

GOODE Component Architecture: Structures d'accueil pour la mobilité et le cache d'objets métier.

(Mémoire de D.E.A.)

Raphaël MARVIE

5 juillet 1999

COMPOSANT n.m. Élément constitutif. **1.** CHIM. Élément qui, combiné avec un ou plusieurs autres, forme un corps composé. **2.** TECHN. Constituant élémentaire d'une machine, d'un appareil ou d'un circuit électrique ou électronique. **3.** CONSTR. Élément de construction industrialisé.

—*Le petit Larousse*—

Remerciements

De nombreuses personnes ont permis la réalisation de ce travail, la liste qui suit n'est pas exhaustive.

Tout d'abord, je tiens à remercier **Jean-Marc GEIB** pour m'avoir accueilli dans son équipe, pour m'avoir proposé un sujet de mémoire passionnant, pour m'avoir offert la possibilité de rencontrer des membres d'autres équipes, pour m'offrir une poursuite en thèse avec tout le groupe. Merci aussi pour les discussions, les remarques, les conseils prodigués depuis deux ans, le recul sur les choses, j'espère que j'en prendrai de la graine.

Ensuite, mes remerciements vont aux deux initiateurs de ce D.E.A., si ce mémoire existe aujourd'hui c'est à cause d'eux ; -). D'une part, **Christophe GRANSART** pour m'avoir invité il y a plus de deux ans, suite à des discussions post-cours de réseau sur mes centres d'intérêts, à venir découvrir leurs activités de recherche. Et d'autre part, **Philippe MERLE** pour m'avoir encadré en dernière année d'IUP GMI, et m'avoir encouragé tout au long de ces deux années. Merci à tous les deux pour les discussions, les encouragements, les relectures de mes documents, etc.

Ce stage s'est déroulé dans une équipe chaleureuse (et très enfumée). Cette atmosphère est donc aussi due à la participation de **Jean-François ROOS** et **Gaëtan SCOTTO DI APOLONNIA**. Merci à tous les deux. Merci aussi à **Bernard CARRÉ** pour les divers échanges, les conseils et les encouragements et à **Gilles VAN WORMHOUT** pour les méta-discussions.

L'ambiance cette année a été particulièrement agréable. Mes remerciements vont donc aussi à tous les étudiants du DEA 1999, avec une petite note supplémentaire pour **Yannick LENIR**, à qui je dois des progrès en pédagogie, et **Yann SECQ**, avec qui plusieurs des idées développées ici ont germé.

Pour terminer, un grand merci à tous les givrés du PARC de Xerox (Palo Alto Research Center). C'est en découvrant il y a quelques années la vie, et les travaux, des équipes des années 70 que j'ai eu envie de faire de la recherche. Et comme a dit Alan KAY "*The best way to predict the future is to invent it*".

Table des matières

1	Cadre et objectifs	1
1.1	Cadre	1
1.2	Objectifs de ce travail	1
1.3	Plan de ce mémoire	2
2	Construction d'applications à l'aide de composants	3
2.1	Une vue métier de la production de logiciel	3
2.1.1	Acteurs de l'industrie logicielle	3
2.1.2	Le développement orienté objet	6
2.1.3	Le développement orienté composant	8
2.2	Quelques modèles de composants	9
2.2.1	Java Beans	10
2.2.2	Enterprise Java Beans	11
2.2.3	Les composants CORBA	14
2.2.4	Vue d'ensemble	18
2.3	Premières conclusions de cette étude	18
3	L'architecture à composants de GOODE	21
3.1	Vue d'ensemble	21
3.1.1	Applications	21
3.1.2	Composants	21
3.1.3	Interactions	22
3.2	Composant logiciel	23
3.2.1	Définition d'un composant GOODE	23
3.2.2	Parties d'un composant GOODE	24
3.3	Structure d'Accueil de Composants	25
3.3.1	Composition d'une SAC	25
3.3.2	Interactions SAC-composants	28
3.3.3	Interactions avec l'environnement	28
3.4	Modèle d'exécution	30
3.4.1	L'insertion de composant	30
3.4.2	Le clonage d'une instance de composant	30
3.4.3	Persistance d'une instance de composant	32
3.5	Conclusion	33

4	Gestion du nommage	35
4.1	Le service de nommage	35
4.1.1	Kesako	35
4.1.2	Vue macroscopique	35
4.1.3	Vue détaillée	36
4.2	Espaces de Nommage	37
4.2.1	Utilisation des espaces de nommage	37
4.2.2	Un seul ou plusieurs espaces de nommage?	37
4.3	Fédération d'espaces de nommage	38
4.3.1	Fédération à gros grain	39
4.3.2	Fédération à un grain très fin	40
4.3.3	Discussion sur les deux approches	42
5	Cache, mobilité et expérimentation	45
5.1	Cache d'objets	45
5.1.1	Objectif des caches	45
5.1.2	Ce qui se fait	45
5.1.3	L'utilisation de GCA pour le cache	47
5.2	Mobilité de composants	48
5.2.1	Un existant	48
5.2.2	La mobilité dans le contexte de GCA	48
5.3	Expérimentation	49
5.3.1	Prototype	49
5.3.2	Le projet des physiciens	49
5.3.3	Proposition les implantations de composants en C++	50
5.4	Synthèse	51
6	Conclusion et poursuite	53
6.1	Conclusion	53
6.2	Poursuite	53
A	structure d'accueil de composants	55
B	Service de Nommage étendu	57
C	Service de Nommage fédéré	59

Table des figures

2.1	Vue d'ensemble des acteurs de l'industrie du logiciel.	3
2.2	Les rôles des acteurs de l'industrie du logiciel.	5
2.3	Le modèle abstrait d'objet.	7
2.4	Le modèle abstrait de composant.	8
2.5	Le modèle de composant JavaBean.	10
2.6	Modèle abstrait des Enterprise Java Beans.	12
2.7	Environnement d'exécution des Enterprise Java Beans.	12
2.8	Modèle très abstrait des composants CORBA.	14
2.9	Modèle abstrait des composants CORBA.	15
2.10	L'architecture d'un container.	16
3.1	Les différentes parties d'un composant GOODE.	24
3.2	Vue d'ensemble d'une Structure d'Accueil de Composants.	25
4.1	Arborescence de contextes de nommages dans un NS.	36
4.2	Fédération de services de nommages à gros grain.	39
4.3	Fédération de services de nommages à grain très fin.	40

Chapitre 1

Cadre et objectifs

1.1 Cadre

Les travaux de recherche présentés dans ce mémoire ont pris place dans le contexte de l'équipe CIM-GOAL¹ du Laboratoire d'Informatique Fondamentale de Lille. Cette équipe travaille sur les modèles et langages à objets et/ou à acteurs. Les deux projets actuellement en cours dans cette équipe sont CROME² et GOODE³.

Le projet CROME s'intéresse à la conception et à la modélisation des composants. Les travaux courants portent sur l'application de ce modèle aux systèmes d'information pour rendre compte de la structuration en contextes fonctionnels de l'organisation sous-jacente. L'étape suivante s'attachera à la conception de systèmes d'objets distribués (évolution et adaptation du modèle CROME).

Le projet GOODE est une réflexion sur les plate-formes dynamiques d'objets distribués. Les aspects mis en avant dans ce projet sont l'exploitation de la dynamique et des objets pour l'intégration de ressources hétérogènes et l'administration d'applications distribuées. Un des principaux fruits de ce projet est la plateforme GoodeScript dont l'instance la plus aboutie est l'environnement CorbaScript [LIF99]. Cet environnement offre un langage interprété orienté objet pour le monde CORBA⁴.

Ce mémoire de DEA se situe dans le second projet. Il a pour contexte de la conception, le développement et l'administration d'applications dans les environnements distribués à grande échelle.

1.2 Objectifs de ce travail

Le premier objectif de ce travail est de réfléchir sur le concept de composant logiciel mobile. Le but de cette réflexion est de tirer parti des composants pour faciliter la conception, le développement, le déploiement et l'administration d'applications réparties à grande échelle. Le choix d'un modèle de composants doit donc prendre en compte ces différents aspects.

Le second objectif est de voir comment offrir des possibilités de cache pour les composants. Les deux buts de ces caches sont d'offrir des moyens de répliquions pour améliorer les

1. Axe : Coopération, Image et Mobilité, équipe : Groupe Objets et Acteurs de Lille

2. Contexte en ROME –Représentation d'Objets Multiple et Evolutive

3. Generic Object Oriented Distributed Environment

4. Common Object Request Broker Architecture, architecture à objets distribués de l'OMG –Object Management Group.

performances et permettre la mobilité à l'aide de structures d'accueil. Ces structures d'accueil doivent permettre, en plus du cache et de la mobilité, le déploiement plus facile des applications et offrir des facilités d'administrations.

Ces deux objectifs peuvent se résumer en la proposition d'une architecture simple à mettre en place et la plus transparente possible pour l'utilisateur final. Ce dernier a pour unique souci d'avoir un système global lui fournissant correctement les services dont il a besoin. La technique sous-jacente l'importe peu.

1.3 Plan de ce mémoire

Nous allons dans un premier temps (dans le chapitre 2) présenter une première classification de différents modèles de composants existants. Pour cela nous avons défini une grille de classification correspondant à une vue métier de l'industrie du logiciel. Le but est de comparer les modèles de composants par rapport aux acteurs de l'industrie du logiciel concernés.

Le chapitre 3 présentera l'état de nos travaux sur un modèle de composant pour la mobilité. Ce chapitre parlera, dans une seconde partie, de l'infrastructure que nous avons mise en œuvre pour ces composants mobiles. Enfin, les structures d'accueil et les mécanismes mis en place seront décrits et commentés.

Un élément important dans le cadre des applications distribuées est le nommage. Le chapitre 4 présentera la mise en œuvre du nommage dans le contexte de l'architecture CORBA. Puis, il s'attardera sur les extensions à ce service de nommage pensées dans le cadre de nos travaux. La fin de ce chapitre discutera des différentes approches sur lesquelles nous avons travaillé.

Enfin, nous présenterons dans le chapitre 5, l'utilisation du cache et de la mobilité dans le cadre des applications distribuées. Nos premières expérimentations relatives au clonage, au cache et à la mobilité de composants seront discutées. Nous parlerons aussi du projet de collaboration qui doit participer à la validation de la mise en œuvre de notre proposition.

Un présentation synthétique, premier bilan, de ces travaux et les poursuites que nous envisageons seront présentées dans le chapitre 6.

Chapitre 2

Construction d'applications à l'aide de composants

Aujourd'hui, les composants sont en vogue dans le milieu industriel. L'utilisation du mot *composant* sert de support au marketing, au même titre que le mot *objet* il y a quelques années. Nous allons essayer de faire abstraction de cet effet de mode pour traiter ici de l'évolution et des apports, que représentent les composants dans le domaine de la conception d'applications.

Après avoir présenté les acteurs de l'industrie du logiciel et un modèle abstrait de composant, nous allons présenter une première classification de trois modèles de composant existants : les JavaBeans, les Enterprise Java Beans et les Composants CORBA. Comme cela a été fait dans la spécification Enterprise Java Beans, nous allons aborder ce chapitre d'un point de vue métiers. Et essayer ainsi, de voir ce qu'apportent les composants, et à qui.

2.1 Une vue métier de la production de logiciel

2.1.1 Acteurs de l'industrie logicielle

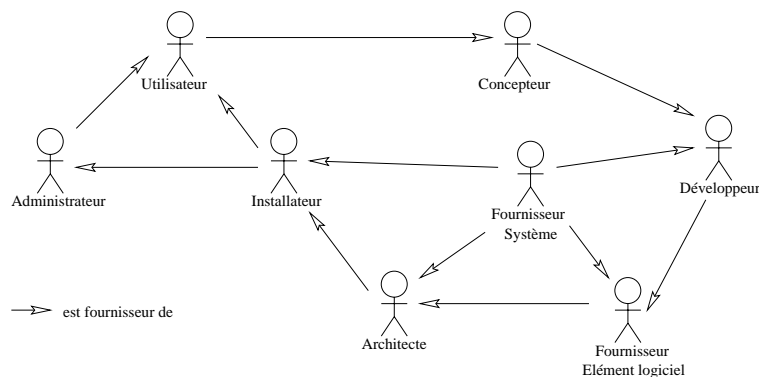


FIG. 2.1 – Vue d'ensemble des acteurs de l'industrie du logiciel.

Les acteurs de l'industrie du logiciel sont nombreux. Nous en avons défini huit catégories qui nous semblent essentielles. Il est évidemment possible de découper plus finement ces huit

métiers et définir plus de catégories, mais nous allons dans le cadre de ce chapitre nous limiter aux huit catégories suivantes :

1. Le fournisseur système
2. Le concepteur d'élément logiciel
3. Le développeur d'élément logiciel
4. Le fournisseur d'élément logiciel
5. L'architecte d'application
6. L'installateur d'application
7. L'administrateur d'application
8. L'utilisateur final

Ces huit catégories vont être présentées une par une. Un diagramme global résumera tous ces métiers dans la figure 2.2. La figure 2.1 présente une vue d'ensemble des dépendances entre ces huit métiers, principalement ce qui est fourni par un acteur d'un métier à destination d'un autre.

Le fournisseur système

Ce métier n'est pas le début de la chaîne *conception de logiciel*, mais il y intervient à plusieurs reprises. Le but de ce métier est de fournir des outils pour la production de logiciel, comme par exemple des environnement de développement, des outils de packaging, de configuration, des serveurs d'application, . . . En fait, ce métier s'occupe en quelque sorte du boulot que les autres ne veulent pas faire.

Les outils produits ici doivent masquer toute la complexité des aspects système, pour que les autres acteurs puissent en faire abstraction et se concentrer sur leur tâche. Ces outils sont à destination du développeur, du fournisseur, de l'architecte, de l'installateur et peuvent servir à l'administrateur.

Le concepteur d'élément logiciel

Le concepteur d'élément logiciel est le premier maillon de la chaîne de production. Il va définir les spécifications des éléments logiciels. Ces spécifications fournissent les interfaces des entités logicielles. Ces interfaces sont d'ordre fonctionnelles, mais peuvent aussi être d'ordre connectiques¹. (Quels sont les services offerts, quels sont les services nécessaires à l'entité spécifiée?) Ce travail s'effectue par rapport à un modèle qui décrit le processus métier, ou la tâche élémentaire, que doit effectuer le composant.

Le développeur d'élément logiciel

Le développeur d'élément logiciel implante l'aspect fonctionnel d'une entité logicielle. Il ne devrait d'ailleurs ne faire que cela. Cette implantation est une partie de l'implantation de l'élément logiciel spécifié par le concepteur. Il est souhaitable que l'autre partie, regroupant les aspects dépendants du système, soit générée par des outils de développement.

1. Liens qui existent entre composants, ou entre un composant et son environnement. Dans le cadre des objets, la connectique est représentée par des pointeurs, ou références, attributs de l'objet.

Le fournisseur d'élément logiciel

Le fournisseur d'élément logiciel package l'entité logicielle implantée par le développeur. Ce package a pour but de faciliter la diffusion et de permettre la vente d'une entité logicielle réutilisable. En plus de fournir l'implantation, le package contient la description du code, son mode d'emploi, les prérequis (OS, moniteur transactionnel, gestionnaire de persistance, ...) associés à l'entité logicielle. Ces éléments complémentaires sont générés, ou écrits par le fournisseur. Un utilisateur pourra facilement mettre en œuvre (intégration, connexion, ...) l'entité logicielle.

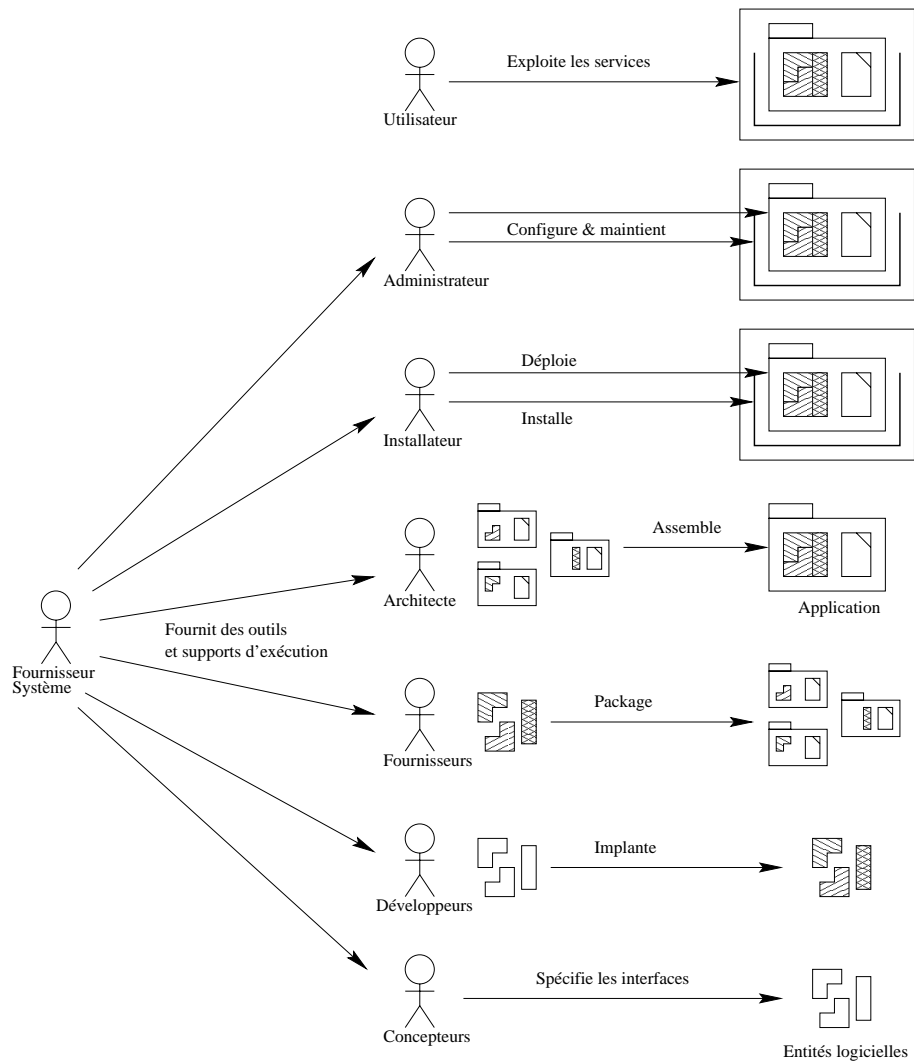


FIG. 2.2 – Les rôles des acteurs de l'industrie du logiciel.

L'architecte d'application

Une fois des packages d'élément logiciel disponibles, l'architecte peut concevoir des applications à l'aide de ceux-ci. L'architecte va assembler ces entités, en utilisant un outil d'assemblage ou un langage de script. Ces assemblages peuvent avoir deux buts. Premièrement, ils peuvent être vendus tels quels au sein d'un package. D'autres architectes pourront alors les utiliser pour bâtir leurs applications. Deuxièmement, les assemblages forment des applications qui seront diffusées ou vendues.

L'architecte a aussi un rôle de configurateur. En assemblant les packages, il configure la connectique qui se trouve entre les différentes entités logicielles mises en œuvre. Il peut aussi fournir une configuration de base pour la périphérie de l'assemblage (configuration par défaut pour l'utilisateur). Cette configuration est un des éléments du package d'assemblage (en plus de la description, de la connectique et des implantations).

L'installateur d'application

L'installateur d'application met en place les supports d'exécution fournis par le fournisseur système (serveurs d'applications, structures d'accueils, ...). Une fois ces supports d'exécution mis en place, il déploie au sein de ces hôtes les applications bâties par l'architecte. La configuration de base est alors appliquée pour permettre l'exploitation des applications, dont l'exécution est démarrée.

L'administrateur d'application

Tout comme un gros système ou un réseau, les applications distribuées requièrent un administrateur. Il doit faire en sorte que l'application se comporte comme il se doit, que les utilisateurs puissent avoir accès aux services, ... Cela implique, pour l'administrateur, de faire de la supervision, de la gestion, du tuning (en modifiant la configuration), de la maintenance, ... Il va donc pouvoir agir sur les hôtes et sur les applications.

L'utilisateur final

Nous présentons l'utilisateur en fin de chaîne, mais il faut être conscient qu'il est aussi le premier pas de la chaîne de production de logiciel : sans utilisateur, pas de besoin donc pas de production. C'est donc lui (ou elle) qui va exposer le besoin auquel l'application doit répondre.

Une fois l'application mise en œuvre, l'utilisateur final pourra l'exploiter. Il n'est à aucun moment conscient du parcours, de la composition, de l'architecture de l'applicatif. Il voit l'applicatif comme une boîte noire qui, en général, réalise les traitements dont il a besoin. La boucle est bouclée.

2.1.2 Le développement orienté objet

Définition

Une définition bien acceptée pour *objet* peut s'énoncer comme suit :

Un objet est un type abstrait de données décrit par une interface offrant un certain nombre de services sans contraintes par rapport à l'implantation de ces derniers.

Ce qui donne le modèle abstrait décrit dans la figure 2.3.

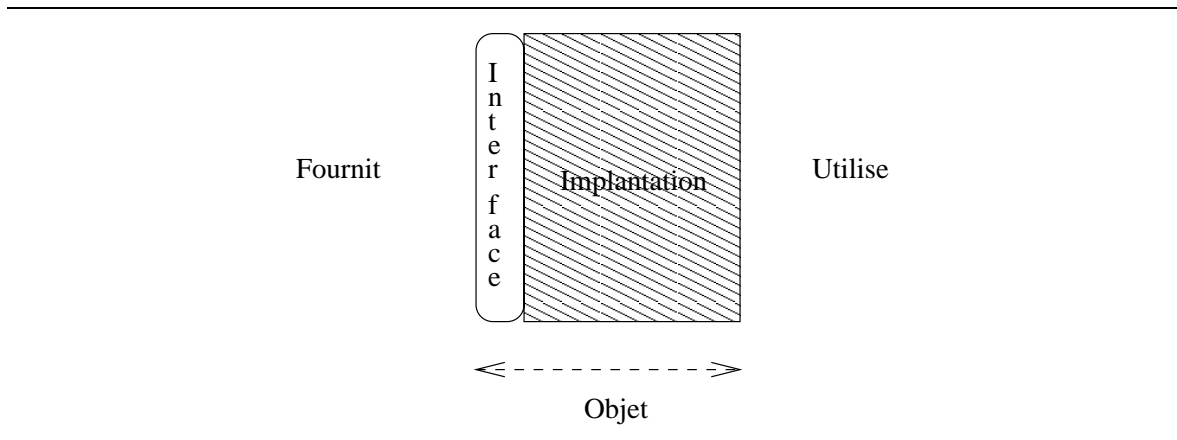


FIG. 2.3 – *Le modèle abstrait d'objet.*

Public visé

Dans [Mey90], B. Meyer définit la conception orientée objet comme suit :

La conception par objets est la construction de systèmes logiciels prenant la forme de collections structurées d'implémentations de types de données abstraits.

Cette assertion définit le public visé par le développement orienté objet : le concepteur, le développeur et éventuellement l'architecte.

Le développement (conception et programmation) orienté objet n'adresse donc pas tous les métiers que nous venons de présenter. Le développement OO s'adresse concrètement au concepteur et au développeur d'élément logiciel. Les méthodes de conceptions OO permettent de bien définir les interfaces des entités logicielles (les objets) et de les rendre réutilisables. La programmation OO offre, via l'encapsulation, une certaine indépendance de l'implantation. Un objet bien implanté ne dépend d'autres objets qu'au travers de leur interface.

Pour les autres métiers, rien, dans le développement OO, ne leur est vraiment destiné. Le fournisseur système offre bien évidemment un environnement de développement. Cet environnement facilite la tâche du développeur, et permet au fournisseur d'entité logicielle de construire des packages : les librairies. Mais le modèle ne définit pas de facilités pour construire, ni pour décrire, les librairies. La description est en générale une documentation sous un format libre.

L'architecte devra, pour bâtir une application, programmer les parties de l'application qui utilisent les fonctionnalités offertes par les librairies. Il se retrouve donc à la place du développeur, ce qui indique qu'il y a beaucoup de développeurs de types différents. L'installateur, quant à lui, ne dispose d'aucune facilité pour installer une application. Il doit en fait tout faire *à la main*, à moins que les développeurs ne lui fournissent l'application sous forme facilement installable.

2.1.3 Le développement orienté composant

Définition

Il n'existe pas de définition universelle d'un composant. Le concept de composant est assez récent, et chacun propose sa définition par rapport au domaine dans lequel il travaille : conception d'applications graphiques, client/serveur, objets distribués, . . .

Dans ce contexte, une définition de type "plus petit dénominateur commun" a été proposée. Lors de ECOOP (European Conference on Object-Oriented Programming) 1996, le workshop WCOP (Workshop on Component Oriented Programming) a proposé la définition suivante :

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.

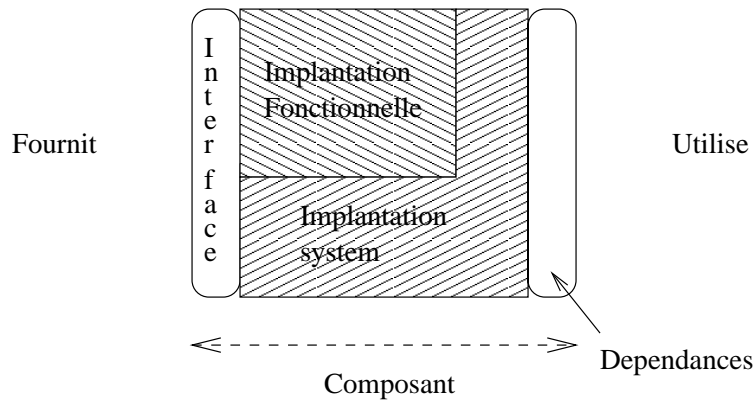


FIG. 2.4 – *Le modèle abstrait de composant.*

Modèle abstraite d'un composant

Nous avons vu dans la section 2.1.2 que les objets se découpent en deux parties : l'interface et l'implantation. Le modèle abstrait d'un composant est peu différent. Dans le cadre des objets, l'implantation est présentée comme le code qui offre les services. Dans le cadre des composants, cette implantation est sous-spécifiée : il y a d'une part le code fonctionnel, et d'autre part le code plus système dont tout développeur souhaiterait disposer automatiquement. La figure 2.4 présente schématiquement ce modèle de composant.

Les métiers adressés par les composants

L'objectif des composants est d'adresser tous les métiers de l'industrie du logiciel. Regardons d'un peu plus près ce que chaque métier peut espérer trouver dans un environnement composants.

Le travail du concepteur n'est pas fondamentalement changé dans l'approche à base de composants. Le but est de fournir les interfaces des services offerts par les composants. La plus

grosse différence est sur la connectique des composants. L'expression de cette connectique doit être plus claire. Il est aussi souhaitable que la connectique soit exprimée indépendamment du composant (un attribut dans le cas des objets).

Le développeur est un des grands gagnants de l'approche à base de composants. Le modèle abstrait de la section précédente présente l'implantation en deux parties. Le développeur va pouvoir se consacrer à la partie fonctionnelle en oubliant les contraintes et astreintes système. L'outil de développement *idéal* pour les composants est un outil qui générerait automatiquement la partie système à partir de la spécification d'un composant.

L'approche composant apporte au fournisseur de composants la granularité. Dans le cas des objets, le moyen de distribution était la librairie. Ici, un composant logiciel doit pouvoir être diffusé seul. Le travail du développeur et le travail du concepteur seraient repris pour décrire les services offerts par le composant, les pré-requis (système, OS, matériel, . . .), la connectique et les implantations (binaires) des composants. Toutes ces informations ont leur place dans un descripteur de composant.

L'architecte d'application est le second grand gagnant de la saga composants. En effet, plus besoin de coder son application en utilisant les bibliothèques fournies par les autres métiers. Il lui suffirait d'acheter les composants dont il a besoin, puis de les assembler à l'aide d'un outil de composition (graphique ou sous forme de script). Cela est possible grâce aux descriptions de composants fournies par son prédécesseur (dans la chaîne de production).

L'apport des composants pour l'installateur d'applications est le fait de disposer de structures d'accueil pour recevoir les applications. Une fois les supports mis en place, le déploiement d'applicatifs peut être automatisé (au moins une partie de ce processus). Il faut pour cela des structures d'accueil bien définies et génériques, capables de recevoir différents types de composants.

Dans le cadre d'une application construite à partir d'objets, la modification d'une partie de l'application implique de recompiler et relinker l'application. Avec les composants, l'administrateur n'aurait plus qu'à débrancher le *vieux composant*, charger le composant qui a évolué (si l'interface n'a pas perdu de services) et refaire les branchements relatifs au composant.

L'utilisateur final ne doit pas voir de différence entre une application à base d'objets et une application à base de composants, si ce n'est qu'il disposera de son application plus vite, et qu'elle répondra mieux à son besoin (objectifs récurrents).

Le fournisseur système est le plus grand gagnant dans l'histoire : il va fournir de nouveaux environnements plus complets, plus performants, plus conviviaux (autant d'arguments commerciaux). D'autre part, il peut ouvrir une nouvelle division, puisque les composants ont fait naître les serveurs d'applications, donc un nouveau créneau (voire un nouveau métier).

2.2 Quelques modèles de composants

Dans toute cette section, nous allons présenter, pour chaque modèle de composant, le modèle abstrait ainsi que l'environnement d'exécution des composants. Ensuite, une vision métier sera présentée. Enfin, nous considérerons les apports de chaque modèle de composants pour chacun des huit métiers que nous avons défini dans la section 2.1.

2.2.1 Java Beans

Le modèle des JavaBeans[Ham97] est un modèle de composants proposé par Sun. Les JavaBeans se présentent comme un mode d'utilisation de l'environnement de développement Java. Bien que certains traitements soient réalisable en Beans, le marché des JavaBeans correspond essentiellement au marché des interfaces graphiques. Le mode d'utilisation est principalement la réutilisation de widgets existants pour en concevoir des nouveaux plus complexes.

Le modèles abstrait

Comme tout objet Java, un Bean possède une interface de classe. Il peut disposer de méthodes pour offrir des traitements et d'attributs pour représenter ses propriétés (voir figure 2.5). Les méthodes peuvent servir à traiter les événements reçus par le Bean. Au niveau de l'interface, rien ne différencie un Bean de tout autre objet Java. Toute personne intéressée par les fonctionnalités d'un Bean, et ne connaissant pas les JavaBeans, pourra sans difficulté utiliser ce Bean selon le modèle objet *traditionnel* de Java.

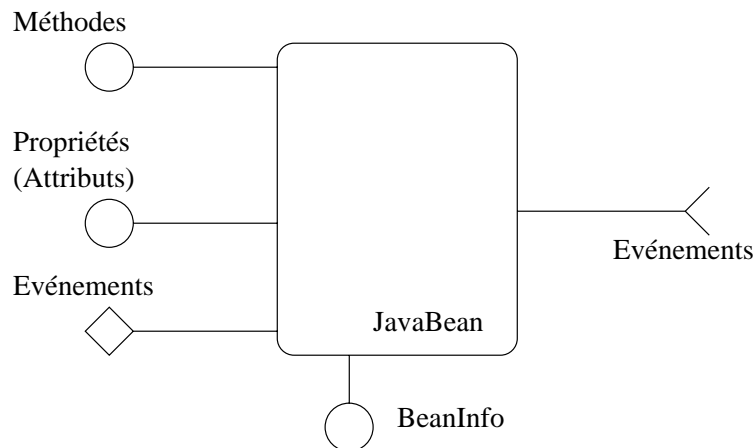


FIG. 2.5 – Le modèle de composant *JavaBean*.

L'environnement d'exécution

L'environnement d'exécution des JavaBeans est simplifié au maximum : un container graphique. Pour qu'un *JavaBean* puisse s'exécuter, il lui faut une fenêtre graphique (*Frame*, *Browser Web*, ...). Un *JavaBean* fait donc parti d'une application, qui lui permettra de s'afficher.

La vue métier

Dans le cadre des JavaBeans, le fournisseur système ne produit pas d'environnement d'exécution (puisque une simple fenêtre sert de container). Le produit d'un fournisseur système est ici un environnement graphique complet pour le développeur (conception par composition,

drag-and-drop, écriture de code, ...). Cet environnement doit permettre de positionner/lire les méta-informations sur le Bean –contenus dans le BeanInfo. Ces méta-information offrent au développeur une liste des services fournis par le Bean, guidant ainsi son intégration dans une application.

Le concepteur d'application définit l'interface du Bean. Il spécifie, en plus des méthodes de traitement, les événements auxquels le Bean doit s'abonner, et les événements que le Bean produit. L'outil nécessaire au concepteur est un outil de conception objet qui générera les squelettes des Beans.

Le développeur dispose, pour implanter les Beans de deux manières: il implante tout le Bean ou il étend un Bean existant. Dans les deux cas, l'implantation est relative à l'interface définie par le concepteur.

Le fournisseur de composant package l'implantation d'un Bean sous forme de fichier archive (.jar) contenant la classe BeanInfo. Cette archive pourra être distribuée / vendue.

L'architecte va à l'aide d'un outil d'assemblage graphique construire des Beans plus complexes, ou des applications. Pour cela, il exploitera les méta-informations contenues dans les BeanInfo sur les services offerts, la gestion des événements par le Bean.

Dans le cas des JavaBeans, les trois (voire quatre) métiers précédents sont, semble t'il, regroupés en un seul. Le développeur fait de la conception et du packaging. Toutes ces facilités sont offertes par l'environnement de développement (qui devient un atelier de production de logiciel).

L'installateur ne fait que de la mise en place d'une application sur un serveur de JavaBeans (serveur Web ou serveur de fichiers). Le client pourra alors utiliser les applicatifs en invoquant sa JVM préférée. L'administrateur n'a pas vraiment de rôle dans cette chaîne. Les applications visées sont surtout monolithiques. Il n'y a pas de répartition de l'application, éventuellement de la coopération entre applications composées de Beans.

2.2.2 Enterprise Java Beans

Les JavaBeans de Sun n'ayant pas reçu le titre de *composant universel*, Sun a passé un an à mettre au point les Enterprise Java Beans[MH99] (EJB) dont la spécification 1.0 est sorti en novembre 1997. Les EJB ont un cadre différent des JavaBeans, les EJB représentent un framework de composants métier (composants serveurs).

Le modèles abstrait

Tout comme un JavaBean, un EJB est un composant représenté par une interface Java. Cette interface hérite de plusieurs interfaces prédéfinies, comme EJBObject, ou définies par l'utilisateur, interface regroupant les opérations métier d'un EJB. La fourniture d'opérations *métier* est le but d'un EJB. L'implantation de ces opérations représentent le cœur du composant.

Un utilisateur ne dialogue pas directement avec une instance d'EJB. Le client utilise une interface distante (remote interface). Cette interface délègue les requêtes à l'implantation (l'instance d'EJB). Cette délégation est prise en charge par le container (voir section suivante) qui a un rôle de médiateur.

En plus de cette interface métier, un EJB offre une interface pour accéder aux méta-données de l'instance de composant et une interface qui gère le cycle de vie d'un composant (home interface). L'interface de maison permet de créer (ou rechercher dans le cas d'un composant persistant) et de détruire une instance de composant. Elle offre aussi une opération retournant une référence sur l'interface de méta-données. Cette dernière interface permettra par la suite de construire dynamiquement des requêtes sur le composant.

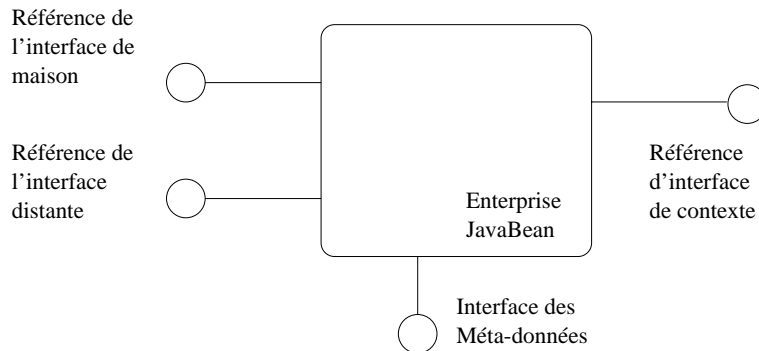


FIG. 2.6 – *Modèle abstrait des Enterprise Java Beans.*

Les composants sont de deux types : composants de session et composants entités. Les instances des premiers sont créées à chaque connexion d'un client. Elle peuvent être avec, ou sans état (composant de traitement). Lorsque le client termine sa session, l'instance de composant est détruite. Les instances du second type, sont organisées sous forme de pool. Chaque instance dispose d'un identifiant unique qui permet au client de retrouver une instance particulière.

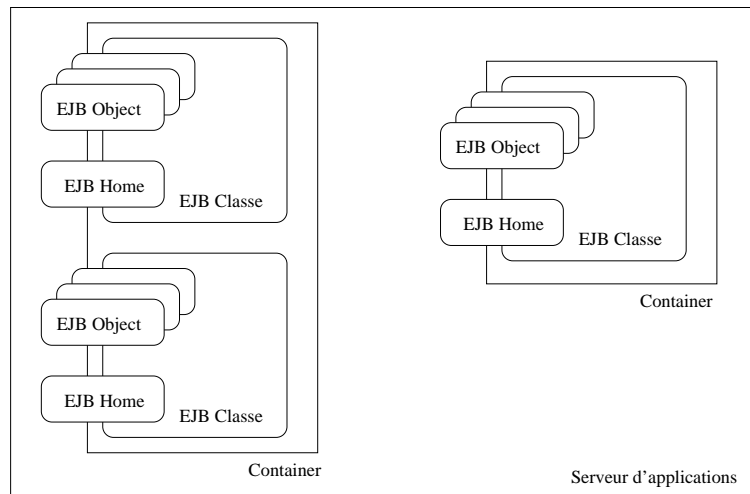


FIG. 2.7 – *Environnement d'exécution des Enterprise Java Beans.*

L'environnement d'exécution

L'environnement d'exécution se découpe en deux parties : un ou plusieurs container (structure d'accueil) et un serveur (type serveur d'application). Les différents containers sont contenus dans le serveur. Plusieurs serveurs peuvent être mise en œuvre pour une application, mais en général il semble qu'un seul serveur par hôte soit le bon choix (un serveur peut être multi-threads).

Un container peut recevoir plusieurs types d'EJBs. Comme le montre la figure 2.7, l'architecture de l'environnement d'exécution est en couche : les EJBs, les containers et les serveurs. Les containers interviennent dans les échanges entre client et instance d'EJB (médiateur). Les serveurs sont uniquement mis en œuvre pour héberger les containers (qui hébergent les instances d'EJB). Les containers sont localisés par l'application via l'API JNDI (Java Naming and Directory Interface).

La vue métier

Bien que notre découpage de l'industrie du logiciel soit inspiré de la spécification des EJBs, les deux découpages ne sont pas identiques. La spécification regroupe les trois métiers concepteur, développeur et fournisseur de composants sous une seule casquette : fournisseur de Bean. Le fournisseur système est découpé en deux rôles : fournisseur de serveur et fournisseur de container. Les rôles sont très proche du découpage *technique* de la proposition. Nous allons dans cette section présenter les rôles de l'industrie EJB selon notre découpage, pour avoir une vue globale uniforme des trois modèles.

Le concepteur d'EJB conçoit les interfaces des EJBs. Il va donc exprimer l'aspect métier d'un composant à l'aide d'une convention syntaxique (préfixage des opérations avec `ejb`) au sein d'une interface fonctionnelle. Ensuite il va définir les interfaces de maison et distante. Toute cette phase s'exprime sous forme d'interfaces Java.

Le développeur de composant va implanter la partie fonctionnelle des EJBs. Il est supposé pouvoir se concentrer sur cette partie fonctionnelle, et oublier les contraintes de transactionnel, les aspects systèmes, qui sont pris en compte par le container. La production du développeur correspond à la classe implantant les aspects métiers.

Ces différentes interfaces, et implantations sont reprises par le fournisseur de composants. Il va packager ces différents éléments au sein d'une archive (`.jar`). Pour permettre l'utilisation d'un EJB, le fournisseur produit aussi un descripteur de Bean. Ce descripteur donne des informations sur le producteur, sur ce que fait le Bean. Le descripteur fournit aussi une configuration initiale pour les instances de ce Bean et indique les ressources dont les instances ont besoin pour s'exécuter.

L'architecte d'application va, à partir de packages d'EJB, concevoir l'application. Il va produire un, ou plusieurs, fichiers archives de Beans. En plus des descripteurs existants, il va fournir les instructions de déploiement dans les descripteurs de déploiement.

L'installateur d'applicatif va mettre en place les packages produits par les fournisseurs d'EJB ou les architectes. Une fois les environnements d'exécution opérationnels, il va installer les packages et résoudre les dépendances. Ces dépendances sont exprimées dans les descripteurs de déploiement. L'installateur d'EJB va générer certaines parties de l'applicatif à l'aide d'outils fournis par le fournisseur de container (implantation des maisons et des interfaces distantes).

L'administrateur système se charge de rendre opérationnel les serveurs et les containers d'EJB. Son rôle est de configurer et de maintenir ces éléments pour que l'installateur déploie les applications et pour que les clients puissent les utiliser. Il doit aussi maintenir et monitorer l'exécution des applications déployées.

Le fournisseur système regroupe deux rôles (fournisseur de serveur et de containers) selon la spécification, mais aucune interface n'est définie entre ces deux entités : la spécification assume donc que les deux parties sont regroupée au sein d'une même entité. Dans le cadre des containers, le fournisseur système doit offrir un environnement d'exécution et les outils nécessaires au déploiement des applicatifs (générateur d'implantation de maison et d'interfaces distantes). Il doit de plus fournir les facilités de transaction, sécurité, persistance, ...

2.2.3 Les composants CORBA

Les composants CORBA ne peuvent pas être présentés au même titre que les JavaBeans ou les EJBs. La raison est simple : la spécification n'est pas terminée, ni voté (acceptée) par les membres concernés de l'OMG. Il n'y a donc actuellement aucune implantation des composants CORBA. Les informations contenues ici sont le résultat de l'étude de [BSCC⁺99a, BSCC⁺99b], qui contient l'état actuel de la proposition.

Dans le RFP² émis par l'OMG, la spécification des composants devait se baser sur le modèle des JavaBeans. Or, entre le RFP et les premières propositions, Sun a sorti les EJBs. C'est donc assez naturellement que la proposition actuelle soit basée sur le modèle des EJBs, plus complet et qui adresse des problèmes plus *industriels*. Le modèle actuel définit donc un framework de composants métiers qui sont totalement compatibles avec les EJBs.

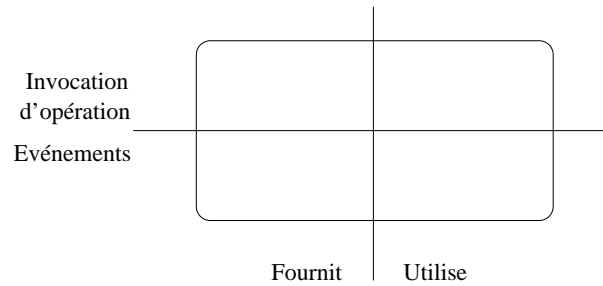


FIG. 2.8 – *Modèle très abstrait des composants CORBA.*

Le modèles abstrait

Le modèle abstrait des composants CORBA peut se lire selon deux axes. Premièrement, la lecture peut être verticale (par rapport à la figure 2.8). D'un premier côté, il y a ce que le composant offre comme services. C'est l'interface du composant pour les clients. D'un autre côté, il y a ce que le composant utilise. Le fait d'utiliser des services offerts par d'autres entités logicielles permet au composant de faire de la délégation.

2. Request For Proposal

Le second axe de lecture est horizontal (toujours par rapport à la figure 2.8). Une première possibilité est l'utilisation d'interfaces CORBA pour l'invocation d'opérations à distance. Cela représente l'approche *traditionnelle* dans le monde CORBA. La seconde possibilité est l'utilisation d'événements. Elle apporte l'asynchronisme dans les échanges entre deux entités logicielles. Les événements du modèle de composant sont compatibles avec le modèle *publish/subscribe* de CORBA.

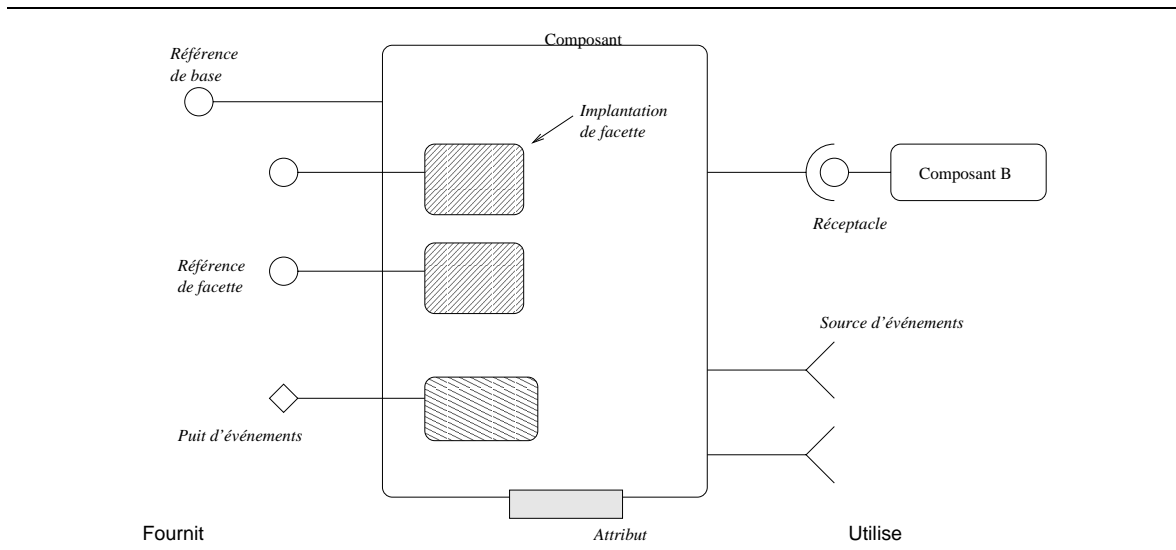


FIG. 2.9 – Modèle abstrait des composants CORBA.

La figure 2.9 présente plus en détail le modèle abstrait de composants CORBA. Le côté *fournit* d'un composant propose au client la référence de base du composant. C'est toujours par cette référence que commence une interaction entre un client et un composant. Cette référence de base permet au client de récupérer des références de facette sur le composant. Une facette est une vue sur le composant. Elle propose une partie des services du composant. Un client du composant peut naviguer entre les facettes pour changer la vue qu'il souhaite avoir sur le composant. Une fois qu'un client dispose d'une facette il peut invoquer toutes les opérations qu'elle offre. Les événements représentent le second type d'interactions qu'un client peut avoir avec un composant. Dans cette seconde approche, le client s'abonne aux événements générés par le composant. L'émission d'événements se fait au travers d'un canal d'événements géré par l'environnement d'exécution sous-jacent.

Le côté *utilise* d'un composant exploite aussi les deux types d'interactions. D'une part un composant peut disposer de réceptacles. Ces réceptacles permettent la composition par *branchement* d'un composant sur un autre (via la référence de base du second). Ce branchement permettra à un composant d'utiliser les services offerts par un autre composant (ou tout entité logicielle disposant d'une interface IDL). De ce côté aussi, les événements peuvent être mis en œuvre. Deux manières sont possibles. Premièrement, l'émission d'événements peut être directe entre le composant et le consommateur (connection un vers un). Dans ce cas il n'y a pas de canal d'événement mis en œuvre et un seul client peut être connecté à la fois. Deuxièmement, l'émission d'événements peut être réalisée via un canal d'événements. Dans

ce cas, il peut y avoir plusieurs consommateur simultanés d'un même événement. Le canal est mis en place et géré par l'environnement d'exécution.

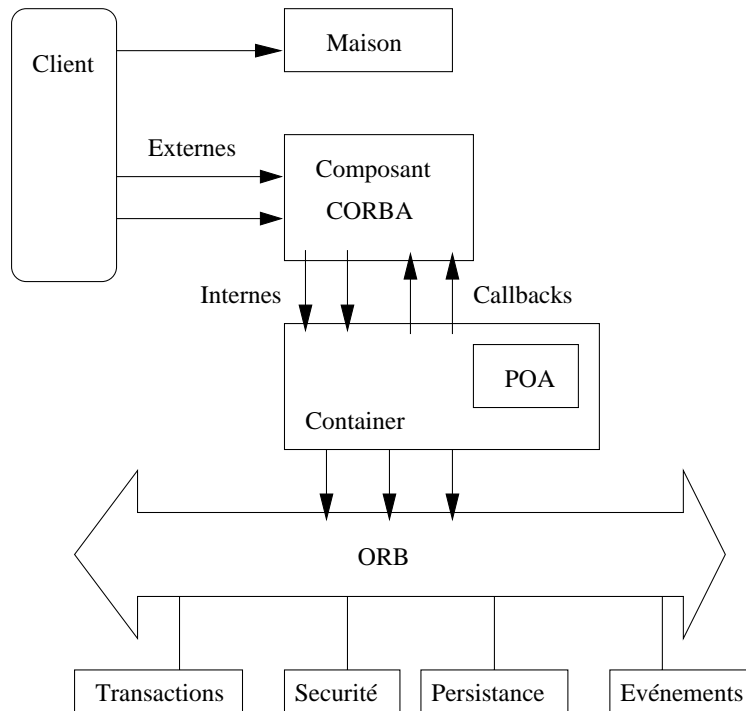


FIG. 2.10 – L'architecture d'un container.

L'environnement d'exécution

Comme dans le cadre des EJBs, les composants CORBA s'exécutent dans un container qui leur offre d'une part un environnement d'exécution (espace mémoire, flot d'exécution), et d'autre part les services systèmes nécessaires au composant. La complexité de ces derniers est masquée au composant, qui ne se préoccupe plus de maîtriser ces aspects. Les containers se déclinent selon différents types de base, mais sont génériques par rapport à une même famille de composant (service, session, processus, entité).

Les échanges entre composant et container sont définies par des APIs standardisées. Le container offre les aspects système. Le composant dispose d'une interface de callback permettant au container d'agir dessus (principalement pour rendre le composant persistant).

Un container n'instancie pas directement un composant. Un composant est fourni avec une maison de composant qui est une implantation du pattern fabrique. Cette maison permet d'instancier des composants en début d'exploitation (instances de composants persistantes – processus, entité) ou à la demande de l'utilisateur (instances de composants volatiles – service, session). Une maison de composant n'est pas implantée mais générée pour chaque type d'instance de composant. Une maison ne peut donc gérer qu'un seul type d'instances.

La vue métier des composants CORBA

Pour exprimer sa conception, le concepteur de composant dispose du langage IDL étendu (avec les nouveaux méta-types de la spécification composants). Ce langage de description va permettre d'exprimer les services fournis par un composant ainsi que les services requis par un composant. La connectique entre instances de composants est exprimée sémantiquement (par des mots clés réservés) et non syntaxiquement (par des règles de nommage des méthodes).

Le développeur de composant va implanter le composant défini par le concepteur en plusieurs étapes. Premièrement, il va projeter la définition en IDL vers son langage de programmation favori. Ensuite, il va implanter la partie fonctionnelle du composant. Puis, il décrira son implantation en CIDL (Component Implantation Description Language) dans le cas où celle-ci requiert des besoins en persistance. Ce langage décrit les structures de données internes au composant pour permettre la persistance des instances de celui-ci. Enfin, le code *système* (sources, squelettes, ...) sera généré à partir de la spécification et de la description de l'implantation du composant.

Le fournisseur de composant va reprendre la spécification, l'implantation et la description de l'implantation pour générer un descripteur de composant. Ce descripteur regroupera des informations sur la spécification du composant, les différentes parties du composant (code exécutable), les prérequis (OS, machine, bibliothèques, ...), ainsi que des informations sur la conception : entreprise, auteur, langage, compilateur, ... Ce descripteur sera ensuite ajouté à une archive contenant l'interface IDL et l'implantation du composant. Les descripteurs sont écrits en OSD (Open Software Descriptor) qui est un langage de description défini sous la forme d'une DTD (Data Type Definition) XML (eXtensible Markup Language).

Pour concevoir une application, l'architecte va récupérer (ou acheter) les composants nécessaires pour mettre en place ses services. Il va les assembler entre eux en exploitant les descripteurs de composants. Une fois les assemblages mis en œuvre, l'outil d'assemblage doit permettre à l'architecte de générer un, ou plusieurs packages. Ces packages regroupent les différents composants nécessaires, un descripteur d'assemblage (écrit lui aussi en OSD) et une description de la configuration de l'ensemble par défaut. Un package devient alors une application ou une nouvelle unité réutilisable.

Une fois les applications conçues, elles sont diffusables (vendables). L'installateur va donc mettre en place une, ou plusieurs structures d'accueils (container) sur les sites cibles. Une fois ces containers prêts, les applications seront installées, et configurées. Cette seconde phase est réalisée à partir des informations contenues dans les descripteurs de package. Il est intéressant que cette phase soit automatisable au maximum pour pouvoir faire de la diffusion automatique de composants... La phase d'exploitation de l'applicatif, et donc des composants, peut alors commencer.

Pour que cette phase d'exploitation se passe bien, l'administrateur va superviser, faire évoluer, régler finement les paramètres, ... Pour cela il doit disposer d'une console de surveillance et d'action sur les paramètres de l'applicatif. En d'autres termes il doit pouvoir agir sur la configuration des containers et des composants. Le remplacement de composants à *chaud* est une facilité qu'il semble possible d'attendre pour un administrateur. Cela permettrait de suspendre une application plutôt que de l'arrêter lors de la mise à jour de certains composants.

Transversalement à tous ces métiers, le fournisseur système va offrir un environnement de conception (passage d'UML à l'IDL), un environnement de développement (projection

de l'IDL, compilateur, ...), des outils pour faire du packaging (production d'archives et de descripteurs), des outils de composition graphique (assemblage de composant, génération d'archives et de descripteurs) et bien sûr les structures d'accueil et les outils associés (container, tableau de bord, services systèmes, ...).

2.2.4 Vue d'ensemble

Le tableau suivant présente un récapitulatif de l'analyse des trois modèles de composants par rapport aux métiers.

	Java Beans	Enterprise Java Beans	Composants CORBA
Concepteur de composant	Interface Java, règles syntaxiques	Interface Java, règles syntaxiques	Langage IDL étendu
Développeur de composant			Projections, CIDL
Fournisseur de composant	Outil de génération de .jar, BeanInfo	Outil de génération de .jar, descripteur	Outil de génération de .zip, OSD
Architecte d'application	Outil de composition graphique, BeanInfo	Outil de composition, descripteur	Outil de composition graphique, OSD
Installateur d'applicatif		Containers, descripteurs	Containers, OSD
Administrateur d'applicatif		tableau de bord	tableau de bord
Fournisseur système	Wizzard, outil d'assemblage graphique, générateur/lecteur de BeanInfo	container, serveur, OTM, générateur de code	Outil d'assemblage graphique, générateur d'OSD, containers, implantation des services

2.3 Premières conclusions de cette étude

Les composants représentent un pas de plus par rapport aux objets dans la voie de l'unité de développement réutilisable. Les composants apportent une certaine indépendance de contexte dont les objets ne disposaient pas. [CS96] propose une définition qui va dans ce sens :

Components are software units that are context independent both in the conceptual and the technical domains.

L'autre avancée (mis en avant ici) se résume en "*Les composants adressent mieux l'industrie du logiciel que les objets*" dans le sens où il touchent un plus grand nombre de métiers.

Un autre aspect positif des composants, qui nous intéresse tout particulièrement, est la possibilité de faire de la distribution de code. Les composants facilitent la mise en œuvre d'applications réparties. Les composants explosent l'aspect monolithique de la programmation par objets, les objets étant moins facile à distribuer que les composants.

Les composants changent la manière dont nous allons concevoir des applications dans le futur. Le métier de programmeur tend déjà à disparaître. Les composants vont accentuer ce

phénomène. En effet, lorsque l'on voudra bâtir une application, le travail (après la conception, bien sûr) sera d'assembler des composants existants. Une fois les assemblages réalisés, le plus gros du travail sera de les configurer. Le producteur d'application sera donc un assembleur/configurateur qui travaillera avec des outils graphiques et des langages de scripts.

Les composants sont donc très prometteurs, cependant il reste une ombre au tableau : il n'existe pas de modèle universel de composants. Il faudra donc choisir un modèle en fonction des applications que l'on souhaite mettre en œuvre. Ce défaut ne sera pas trop gênant aussi longtemps que les composants seront interopérables (cas des EJBs et des composants CORBA pour l'instant) pour ne pas tomber dans le travers de composants propriétaires. Ce travers nous ramènerait au même problème : un choix de modèle implique un choix de fournisseur, d'outils, de services, . . . et donc un avenir lié à ce fournisseur.

Certains éléments sont toutefois récurrents dans les trois modèles que nous avons présenté. Premièrement, les trois modèles exploitent le concept d'introspection. Dans le cadre des Beans de Java, l'API de réflexion du langage est mise en œuvre. Dans le cadre des composants CORBA, l'introspection se fait au travers de descripteurs. Deuxièmement, chaque modèle met en œuvre des structures d'accueil pour recevoir les composants. Les structures d'accueil présentées dans le cadre de ce chapitre sont variées : du simple container graphique (JavaBeans) au serveur d'application (EJB et composants CORBA). Ces deux éléments seront repris dans la conception de notre modèle de composants, et présentés dans le chapitre 3.

Chapitre 3

L'architecture à composants de GOODE

GOODE Component Architecture (GCA) est l'architecture à composants pensée durant ce mémoire de DEA. Cette architecture résulte de nos expérimentations. Elle se différencie un peu des environnements présentés dans le chapitre précédent dans le sens où elle ne propose pas un environnement de développement, mais uniquement un support d'exécution (dans un premier temps).

La première partie de ce chapitre va s'attarder à la définition d'un composant à *la sauce* GOODE. Une seconde partie, plus importante, va présenter le support d'exécution. Enfin, nous verrons le modèle d'exécution sous-jacent à nos composants.

3.1 Vue d'ensemble

Nous allons présenter ici l'architecture à composants de GOODE. Le but n'est pas la comparaison point par point avec les autres architectures, mais plutôt de fixer le cadre de notre travail : les contextes d'utilisation de GCA. Il est néanmoins évident que notre réflexion a été influencée par les modèles existants.

3.1.1 Applications

Dans le contexte du projet GOODE, nous travaillons sur les applications distribuées à objets. C'est donc naturellement que GCA vise à offrir un support d'exécution pour de telles applications. Le projet dans lequel GCA s'intègre, vise à aller encore un peu plus loin dans les travaux sur la modularité, la dynamique et la *scalability*¹ des applications distribuées.

De plus en plus, la réflexion au sein du projet GOODE s'attarde sur les applications réparties à grande échelle. Aujourd'hui, la majorité des applications réparties se trouvent sur un réseau local ou sur des réseaux plus importants, mais avec des débits rapides. Dans le cadre de GCA, nous visons les applications à grande échelle (à l'échelle d'un pays par exemple) pour lesquelles les interactions entre certaines entités sont coûteuses en temps.

3.1.2 Composants

Comme dans les modèles à composants en général, les composants GOODE sont des morceaux d'applications. Les composants GOODE peuvent être implantés sur des sites hôtes distribués tout en formant une même application. Ces sites hôtes n'ont pas obligation, dans

1. Au sens variation de l'échelle d'implantation.

notre cas, d'être reliés par des liaisons performantes, bien que celles-ci restent toujours un avantage indéniable.

Un autre aspect qui nous semble intéressant dans notre approche est le fait que la composition des différentes parties d'une application se fait toujours à l'exécution (les applications ne sont jamais monolithiques). Cette composition peut être mise en place de deux manières. Soit une composition statique, fixée à la conception, par l'utilisation explicite d'un composant nommé. Soit une composition dynamique, prévue à la conception, par recherche d'un composant qui propose un certain type de services. Dans ce dernier cas, l'instance de composant qui sera utilisée est totalement inconnue à la conception de l'application.

Exemple :

Lors d'un problème de baignoire, le problème est toujours le même, il faut un plombier pour réparer. Dans tous les cas, ou presque, le plombier n'est pas déjà dans la pièce, mais il faut le faire venir : composition à l'exécution. Néanmoins il y a deux manières de faire venir le plombier :

1. Je connais le nom du plombier que je vais appeler, j'appelle **ce** plombier : ce que nous appelons la *composition statique*.
2. Je ne connais pas le plombier que je vais appeler, je recherche **un** plombier : ce que nous appelons la *composition dynamique*.

Dans les deux cas, le but est de réparer la canalisation. Ce que tout plombier peut faire.

3.1.3 Interactions

Dans toutes les interactions, le modèle utilisé est celui des invocations à distance de CORBA. Ce modèle peut être interprété de deux façons :

- Interaction par invocation de méthode sur un objet : on dispose d'une référence sur les entités logicielles dont j'ai besoin des services. Cette interprétation est comparable aux RPC² de Sun.
- Interaction par envoi de messages : on connaît les autres entités et je leur envoie des requêtes de traitement à l'aide de messages. Les entités me répondent de la même manière. C'est une interprétation à la *Smalltalk*.

Que ce soit entre l'application et les composants, ou entre composants, ce modèle est toujours le même. Dans le cas où un modèle événementiel est requis, le service CORBA Events est utilisé, ce qui nous ramène à ce même modèle d'interaction, avec l'ajout de l'asynchronisme. Tout comme une application peut avoir besoin des services d'un composant, un composant peut avoir besoin de services d'un autre composant. Cette dépendance n'est pas explicite au niveau de l'application, et elle est gérée de manière transparente par le composant.

Exemple :

Revenons à notre problème de baignoire qui fuit. Dans le cas où la baignoire est encastrée dans le mur, le plombier ne peut y accéder facilement. Le plombier a besoin d'un maçon pour faire une saignée dans le mur, –pour avoir accès à la

2. Remote Procedure Call

tuyauterie– puis la reboucher. Ceci n’est pas le problème de l’usager qui a appelé le plombier pour corriger son problème³.

Le plombier va donc sous-traiter les opérations sur le mur à un maçon sans que l’usager soit nécessairement au courant. Le plombier dispose de deux méthodes de composition : il connaît un maçon (statique) ou il recherche un maçon (dynamique). Les interactions entre plombier et maçon sont sensiblement similaires aux interactions entre l’usager (de la baignoire) et le plombier.

3.2 Composant logiciel

Nous allons ici définir plus précisément ce que nous appelons composant dans le cadre du projet GOODE.

3.2.1 Définition d’un composant GOODE

Un composant GOODE est défini essentiellement par trois critères.

Entité logicielle autonome

Un composant GOODE est une entité logicielle autonome. C’est un ensemble d’objets masqués derrière une interface unique et non liée à une application ou un programme particulier. Dans le cas où un composant utilise les services d’un autre composant, il n’y est pas fortement lié. Il y a indépendance du code et indépendance de l’utilisation. L’un peut être utilisé sans l’autre et vice-versa.

Dans le cadre des composants GOODE, il n’existe qu’un lien entre deux entités : l’interface de l’entité qui fournit des services. Toute interaction se fait au travers de cette interface. Il n’existe donc pas de dépendance entre les deux composants au niveau du code, si ce n’est le fait d’avoir une référence sur une interface connue. Cette approche est l’approche *standard* dans l’environnement CORBA.

Indépendance vis-à-vis de la structure hôte

Un composant GOODE est indépendant de la structure d’accueil dans laquelle il s’exécute. Un composant peut donc être utilisé dans n’importe quelle structure d’accueil du système. De manière plus concrète, un composant ne dépend pas de ressources locales à une structure d’accueil, ressources qui seraient inaccessibles à distance.

Nous pensons que cette indépendance est toujours réalisable. Les ressources locales, par exemple des ressources matérielles, sont encapsulées dans des proxys logiciels accessibles de n’importe quelle structure d’accueil du système. Ceci est très important pour permettre la diffusion de composants, sans contraintes fortes, et surtout pour permettre leur mobilité⁴.

3. Si c’est pas malheureux de devoir appeler un plombier pour corriger ce problème de la baignoire qui fuit. Pour information : le temps de remplissage est inversement proportionnel au débit de la fuite.

4. A l’heure où ce document est apperçû, il n’y a pas encore de guideline d’utilisation pour ces proxys.

Etat externalisable

Une instance de composant se différencie d'une autre instance de ce même composant par son état. Chaque instance est unique dans le sens où elle dispose de son propre état qui évolue indépendamment des autres instances. Deux instances de composants sont donc logiquement différenciables par leur état, une différence technique prendrait en compte le fait qu'elles ont des noms et des références différents.

Un composant GOODE dispose d'un état externalisable. Il est possible d'extraire l'état d'une instance de composant. Une fois cet état extrait, il est possible de le rendre persistant, de l'émettre vers une autre structure d'accueil, de le réinjecter dans une autre instance de composant. Cet état est représenté sous forme d'une structure IDL. Il peut donc être exploité dans différentes implantations de composants.

3.2.2 Parties d'un composant GOODE

L'interface d'un composant GOODE est une interface OMG IDL *classique*. Nous n'avons pas étendu le langage IDL pour l'utilisation de nos composants. Ce choix vient principalement du fait que nous souhaitions pouvoir réutiliser tout objet CORBA *bien écrit*⁵.

L'implantation d'un composant GOODE est une implantation d'objet à la CORBA. C'est à dire qu'elle résulte de la projection de l'interface IDL sus-mentionnée. Cette implantation peut être réalisée dans plusieurs langages cibles. Le but de cette caractéristique est la possibilité de pouvoir utiliser un composant sur un grand nombre d'architectures.

L'état d'un composant GOODE est, comme nous venons de le voir, externalisable. Il faut donc offrir des mécanismes permettant d'extraire cet état. Dans le cas présent, le concepteur d'un composant doit fournir deux scripts qui exploitent l'interface de l'objet pour permettre l'extraction et la réinsertion de l'état d'un composant.

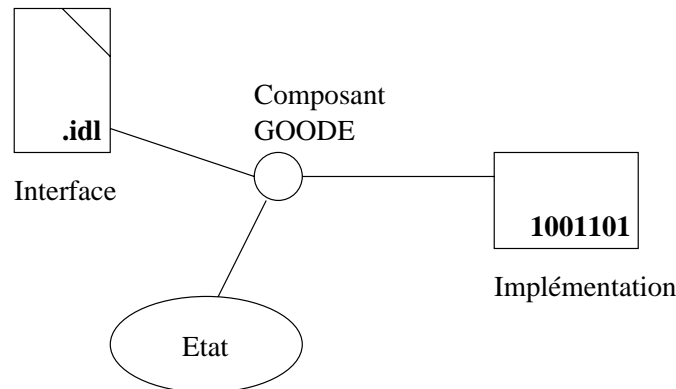


FIG. 3.1 – Les différentes parties d'un composant GOODE.

Les informations sur ces parties sont regroupées dans un descripteur de composant.

5. Bien écrit dans le sens qui répond au critère énoncés dans la section 3.2.1.

Le descripteur de composant

Un descripteur de composant⁶ regroupe des informations sur les trois éléments du composant que nous venons de voir. Ces informations localisent le site où sont stockées l'interface et les diverses implantations du composant. D'autres types d'informations sont aussi contenues dans le descripteur. Il y a d'une part des renseignements sur le concepteur du composant (nom, adresse W3, pointeur sur la licence d'utilisation, aperçu de ce que fait le composant, ...) et d'autre part les méta-informations sur l'état d'une instance de composant. Ces méta-informations se matérialisent en deux scripts fournis par l'utilisateur pour extraire/injecter l'état d'une instance de composant.

3.3 Structure d'Accueil de Composants

La structure d'accueil de composants (SAC) est le cœur de notre architecture. Une SAC offre un environnement d'exécution aux composants qu'elle *héberge*. Elle offre des facilités d'administration pour une application basée sur des composants. Elle permet enfin, de faciliter l'interconnexion de sites qui servent de support à l'exécution des applications. Une vue d'ensemble de la SAC est présentée par la figure 3.2.

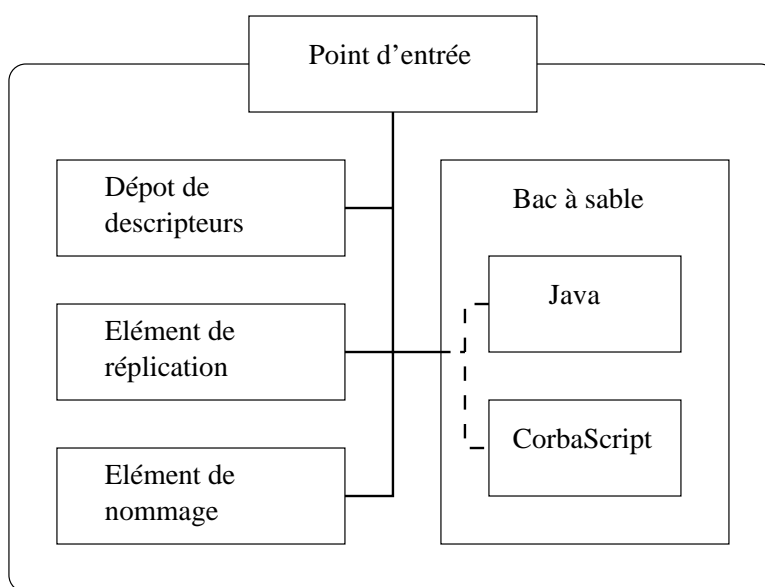


FIG. 3.2 – Vue d'ensemble d'une Structure d'Accueil de Composants.

3.3.1 Composition d'une SAC

Une SAC est composée principalement de quatre éléments : un dépôt de descripteurs, un élément de réplication, un bac à sable et un élément de nommage. En plus de ces quatre

6. L'interface IDL de ce descripteur est présentée dans l'annexe A.

éléments, un point d'entrée est intégré pour permettre l'utilisation, l'administration et la configuration d'une SAC à partir d'une console ou d'un navigateur Web.

Dépôt de descripteurs

Le dépôt de descripteurs regroupe les différents descripteurs de composants *connus* d'une SAC. Un descripteur de composant est *connu* d'une SAC si au moins une instance du composant qu'il décrit a été chargé dans la SAC, ou si le descripteur a été créé dans cette SAC. Le fait de conserver les descripteurs de composants dans chaque SAC permet à l'utilisateur d'avoir facilement accès aux descripteurs des composants qu'il utilise localement.

Le dépôt de descripteurs permet d'autre part d'avoir accès aux descripteurs de composants distants. Etant donné que nous sommes dans un contexte réparti, l'utilisation de SACs se fera majoritairement sous forme de grappes⁷. L'interconnexion de dépôts de descripteurs permet donc à l'utilisateur de rechercher les descriptions des différents composants disponibles dans l'ensemble de la grappe de SACs dont il fait parti.

Le dépôt de descripteurs permet la gestion des composants du système. Lorsque l'utilisateur souhaite charger ou cloner un composant distant, le dépôt de descripteurs sera utilisé par la SAC pour trouver les différentes parties du composants dans le système. Dans le dépôt de descripteurs, chaque descripteur est encapsulé dans un objet. Cet objet offre en plus de l'accès aux informations concernant le composant, des méthodes pour rapatrier localement l'interface et les implantations d'un composant.

Élément de réplication

Cet élément de la SAC exploite les méta-informations des composants contenus dans les descripteurs. Ces méta-informations permettent d'externaliser l'état d'un composant en vue de faciliter sa diffusion ou de rendre cet état persistant. L'élément de réplication utilise les deux scripts contenus dans le descripteur de composant pour extraire ou injecter l'état d'une instance de composant.

En plus d'exploiter ces deux primitives d'externalisation de l'état, l'élément de réplication permet de rechercher dans les différentes SACs de la grappe quelle est la SAC contenant l'instance de composant. Une fois cette SAC trouvée, l'élément de réplication va dialoguer avec le dépôt de descripteurs de cette dernière pour rapatrier l'interface et l'implantation du composant. Une fois ces deux parties rapatriées, l'élément de réplication va dialoguer avec son vis-à-vis pour recevoir l'état de l'instance du composant⁸.

Bac à sable

Le bac à sable est un élément optionel de la SAC. Il n'est nécessaire que pour faire de l'utilisation locale de composants définis à distance. Si une SAC ne sert qu'à faire de la diffusion de composants existants —des composants qui n'ont pas été conçus dans le cadre de GCA— l'utilisation d'un bac à sable n'est pas obligatoire. Néanmoins son utilisation facilite l'ajout de nouveaux composants dans le système.

7. Ensemble de SACs interconnectées –clusters.

8. Uniquement dans le cas du clonage de composant, pour le chargement de code cette dernière phase n'existe pas. Voir section 3.4.2 pour une présentation complète du clonage.

En effet, le bac à sable est pour les composants de GCA un environnement d'exécution. L'utilisation du bac à sable permet d'instancier un composant dans le système sans avoir à écrire un *programme serveur* qui héberge le composant. Ainsi lors de l'exécution, il est possible de charger dynamiquement un composant dans une SAC. Ce dernier point est très important car il permet le chargement de code dans le cadre de la diffusion de composants et dans le cadre du clonage.

Le bac à sable est une entité qui permet de charger des composants écrits dans différents langages. Toutefois, ce bac à sable n'est pas une machine virtuelle universelle. Il est composé de plusieurs *sous-bacs*, chacun étant une machine virtuelle pour un langage précis⁹.

Élément de nommage

L'élément de nommage permet de trouver les différents composants de la structure d'accueil. A chaque fois qu'une instance de composant est insérée dans une SAC, un lien est automatiquement fait entre son identifiant —au sens du descripteur— et sa référence CORBA dans le service de nommage. L'utilisateur pourra de plus donner un nom symbolique —en général plus explicite— à l'instance de composant. Lors du clonage d'une instance de composant, ce nom symbolique sera aussi dupliqué vers la SAC de destination.

L'élément de nommage est une extension du service de nommage¹⁰ défini dans les CORBA Services. Les extensions au service de nommage ne sont pas des ajouts de fonctionnalités, mais des modifications de comportements. Comme nous le décrivons plus en détail dans la section 4.3 l'extension au service de nommage tient dans l'interconnexion des éléments de nommage de chaque SAC.

Point d'entrée sur la SAC

Lorsqu'une application utilise des composants contenus dans une SAC, l'application n'utilise pas explicitement les services de la SAC. Elle utilise directement des objets CORBA via le bus. Le point d'entrée n'est donc pas à destination des applications. Le point d'entrée est en fait une *prise* pour *brancher* une console d'administration sur une SAC, et indirectement sur l'ensemble du système.

Cette console d'administration peut être de deux type. Elle peut être une application cliente CORBA *classique*. Elle se présente alors sous forme d'un programme (éventuellement graphique) qui comprend les souches de la SAC. L'utilisateur dispose alors de fonctionnalités prédéfinies sur une interface de la SAC. L'utilisation d'un interpréteur CorbaScript [LIF99] comme console texte de SAC est aussi une possibilité qui présente deux avantages: d'une part il n'est pas nécessaire d'écrire un programme client et d'autre part l'interpréteur permet d'exploiter facilement toutes les possibilités de la SAC, quelles que soient ses évolutions.

Le second type de console associe les deux aspects que nous venons d'aborder: d'une part une interface graphique simple et conviviale, et d'autre part la puissance et la souplesse des scripts. Cette solution s'appelle CorbaWeb[GGM95, Mer97]. CorbaWeb est une passerelle entre le monde CORBA et le Web. Elle permet au travers d'un navigateur d'interagir avec des

9. Dans le cadre de notre travail, les langages supportés sont CorbaScript et Java.

10. Voir chapitre 4 pour une présentation de ce service.

objets CORBA via des scripts. Ces scripts sont exécutés sous forme de CGI¹¹, ou sous forme de Servlets.

3.3.2 Interactions SAC-composants

Dans l'état actuel de l'architecture des SACs, les types d'interactions entre SAC et composants sont peu nombreuses. Contrairement au modèle des EJB et des CORBA Components, la SAC ne propose pas encore vraiment d'interface offrant des services au composant.

SAC → Composants

Les interactions SAC→composants sont principalement des interactions dont le but est d'exploiter l'état d'une instance de composant. Comme nous l'avons déjà vu dans la section 3.3.1, L'état d'une instance de composant est externalisable. La SAC est capable d'extraire cet état puis de le réinjecter dans toute instance de ce même composant. La SAC n'utilise pas directement l'interface du composant (comme dans les EJBs ou les CORBA Components) mais des méta-informations sur le composant.

Ce type d'interaction est utilisé dans le cadre du clonage d'instance de composant¹². Il peut aussi être exploitée dans le cadre de la persistance d'instance de composant. Une SAC peut implanter cette persistance de manière relativement simple¹³ pour permettre l'arrêt de SAC, et pour offrir des moyens de sauvegarde des instances de composants.

Composants → SAC

Actuellement les interactions composants→SAC sont des requêtes du composant vers la SAC pour se faire cloner ou migrer. Les opérations de clonage et de migration sont effectuées par la SAC. L'instance de composant demande à la SAC d'effectuer l'opération (ou les opérations) adéquates.

Un autre type d'interaction composants→SAC est la demande de ressources par une instance de composant à la SAC. Comme nous l'avons déjà mentionné, un composant est indépendant de toute ressource "physique" de la SAC. Il faut donc que la SAC lui offre des ressources qui soient accessibles quelque soit l'emplacement de composant dans le système (principalement pour permettre la migration de composants). L'instance fait donc une demande d'accès à une ressource, plutôt que d'utiliser directement cette ressource. Ce point est discuté plus longuement dans la section 5.2.2.

3.3.3 Interactions avec l'environnement

Il y a deux types d'interactions entre utilisateurs et SACs. Il y a les interactions utilisateur-SAC pour l'administration et les interactions utilisateurs-composants présents dans la SAC. Enfin il y a les interactions entre SACs, interactions qui résultent de l'interconnexion de SACs

11. Common Gateway Interface

12. Voir section 3.4.2.

13. Voir section 3.4.3.

Administration de SACs

Dans le cas de l'administration de SAC, l'utilisation de la SAC est explicite. L'administrateur utilise le point d'entrée de la SAC pour charger ou supprimer des instances de composants de la SAC. Via le point d'entrée, l'administrateur peut aussi demander le clonage ou la migration de composants du système. Ces interactions peuvent se faire avec tout type de client prévu à cet effet¹⁴.

Les outils d'administration permettent aussi d'avoir une vision de l'état de la SAC local¹⁵. Par extension, cette vision d'état peut s'étendre à tout le système, en fait à toutes les SACs connectées (voir un peu plus loin *Interconnexion de SACs*). Cette vision, en général au travers d'un browser, permet de prendre connaissances des composants offerts par l'ensemble des SACs. Il est possible d'implanter un trader qui exploite la vision du système pour faire de la recherche de composants par rapport à un besoin d'utilisateur.

Interactions entre l'utilisateur et les composants

Dans le cas des interactions entre utilisateurs (applications utilisateurs) et composants, la SAC ne joue plus de rôle central. En fait, l'utilisation de composants par les applications est faite de manière transparente par rapport à la SAC. Une application n'a pas connaissance de la SAC lors de l'utilisation de composants. Les composants sont utilisés comme des objets distants CORBA.

La connexion d'une application sur un composant se fait via le service de nommage. Tout composant présent dans une SAC est enregistré dans le service de nommage global du système¹⁶. Ainsi une application qui souhaite utiliser un composant va faire une recherche dans le service de nommage avec le nom d'une instance du composant souhaité. Dans ce contexte, la SAC héberge l'instance de composant, mais n'intervient pas dans l'utilisation de cette instance.

Interconnexion de SACs

Ce point est essentiel dans notre architecture. L'utilisation de SAC en mode *stand-alone* n'apporte qu'un container à l'utilisateur : cela évite uniquement l'écriture de serveurs pour contenir les composants. L'intérêt majeur des SACs est la possibilité de les connecter les unes aux autres. Cette connexion offre un système global, ou macro-système. Ce macro-système est la somme des sous-systèmes que sont les SACs.

Le but de notre architecture est de proposer le système global tout en masquant qu'il est la somme des sous-systèmes. Il y a existence globale du système qui doit être vu comme une entité "unique". Chaque SAC quand à elle devient une prise pour brancher une console sur le système et pouvoir connecter des applications aux composants offerts.

14. voir section 3.3.1.

15. Sur laquelle on branche l'outil d'administration.

16. Voir section 4.2.2 pour une discussion plus longue sur le service de nommage global du système.

3.4 Modèle d'exécution

Nous allons présenter tout au long de cette section le modèle d'exécution des Composants GOODE. Les deux éléments importants du modèle sont : l'instanciation de composant (insertion dans le système) et le clonage d'instances de composants. Ces différentes opérations sont effectuées à partir de la SAC. L'utilisateur demande, de manière explicite ou non, à la SAC de lui effectuer les opérations.

3.4.1 L'insertion de composant

Comme nous l'avons présenté dans la section 3.2.2, des informations sur les différents éléments d'un composant sont regroupées au sein d'un descripteur. Lorsqu'un nouveau composant est introduit dans le système, il est nécessaire de fournir un tel descripteur. Actuellement, le descripteur est fait *en ligne* à partir d'un interpréteur, mais la fourniture d'un outil visuel, simple de description de composant est un des prochains objectifs. Cette outil doit non seulement permettre l'écriture d'un composant, mais il doit aussi guider le concepteur du composant dans la description.

Une fois le composant décrit, les références sur les éléments fournies, l'outil offre une seconde fonctionnalité : il permet de charger le composant dans le système. Le chargement du composant peut se faire de deux manières. Il est possible soit de charger le descripteur uniquement, soit de charger le descripteur puis de créer une première instance (ou l'inverse). Ce choix dépend du type de composant chargé. Si le composant est un composant de service (sans état) alors il n'est pas nécessaire de créer une instance. Par contre, si le composant dispose d'un état qui caractérise ses propriétés, il est alors nécessaire de créer une instance, puis de la configurer. Dans ce dernier cas, il est aussi nécessaire de charger l'IDL du composant dans le dépôt des interfaces.

L'insertion d'un composant peut donc se résumer à l'écriture d'un descripteur de composant, et au fait de rendre accessibles les différents constituants (IDL, code source) pour qu'un utilisateur du système puisse faire une copie locale d'un composant ou un clone d'une instance de composant.

3.4.2 Le clonage d'une instance de composant

Après avoir brièvement présenté l'intérêt que nous voyons dans le clonage d'instances de composants, nous présenterons la manière dont le clonage s'effectue.

Objectif

Le but de cloner des instances de composants est de permettre la réalisation de copies conformes d'une instance. Cette copie conforme peut avoir deux utilités. D'un côté, elle permet d'avoir une instance de composant sur laquelle on peut appliquer des opérations sans nécessairement les conserver. Cette instance devient un brouillon permettant l'expérimentation.

Soit un composant décrivant le résultat d'expériences de physique, une instance sera alors le résultat d'une simulation. Elle peut servir à un physicien pour faire une étude sur le changement de comportement lors de modification de paramètres.

Seulement le résultat de la simulation ne doit pas être changé, d'une part parce qu'il est résultat de la simulation, et d'autre part car il sert de référence. Il est donc nécessaire que le physicien fasse une copie de la simulation sur laquelle il pourra travailler.

D'un autre côté, une copie conforme d'instance de composant n'est pas nécessairement localisé au même endroit que son instance mère. Cela nous apporte un atout dans le cadre d'applications à grande échelle : cela coûte cher d'invoquer de multiples opérations sur une instance de composant éloignée¹⁷, il est alors intéressant d'avoir une copie locale pour laquelle les invocations d'opérations sont peu coûteuses. Nous discuterons plus longuement de ce point dans le chapitre 5.

Dans l'état actuel des choses, il est possible de cloner une instance de composant pour le même langage que l'original, ou pour un langage différent. L'objectif est ici de permettre d'une part, de tirer profit d'implantations performantes par rapport aux machines, et d'autre part, d'offrir une plus grande variété d'implantations pour viser un plus large public.

Dans l'éventualité où je dispose sur une machine A d'une instance de composant écrite en Java que je souhaite cloner sur une machine B qui ne dispose pas de JVM. Il n'est pas possible de faire un clone de l'instance en Java. Supposons que B dispose d'un interpréteur CorbaScript, je peux alors faire un clone de mon instance à partir de Java et vers CorbaScript. Cela nécessite tout de même de disposer des deux implantations. Ce clonage inter-langage sera discuté à nouveau dans la section 5.3.3 à propos des implantations de composants en C++.

Le clonage d'une instance de composant se fait de manière similaire à l'injection d'une instance de composant dans le système. La différence la plus notable se trouve au niveau de la configuration. Ici, la configuration n'est pas effectuée par l'utilisateur. Elle est réalisée par le système pour avoir une configuration identique à l'instance que l'on souhaite cloner. Détaillons un peu les deux phases du clonage.

Clonage de l'implantation

Pour permettre l'utilisation du composant par la SAC, il faut que son interface soit connue. Pour cela, la première étape du clonage est la récupération de l'interface IDL du composant. La localisation de cette interface est extraite du descripteur de composant. Une fois l'interface rappatriée, elle est chargée dans le dépôt des interfaces pour être exploitée par la SAC, et par tout interpréteur CorbaScript. Connaissant cette interface, la SAC peut désormais interagir avec le composant d'origine.

Cloner une instance de composant implique avant tout de pouvoir créer une instance de composant (oublions la sérialisation qui n'offre pas les possibilité de clonage inter-langage). Pour être en mesure de créer une instance de composant il faut disposer de son implantation. La SAC va donc rapatrier (télécharger) le code d'implantation du composant. Ce code va ensuite servir à instancier le composant désiré.

Cette instanciation est possible car la SAC dispose de tout : interface et code. Elle peut donc charger l'implantation du composant dans le bac à sable et demander à ce dernier de

17. Dans le sens *au dessus d'un réseau lent*

créer une instance du composant. Arrivé à ce point, nous disposons d'une instance neutre (non configurée) d'un composant, qu'il faut donc configurer.

Clonage de l'état

La configuration de l'instance de composant n'est pas une configuration *standard* (appliquée à toute nouvelle instance). La configuration de la nouvelle instance doit être identique à celle de l'instance originale. Pour cela, le fait que l'état d'un composant soit externalisable est exploité. La SAC duplique sous forme de structure IDL (de manière indépendante du langage) l'état de l'instance originale. Elle peut ensuite réinjecter dans la nouvelle instance de composant cette représentation de l'état.

Cela donne désormais deux instances de composants identiques : d'où le terme de clone. Cette opération peut être effectuée plusieurs fois, et peut ainsi permettre de produire autant de clones que souhaité. La SAC termine son traitement en générant un identifiant de la nouvelle instance de composant qui sera enregistré dans le service de nommage. Cet identifiant, unique dans le système global, sera utilisé pour retrouver cette instance particulière et l'utiliser (voire la cloner).

Cohérence

Le clonage étant désormais terminé, les deux instances de composants sont deux entités autonomes. Tout comme les clones végétal/animal, ces deux entités vont *vivre leur vie*, plus rien ne les relie plus qu'une autre instance du composant. Cela veut dire qu'il n'y a pas de cohérence entre ces deux instances. Une modification de l'état d'une des instances ne sera pas reportée sur l'autre (et vice versa).

Ce choix se justifie par le type de composants adressé par le projet : les composants métiers. Les composants métiers sont des composants de traitement plus que des composants de données (bien que les deux soient fortement liés dans le monde objet). D'autre part, les composants de *physiciens* présentés dans la section précédente, montre que même pour des composants dont le but premier est de fournir des données, ce choix reste adéquate dans certains cas. Une piste intéressante à explorer est celle du composant multi-comportemental : l'utilisateur choisit s'il souhaite ou non de la cohérence entre les différentes instances lors de clonage.

3.4.3 Persistance d'une instance de composant

La persistance est un élément qui n'est pas encore totalement intégré à l'architecture, mais sa prise en compte est déjà *prête*. De manière assez simpliste, la persistance est la possibilité de conserver l'état d'une entité dans le but de la réutiliser ultérieurement, principalement pour que sa durée de vie soit supérieur à la durée de vie du programme. Nous n'adressons pas ici les notions de reprise sur erreur, commit, rollback, ... Ce que nous souhaitons offrir est une persistance souple qui offre à l'utilisateur de sauvegarder ses objets explicitement.

Nous avons vu dans le cadre du clonage, comment nous exploitons l'externalisation de l'état d'un composant. Cette technique est à notre avis une manière simple, dans le contexte actuel, de fournir la persistance d'instances de composants. Il est très facile en CorbaScript de sauvegarder l'état d'une instance exprimé sous forme de structure IDL. Rendre une instance

persistante revient donc à sauvegarder cet état dans un fichier. Pour réutiliser une telle instance de composant, la SAC utilise le modèle d'insertion d'instance de composant dans le système (comme décrit en 3.4.1) puis remplace la phase de configuration par le chargement du fichier pour reconstruire l'état, et l'injection de cet état dans l'instance du composant (comme cela est fait pour le clonage).

3.5 Conclusion

Dans l'état actuel des choses, l'architecture décrite offre des moyens, mais ne résoud pas tous les problèmes (ce qui est plutôt normal). Beaucoup trop d'opérations sont *manuelles*, nous souhaitons aller plus loin en automatisant un maximum de traitement : le clonage n'est plus explicite, mais régit par un ensemble de règles.

Nous souhaitons aussi offrir des facilités qui ne sont pas encore mises en œuvre : la cohérence peut exister entre deux instances après un clonage, il est possible d'utiliser des assemblages de composants (actuellement le clonage n'est réalisable que sur un objet unique, pas sur un graphe de composants) ...

D'un autre côté, nous souhaitons continuer dans la voie de l'indépendance des composants. La composition, comme dans le cadre des CORBA Components, doit être totalement externe au composant. Ainsi la déconnexion et la reconnexion de composants peuvent être grandement facilitées voire automatisée dans un certain nombre de cas.

Des mécanismes peuvent aussi être mis en place pour faire du chargement et déchargement automatique d'instances lorsqu'un utilisateur se connecte à une SAC. Cela permet à un utilisateur de configurer son environnement et de le retrouver à chaque connexion, quelque soit la SAC où il se *branche*. Mais cela est une autre histoire, qui démarrera très prochainement ...

Enfin, les SACs sont destinées à être connectées entre elles. Comme elles ne sont que des points d'entrée dans le système une cohérence globale est nécessaire. Il faut donc permettre une vision globale du système. Cette vision globale nécessite une gestion du nommage à l'échelle du système, et pas seulement à l'échelle d'une SAC. Nous allons discuter de ce point tout au long du prochain chapitre.

Chapitre 4

Gestion du nommage

L'utilisation de composants distribués offre une multitude de possibilités : profiter de la puissance de différentes machines, tirer parti des atouts d'une architecture et d'un langage pour concevoir des applications plus performantes, . . . Cependant, avoir des composants distribués n'offre pas grand intérêt si ces composants ne sont pas facilement accessibles.

Tout comme le DNS dans le cadre des réseaux informatiques, le Naming Service (NS) de CORBA est un élément clé de l'architecture. Il permet de prendre connaissance des composants disponibles sur le bus, et donc d'y accéder.

Ce chapitre va dans un premier temps présenter le NS de CORBA. Nous verrons dans un deuxième temps l'utilisation d'espaces de nommage dans le cadre de notre projet, ainsi que le pourquoi de leur fédération et sa mise en place. Pour terminer, nous discuterons des propositions mises en œuvre dans ce projet.

4.1 Le service de nommage

4.1.1 Kesako

Le service de nommage est un service de désignation, il permet à partir d'un nom de retrouver un objet dans un système. Ce service est un annuaire similaire aux pages blanches. Lorsque l'on souhaite rendre un objet accessible à tous, on l'enregistre dans le NS en lui associant un nom. Le NS permet, à l'exécution, de retrouver dynamiquement des objets du système.

Un atout du NS est l'utilisation d'un nommage uniforme des objets dans le système. Le nommage des objets ne dépend pas du langage d'implémentation des objets, du système d'exploitation sur lequel il s'exécute, ni du type d'objet (persistant, volatile, . . .) ni de la localisation.

4.1.2 Vue macroscopique

Si l'on regarde le NS de manière globale, il se présente comme un objet contenu dans un serveur. Ce serveur est, en général, un processus qui s'exécute comme démon sur une machine du réseau. L'objet est un quant à lui un "objet particulier" du bus. Il est accessible grâce à l'appel de la méthode `CORBA::ORB::resolve_initial_references()` avec le nom `NameService` comme argument.

L'objet NS est un objet avec une interface simple : il propose principalement deux méthodes. La première méthode permet d'ajouter une relation entre un nom et une référence.

Cette méthode est utilisée par chaque programme qui veut offrir un accès facile à ses objets. La seconde méthode permet de retrouver une référence à partir d'un nom symbolique.

4.1.3 Vue détaillée

Si nous regardons le NS avec un peu plus de détails, nous pouvons constater que le service de nommage est moins élémentaire. Le NS est en fait composé d'un ensemble de contextes de nommages. Ces contextes de nommages permettent l'organisation du NS.

Contexte de nommage

Le contexte de nommage est la *particule élémentaire* du service de nommage. Tout, dans le NS, est contexte de nommage. L'interface du NS est elle même une interface de contexte de nommage. Chaque contexte peut être vu comme un répertoire de système de fichier. Ce répertoire peut contenir des sous-répertoires et des informations.

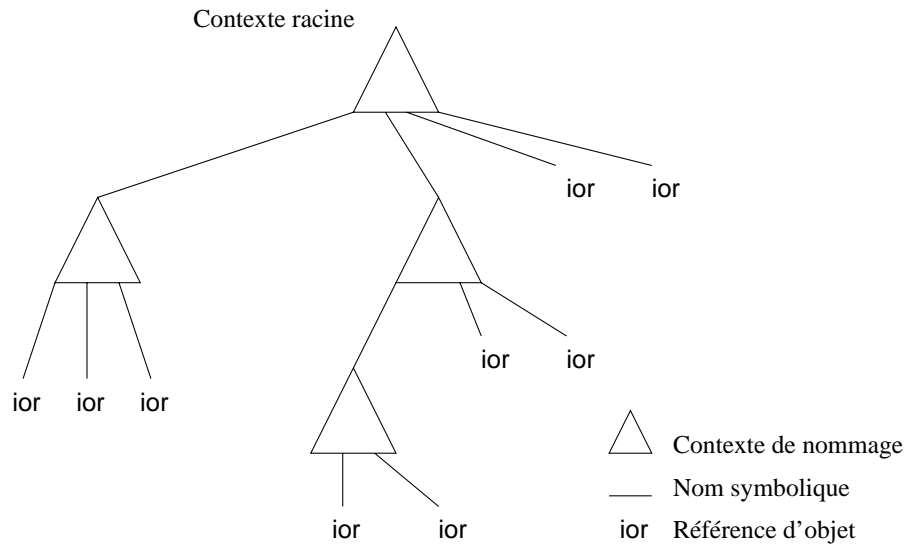


FIG. 4.1 – Arborescence de contextes de nommages dans un NS.

Ces informations sont en fait les associations nom symbolique-référence d'objet. Les références d'objets sont soit vers des objets applicatifs, soit vers des contextes de nommage. Comme les contextes de nommage sont des objets "classiques" CORBA, le nommage au sein du NS est uniforme quels que soient les objets référencés, (voir FIG. 4.1). Un nom symbolique dans le NS référence soit un objet utilisateur du bus, soit un contexte de nommage.

Organisation du NS

Comme déjà mentionné, le NS peut être comparé à un système de fichier. Les contextes de nommages sont organisés sous forme de graphe. Un nom symbolique complet est équivalent au chemin absolu d'accès à une référence à partir du contexte racine du graphe de contextes de

nommage. Il est composé du nom de l'objet préfixé par l'ensemble des contextes de noms qui le contiennent. Le graphe du NS est parcourable de manière récursive. Comme ce graphe est un graphe d'objets, le parcours peut se faire par délégation.

Lorsque l'on souhaite faire une résolution sur un nom d'objet, on interroge un contexte de nommage. Si ce contexte ne contient pas le nom symbolique ni un de ses préfixes, il va lever une exception. Cette exception remonte jusqu'au client qui a invoqué le contexte racine. Dans le cas contraire, il regarde si le nom symbolique est une base (pas de préfixe). Si oui le contexte de nommage retourne la référence de l'objet, sinon le contexte de nommage délègue la résolution au contexte nommé par le premier préfixe.

4.2 Espaces de Nommage

Nous allons, dans cette section, étudier un peu plus longuement les espaces de nommage. Un espace de nommage correspond à un ensemble de contextes (un sous-graphe). Ils correspondent à une vue sur l'ensemble des informations. Nous parlerons plus précisément de leur utilité et de leur utilisation dans le cadre des systèmes à grande échelle.

4.2.1 Utilisation des espaces de nommage

Dans tous les cas, les espaces de nommage permettent d'identifier de manière unique les instances de composants du système. Dans le cas où un même nom (base) est donné à deux composants, les noms symboliques complets seront différents car au moins un de leurs préfixes seront différents. Dans l'éventualité contraire, le deuxième composant lèvera une exception au moment de son enregistrement dans le NS.

Un nom symbolique permet d'accéder à un seul objet, mais un objet peut être référencé par plusieurs noms symboliques. Ce point permet d'utiliser des espaces de nommage pour faire de la diffusion. Un site qui souhaite faire de la diffusion pourra utiliser un espace de nommage par système vers lequel il veut exporter des composants. Pour un composant donné, un nom symbolique (sans doute avec la même base) sera ajouté dans chaque espace de nommage concerné. Plusieurs noms pointeront donc vers le même composant.

Cette approche est intéressante et pratique. Toutefois, des problèmes apparaissent. Comment les différents sites se connaissent-ils? Chaque site dispose t'il de son propre espace de nommage, ou est ce que tous les sites partagent un même espace de nommage commun? Un site doit-il utiliser explicitement les différents services de noms dont il a besoin?

4.2.2 Un seul ou plusieurs espaces de nommage?

Un espace global nécessaire

Dans tout système d'information, un espace global de nommage est nécessaire. D'un côté, un espace global implique une commodité d'utilisation. Tout comme il n'est pas envisageable d'avoir plusieurs conventions de noms, il n'est pas souhaitable d'avoir plusieurs espaces de nommage. Dans le cas contraire cela reviendrait à avoir plusieurs systèmes quasiment sans rapport les uns avec les autres.

D'un autre côté, plus conceptuel celui-là, l'ensemble des sous-systèmes forment **le** système d'information. Dans ce cadre, pourquoi les différents sous-systèmes seraient isolés dans des espaces de nommages différents? Ils ont un sens réunis tous ensembles au sein du système global, donc il ne doit pas y avoir de cloisonnement dans l'utilisation de ces sous-ensembles. L'utilisateur a besoin d'une vue logique unifiée.

Un exemple de tel système est le Web, l'accès aux ressources se fait de manière unifié (les URLs) au sein d'un unique espace de nommage. Les morceaux du système sont quant à eux vraiment hétérogènes et dispersés aux quatre coins de la planète. Sans l'utilisation d'un espace de nommage global, le Web serait réservé à une minorité d'experts en accès non uniforme à des ressources distribuées. Le nombre de ressources accédées seraient en nombre plus faible car il serait plus difficile de les connaître.

Des implantations éparses

La vue unifiée et globale du système est un besoin indéniable. Toutefois, la réalité d'implantation est toute autre. Lorsqu'un système est mis en place sur un réseau local, la vue globale est simple à mettre en œuvre. Une machine du système sert de *référence* ou *point d'entrée*. Dans le cas d'un système à grande échelle, il n'est plus possible d'avoir un unique hôte de référence.

Premièrement, comment choisir sur quel site l'hôte de référence doit être situé? Deuxièmement, si les connexions entre les différents points du système ne sont pas fiables, ou si les connexions ne sont pas permanentes, comment les applications peuvent-elles fonctionner lorsque le site de référence n'est pas accessible?

Il y a donc, dans le cadre des applications distribuées à grande échelle, un éclatement physique de l'application qui est difficile à masquer au niveau du nommage, bien que cela soit nécessaire pour tout utilisateur du système. Nous pensons que l'utilisation d'espaces de nommage fédérés est une solution à ce problème.

4.3 Fédération d'espaces de nommage

La fédération d'espaces de nommage est le fait de simuler un unique espace de nommage global à l'utilisateur pour masquer la répartition des différents *morceaux* de son système d'information. Dans le cas présent, le but premier est d'offrir un contexte de nommage racine qui soit unique pour tous les utilisateurs du système. Cette unicité ne tient pas dans la localisation, mais dans son identité. Le leitmotiv de la fédération d'espaces de nommage tient dans l'interconnexion des différents NS.

Plusieurs approches sont possibles, nous en avons étudié deux. Ces deux approches sont relativement extrêmes. Il est possible de mixer ces deux approches selon le besoin d'utilisation. Cependant, devant le grand nombre de variantes possibles, nous ne les discuterons pas ici. Nous terminerons cette section par une discussion sur ces deux approches qui, nous l'espérons, permettra au lecteur intéressé de définir sa variante en fonction de ses besoins.

4.3.1 Fédération à gros grain

Présentation

La fédération d'espaces de nommage à gros grain est une fédération qui, physiquement, se traduit par une fédération au niveau du processus. Chaque site, au sens ensemble de ressources relative à un même NS, dispose d'une politique unique de fédération. Cette politique de fédération décrit le comportement du Service de Nommage. Elle peut varier selon l'utilisation que l'on souhaite faire du NS. La politique que nous avons implanté correspond à une approche fédérante. Tant que les recherches se passent bien en local, le NS ne fait rien de particulier. Dans le cas contraire, le NS peut *forwarder* les requêtes à distance vers les autres NSs. Les ajouts sont quant à eux toujours locaux.

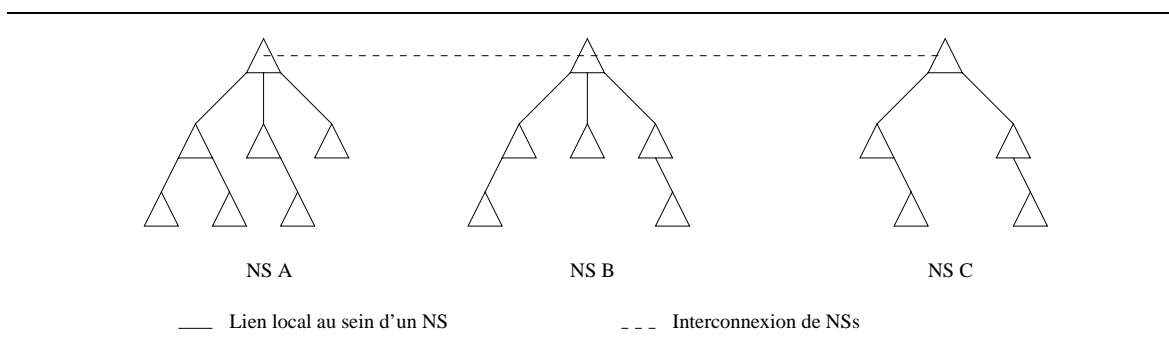


FIG. 4.2 – Fédération de services de nomenclatures à gros grain.

Au niveau de l'implantation, cette politique unique se traduit par une interconnexion globale des services de nomenclatures (voir figure 4.2). Lors de la recherche d'une information dans le NS, si l'information n'est pas trouvée localement, cette information sera recherchée dans les autres services de nomenclature interconnectés. Si l'information est trouvée à distance, elle sera recopiée dans le service local, puis retournée à l'utilisateur. Dans le cas contraire, une exception (NotFound) sera levée.

Cette utilisation du service de nomenclature est standard pour l'utilisateur qui, si l'on omet les temps de réponse dans le cas de recherche distante, ne sera pas conscient de l'interconnexion des NSs. L'implantation du *NS++* s'appuie sur un NS standard. Les implantations des contextes de nomenclatures sont identiques au NS classique. La seule différence tient dans le contexte racine qui fera des interrogations distantes avant de répondre que le binding n'est pas défini.

Avantages

Cette approche offre des avantages intéressants. D'une part la gestion de l'interconnexion est simple. Une fois le NS interconnecté, il n'y a plus d'opération à effectuer sur le NS pour le maintenir ou faire du tuning. L'administration du NS est donc pratiquement aussi simple que pour un NS standard. L'action supplémentaire tient dans l'interconnexion des NSs (qui se fait en invoquant une opération de connexion sur un NS avec les IORs des NSs distants comme paramètre).

Au niveau de l'utilisateur, l'utilisation ne diffère pas du tout de l'utilisation d'un NS standard. L'utilisateur ne voit donc aucune différence et, lorsqu'il n'y a pas d'interconnexion, le NS est un vrai NS standard. Cette approche offrant une fédération globale, comme pour un NS standard, l'utilisateur ne sait pas où sont localisés ses objets.

Inconvénients

Le principal inconvénient de cette approche tient dans sa conception. Le fait d'avoir une politique unique pour l'ensemble du NS implique que la gestion du NS est figée. Dans le cas où la politique est encapsulée dans un objet, elle pourra être modifiée à l'exécution, la gestion devient au mieux peu souple. La conception de ce NS découle d'une vue et d'une approche globale de la gestion du NS. Cet inconvénient est donc relativement inhérent à ce choix.

Pour ce qui est de l'implantation, l'approche que nous avons suivie est relativement coûteuse au niveau de la résolution. Si je ne trouve pas un binding en local et que le NS est connecté à cinq autres, alors je peux faire cinq invocations à distance. Ces invocations étant séquentielles, les performances ne semblent pas très prometteuses. Les implantations d'une politique particulière peuvent améliorer ces performances, par exemple rappatriement de tout un contexte lors d'un resolve distant, notification des ajouts, . . .

4.3.2 Fédération à un grain très fin

Présentation

A l'opposé de la fédération à gros grain que nous venons de présenter, nous avons aussi expérimenté la fédération à un grain très fin. Dans ce cas, la connexion entre NSs se fait au niveau des contextes. Plutôt que d'avoir une connexion au niveau de la racine, il n'y a que certains contextes de l'arborescence qui sont connectés avec un NS distant (voir figure 4.3). Pour reprendre notre parallèle avec un système de fichier (FS), dans le cas d'un FS Unix il existe des répertoires particuliers pour faire du partage via NFS¹, répertoires qui sont déclarés dans `/etc/export` en général. Cela permet de partager une partie d'un FS entre plusieurs machines.

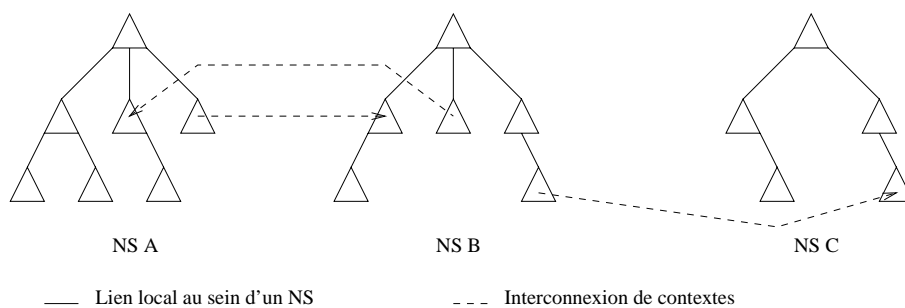


FIG. 4.3 – Fédération de services de nommages à grain très fin.

1. Network File System

Dans le cas de la fédération à gros grain nous avons vu que l'implantation du NS est classique. Dans le cas du grain très fin, il nous a été nécessaire de faire une implantation particulière des contextes de nommage. Les contextes ont toujours l'interface standard d'un contexte de nommage, ce qui ne change rien pour l'utilisateur, mais des opérations ont été ajoutées pour la gestion des connexions. Cette gestion nécessitera la présence d'un administrateur qui pourra ajouter/supprimer des liaisons entre contextes au cours de la vie du NS (de son processus hôte).

La gestion n'est plus, ici, une gestion globale et figée au lancement du NS. La politique de fédération peut être la même pour tous les contextes interconnectés, bien sûr, mais plusieurs politiques peuvent être mises en place pour différentes inter-connexions. Cela permet une gestion fine du comportement. Une interconnexion entre deux sites pourra être gérée de manière fénéante² pour certains contextes qui évoluent peu, et une autre interconnexion gérée de manière active (notification des changements à tous les contextes connectés/abonnés) pour des contextes dont les bindings changent régulièrement.

Pour permettre une telle souplesse, la conception des contextes de nommage du **Federated-NameServer** (tel est son nom) a suivi le pattern stratégie. La politique de fédération est implantée comme une stratégie du contexte et peut donc être changée (même dynamiquement). Ensuite, les contextes sont d'un type particulier (export ou import) et instanciés à l'aide du pattern fabrique. Enfin, les notifications peuvent être réalisées en mode push et/ou en mode pull. L'administrateur a donc toutes les cartes en main pour mettre au point sa politique de fédération.

Avantages

Les avantages de cette approche correspondent dans l'ensemble aux inconvénients de l'approche à gros grain. L'approche à grain très fin apporte une gestion fine de la politique de fédération et donc une souplesse dans la gestion de l'interconnexion des NSs. Un grand nombre de comportements peuvent être utilisés en simultanément.

Le fait qu'un contexte n'est interconnecté qu'avec (au plus) un autre contexte, la recherche dans le cas de politique fénéante ne sera pas une séquence d'invocation à distance. Les recherches à distances sont ciblées sur l'unique endroit où le binding est susceptible d'être trouvé. Cela apporte, en théorie, des performances meilleures que dans l'approche à gros grain.

Inconvénients

Les inconvénients de l'approche à gros grain sont devenus des avantages de l'approche à grain fin, mais malheureusement, les avantages de la première approche ont suivi le chemin inverse. La facilité, voire l'absence d'administration se transforme ici en véritable travail à temps complet. L'administrateur n'est plus un simple *démarrateur de processus*, il doit répondre au besoin d'interconnexion des utilisateurs, tout comme dans un réseau de stations Unix.

Un autre attrait qui disparaît est l'aspect *simple* du NS++ : Je lance le NS, je le branche et l'utilisation est directe. Dans le cas présent le **FederatedNameServer** paraît complexe : différents types de contextes, des opérations supplémentaires sur les interfaces, une interconnexion qui met en œuvre autant de modes d'interconnexion que de liens, . . .

2. Voir section 4.3.1.

4.3.3 Discussion sur les deux approches

Quelquesoit l'approche choisie, un aspect commun est disponible. Bien que la réplication d'informations puisse se faire à la racine, il est possible de nommer le premier niveau de l'arborescence et ainsi préfixer tous les noms de binding par le sous-système qui les contient. Le nommage se fait alors par rapport au NS dans lequel sont créés les bindings. Il faut cependant être conscient que dans l'approche à gros grain, cette approche de nommage n'améliore en rien les performances. Il faudrait pour cela que l'interconnexion ne se fasse pas à partir de la racine, mais à partir du premier niveau de contextes.

Cette dernière proposition est un bon compromis entre les deux approches pour une utilisation généraliste. Cependant, d'une manière générale, chaque approche ayant ses avantages et inconvénients, elles visent des types d'applications différents.

Gros grain

Ce genre d'approche est idéale pour les applications baties autour d'un faible nombre de composants, ou d'applications pour lesquelles la politique de gestion du nommage est unique et uniforme. Cette approche a aussi un sens dans le cadre d'application où les composants peuvent être mobiles. Dans ce cas il n'est pas envisageable de préfixer le nommage des composants par leur localisation géographique. Enfin, en raison des risques de performances faibles, il est préférable que les performances des réseaux soient acceptables.

Grain fin

L'approche à grain fin permet de prendre en compte les applications regroupant un grand nombre de composants. Il sera alors possible de les regrouper, de les classifier (en utilisant une politique de nommage adéquate). Dans ce cas, il est rarement possible, ou souhaitable, de mettre tous les composants à la même enseigne. Une gestion fine est nécessaire pour que le nommage chaque composant soit le plus approprié. Ainsi la solution n'est pas *un compromis globalement bon* mais une réponse globale adéquate qui prend en compte les différents besoins. D'autre part, dans le cas d'applications à très grande échelle (pays, continent, terre) moins on fait d'invocations à grande distance, mieux l'application se porte.

Conclusion

L'interconnexion de services de nommage est un concept qui se met en œuvre de multiples manières. Nous ne revendiquons pas la bonne solution, mais des outils pour mettre une bonne solution pour chaque utilisateur en œuvre. Nous continuons à travailler sur les solutions à utiliser pour l'implantation de services de nommage interconnectés. Un des points qui nous intéresse le plus pour l'instant est la notification des changements : canaux d'événements et utilisation du MulticastIOP [GL99].

La fourniture d'un service de nommage distribué n'est pas une fin en soit, mais un élément essentiel de l'architecture globale. Nous avons toutefois essayé de rendre cette partie la plus indépendante possible par rapport à notre travail. Ainsi elle peut être réutilisable dans d'autres contextes. Il faudra peut-être quand même étendre la proposition actuelle qui ne doit pas avoir pris en compte tous les besoins des différentes utilisations possibles.

Un service de nommage distribué est déjà utilisable (ce n'est plus uniquement un prototype). Il est basé sur la fédération à gros grain et permet de mettre en œuvre le NS généraliste mentionné dans la section 4.3.3. Ce NS est totalement écrit en Java au dessus d'ORBacus.

Chapitre 5

Cache, mobilité et expérimentation

Ce chapitre va présenter la mise en œuvre des aspects décrits dans les chapitres précédents pour faire du cache de composants métiers. Nous aborderons un peu le problème de la mobilité dans le cadre de nos objectifs. Puis nous terminerons par une rapide description de notre collaboration avec une équipe de physiciens, utilisateurs intéressés par l'utilisation du cache de composants.

5.1 Cache d'objets

5.1.1 Objectif des caches

Le principal objectif des caches d'objets est de permettre l'amélioration des performances d'un système. Le principe est de réduire les temps de transfert entre deux entités du système. Ainsi lorsqu'un utilisateur travail sur une donnée, il en fait une copie sur laquelle il a un accès rapide. Une fois ses opérations terminées, la copié remplace la version originale. La manière dont l'originale est remplacée peut être relativement complexe (utilisation de moteur transactionnels, gestion de la cohérence, ...).

5.1.2 Ce qui se fait

Nous passerons dans cette section l'utilisation de cache dans les PCs pour améliorer les performances de la mémoire. La raison est que cette utilisation n'est pas répartie et limitée à un utilisateur : le système d'exploitation.

Le cache d'objets sur le Web

Le cache d'objets sur le web est une des utilisations les plus courantes du caching dans un système réparti. Ce modèle est un modèle simple, il ne diffuse que des documents en lecture seule et il n'y a aucune garantie de cohérence entre une page et ses copies. Lorsqu'un utilisateur récupère une page HTML contenant des images et des fichiers de son il y a plusieurs niveaux de cache mis en œuvre.

Premièrement, il y a le serveur Web original, fournisseur de la page et de son contenu. Lorsqu'un client (browser) demande une page, et les différents fichiers liés, le serveur retourne ces informations. Ensuite, il y a le proxy (souvent géré par le provider internet) qui va recevoir ces informations, et les stocker localement avant de les retourner au client demandeur. Enfin,

il y a le browser qui va, lui aussi, faire une copie locale des données dans un cache et afficher la page et son contenu.

Si le même client demande à nouveau la page, le browser ne va pas interroger le serveur, il va prendre les informations dans son cache local pour les réafficher. Les données sont donc accessibles facilement et rapidement. Si un autre client du provider internet demande cette même page, le proxy va retourner ces informations sans les demander au serveur. Le temps de réponse sera donc plus long qu'en local, mais moins long que si le serveur avait été interrogé.

Dans l'ensemble il faut remarquer que les objets du Web sont des objets simples (pages html, images, sons, ...). Cependant avec la généralisation de XML les objets du Web peuvent devenir plus conséquents. D'une part leur complexité s'accroît, et d'autre part les possibilités offertes par ces objets augmentent. Un objet XML peut servir à exprimer des notions plus complexes que du texte (limite pour les pages Web). Des travaux ont été réalisés pour exploiter les caches dans le monde du World Wide Web. Le but est de mettre en place des caches de manière hiérarchique pour améliorer des temps de réponse du Web et réduire la consommation de bande-passante (voir [Wes95]).

Le cache d'objets dans les OS répartis

Dans le cadre des systèmes d'exploitation répartis, les caches sont mis en œuvre pour améliorer les performances d'accès à un système de fichier global réparti. D'une manière générale, le caching des données du filesystem se font sous la forme de page mémoire. Lorsqu'un applicatif accède à une donnée, cette donnée est rapatrié en local avec les données adjacentes. La quantité d'information rapatriée correspond à la quantité d'information stockable dans une page mémoire. C'est pour cette raison que les données sont souvent organisées. Ainsi un maximum de données nécessaires à l'applicatif sont rapatriés en une seule opération.

Dans le cadre du POSM[Mar98] (Persistent Object Store Manager) du système d'exploitation Arena[MB97, MQBN94], la mise en œuvre des caches a été un peu différente. L'approche a été de faire du cache par objet et non par page. Lorsqu'un applicatif requiert un objet, cet objet est copié à partir de la version du médium stable vers une version volatile en mémoire. Une fois une copie d'objet chargée en mémoire, la cohérence n'est pas gérée par le manager de persistance, mais par l'applicatif. Seule l'atomicité de la phase de commit (retour de l'état volatile à l'état persistant d'un objet) était garantie par le système.

Le POSM permet à différents clients d'accéder à un dépôt d'objets persistants distribué. Chaque client de ce dépôt dispose d'un cache qui lui permet d'utiliser les objets. L'utilisation de cache n'a pas seulement pour but d'augmenter les performances du système. En effet, le dépôt d'objets est implanté sur un disque, il faut donc un moyen de charger un objet en mémoire pour pouvoir l'utiliser. L'utilisation directement sur le disque semble d'une autre époque... D'autre part, les clients pouvant être multiples à un instant donné, et le support stable distribué, un accès centralisé –partage d'un espace mémoire pour le chargement d'objets à destination de tous les clients– ne serait pas adapté.

5.1.3 L'utilisation de GCA pour le cache

SAC comme gestionnaire de cache

Dans le cadre des applications à grande échelle, l'utilisation de services distant peut avoir un coût non négligeable (voire très gênant) lors d'invocations d'opérations. Ce problème est adressé par l'utilisation de caches comme nous venons de le mentionner dans la section précédente. Nous avons donc cherché à mettre en œuvre des mécanismes de cache pour améliorer l'utilisation des composants dans le cadre des applications réparties à grande échelle.

Les performances de telles applications se dégradent très vite lors d'invocations multiples et répétées sur des services lointains. La définition de lointain repose sur le temps de réponse lors d'une invocation d'opération. La distance est donc relative à la performance des réseaux de communication. Dans cette situation, il serait très intéressant de factoriser les invocations d'opérations. L'applicatif envoie un groupe d'invocations et reçoit un groupe de réponses. Cependant, cette approche nécessite la prise en compte du regroupement d'invocations lors de la conception. Une application existante ne peut être facilement modifiée pour prendre en compte ces facilités.

Dans le cas d'applications existantes, une seconde approche est possible. Cette approche met en œuvre l'utilisation de caches. En effet, si de multiples invocations distantes coûtent cher, il peut être intéressant de rapprocher le composant, puis de faire les invocations de manière plus locale. Le choix dépend du coût de déplacement du composant. Comme nous n'avons pas encore défini de critères qui permettent de faire un choix automatiquement, le rapprochement de composant est laissé à la charge de l'utilisateur. Ce dernier dispose de la possibilité de copier localement un composant distant. Pour cela, la SAC offre les mécanismes de clonage¹.

Cohérence...

L'utilisation de cache soulève toujours la question de la cohérence entre les éléments originaux et les éléments *cachés*. Nous n'avons pas, dans le cadre de ce travail, pris en compte ces éléments pour deux raisons. D'une part, notre approche avait pour but d'expérimenter des mécanismes de clonage. D'autre part, les composants que nous souhaitions adresser sont des composants qui ne nécessitent pas (ou peu) de cohérence entre version du cache et version originale.

Les composants métiers sont principalement des composants de traitement. Leur état reflète une configuration qui a pour but de fixer leur comportement. Cette configuration n'est pas constamment changée par l'utilisateur et elle peut être différente pour deux clients d'un même composant. Donc le problème de cohérence était secondaire. Le deuxième type de composants que nous avons cherché à adresser sont les composants *de physiciens*. Ces composants reflètent le résultat de mesures physiques et permettent de faire de l'expérimentation. Il faut donc que les modifications apportées par un client ne soient pas reportées sur le composant original (le composant résultat de la mesure physique).

Cette approche nous a permis d'aborder le problème de manière simplifiée. Cependant il existe des situations où il faudra de la cohérence, et il va donc falloir maintenant penser à des

1. Voir section 3.4.2

mécanismes pour la prendre en compte.

5.2 Mobilité de composants

5.2.1 Un existant

Dans [Pet98], Y. PETER propose une mise en œuvre de la migration d'objets à grain gros ou moyen dans un environnement standard et homogène. Dans ce cadre, les objets pris en compte ne sont pas fortement liés à des ressources locales, contrainte qui limiterait les possibilités de migration. L'objectif de cette proposition est d'offrir des facilités de régulation de charge pour les applications orientées objet et distribuées.

Les facilités de migration mises en œuvre offrent la transparence à l'utilisateur. Ce dernier ne sait pas nécessairement lorsqu'un objet est migré. Le système lui garantit une continuité d'utilisation. Cependant, la migration n'est pas une opération réalisée de manière automatique par l'objet, ni par le système. Une migration est toujours initiée par un utilisateur (ou par un administrateur). Une migration d'objet peut se dérouler avec ou sans migration d'activité. Lors de la migration d'un objet, un traitement peut être interrompu, puis redémarré sur le site distant (migration d'activité).

La mise en œuvre de cette architecture repose sur les patterns *Factory* et *Finder*. Les *finders* définissent des domaines du système global. Ils permettent de retrouver les fabriques d'un domaine précis. Une fabrique représente un potentiel site destinataire lors d'une migration. Les migrations reposent sur l'instanciation d'objets à distance, instanciations rendues possibles grâce aux fabriques. Lors de la migration d'un objet, le site d'origine est *lié* au nouvel emplacement de l'objet par une référence de *forward*. Ainsi, lors d'une migration, l'objet est toujours disponible pour ses différents utilisateurs (l'éventuel initiateur et les autres). Les performances peuvent cependant se dégrader à mesure que le nombre d'indirections augmente (chaîne de références de *forward*).

5.2.2 La mobilité dans le contexte de GCA

Il n'y a actuellement pas d'expérimentation sur la mobilité des composants, seulement des idées. La mobilité peut être vue de manière simple comme une autre utilisation des mécanismes mis en œuvre pour faire le clonage de composants. La principale différence entre le clonage (comme nous l'avons abordé) et la migration est le fait que l'objet original n'existe plus à la fin de l'opération. La mobilité peut donc se résumer comme suit. La migration correspond à la séquence : clonage d'une instance de composant vers une SAC cible, puis destruction de cette instance originale.

Un travail supplémentaire est cependant nécessaire. La gestion du nommage est quelque peu différente dans le cadre de la migration : il n'y a pas création d'un nouveau identifiant pour l'instance cible. Le nom est identique à celui de l'instance originale. D'autre part, si l'instance de composant est destinée à un seul utilisateur, ces opérations sont suffisantes. Mais, si l'instance de composant est destinée à plusieurs utilisateurs, il faut alors mettre en place des mécanismes supplémentaires.

Dans le cas où une instance de composant est utilisée par plusieurs clients de manière simultanée, un déplacement de cette instance par un client peut engendrer des problèmes pour

d'autres clients. Il est donc nécessaire de mettre en place des mécanismes qui permettront à tous les clients de conserver un accès sur l'instance de composant. Un mécanisme possible est l'utilisation d'objets qui se contentent de ré-émettre les requêtes vers la nouvelle localisation du composant. Un autre mécanisme est de lever une exception (CORBA s'en charge) et d'imposer au client de refaire une recherche dans le service de nommage.

Il faut toutefois se poser la question : pourquoi avoir plusieurs clients pour une même instance mobile ? Il est si facile de faire des clones d'instances pour chaque client désireux d'utiliser l'instance de composant en question. La mobilité d'une instance partagée semble avoir peu de sens : à chaque fois que je favorise un client, je défavorise plusieurs autres.

5.3 Expérimentation

5.3.1 Prototype

Dans un premier temps, les objectifs du prototype ont été simplifiés. Les composants que nous avons pris en compte sont des composants *monoblocs*. Ce sont des composants dont l'implantation représente une seule entité. Les composants qui sont implantés sous la forme de plusieurs objets doivent avoir un état externalisable à partir de l'interface de base, et ils doivent être capable de reconstruire leur état composite.

Dans l'état actuel des choses, le prototype nous a permis de constater que les mécanismes pour le clonage étaient réalisables. L'objectif était surtout de se faire la main. Des éléments seront réutilisables, d'autres moins. Nous savons, désormais, que notre approche dispose d'avantages –expression de l'état dans des méta-scripts, sandbox composite, généricité de la structure d'accueil, . . . –, mais qu'il y a des points qui ne vont pas –Pas encore de modèle de composant mobile, pas assez d'automatisation (clonage manuel), interconnexion de SAC manuelle, . . .

Nous avons un prototype qui démontre la faisabilité de certains éléments. Il nous reste maintenant à le finaliser, le rendre utilisable de manière globale. Puis par la suite de le compléter (ou modifier) pour prendre en compte les aspects qui ont été mis de côté.

5.3.2 Le projet des physiciens

Présentation

Dans le cadre d'une collaboration entre le LAL (Laboratoire de l'Accélérateur Linéaire, Université d'Orsay - Paris IX), SLAC (Stanford Linear Accelerator Center, Etats-Unis) et le CERN (Centre Européen de Recherche Nucléaire, Genève) des physiciens souhaitent et peuvent partager le résultat d'expériences de physique. Le but est d'exploiter ces mesures pour faire de la simulation, et ainsi refaire d'autres mesures avec de nouveaux paramètres. Le but de la simulation est, en partie, de compenser l'impossibilité de faire beaucoup d'expérimentation (chères et complexes).

Leur mode de fonctionnement est simple. Le SLAC diffuse des résultats d'expériences réalisées avec un nouveau type de détecteur. Ces résultats sont représentés sous formes de graphes d'objets : les traces. Lorsque des particules frappent le détecteur, cela génère un hit. Une trace est un ensemble de hits relatifs à une même expérience. Les physiciens de Stanford, Paris et Genève peuvent ensuite travailler sur les traces pour reconstruire le chemin des particules

dans le détecteur. Ils peuvent aussi modifier certains paramètres et faire de la simulation pour voir comment la trace évolue. Toutes ces modifications ne doivent pas être reportées sur la trace originale, mais la nouvelle trace peut éventuellement être ajoutée dans un référentiel de traces.

La collaboration

Le mode de fonctionnement de cette communauté est en phase avec les travaux que nous avons mené sur les composants et leur clonage. Des serveurs offrent des composants (les traces) que les utilisateurs peuvent recopier localement pour travailler. Les composants originaux ne sont pas modifiés, mais les nouveaux composants peuvent être partagés dans le système. Les structures d'accueil que nous avons proposé se prêtent parfaitement à ce problème. Nous avons donc eu des échanges avec le LAL pour chercher à fournir des SAC pour leur projet, et avoir ainsi de notre côté une expérimentation grandeur nature.

La collaboration était idéal sur le papier, malheureusement la réalisation n'a pas pu aboutir. Le premier problème est que l'implantation du projet est en C++. Le point noir est ici le chargement dynamique de code que nous exploitons pour permettre la mobilité de code. Ce chargement dynamique n'est possible en C++ qu'au prix de très gros efforts et la généricité n'existe quasiment plus.

L'autre problème est l'architecture actuelle de leur application. L'application des physiciens est monolithique. Lorsqu'un utilisateur souhaite travailler sur des données, un clone de l'application complète est réalisé sur une machine serveur où tous les utilisateurs travaillent. L'application n'est donc pas décomposable pour en déplacer certaines parties. Il devient donc inutile de déplacer les données si l'application est toujours à l'autre bout du monde.

Nous sommes cependant toujours en contact. Le LAL essaye de changer progressivement leur architecture pour pouvoir la décomposer. De notre côté, nous réfléchissons un peu à des moyens pour palier les défauts intrinsèques au C++. Pour l'instant des outils basés sur CorbaScript permettent de gérer des sessions pour les utilisateurs. D'autre part, le principe d'extraction de l'état des composants sert de base pour faire de l'échange de traces en sites serveurs. Plutôt que de rapatrier les hits un par un, la trace est représentée dans une structure IDL qui sera échangée entre les deux sites. La trace étant un graphe, la gestion des boucles est un détail sur lequel travaille un des informaticiens d'Orsay.

5.3.3 Proposition des implantations de composants en C++

L'utilisation d'implantations de composants en C++ pose des problèmes pour la mobilité. Le chargement dynamique de code en C++ est possible au travers de bibliothèques dynamiques. Cependant, exploiter ces bibliothèques n'est pas une opération simple. La première remarque est qu'il n'y a pas de standard pour le nommage des classes au sein de la bibliothèque : les noms de classe sont préfixés, et les préfixes variables. Le second point est la quasi impossibilité de faire du chargement générique. Même si une bibliothèque ne contient qu'une seule implantation de composant et qu'elle dispose d'une méthode d'instanciation, il ne sera très difficile de faire un cast sur l'instance de composant créée, donc de l'utiliser. La conception d'un chargeur de composants C++ générique est donc totalement impossible.

Il reste une alternative pour permettre l'utilisation de composants implantés en C++.

Lorsque l'utilisation d'une implantation C++ de composant est courante, il est envisageable de rendre disponible cette implantation ainsi qu'un moyen de l'utiliser (moyen dépendant de l'implantation). Pour cela l'utilisation de la pattern Fabrique est une bonne solution. Lorsqu'un développeur fournit un composant en C++, il fournit avec une fabrique pour ce composant.

Pour permettre cette utilisation, plusieurs choses doivent être prises en compte. Premièrement, il est souhaitable que nous fournissions un template C++ pour concevoir les fabriques. Le développeur écrit le code relatif à l'instanciation, et nous sommes sûr que la fabrique aura la bonne API. Deuxièmement, il faut légèrement modifier la SAC. Il sera dans le cas de composant en C++ démarrer un nouveau processus et non émettre une requête de chargement vers un processus existant. Enfin, cloner un composant écrit en C++ sera possible, mais le cloner vers un autre composant C++ sera coûteux (transfert des binaires et création de processus en plus de l'instanciation de composant, ...).

5.4 Synthèse

Nous venons de parler d'expérimentations qui ont déjà eu lieu, et de quelques expérimentations que nous souhaitons faire dans l'avenir. Cette partie du travail n'a pas été un brouillon, mais les premiers pas dans le monde des composants, du cache et de la mobilité. Ces expérimentations ont levé plus de questions qu'offert de réponses.

Quelques points à traiter dans l'avenir sont :

- Le choix de quand faire du clonage automatique pour avoir des caches *intelligents*.
- Réellement entamer la réflexion sur la mobilité : quand, pourquoi, de quelle manière ?
- Définir précisément notre modèle de composant, premièrement en reprenant le chapitre 2 avec d'autres modèles de composants.
- Travailler sur le niveau méta de l'état d'un composant.
- Prendre réellement en compte l'aspect persistance.
- Etc.

Chapitre 6

Conclusion et poursuite

Le but de ce travail était au départ de réfléchir à la définition de structures d'accueil pour le cache et la mobilité de composant métiers. Nous ne disposons pas aujourd'hui d'une structure d'accueil prête à l'emploi. Cependant, nous avons déjà une première définition des mécanismes nécessaires à sa mise en œuvre.

6.1 Conclusion

En plus de réfléchir à la définition des structures d'accueil, ce travail a nécessité une réflexion sur la définition d'un modèle de composants métiers. Ce modèle servant à définir les composants qui peuvent être accueillis par les structures d'accueil. Nous avons, à l'heure actuelle, uniquement fait les premiers pas de cette réflexion. Des besoins se sont détachés quant à certains points important pour un modèle de composants mobiles. Le besoin d'un niveau méta pour décrire au moins l'état d'un composant est un point que je ne m'attendais pas à aborder.

Les travaux relatés dans ce document représentent pour moi une première acquisition d'expertise sur différents modèles de composants. Cette expertise m'a permis de me familiariser avec l'utilisation actuelle des composants pour bâtir des applications distribuées. Cela a permis aussi d'entrevoir les besoins supplémentaires intrinsèques à la mobilité des composants, les modèles actuels étant relativement statiques.

Ces *défauts* des modèles statiques par rapport à nos besoins définissent des directions de recherche pour nos travaux futurs. Il est nécessaire dans un premier temps de voir comment palier ces défauts (puisqu'ils sont identifiés) puis, dans un second temps, de réfléchir aux éventuels problèmes non encore identifiés, mais induits par la mobilité.

6.2 Poursuite

Dans l'état actuelle des choses, l'opération de clonage d'une instance de composant est explicitement invoquée par l'utilisateur. Nous souhaitons maintenant travailler sur la transparence d'utilisation du clonage, il en est de même pour la mobilité et le nommage. Notre objectif n'est pas nécessairement de mettre de *l'intelligence* dans le système, mais de rendre le système le plus souple possible. Le système (au sens global) doit inclure un maximum d'automatismes pour rendre son utilisation la plus simple possible à l'utilisateur.

Pour rendre le système souple à souhait, il est aussi important de continuer notre réflexion sur les composants. Nous devons les rendre le plus autonomes, et le plus génériques possibles.

La généralité recherchée est la faculté de couvrir le plus grand nombre possible d'applications réalisables avec notre modèle de composant. D'autre part, nous avons abordé le niveau méta pour extraire l'état d'un composant. Cette réflexion doit être poursuivie vers une généralisation du niveau méta : un méta-modèle de composants prenant en compte la mobilité.

Cette poursuite de travail se dessine au sein du projet CESURE¹. Ce projet vise à fournir un ensemble de services pour les utilisateurs nomades, principalement la fourniture de leur environnement de travail favori quel que soit le site de connexion. Cela implique, en plus des besoins déjà énoncés, la prise en compte de la configuration des composants. La projet nécessitera la définition de moyens pour permettre la configuration, mais surtout la reconfiguration des composants lors de leurs déplacements. Cette configuration devra se faire en fonction de leur environnement, des services à leur disposition, du contexte, . . . Autant de points que nous chercherons à définir et à décrire au niveau du méta-modèle de composants.

En plus de me garantir une poursuite intéressante tant le sujet est passionnant, les travaux actuels et ceux qui se profilent à l'aube des trois prochaines années me confortent dans le choix de faire de la recherche universitaire. Faire de la recherche universitaire, mais tout en conservant la volonté de rendre possible le transfert dans l'industrie des résultats de nos travaux.

1. Configuration et Execution de Services pour les Usagers mobiles des Réseaux Etendus

Annexe A

structure d'accueil de composants

```
module GOODEComponentArchitecture {

    typedef string Identifier;

    typedef sequence<Identifier> IdentifierSeq;

    typedef sequence<octet> ImplCode;

    struct ComponentAuthor {
        string name;      // GOODE
        string company;  // LIFL
        string webpage;  // http://corbaweb.lifl.fr
    };

    enum ComponentCodeType {
        CLASS,
        JAR,
        CS,
        SO,
        DLL
    };

    struct ComponentCode {
        ComponentCodeType type; // CLASS, CS, SO, DLL...
        string filename;       // toto.class, toto.cs, toto.so, toto.dll...
        string location;       // directory (/export/corba/components) or
                               // URL (http://corbaweb.lifl.fr/components)
    };

    struct ComponentInterface {
        string filename; // toto.idl
        string location; // directory (/export/corba/components) or
                          // URL (http://corbaweb.lifl.fr/components)
    };
};
```

```
struct Implementation {
    string version;           // version of the implementation
    string os;               // not useful for Java and CorbaScript
    string processor;       // not useful for Java and CorbaScript
    string language;       // Java, C++, CorbaScript...
    string compiler;       // jdk1.1.6, cssh1.2, egcs1.0.1
    ComponentCode code;     // various information about the code
};

struct ComponentState {    // allow to externalize a component state
    string getScript;      // script to extract the state of a component
    string setScript;      // script to fix the state of a component
};

typedef sequence<Implementation> ImplementationSeq;

struct ComponentDescription {
    string name;           // GOODEbank -- interface name
    string type;          // Corba component
    ComponentAuthor author; // GOODE, LIFL, corbaweb.lifl.fr
    string abstract;      // bank component example.
    string license;       // file that describes the licence
    string idlId;         // "IDL:GOODE/Bank:1.0"
    string source;        // component's site of creation
    ComponentInterface idl; // IDL file (mainly useful for cssh)
    ImplementationSeq impl; // infos on the implementations
    ComponentState state; // state of the component
};

};
```

Annexe B

Service de Nommage étendu

```
module CosNaming {  
    exception NotConnected { };  
    exception AlreadyConnected { };  
    interface RetNamingContext : NamingContext {  
        string connect ( in Object ref ) raises (AlreadyConnected);  
        void disconnect ( in string id ) raises (NotConnected);  
    };  
};
```

Annexe C

Service de Nommage fédéré

```
module FederatedCosNaming {

    //
    // Some module definitions

    typedef sequence<CosNaming::Name> NameSeq;

    struct Bind {
        CosNaming::Name  name;
        Object            ref;
    };

    typedef sequence<Bind> BindSeq;

    exception IncompatibleContext {
        string expected_type;
    };

    interface FederatedNamingContext;

    //
    // Strategies

    interface Strategy {
        void setContext ( in FederatedNamingContext ctxt );
    };

    interface PullStrategy : Strategy {
        Object resolve ( in CosNaming::Name n )
            raises (CosNaming::NamingContext::NotFound,
                  CosNaming::NamingContext::CannotProceed,
                  CosNaming::NamingContext::InvalidName);
    };
};
```

```
interface PushStrategy : Strategy {
    void bind ( in CosNaming::Name n, in Object obj )
        raises (CosNaming::NamingContext::NotFound,
                CosNaming::NamingContext::CannotProceed,
                CosNaming::NamingContext::InvalidName,
                CosNaming::NamingContext::AlreadyBound);
    void rebind ( in CosNaming::Name n, in Object obj )
        raises (CosNaming::NamingContext::NotFound,
                CosNaming::NamingContext::CannotProceed,
                CosNaming::NamingContext::InvalidName);
    void unbind ( in CosNaming::Name n )
        raises (CosNaming::NamingContext::NotFound,
                CosNaming::NamingContext::CannotProceed,
                CosNaming::NamingContext::InvalidName);
};

/////
//
// FederatedNamingContext and subclasses
//
// These interfaces describe the contexts that can be federated

interface FederatedNamingContext : CosNaming::NamingContext {
};

interface NamingContextCache : FederatedNamingContext {
    void connectToRemoteContext ( in FederatedNamingContext rnc )
        raises (IncompatibleContext);
};

interface NamingContextHost : FederatedNamingContext {
    BindSeq resolve_all ( );
};

enum FNCKind {
    PULL,
    PUSH
};

interface PullNamingContext : FederatedNamingContext {
    void setStrategy ( in Strategy s );

    FNCKind kind ( );
};
```

```
interface PushNamingContext : FederatedNamingContext {
    void setStrategy ( in Strategy s );

    FNCKind kind ( );
};

interface PullNamingContextCache : PullNamingContext, NamingContextCache {
};

interface PullNamingContextHost : PullNamingContext, NamingContextHost {
};

interface PushNamingContextHost : PushNamingContext, NamingContextHost {
};

interface PushNamingContextCache : PushNamingContext, NamingContextCache {
};

/////
//
// AdminNamingContext and subclasses
//
// These interfaces are used by the administrator to manage new context
// and to federate them.

interface AdminNamingContext : CosNaming::NamingContext {
};

interface ImportNamingContext : AdminNamingContext {
    PullNamingContextCache
    bind_new_pull_context ( in CosNaming::Name name,
                          in PullStrategy strategy );

    PushNamingContextCache
    bind_new_push_context ( in CosNaming::Name name );
};

interface ExportNamingContext : AdminNamingContext {
    PullNamingContextHost
    bind_new_pull_context ( in CosNaming::Name name );

    PushNamingContextHost
};
```

```
    bind_new_push_context ( in CosNaming::Name name,
                           in PushStrategy strategy );

    void
    setExportPolicy ( in CosNaming::Name name,
                    in NameSeq contexts );
};
};
```

Bibliographie

- [BSCC⁺99a] Inc. BEA Systems, CRC DST, Expertsoft Corporation, Genesis Development Corporation, IBM Corporation Inprise Corporation, PLC IONA Technologies, Oracle Corporation, Inc. Rogue Wave Software, and Unisys Corporation. Corba components - joint revised submission. OMG-tcdocument orbos/99-02-05, Object Management Group, March 1 1999.
- [BSCC⁺99b] Inc. BEA Systems, CRC DST, Expertsoft Corporation, Genesis Development Corporation, IBM Corporation Inprise Corporation, PLC IONA Technologies, Oracle Corporation, Inc. Rogue Wave Software, and Unisys Corporation. Corba components - joint revised submission - errata. OMG-tcdocument orbos/99-02-09, Object Management Group, March 21 1999.
- [CS96] Olivier Ciupke and Rainer Schmidt. Component as context-independent units of software. In Max Mühlhäuser, editor, *Special issues in Object-Oriented Programming*, pages 139–143. Dpunkt, July 1996. WCOP'96 (Workshop on Component Oriented Programming).
- [GGM95] Jean-Marc Geib, Christophe Gransart, and Philippe Merle. Corbascript and corbaweb: A generic objetc oriented dynamic environnement upon corba. In *Proc. of TOOLS Europe 1996*, Paris, juin 1995.
- [GGM97] Jean-Marc Geib, Christophe Gransart, and Philippe Merle. *Corba: Des concepts à la pratique*. InterEditions, 1997.
- [GL99] C. Gransart and M. Laukien. Mutlicast iop. In *Proceedings...*, August 1999. Not yet published.
- [Ham97] Graham Hamilton. *JavaBeans*. Sun Microsystems, July 1997. Version 1.01.
- [LIF99] LIFL. Omg corba scripting language rfp initial submission. OMG-tcdocument orbos/99-01-25, Laboratoire d'Informatique Fondamentale de Lille, janvier 1999.
- [Mar98] Raphaël Marvie. A distributed persistent object store manager for the arena system. Master's thesis, University of Lille 1 & University of Manchester, June 1998.
- [Mar99] Raphaël Marvie. Corba components: la proposition unifiée ("du modèle d'objets au modèle de composants"). Technical report, Laboratoire d'Informatique Fondamentale de Lille, mai 1999.
- [MB97] Ken R. Mayes and J. Bridgland. Arena - a run-time operating system for parallel applications. In *Proceeding of the fifth Euromicro Workshop on Parallel and Distributed Processing*, pages 253–258, London, January 1997.
- [Mer97] Philippe Merle. *CorbaScript - CorbaWeb: Propositions pour l'accès à des objets et services distribués*. Thèse de doctorat en informatique, Laboratoire d'Informatique Fondamentale de Lille, Villeneuve d'ascq-France, janvier 1997.

- [Mey90] Bertrand Meyer. *Conception et programmation par objets*. InterEditions, Paris, 1990. ISBN 2-7296-0272-0.
- [MH99] Vlada Matena and Mark Hapner. *Enterprise JavaBeans Specification*. Sun Microsystems, May 1999. Version 1.1, public draft.
- [MQBN94] K. Mayes, S. Quick, J. Bridgland, and A. Nisbet. Language- and application-oriented resource management for parallel architecture. In *Proceeding of the sixth ACM SIGPLAN European Workshop*, pages 172–177, September 1994.
- [Pet98] Yvan Peter. *Gestion de la mobilité dans les environnements de communication à objets*. PhD thesis, UFR des Sciences et Techniques de l'Université de Franche-Comté, novembre 1998.
- [Wes95] Duane Wessels. Intelligent caching for world-wide web objects, May 1995. <http://www.isoc.org/HMP/PAPER/139/html/paper.html>.