consensus. La figure 9.6 représente le temps pris par notre algorithme pour valider une décision de placement en fonction du nombre de machines concernées par ce déploiement. Ces expériences ont été réalisées au sein d'un réseau filaire complètement maillé. Dans un premier temps, courbe (1), nous avons considéré qu'une seule machine se portait candidate. Nous avons ensuite pris en compte, courbes (2) et (3), des candidatures concurrentes. Pour ce faire, nous avons modifié l'algorithme de [MRR01] afin de gérer des lancements concurrents de consensus ainsi que l'arrivée de nouvelles machines pendant un consensus. Cette étude a permis de vérifier que des détections simultanées de solution de placement n'entraînaient pas de surcoût prohibitif.

9.3.6 Mise en œuvre des liaisons distantes

Nous avons défini dans le chapitre 5 de ce mémoire un schéma de distribution permettant de rendre les composants (composites et primitifs) ubiquitaires. Ces composants définissent des communications distantes pour rendre leur services sur l'ensemble des machines du réseau. La mise en œuvre de la communication distante s'appuie sur deux concepts :

- la duplication de la membrane des composites, contenant les informations de placement des sous-composants, sur un ensemble de machines ;
- et les composants mandataires.

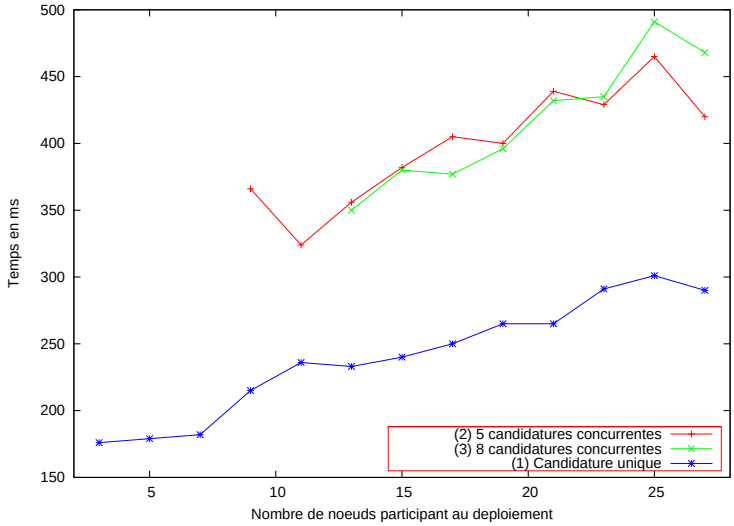


FIG. 9.6: Évaluation de la durée d'une prise de décision pour le placement de composants

Nous avons ajouté à Julia le support pour des composants Fractal ubiquitaires. La première modification que nous avons apportée consiste à ajouter au niveau de la membrane des composants (au niveau du `cubik-controller` l'information de placement des composants, i.e. la liste des machines sur lesquelles le composant est rendu ubiquitaire ainsi que la localisation des sous-composants. Sur toutes les machines où un composant ne peut être instancié (i.e. des ressources sont manquantes ou la machine ne fait pas partie des cibles de déploiement), un composant mandataire est créé. Ces composants mandataires sont générés dynamiquement à l'aide de la bibliothèque ASM [30] et possèdent comme interfaces fournies et requises, les mêmes que celles du composant qu'ils représentent.

La figure 9.7 illustre la structure d'un composant mandataire généré à partir de la description du composant distant qu'il représente. Ce composant est un composite et possède les mêmes interfaces fournies et requises que le composant distant. Comme nous l'avons justifié dans le chapitre 5, l'intégration d'un composant ayant la même « forme » (en termes d'interfaces) que le composant distant permet de répercuter les reconfigurations et les (re/dé)connexions réseau de manière uniforme sur toutes les entités de l'architecture. Ainsi si le composant distant n'est plus accessible, l'interface requise du composant mandataire est désactivée ce qui va désactiver ensuite ses interfaces fournies.

L'architecture interne du mandataire comprend un composant primitif également généré dynamiquement. Ce composant par défaut n'effectue qu'une simple délégation vers le composant distant. Il est prévu que l'on puisse définir ici des stratégies d'adaptation spécifiques.

La réalisation de la liaison distante entre un composant client et serveur se fait en deux temps :

- tout d'abord chez le client, un composant mandataire représentant le composant distant est créé. Les liaisons du composant client vers le composant mandataire peuvent également être réalisées. La mise en place de ces liaisons ne diffère pas de la situation où à la place du composant mandataire avait été instancié le composant serveur ;
- puis, des objets talons et squelettes sont respectivement créés sur la machine cliente et

[30] `http://asm.objectweb.org/`

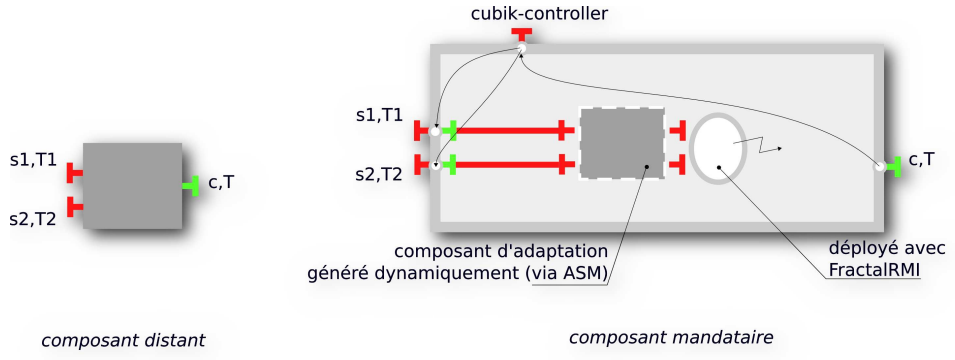


FIG. 9.7: Interfaces requises et fournies d'un composant mandataire

serveur à l'aide de FractalRMI[31]. FractalRMI, fourni dans la distribution Java de Fractal, offre un certain nombre de composants permettant la création de liaisons distantes entre composants Fractal.

Le listing 9.9 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() ;
}
```

Listing 9.9: Introspection sur l'état d'une interface

[31] http://fractal.objectweb.org/fractalrmi/index.html

Conclusion

L'émergence des réseaux dynamiques, c'est-à-dire des réseaux constitués d'équipements hétérogènes, volatiles et mobiles, amène à repenser notre utilisation des logiciels. L'omniprésence des équipements informatiques fait émerger en effet un nouveau besoin, celui de l'ubiquité des services offerts par les applications. Cependant, l'accès aux fonctionnalités d'une même application depuis n'importe quel équipement qui nous entoure reste difficile à concevoir et à mettre en œuvre dans des réseaux dynamiques. Dans de tels réseaux, les machines sont mobiles et volatiles et aucune hypothèse sur la disponibilité des ressources requises ne peut être faite. La connectivité entre les différentes machines fluctue et ne peut donc être garantie.

L'objectif de ce travail de thèse était de proposer des abstractions et des mécanismes facilitant la conception et le déploiement d'applications ubiquitaires dans des réseaux dynamiques. Nous nous sommes focalisés sur les applications construites à l'aide de composants logiciels dont les avantages ont maintenant été démontrés tant pour faciliter la conception des applications que pour contrôler leur maintenance et évolution.

Notre travail a permis de dégager des contributions dans plusieurs volets que nous développons ci-dessous.

Composants logiciels ubiquitaires

Notre première contribution porte sur la définition d'un schéma de distribution dans un modèle de composants hiérarchiques. La mise à disposition des fonctionnalités d'un composant sur un ensemble de machines permet de rendre ce composant ubiquitaire. Nous avons exploité la structure hiérarchique du modèle à la fois dans la gestion de la distribution et dans celle des déconnexions réseau. 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 déployés sur une unique machine. La gestion décentralisée de l'architecture du composite offre un support pour la prise en compte des déconnexions réseau, déconnexions qui sont susceptibles d'entraîner des ruptures de liaisons entre sous-composants.

Nous avons ajouté au modèle hiérarchique la notion d'interface active 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.

Spécification du déploiement sous forme de contraintes

Dans des environnements constitués d'équipements mobiles et aux ressources fluctuantes, le choix du placement de chaque composant est rendu difficile voire impossible dans des phases antérieures au déploiement. En effet, dans ce genre de réseau, aucune hypothèse ne peut être faite sur la présence d'une machine particulière ou sur la disponibilité d'une ressource requise par un composant.

Nous avons proposé de différer la détermination de la correspondance entre composants et machines au moment de l'exécution et de fonder sa spécification sur un langage de description d'architecture baptisé CDL (*Constrained Deployment Language*). Ainsi au moment de la conception de l'application, chaque composant est annoté des propriétés des ressources que doivent satisfaire les machines pouvant les héberger. Il s'agit d'une approche purement déclarative. Afin de contrôler le placement des composants, CDL permet l'utilisation de variables permettant de spécifier des contraintes de localisation comme par exemple : deux composants doivent résider sur des machines distinctes ou encore, tel composant doit être héberger sur la machine ayant la plus grande puissance de calcul. Ainsi, il n'est plus nécessaire de spécifier des noms ou adresses de machines explicitement dans la description du déploiement. Le placement des composants est décidé à l'exécution, conformément aux contraintes associées aux composants.

Déploiement propagatif et autonome

Du fait du caractère dynamique des plates-formes sur lesquelles doivent s'exécuter les applications considérées, des ressources nécessaires à certains composants peuvent ne pas être disponibles et certaines machines peuvent ne pas être présentes au moment du déploiement. Le processus de déploiement proposé permet l'instanciation des composants au fur et à mesure de la découverte des ressources présentes dans l'environnement (par exemple lorsque de nouvelles machines se connectent au réseau). Ce déploiement, qualifié de *propagatif*, tient compte des contraintes de ressources et de localisation spécifiées via CDL. Cette installation progressive permet de bénéficier au plus tôt des fonctionnalités de l'application, la gestion des dépendances au sein de l'architecture autorisant cette dernière à s'exécuter en mode dégradé.

Cependant, après l'installation et l'instanciation d'un composant, il se peut que la machine choisie pour son hébergement ne soit plus en mesure de fournir les ressources nécessaires pour celui-ci (par exemple, la quantité de mémoire disponible n'est plus suffisante). Lorsque les contraintes de déploiement ne sont plus vérifiées, une nouvelle machine doit être sélectionnée. En diffusant au sein du réseau cet ordre de redéploiement, le processus de déploiement se retrouve dans la phase *propagative*, rendant de fait le redéploiement des composants totalement automatique.

Consensus dans des environnements dynamiques

Le processus de déploiement présenté dans ce mémoire nécessite une prise de décision entre les différentes machines constituant le réseau. En effet, plusieurs machines peuvent héberger un même composant et il faut empêcher que l'instanciation d'un composant ne soit réalisée par toutes ces machines candidates : l'architecture obtenue ne serait plus conforme à sa spécification. Ce problème d'accord est rendu difficile du fait de la dynamicité du ré-

seau dans lequel des îlots peuvent apparaître. Nous avons adapté un algorithme de consensus tolérant les pannes à notre problématique de déploiement. Ainsi, notre processus de déploiement garantit que les décisions portant sur l'instanciation des composants et d'une manière générale sur toutes les opérations modifiant l'architecture de l'application seront validées par toutes les machines du réseau. L'algorithme de consensus a été entièrement implanté au sein de notre intergiciel de déploiement.

Programmation par contraintes pour les adaptations orientées architecture

L'architecture de l'application, spécifiée via CDL, constitue un objectif à atteindre par le processus de déploiement. Les fluctuations de l'environnement doivent être prises en compte pour d'une part profiter des nouvelles ressources afin d'instancier de nouveaux composants, et d'autre part pour redéployer les composants dès lors que des contraintes de ressources ne sont plus respectées. Le moteur qui guide les choix d'instanciation des composants et de leur redéploiement est constitué d'un ensemble de résolveurs de contraintes. Ces derniers ont pour rôle de résoudre les problèmes de satisfaction de contraintes (CSP) générés à partir de la spécification de l'assemblage des composants, des contraintes de ressources et des contraintes de localisation. Ainsi, lorsqu'une nouvelle machine se connecte ou bien qu'un composant n'est plus accessible, ces événements sont pris en compte directement au niveau des résolveurs qui peuvent alors générer une nouvelle configuration. Nous avons utilisé la bibliothèque Cream pour ajouter à notre intergiciel les fonctions relatives à la résolution des contraintes de ressources et de localisation. Nous avons également développé une suite d'outils qui permet la compilation des descripteurs de déploiement écrits via CDL. Ces outils ont été réalisés en ajoutant des extensions à FractalADL.

Un intergiciel pour le support des applications dans des environnements dynamiques

Le support et le déploiement des composants sont assurés par un intergiciel, baptisé Ubiware, capable de prendre en compte les caractéristiques logicielles et matérielles des machines. Cet intergiciel, à partir de la description des besoins en ressources des composants, réifie ces ressources sous la forme d'objets (e.g. processeur, mémoire, batterie) et permet d'observer leur état. Comme nous l'avons vu précédemment, des reconfigurations de l'architecture ainsi qu'un redéploiement automatique des composants sont réalisés dès lors que l'état d'une ressource sur un équipement est en contradiction avec les contraintes de déploiement. L'intergiciel que nous avons développé est complètement modulaire : ubiware est en effet conçu à l'aide de composants Fractal. Les entités prenant en charge la distribution des composants sont générées dynamiquement et des politiques d'adaptation peuvent être précisées afin de réagir de manière spécifique aux déconnexions réseau.

Perspectives

Nous avons proposé dans ce mémoire des concepts ainsi que des mécanismes facilitant la conception et le déploiement d'applications ubiquitaires. Cependant, il reste encore beaucoup de travail pour trouver sur les « étagères de composants » de telles applications. Le

travail que nous avons mené durant cette thèse ouvre plusieurs perspectives de recherche qui permettraient de faciliter davantage la mise en œuvre d'applications ubiquitaires. Dans cette optique, nous donnons ici quelques pistes de travail sur l'extension de notre approche.

Prise en compte des styles architecturaux

Les applications ubiquitaires que nous avons considérées dans nos travaux sont définies par un assemblage de composants tel que, à chaque composant spécifié dans le descripteur d'architecture doit correspondre une instance créée par notre plate-forme de déploiement. La description de l'application est donc figée en termes de nombre de composants. Le langage de description d'architecture que nous avons considéré ne permet pas la définition d'architecture dynamique, c'est-à-dire des architectures où le nombre de composants n'est pas fixé mais guidé par des contraintes. Par exemple, des contraintes définissant le nombre maximum de composants d'un certain type ou encore le nombre de liaisons qu'un composant peut accepter. Ces aspects dynamiques peuvent être pris en compte dans les langages de description d'architecture comme DynamicWright [ADG98], C2 [MORT96], ou ASL [Ley04]. Dans ces langages il est possible de définir des styles architecturaux, c'est-à-dire un ensemble de règles qui dicte comment la composition doit être faite. L'utilisation des styles architecturaux simplifie non seulement la définition des assemblages de composants mais permet également la spécification de propriétés architecturales, ces dernières garantissant une certaine qualité de l'application.

La prise en compte des styles architecturaux dans notre travail amène à définir ou réutiliser un langage de description d'architecture permettant l'expression de contraintes sur les éléments d'une architecture de composants. Une première extension est alors nécessaire au niveau de la génération des problèmes de satisfaction de contraintes (cf. chapitre 7) pour que les contraintes et résolveurs de contraintes prennent en considération les nouvelles contraintes architecturales. Nous avons commencé à travailler sur cet aspect dans [THK07] où un langage de description d'architecture basé sur OCL a été utilisé. Nous avons déjà formalisé sous la forme de CSP certaines contraintes identifiées dans des styles architecturaux : celles portant sur le nombre d'instances de composants et sur le nombre de liaisons autorisé entre composants.

Généralisation du schéma de distribution des composants ubiquitaires

Le schéma de distribution des composants composites et primitifs que nous avons défini (cf. chapitre 5) ne considère pas la réplication des composants primitifs. Ces composants sont en effet rendus ubiquitaires en créant, sur chaque machine du réseau, des composants mandataires qui exhibent les fonctionnalités du composant qu'ils représentent. Dès lors qu'un composant primitif n'est plus accessible de son représentant, ses fonctionnalités ne sont plus disponibles sur la machine hébergeant le composant mandataire. Nous pouvons envisager de répliquer les composants primitifs sur l'ensemble (ou une partie) des machines par rapport auxquelles ils sont définis comme ubiquitaires afin de disposer de leurs services, indépendamment des déconnexions réseau. Le problème qui se pose alors est que les données manipulées par les composants primitifs sont dupliquées et prendront des valeurs différentes selon les machines où elles se trouvent. Il peut être nécessaire de maintenir la cohérence de ces données. Pour cela, nous envisageons de définir des composants spécialisés dans la gestion de la

cohérence de données. Ces composants, spécifiques à chaque application, devront ensuite être intégrés à l'architecture de l'application afin que les composants qui sont répliqués puissent être gérés par ces composants spécifiques.

Contrôle des composants ubiquitaires

Le modèle de composant hiérarchique sur lequel nous nous appuyons est largement inspiré du modèle de composant Fractal. Un des aspects de ce modèle que nous n'avons pas conservé est l'utilisation des contrôleurs définis au niveau de la membrane des composants. Ainsi, nous avons retiré la plupart des contrôleurs présents par défaut sur un composant (c'est-à-dire les contrôleurs prenant en charge la gestion du cycle de vie des composants, la modification de leurs attributs, la modification des liaisons, etc.). La raison principale de ce retrait était que l'utilisation de ces contrôleurs aurait nécessité une synchronisation des données qu'ils manipulent puisque la membrane des composants composites est dupliquée sur plusieurs machines.

Comme nous l'envisageons dans le cadre de la duplication des composants primitifs (voir paragraphe précédent), des composants spécifiques qui traitent de la cohérence des informations manipulées par les contrôleurs peuvent être définis. La plate-forme Koch (une autre implantation Java du modèle de composant Fractal), dans laquelle les membranes des composants sont programmées en tant que composants, pourrait alors servir à gérer la cohérence des contrôleurs de la même manière que celles des composants primitifs.

Optimisation du déploiement de composants

Le processus de déploiement propagatif que nous avons proposé dans ce mémoire permet l'instanciation des composants au fur et à mesure de la disponibilité des ressources ainsi que l'accessibilité des machines. Notre proposition permet de décider pour chaque composant d'une machine hôte. Le redéploiement d'un composant, c'est-à-dire la remise en question de la décision de son placement est réalisé lorsqu'il tombe en panne ou bien lorsque la machine qui l'héberge ne dispose plus des ressources nécessaires à son fonctionnement. Lors du calcul d'une nouvelle solution de placement, le processus que nous avons présenté considère le placement des composants déjà instanciés comme définitif. De ce fait, le processus de redéploiement ne reconsidère pas le placement des composants déjà instanciés. Cela peut allonger le temps de calcul d'une nouvelle solution de placement pour le composant à redéployer. Par exemple, la présence d'une contrainte de localisation impliquant l'opérateur `alldiff` empêche les machines déjà choisies pour l'hébergement des composants concernés par cette contrainte d'être sélectionnées pour le placement du composant à redéployer.

Nous envisageons donc d'étendre notre algorithme de redéploiement en considérant qu'un composant déjà instancié et pour lequel les contraintes de ressources sont toujours vérifiées peut être redéployer. Nous pourrions également alors considérer le redéploiement d'un composant lorsqu'une machine offrant de meilleures conditions d'hébergement devient accessible.

A

Grammaire XML du langage CDL

La grammaire XML donnée ci-dessous correspond à l'extension que nous avons apportée à FractalADL pour la prise en compte du langage CDL. Cette grammaire se présente en deux parties. La première contient la spécification des nouveaux modules ajoutés à FractalADL et la seconde définit les éléments et la grammaire du langage CDL.

```
<?xml version="1.0" encoding="ISO-8859-1" ?>

<!-- A DTD that includes only the minimum necessary modules -->
<!-- to define component deployment context -->

<!-- ********************************************************************** -->
<!--                            AST nodes definitions                       -->
<!-- ********************************************************************** -->

<?add ast="definition" itf="org.objectweb.fractal.adl.Definition" ?>
<!-- components module -->
<?add ast="component"  itf="org.objectweb.fractal.adl.components.Component" ?>
<?add ast="definition"
      itf="org.objectweb.fractal.adl.components.ComponentDefinition" ?>

<!-- bindings module -->
<?add ast="binding"    itf="org.objectweb.fractal.adl.bindings.Binding" ?>
<?add ast="definition"
      itf="org.objectweb.fractal.adl.bindings.BindingContainer" ?>
<?add ast="component"
      itf="org.objectweb.fractal.adl.bindings.BindingContainer" ?>

<!-- deployment context module -->
<?add ast="deploymentContext"
      itf="fr.valoria.cubik.adl.deploymentcontext.DeploymentContext" ?>
<?add ast="definition"
      itf="fr.valoria.cubik.adl.deploymentcontext.DeploymentContextContainer" ?>
<?add ast="component"
```

```
      itf="fr.valoria.cubik.adl.deploymentcontext.DeploymentContextContainer" ?>


<!-- resource constraint module -->
<?add ast="resourceConstraint"
      itf="fr.valoria.cubik.adl.resourceconstraints.ResourceConstraint" ?>
<?add ast="deploymentContext"
      itf="fr.valoria.cubik.adl.resourceconstraints.ResourceConstraintContainer"?>

<!-- location constraint module -->
<?add ast="locationConstraint"
      itf="fr.valoria.cubik.adl.locationconstraints.LocationConstraint" ?>
<?add ast="deploymentContext"
      itf="fr.valoria.cubik.adl.locationconstraints.LocationConstraintContainer"?>

<!-- Target module -->
<?add ast="target" itf="fr.valoria.cubik.adl.locationconstraints.Target" ?>
<?add ast="locationConstraint"
      itf="fr.valoria.cubik.adl.locationconstraints.TargetContainer" ?>

<?add ast="operator"
      itf="fr.valoria.cubik.adl.locationconstraints.Operator" ?>
<?add ast="locationConstraint"
      itf="fr.valoria.cubik.adl.locationconstraints.OperatorContainer" ?>

<?add ast="target" itf="fr.valoria.cubik.adl.locationconstraints.Target" ?>
<?add ast="locationConstraint"
      itf="fr.valoria.cubik.adl.locationconstraints.TargetContainer" ?>

<?add ast="argument"
      itf="fr.valoria.cubik.adl.locationconstraints.Argument" ?>
<?add ast="operator"
      itf="fr.valoria.cubik.adl.locationconstraints.ArgumentContainer" ?>


<!-- ********************************************************************* -->
<!--                    Mapping from XML names to AST names                -->
<!-- ********************************************************************* -->

<?map xml="binding.client" ast="binding.from" ?>
<?map xml="binding.server" ast="binding.to" ?>

<?map xml="deployment-context"      ast="deploymentContext" ?>
<?map xml="resource-constraint"     ast="resourceConstraint" ?>
<?map xml="location-constraint"     ast="locationConstraint" ?>
<?map xml="arg"     ast="argument" ?>


<!-- ********************************************************************* -->
<!--                            XML syntax definition                      -->
<!-- ********************************************************************* -->
<!ELEMENT definition (comment*,interface*,component*,binding*,content?,
                      attributes?,controller?,template-controller?,logger?,
                      virtual-node?, coordinates*,deployment-context?) >
<!ATTLIST definition
  name CDATA #REQUIRED
```

```
   arguments CDATA #IMPLIED
   extends CDATA #IMPLIED
>

<!ELEMENT component (comment*,interface*,component*,binding*,content?,attributes?,
                     controller?,template-controller?,logger?,virtual-node?,
                     coordinates*,deployment-context?) >
<!ATTLIST component
   name CDATA #REQUIRED
   definition CDATA #IMPLIED
>

<!ELEMENT binding (comment*) >
<!ATTLIST binding
   client CDATA #REQUIRED
   server CDATA #REQUIRED
>


<!-- XML element related to context awareness -->

<!ELEMENT deployment-context (location-constraint?,resource-constraint+) >
<!ATTLIST deployment-context
         component CDATA #IMPLIED
         version CDATA #IMPLIED
>

<!ELEMENT resource-constraint EMPTY >
<!ATTLIST resource-constraint
   type CDATA #REQUIRED
   value CDATA #REQUIRED
   operator CDATA #REQUIRED
>

<!ELEMENT location-constraint (target*,operator*) >

<!ELEMENT target EMPTY>
<!ATTLIST target
   hostname CDATA #IMPLIED
   inetAddress CDATA #IMPLIED
   varname CDATA #IMPLIED
>

<!ELEMENT operator (arg+) >
<!ATTLIST operator
   name CDATA #REQUIRED
>

<!ELEMENT arg EMPTY>
<!ATTLIST arg
   name CDATA #REQUIRED
>
```

Listing A.10: Grammaire XML du langage CDL

Bibliographie

[AAG93] G. Abowd, R. Allen, et D. Garlan. « Using style to understand descriptions of software architecture ». *SIGSOFT Softw. Eng. Notes*, 18(5) :9–20, 1993.

[Abd06] T. Abdellatif. « *Apport des architectures à composants pour l'administration des intergiciels* ». Thèse de doctorat, Institut National Polytechnique de Grenoble, septembre 2006.

[ADG98] R. Allen, R. Douence, et D. Garlan. « Specifying and analyzing dynamic software architectures ». Dans *Proceedings of the 1998 Conference on Fundamental Approaches to Software Engineering (FASE'98)*, Lisbonne, Portugal, mars 1998.

[AMP04] E. Anceaume, E. Mourgaya, et P. R. Parvédy. « Unreliable distributed timing scrutinizers to converge towards conditions ». *Studia Universalis Informatica*, 3(1) :17–36, 2004.

[Bar05] O. Barais. « *Construire et maîtriser l'évolution d'une architecture logicielle à base de composants* ». Thèse de doctorat, Laboratoire d'Informatique Fondamentale de Lille, Lille, France, novembre 2005.

[BBC+03] D. F. Bantz, C. Bisdikian, D. Challener, J. P. Karidis, S. Mastrianni, A. Mohindra, D. G. Shea, et M. Vanover. « Autonomic personal computing ». *IBM Systems Journal*, 42(1) :165–176, 2003.

[BCL+04] E. Bruneton, T. Coupaye, M. Leclercq, V. Quéma, et J.-B. Stefani. « An open component model and its support in Java ». Dans *Proceedings of the International Symposium on Component-based Software Engineering (CBSE7)*, numéro 3054 dans LNCS, Edinburgh, Écosse, mai 2004.

[BCM03] F. Baude, D. Caromel, et M. Morel. « From distributed objects to hierarchical grid components ». Dans *CoopIS/DOA/ODBASE*, pages 1226–1242, 2003.

[BD07] G. Bell et P. Dourish. « Yesterday's tomorrows : notes on ubiquitous computing's dominant vision ». *Personal and Ubiquitous Computing*, 11(2) :133–143, 2007.

[BdHT06] S. Bouchenak, N. de Palma, D. Hagimont, et C. Taton. « Autonomic management of clustered applications ». Dans *Proceedings of the IEEE International Conference on Cluster Computing (Cluster'06)*, Barcelone, Espagne, septembre 2006.

[Ber96] P. A. Bernstein. « Middleware : a model for distributed system services ». *Commun. ACM*, 39(2) :86–98, 1996.

[BHSR04] C. Becker, M. Handte, G. Schiele, et K. Rothermel. « PCOM - a component system for pervasive computing ». Dans *Second IEEE International Conference*

on Pervasive Computing and Communications (PerCom'04), pages 67–76, Orlando, Floride, mars 2004. IEEE Computer Society Press.

[BJC05] T. V. Batista, A. Joolia, et G. Coulson. « Managing dynamic reconfiguration in component-based systems ». Dans *EWSA*, pages 1–17, 2005.

[BMK+00] B. Brumitt, B. Meyers, J. Krumm, A. Kern, et S. A. Shafer. « EasyLiving : technologies for intelligent environments ». Dans *HUC*, pages 12–29, 2000.

[BO83] M. Ben-Or. « Another advantage of free choice : completely asynchronous agreement protocols ». Dans *PODC*, pages 27–30, 1983.

[BS03] C. Becker et G. Schiele. « Middleware and application adaptation requirements and their support in pervasive computing ». Dans *ICDCS Workshops*, pages 98–103, 2003.

[BSGR03] C. Becker, G. Schiele, H. Gubbels, et K. Rothermel. « BASE - a micro-broker-based middleware for pervasive computing ». Dans *PerCom*, pages 443–451, 2003.

[BTAB05] A. Beloued, C. Taconnet, D. Ayed, et G. Bernard. « Placement automatique des composants lors du déploiement d'applications à base de composants ». Dans *Actes de RenPar'16, CFSE'4, SympAAA'2005*, Le Croisic, Presqu'île de Guérande, France, avril 2005.

[CASD95] F. Cristian, H. Aghili, R. Strong, et D. Dolev. « Atomic broadcast : from simple message diffusion to byzantine agreement ». *Inf. Comput.*, 118(1) :158–179, 1995.

[CBCP02] G. Coulson, G. S. Blair, M. Clarke, et N. Parlavantzas. « The design of a configurable and reconfigurable middleware platform ». *Distributed Computing*, 15(2) :109–126, 2002.

[CCB02] D. Conan, S. Chabridon, et G. Bernaro. « Disconnected operations in mobile environments ». Dans *IPDPS '02 : Proceedings of the 16th International Parallel and Distributed Processing Symposium*, page 118, Washington, DC, USA, 2002. IEEE Computer Society.

[CCM+05] P. Costa, G. Coulson, C. Mascolo, G. P. Picco, et S. Zachariadis. « The RUNES middleware : a reconfigurable component-based approach to networked embedded systems ». Dans *Proceedings of the 16th Annual IEEE International Symposium on Personal Indoor and Mobile Radio Communications (PIMRC'05)*, Berlin, Allemagne, septembre 2005.

[CCM+06] P. Costa, G. Coulson, C. Mascolo, L. Mottola, G. P. Picco, et S. Zachariadis. « A reconfigurable component-based middleware for networked embedded systems ». *Journal of Wireless Information Networks*, 2006.

[CELQ05] E. Cecchet, H. Elmeleegy, O. Layaïda, et V. Quéma. « Implementing probes for J2EE cluster monitoring ». *Studia Informatica*, 4(1), 2005.

[CGLSM03] L. Courtrai, F. Guidec, N. Le Sommer, et Y. Mahéo. « Resource management for parallel adaptive components ». Dans *Workshop on Java for Parallel and Distributed Computing, IPDPS'03*, pages 134–141, Nice, France, avril 2003. IEEE CS.

[CGS+02] S.-W. Cheng, D. Garlan, B. R. Schmerl, J. P. Sousa, B. Spitznagel, P. Steenkiste, et N. Hu. « Software architecture-based adaptation for pervasive systems ». Dans *ARCS*, pages 67–82, 2002.

[CH04] H. Cervantes et R. S. Hall. « Autonomous adaptation to dynamic availability using a service-oriented component model ». Dans *ICSE*, pages 614–623, 2004.

[CKV98] D. Caromel, W. Klauser, et J. Vayssière. « Towards seamless computing and metacomputing in Java ». *Concurrency - Practice and Experience*, 10(11-13) :1043–1061, 1998.

[CQSS07] T. Coupaye, V. Quéma, L. Seinturier, et J.-B. Stefani. « *Intergiciel et construction d'applications réparties* », Chapitre Le système de composants Fractal. janvier 2007. `http://sardes.inrialpes.fr/ecole/livre/pub/`.

[CRS07] D. Conan, R. Rouvoy, et L. Seinturier. « Scalable processing of context information with COSMOS ». Dans *DAIS*, pages 210–224, 2007.

[CSWW04] D. M. Chess, A. Segal, I. Whalley, et S. R. White. « Unity : experiences with a prototype autonomic computing system ». Dans *ICAC*, pages 140–147, 2004.

[CT96] T. D. Chandra et S. Toueg. « Unreliable failure detectors for reliable distributed systems ». *J. ACM*, 43(2) :225–267, 1996.

[DC01] D. Diaz et P. Codognet. « Design and implementation of the GNU Prolog system ». *Journal of Functional and Logic Programming*, 2001(6), 2001.

[DCD06] M. Desertot, H. Cervantes, et D. Donsez. « FROGi : Fractal components deployment over OSGi ». Dans *Software Composition*, pages 275–290, 2006.

[DES01] F. Duclos, J. Estublier, et R. Sanlaville. « Architectures ouvertes pour l'adaptation des logiciels ». *Revue Génie Logiciel*, 58 :19–25, septembre 2001.

[DFI03] C. Demetrescu, I. Finocchi, et G. Italiano. « *Handbook of graph theory* », Chapitre 10.2, Dynamic Graph Algorithms. J. Yellen and J.L. Gross eds., CRC Press Series, in Discrete Mathematics and Its Applications, ISBN 1-58488-090-2, 2003.

[DKM04a] A. Dearle, G. Kirby, et A. McCarthy. « A framework for constraint-based deployment and autonomic management of distributed applications ». Dans *International Conference on Autonomic Computing (ICAC-04), New York, USA*, pages 300–301. IEEE Computer Society, 2004.

[DKM04b] A. Dearle, G. Kirby, et A. McCarthy. « A middleware framework for constraint-based deployment and autonomic management of distributed applications ». Rapport Technique CS/04/2, Université de St Andrews, 2004.

[DL03] P.-C. David et T. Ledoux. « Towards a framework for self-adaptive component-based applications ». Dans *Proceedings of Distributed Applications and Interoperable Systems 2003, the 4th IFIP WG6.1 International Conference, DAIS 2003*, volume 2893 de *Lecture Notes in Computer Science*, pages 1–14, Paris, novembre 2003. Springer-Verlag.

[DL05] P.-C. David et T. Ledoux. « WildCAT : a generic framework for context-aware applications ». Dans *Proceeding of MPAC'05, the 3rd International Workshop on Middleware for Pervasive and Ad-Hoc Computing*, Grenoble, France, 2005. ACM Press.

[DLS88] C. Dwork, N. Lynch, et L. Stockmeyer. « Consensus in the presence of partial synchrony ». *J. ACM*, 35(2) :288–323, 1988.

[DM06] J. Dubus et P. Merle. « Applying OMG D&C specification and ECA rules for autonomous distributed component-based systems ». Dans *MoDELS Workshops*, pages 242–251, 2006.

[dot] .NET, `http://www.microsoft.com/net/`.

[Dow04] J. Dowling. « *The decentralised coordination of self-adaptive components for autonomic distributed systems* ». Thèse de doctorat, Université de Dublin, Trinity College, octobre 2004.

[DRS90] D. Dolev, R. Reischuk, et H. R. Strong. « Early stopping in byzantine agreement ». *J. ACM*, 37(4) :720–741, 1990.

[DvdHT02] E. M. Dashofy, A. van der Hoek, et R. N. Taylor. « Towards architecture-based self-healing systems ». Dans *First Workshop on Self-Healing Systems (WOSS)*, pages 21–26, Charleston, South Carolina, USA, novembre 2002.

[EIS+96] W. J. Ellis, R. F. H. II, T. F. Saunders, P. T. Poon, D. Rayford, B. Sherlund, et R. L. Wade. « Toward a recommended practice for architectural description ». Dans *ICECCS*, pages 408–413, 1996.

[Emm00] W. Emmerich. « Software engineering and middleware : a roadmap ». Dans *ICSE - Future of SE Track*, pages 117–129, 2000.

[Exe04] F. Exertier. « J2EE deployment : the JOnAS case study ». *CoRR*, cs.NI/0411054, 2004.

[FHMO04] A. Ferscha, M. Hechinger, R. Mayrhofer, et R. Oberhauser. « A light-weight component model for peer-to-peer applications ». Dans *ICDCS Workshops*, pages 520–527, 2004.

[FLP85] M. J. Fischer, N. A. Lynch, et M. Paterson. « Impossibility of distributed consensus with one faulty process ». *J. ACM*, 32(2) :374–382, 1985.

[FM04] A. Flissi et P. Merle. « Vers un environnement multi-personnalités pour la configuration et le déploiement d'applications à base de composants logiciels ». Dans *1re conférence sur le Déploiement et la (Re)Configuration de Logiciels, DECOR 2004*, Grenoble, France, octobre 2004. Hermès Sciences.

[Fré04] S. Frénot. « Gestion du déploiement de composants sur réseau P2P ». *CoRR*, cs.NI/0411060, 2004.

[Fro02] Á. Frohner, éditeur. *Object-Oriented Technology ECOOP 2001 Workshop Reader, ECOOP 2001 Workshops, Panel, and Posters, Budapest, Hungary, June 18-22, 2001, Proceedings*, volume 2323 de *Lecture Notes in Computer Science*. Springer, 2002.

[FTBG06] S. Fontaine, C. Taton, S. Bouchenak, et T. Gautier. « Administration autonome d'applications réparties sur grilles ». Dans *Proceedings of the 17e Rencontres francophones du Parallélisme (RenPar'17)*, Le Canet en Roussillon, France, octobre 2006.

[GAO94] D. Garlan, R. Allen, et J. Ockerbloom. « Exploiting style in architectural design environments ». Dans *SIGSOFT FSE*, pages 175–188, 1994.

[GCS03] D. Garlan, S.-W. Cheng, et B. Schmerl. « Increasing system dependability through architecture-based self repair ». Dans R. D. Lamos, C. Gacek, et A. Romanovsky, éditeurs, *Proceedings of the ICSE Workshop on Software Architecture for Dependable Systems*, mai 2003.

[GDL+04] R. Grimm, J. Davis, E. Lemar, A. MacBeth, S. Swanson, T. E. Anderson, B. N. Bershad, G. Borriello, S. D. Gribble, et D. Wetherall. « System support for pervasive applications ». *ACM Trans. Comput. Syst.*, 22(4) :421–486, 2004.

[Geo02] I. Georgiadis. « *Self-organising distributed component software architectures* ». Thèse de doctorat, Department of Computing, Imperial College, University of London, Londres, UK, janvier 2002.

[GMK02] I. Georgiadis, J. Magee, et J. Kramer. « Self-organising software architectures for distributed systems ». Dans *WOSS '02 : Proceedings of the first workshop on Self-healing systems*, pages 33–38, New York, NY, USA, 2002. ACM Press.

[GMW00a] D. Garlan, R. T. Monroe, et D. Wile. « Acme : architectural description of component-based systems ». pages 47–67, 2000.

[GMW00b] D. Garlan, R. T. Monroe, et D. Wile. « Acme : architectural description of component-based systems ». pages 47–67, 2000.

[Gri02] R. Grimm. « *System Support for Pervasive Applications* ». Thèse de doctorat, Université de Washington, décembre 2002.

[GSC01] D. Garlan, B. Schmerl, et J. Chang. « Using gauges for architecture-based monitoring and adaptation ». Dans *Proceedings of the Working Conference on Complex and Dynamic Systems Architecture*, décembre 2001.

[GSSS02] D. Garlan, D. Siewiorek, A. Smalagic, et P. Steenkiste. « Project Aura : towards distraction-free pervasive computing ». *IEEE Pervasive Computing, special issue on Integrated Pervasive Computing Environments*, 1(2) :22–31, avril 2002.

[Hal99] R. S. Hall. « *Agent-based Software Configuration and Deployment* ». Thèse de doctorat, Université de Colorado, 1999.

[HC01] G. T. Heineman et W. T. Councill, éditeurs. *Component-based Software Engineering : putting the Pieces Together*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001.

[Hei98] G. T. Heineman. « A model for designing adaptable software components ». Dans *Proceedings of the 22nd International Computer Software and Applications Conference (COMPSAC)*, pages 121–127, Vienne, Austriche, août 1998.

[HRR+03] A. Hoffmann, T. Ritter, J. Reznik, M. Born, B. Neubauer, F. Stoinski, H. Boehme, B. Folliot, M. Vadet, U. Lang, P. Merle, et C. Contreras. « Specification of the deployment and configuration ». Rapport Technique D2.2/2.3, Specification of the Telecom CCM Infrastructure IST Programme. Project IST-2001-34445. 1 April 2002 to 31 March 2004, juillet 2003.

[Hér05] C. Hérault. « *Adaptabilité des services techniques dans le modèle à composants* ». Thèse de doctorat, Université de Valenciennes et du Hainaut Cambrésis, LAMIH / ROI / SID, Valenciennes, France, juin 2005.

[ISW02] J. Ivers, N. Sinha, et K. Wallnau. « A basis for composition language cl ». Rapport Technique CMU/SEI-2002-TN-026, Université de Carnegie Mellon (CMU), Pittsburgh, USA, 2002.

[JBCG05] A. Joolia, T. V. Batista, G. Coulson, et A. T. A. Gomes. « Mapping ADL specifications to an efficient and reconfigurable runtime component platform ». Dans *WICSA*, pages 131–140, 2005.

[JHE99] J. Jing, A. S. Helal, et A. Elmagarmid. « Client-server computing in mobile environments ». *ACM Comput. Surv.*, 31(2) :117–157, 1999.

[jsr] Java EE Application Deployment, `http://jcp.org/en/jsr/detail?id=88`.

[KB01] D. Kebbal et G. Bernard. « Component search service and deployment of distributed applications ». Dans *Distributed Object and Applications (DOA)*, pages 125–, 2001.

[KC03] J. O. Kephart et D. M. Chess. « The vision of autonomic computing ». *Computer*, 36(1) :41–50, 2003.

[KCB04] N. Kouici, D. Conan, et G. Bernard. « Caching components for disconnection management in mobile environments ». Dans *Proceedings of the International Symposium on Distributed Objects and Applications (DOA)*, Agia Napa, Chypre, octobre 2004.

[KF02] T. Kindberg et A. Fox. « System software for ubiquitous computing ». *IEEE Pervasive Computing*, 1(1) :70–81, 2002.

[KL00] M. Kamel et S. Leue. « Formalization and validation of the General Inter-ORB protocol (GIOP) using PROMELA and SPIN ». *STTT*, 2(4) :394–409, 2000.

[Kou05] N. Kouici. « *Gestion des déconnexions pour applications réparties à base de composants en environnements mobiles* ». Thèse de doctorat, Institut National des Télécommunications, novembre 2005.

[Kra] S. Krakowiak. « *Middleware Architecture with Patterns and Frameworks* ». `http://sardes.inrialpes.fr/~krakowia/MW-Book/`.

[KW04] J. O. Kephart et W. E. Walsh. « An artificial intelligence perspective on autonomic computing policies ». Dans *POLICY*, pages 3–12, 2004.

[Lac05] S. Lacour. « *Contribution à l'automatisation du déploiement d'applications sur des grilles de calcul* ». Thèse de doctorat, Université de Rennes 1, IRISA, Rennes, France, décembre 2005.

[Lau06] K.-K. Lau. « Software component models ». Dans *ICSE '06 : Proceeding of the 28th international conference on Software engineering*, pages 1081–1082, New York, NY, USA, 2006. ACM Press.

[Les03] V. Lestideau. « *Modèles et environnement pour configurer et déployer des systèmes logiciels* ». Thèse de doctorat, Université Joseph Fourier - Grenoble I - INPG, décembre 2003.

[Ley04] F. Leymonerie. « *ASL : un langage et des outils pour les styles architecturaux - Contribution à la description d'architectures dynamiques* ». Thèse de doctorat, Université de Savoie, décembre 2004.

[LS03] N. Le Sommer. « *Contractualisation des ressources pour les composants logiciels : une approche réflexive* ». Thèse de doctorat, Université de Bretagne-Sud, décembre 2003.

[LY99] T. Lindholm et F. Yellin. *Java Virtual Machine specification*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[Mar03] V. Marangozova. « *Duplication et cohérence configurables dans les applications réparties à base de composants* ». Thèse de doctorat, Universit'e Joseph Fourier - Grenoble I, juin 2003.

[Mar04] C. Marchand. « *Mise au point d'algorithmes répartis dans un environnement fortement variable, et expérimentation dans le contexte des pico-réseaux* ». Thèse de doctorat, Institut National Polytechnique de Grenoble - INPG, décembre 2004.

[MBC02] R. S. Moreira, G. S. Blair, et E. Carrapatoso. « FORMAware : framework of reflective components for managing architecture adaptation ». Dans *SEM*, pages 115–129, 2002.

[MCC04] M. L. Massie, B. N. Chun, et D. E. Culler. « The Ganglia distributed monitoring system : design, implementation, and experience ». *Parallel Computing*, 30(5-6) :817–840, 2004.

[MDK94] J. Magee, N. Dulay, et J. Kramer. « Regis : a constructive development environment for distributed programs ». *Distributed Systems Engineering*, 1(5) :304–312, 1994.

[Mey03] B. Meyer. « The grand challenge of trusted components », 2003.

[MH99] R. Monson-Haefel. *Enterprise JavaBeans*. O'Reilly, 1999.

[Mic] S. Microsystems. « Java remote method invocation (RMI) ». `http://java.sun.com/j2se/1.5.0/docs/guide/rmi/`.

[Mic88] S. Microsystems. « Remote Procedure Call specification version 2 (RPC) », juin 1988. `http://www.faqs.org/rfcs/rfc1057.html`.

[Mic96] Microsoft. « DCOM technical overview ». Microsoft Windows NT Server white paper, Microsoft Corporation, 1996.

[MK96] J. Magee et J. Kramer. « Dynamic structure in software architectures ». *SIGSOFT Softw. Eng. Notes*, 21(6) :3–14, 1996.

[MKB+04] R. Morrison, G. N. C. Kirby, D. Balasubramaniam, K. Mickan, F. Oquendo, S. Cîmpan, B. Warboys, B. Snowdon, et R. M. Greenwood. « Support for evolving software architectures in the ArchWare ADL ». Dans *WICSA*, pages 69–78, 2004.

[MKMG97] R. T. Monroe, A. Kompanek, R. Melton, et D. Garlan. « Architectural styles, design patterns, and objects ». *IEEE Software*, 14(1) :43–52, janvier 1997.

[MM03] J. Miller et J. Mukerji. « MDA guide version 1.0.1 ». Rapport Technique, Object Management Group (OMG), 2003.

[Mon01] R. Monroe. « Capturing software architecture design expertise with Armani ». Rapport Technique CMU-CS-98-163, Carnegie Mellon University School of Computer Science, janvier 2001. Version 2.3.

[MORT96] N. Medvidovic, P. Oreizy, J. E. Robbins, et R. N. Taylor. « Using object-oriented typing to support architectural design in the C2 style ». Dans *SIGSOFT FSE*, pages 24–32, 1996.

[MR04] M. Mikic-Rakic. « *Software architectural support for disconnected operation in distributed environments* ». Thèse de doctorat, Université de Californie du Sud, décembre 2004.

[MRC02] C. H. M. Roman et R. Campbell. « Gaia : an OO middleware infrastructure for ubiquitous computing environments ». Dans *ECOOP Workshop on Object-Orientation and Operating Systems (ECOOP-OOOSWS)*, Malaga, Espagne, juin 2002.

[MRM02] M. Mikic-Rakic et N. Medvidovic. « Architecture-level support for software component deployment in resource constrained environments ». Dans *Component Deployment*, pages 31–50, 2002.

[MRM03] M. Mikic-Rakic et N. Medvidovic. « Toward a framework for classifying disconnected operation techniques ». Dans *Workshop on Software Architecture for Dependable Systems (ICSE/WADS'2003)*, Portland, Oregon, mai 2003. ACM Press.

[MRM04a] M. Mikic-Rakic et N. Medvidovic. « Software architectural support for disconnected operation in highly distributed environments ». Dans *CBSE*, pages 23–39, 2004.

[MRM04b] M. Mikic-Rakic et N. Medvidovic. « Support for disconnected operation via architectural self-reconfiguration ». Dans *Proceedings of the First International Conference on Autonomic Computing (ICAC 2004), 17-19 May 2004, New York, NY, USA*. IEEE Computer Society, 2004.

[MRR01] A. Mostéfaoui, S. Rajsbaum, et M. Raynal. « Conditions on input vectors for consensus solvability in asynchronous distributed systems ». Dans *STOC*, pages 153–162, 2001.

[MRRR01] A. Mostéfaoui, S. Rajsbaum, M. Raynal, et M. Roy. « A hierarchy of conditions for consensus solvability ». Dans *PODC*, pages 151–160, 2001.

[MT00] N. Medvidovic et R. N. Taylor. « A classification and comparison framework for software architecture description languages ». *IEEE Trans. Software Eng.*, 26(1) :70–93, 2000.

[Mue97] M. Muehlhaeuser, éditeur. *Special Issues in Object-Oriented Programming, Workshop Reader of the 10th European Conference on Object-Oriented Programming ECOOP'96*. Linz. dpunkt.verlag, Heidelberg, Allemagne, 1997.

[NAD+02] O. Nierstrasz, G. Arévalo, S. Ducasse, R. Wuyts, A. P. Black, P. O. Müller, C. Zeidler, T. Genssler, et R. van den Born. « A component model for field devices ». Dans *CD'02 : Proceedings of the IFIP/ACM Working Conference on Component Deployment*, pages 200–209, Londres, UK, 2002. Springer-Verlag.

[New03] M. E. J. Newman. « The structure and function of complex networks ». *SIAM Review*, 45 :167, 2003.

[Obj02] Object Management Group. « CORBA components ». Adopted Specification formal/02-06-65, OMG, juin 2002. Version 3.0.

[OGT+99] P. Oreizy, M. M. Gorlick, R. N. Taylor, D. Heimbigner, G. Johnson, N. Medvidovic, A. Quilici, D. S. Rosenblum, et A. L. Wolf. « An architecture-based approach to self-adaptive software ». *IEEE Intelligent Systems*, 14(3) :54–62, 1999.

[omg03] « *Deployment and configuration of component-based distributed applications* », juin 2003. Object Management Group, Inc., Draft Adopted Specification (ptc/03-07-02), `http://www.omg.org/docs/ptc/03-07-02.pdf`.

[OMG05] OMG. « *Object Constraint Language specification, version 2.0* ». Object Modeling Group, juin 2005.

[osg] OSGi, Open Service Gateway initiative, `http://www.osgi.org`.

[Par04] P. R. Parvédy. « *Accords tolérant les fautes dans les systèmes répartis synchrones et asynchrones* ». Thèse de doctorat, Université de Rennes I, octobre 2004.

[PBJ98] F. Plásil, D. Bálek, et R. Janecek. « SOFA/DCUP : architecture for component trading and dynamic updating ». Dans *CDS '98 : Proceedings of the International Conference on Configurable Distributed Systems*, page 43, Washington, DC, USA, 1998. IEEE Computer Society.

[PR04] P. R. Parvédy et M. Raynal. « Optimal early stopping uniform consensus in synchronous systems with process omission failures ». Dans *SPAA*, pages 302–310, 2004.

[PT86] K. J. Perry et S. Toueg. « Distributed agreement in the presence of processor and communication faults ». *IEEE Trans. Softw. Eng.*, 12(3) :477–482, 1986.

[QBB+04] V. Quéma, R. Balter, L. Bellissard, D. Féliot, A. Freyssinet, et S. Lacourte. « Asynchronous, hierarchical and scalable deployment of component-based applications ». Dans *Proceedings of the 2nd International Working Conference on Component Deployment (CD'2004)*, Edinburgh, Écosse, mai 2004.

[RC00] M. Román et R. H. Campbell. « Gaia : enabling active spaces ». Dans *ACM SIGOPS European Workshop*, pages 229–234, 2000.

[RC01] M. Roman et R. H. Campbell. « Unified object bus : providing support for dynamic management of heterogeneous components ». Rapport Technique UIUCDCS-R-2001-2222, UILU-ENG-2001-1729, Université de l'Illinois, Urbana-Champaing, 2001.

[RG05] H. Roussain et F. Guidec. « Cooperative component-based software deployment in wireless ad hoc networks ». Dans *3rd International Working Conference on Component Deployment (CD 2005)*, volume 3798 de *LNCS*, pages 1–16, Grenoble, France, novembre 2005. Springer Verlag.

[Sat01] M. Satyanarayanan. « Pervasive computing : vision and challenges ». *Personal Communications, IEEE*, 8(4) :10–17, 2001.

[SBF02] C. B. Saab, X. Bonnaire, et B. Folliot. « PHOENIX : a self adaptable monitoring platform for cluster management ». *Cluster Computing*, 5(1) :75–85, 2002.

[Sch95] R. Schreiber. « Middleware demystified ». *Datamation*, 41(6) :41–45, 1995.

[SG02] B. R. Schmerl et D. Garlan. « Exploiting architectural design knowledge to support self-repairing systems ». Dans *SEKE*, pages 241–248, 2002.

[Sha86] M. Shapiro. « Structure and encapsulation in distributed systems : the proxy principle ». Dans *Proc. 6th Intl. Conf. on Distributed Computing Systems*, pages 198–204, Cambridge, Mass. (USA), mai 1986. IEEE.

[Sie99] J. Siegel. *CORBA 3 fundamentals and programming*. John Wiley & Sons, Inc., New York, NY, USA, 1999.

[SJSM04] A. Sahai, R. Joshi, S. Singhal, et V. Machiraju. « Automated policy based resource construction in utility computing environments ». Dans *Proceedings of the 2004 IEEE/IFIP Network Operations and Management Symposium (NOMS 2004)*, Séoul, Corée, 2004. IEEE.

[SRSS00] D. C. Schmidt, H. Rohnert, M. Stal, et D. Schultz. *Pattern-Oriented Software Architecture : patterns for Concurrent and Networked Objects*. John Wiley & Sons, Inc., New York, NY, USA, 2000.

[SS01] R. Schantz et D. Schmidt. « Middleware for distributed systems : evolving the common structure for network-centric applications », 2001.

[SS02] R. E. Schantz et D. C. Schmidt. « Research advances in middleware for distributed systems ». Dans *Communication Systems : The State of the Art (IFIP World Computer Congress)*, pages 1–36, 2002.

[St00] R. Soley et the OMG staff. « Mode-driven architecture », novembre 2000. `www.omg.org`.

[Sta95] M. Stal. « The broker architectural framework ». octobre 1995.

[Szy02] C. Szyperski. *Component Software*. Addison-Wesley, 2002. 2nd edition.

[Tam] N. Tamura. « Cream : class library for constraint programming in Java ». `http://bach.istc.kobe-u.ac.jp/cream/`.

[THK07] C. Tibermacine, D. Hoareau, et R. Kadri. « Enforcing architecture and deployment constraints of distributed component-based software ». Dans *Proceedings of the International Conference on Fundamental Approaches to Software Engineering (FASE'07)*, Braga, Portugal, mars 2007. LNCS 4422, Springer-Verlag.

[Tib06] C. Tibermacine. « *Contractualisation de l'évolution architecturale de logiciels à base de composants : une approche pour la préservation de la qualité* ». Thèse de doctorat, Université de Bretagne Sud, Vannes, France, octobre 2006.

[VCK96] J. M. Vlissides, J. O. Coplien, et N. L. Kerth. *Pattern languages of program design 2*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1996.

[vH01] W. J. van Hoeve. « The alldifferent constraint : a survey ». *CoRR*, cs.PL/0105015, 2001.

[vHSD94] P. van Hentenryck, H. Simonis, et M. Dincbas. « Constraint satisfaction using constraint logic programming ». pages 113–159, 1994.

[VK02] G. Valetto et G. Kaiser. « A case study in software adaptation ». Dans *WOSS '02 : Proceedings of the first workshop on Self-healing systems*, pages 73–78, New York, NY, USA, 2002. ACM Press.

[vOvdLKM00] R. van Ommering, F. van der Linden, J. Kramer, et J. Magee. « The Koala component model for consumer electronics software ». *Computer*, 33(3) :78–85, 2000.

[Wei99] M. Weiser. « The computer for the 21st century ». *ACM SIGMOBILE Mobile Computing and Communications Review*, 3(3) :3–11, 1999.

[Whi76] J. E. White. « High-level framework for network-based resource sharing », 1976.

[WRW96] A. Wollrath, R. Riggs, et J. Waldo. « A distributed object model for the Java system ». Dans *2nd Conference on Object-Oriented Technologies & Systems (COOTS)*, pages 219–232. USENIX Association, 1996.

[ZME04] S. Zachariadis, C. Mascolo, et W. Emmerich. « Satin : a component model for mobile self organisation ». Dans *On the Move to Meaningful Internet Systems 2004 : CoopIS/DOA/ODBASE OTM Confederated International Conferences, Part II*, pages 1303–1321, octobre 2004.

Résumé

Résumé

On assiste depuis quelques années à l'émergence de réseaux de machines, qualifiés de dynamiques, qui ne sont plus limités à une interconnexion de stations de travail définissant un réseau stable. Ces plates-formes intègrent de plus en plus des équipements mobiles et à faibles ressources. L'exploitation effective de ces réseaux dynamiques constitue encore un défi. Du fait de la volatilité des équipements et de leur hétérogénéité, on ne peut avoir une connaissance a priori de toutes les ressources logicielles et matérielles qui sont disponibles dans ces réseaux. De plus, la connectivité entre les différentes machines ne peut être garantie : de nombreux équipements sont régulièrement mis en veille et l'utilisation de technologie sans fil limite la portée de communication entre les machines. Ces réseaux dynamiques commencent à être exploités pour fournir des applications censées être ubiquitaires, c'est-à-dire des applications dont les fonctionnalités sont accessibles de partout, indépendamment de l'équipement utilisé. Mais de telles applications sont par nature complexes et leur conception est rendue difficile du fait du caractère dynamique des réseaux visés.

Le travail décrit dans ce mémoire de thèse a permis de développer des méthodes pour concevoir et déployer des applications ubiquitaires dans des réseaux dynamiques.

En nous appuyant sur le modèle de composants Fractal, nous définissons le concept de composants hiérarchiques ubiquitaires, briques d'assemblage des applications ubiquitaires. Un schéma de distribution de ces composants est élaboré afin que ceux-ci puissent offrir leurs services depuis n'importe quelle machine du réseau. Cette distribution des fonctionnalités des composants sur plusieurs machines impose de prendre en compte les déconnexions réseau qui pourraient conduire au dysfonctionnement de l'application. Pour cela, nous avons défini au sein du modèle de composants ubiquitaires le concept d'interface active qui permet de continuer à utiliser certaines parties de l'application tout en isolant celles inutilisables du fait des problèmes de connectivité.

Les approches traditionnelles de déploiement ne sont pas adaptées au déploiement des composants ubiquitaires dans les réseaux dynamiques. Dans ces réseaux, les ressources qui sont présentes ne peuvent être connues à l'avance, ce qui empêche la désignation explicite des machines cibles devant héberger les composants. Dans notre approche, le placement des composants est spécifié à l'aide de contraintes exprimant le besoin des composants vis-à-vis des ressources nécessaires à leur exécution. Après avoir présenté un langage permettant la définition de telles contraintes, nous proposons un support intergiciel qui réalise le déploiement d'une application ubiquitaire même si initialement, les ressources exigées par ses composants ne sont pas disponibles et que certaines machines sont inaccessibles. Le processus de déploiement que nous proposons est qualifié de propagatif : les composants sont instanciés au fur et à mesure de la disponibilité des ressources et de l'arrivée des machines. Lorsque des fluctuations de ressources interviennent dans le réseau, les applications ubiquitaires sont amenées à être reconfigurées. La solution que nous présentons permet de rendre ces reconfigurations complètement autonomes, ne nécessitant ainsi plus aucune intervention manuelle.

L'ensemble des concepts introduits dans cette thèse a fait l'objet du développement d'un prototype fondé sur une implantation Fractal.