

DEA Réseaux et Systèmes Distribués
Université de Nice Sophia - Antipolis

Décrire une application à base de composants sur middleware

François CREVOLA

Encadrement:

Michel Riveill

Juin 2001.

Remerciements

Merci à Michel Riveill, mon encadreur, pour ses conseils et son aide.

Tous mes remerciements vont également à l'ensemble de l'équipe du projet Rainbow (ses enseignants chercheurs, ses thésards et ses stagiaires).

Table des matières

1. INTRODUCTION.....	4
CONTEXTE, DÉFINITIONS	4
<i>Middleware</i>	4
<i>Composants</i>	4
<i>ADLs</i>	5
OBJECTIFS	6
APPORTS	7
DÉMARCHE	8
2. ADLS.....	9
SCÉNARIO	9
OLAN ET LES CONTRAINTES DE PLACEMENT	10
DARWIN ET L'INSTANCIATION DYNAMIQUE.....	11
3. ADLS ET MIDDLEWARE.....	13
INTRODUCTION	13
ASTER.....	13
SOFA	15
ABSTRACT ADL	17
<i>Composants</i>	17
<i>Connecteurs</i>	18
<i>Configurations</i>	18
4. PROPOSITION.....	20
ADL	20
<i>Composants</i>	20
<i>Connecteurs</i>	23
<i>Configuration</i>	24
OUTILS ET MÉCANISMES	24
<i>Architecture</i>	25
<i>SMNP</i>	25
<i>Notre système</i>	26
5. CONCLUSION.....	29
OBJECTIF ET DÉMARCHE DE TRAVAIL	29
ÉVALUATION	29
PERSPECTIVES.....	29
BIBLIOGRAPHIE.....	31

Chapitre 1.

Introduction

Le présent rapport présente le travail que j'ai effectué lors de mon stage de DEA RSD effectué dans le cadre du projet Rainbow au sein du Laboratoire I3S / CNRS - UNSA.

Contexte, Définitions

Middleware

Un middleware (parfois appelé en français intergiciel), est une couche logicielle qui, comme son nom l'indique, se situe entre le système d'exploitation et les applications. Comme le fait remarquer P. Bernstein dans [Bern96], les objectifs poursuivis par les middleware ont considérablement évolués et ce qui est dans un middleware aujourd'hui sera peut-être, à l'avenir, partie intégrante du système d'exploitation et réciproquement. Dans tous les cas, l'objectif d'un middleware est de fournir des services n'existant pas à l'origine dans le système et d'être une couche d'abstraction pour la programmation d'applications.

L'utilisation des services tant à être de plus en plus automatisée, ceci afin que le développeur d'applications puisse se concentrer sur le code métier plutôt que sur des tâches non-spécifiques à une application. Le premier service fourni est généralement, un système de communication (dit aussi "bus logiciel") entre les applications utilisant le middleware. C'est parfois le seul service fourni, comme dans les RPC¹. On peut également trouver, selon les middleware, un service de gestion des transactions, un service de persistance ou encore un service de sécurité. Il ne s'agira pas ici de comparer les middlewares existant ou de lister les services offerts ou non par tel ou tel middleware.

En tant que couche d'abstraction pour la programmation d'applications, un middleware présente une vision uniforme au développeur d'applications quelque soit le système sous-jacent. Il constitue alors un modèle avec ses contraintes et ses avantages. Ainsi, le modèle EJB² défini par Sun, est un middleware dans lequel les applications sont écrites en utilisant un sous-ensemble bien défini des possibilités du langage Java.

Composants

Alors que l'attention était récemment portée sur les objets, on entend aujourd'hui parler de plus en plus de composants. Par exemple, la plate-forme CORBA, généralement considérée comme étant un middleware permettant la communication entre objets distants, propose depuis peu un modèle de composants appelé CCM³.

¹ Remote Procedure Call

² Enterprise Java Beans

³ Corba Component Model

Un objet peut être défini par ce qu'il fait plutôt que par comment il le fait (et les données encapsulés font parties de "comment il le fait"). En ce sens on peut considérer, les composants comme étant une forme plus aboutie d'objets, comme étant une étape supplémentaire dans les outils nécessaires à faire correctement du "*programming in the large*" (voir [Kron76]), c'est à dire de la programmation par assemblage de modules logiciels.

Dans le présent document on s'intéresse aux modèles contemporains de composants sur middleware comme les EJBs (Enterprise Java Beans) de la plate-forme J2EE (Java 2 Enterprise Edition) et les CCM de la plate-forme Corba. Tous deux, sont représentatifs d'une certaines idées des composants et ont suffisamment de caractéristiques communes pour que les propositions faites pour l'un puissent ensuite être appliquées à l'autre ou à de futurs modèles de composants aux caractéristiques similaires, avec bien sûr quelques adaptations. Nous ne nous lancerons pas ici dans une étude détaillée des différences entre CCM et EJB, pour cela voir par exemple [Hub99] qui s'interroge sur les similarités et différences entre CCM et EJB ("*we suspect that EJB and CORC will remain similar and compatible in some ways but will not maintain a clear superset/subset relationship*") ou [O'reil99].

Voyons brièvement certaines caractéristiques des composants évoqués.

* interface: le composant est une entités logicielles qui exhibent ce qu'elle sait faire par le biais de son interface. Aussi bien dans le modèle CCM, que dans le modèle EJB, Il n'existe pas une unique interface mais plusieurs. Chacune ayant une signification particulière. Dans le modèle EJB, chaque composant possède deux interfaces une interface dite "Home" qui à pour fonction de permettre la création, la destruction et la recherche d'instances du composant (dans le cas particulier des ... où les instances survivent aux clients) et une interface dite "Remote" qui permet d'accéder aux fonctions exposées à l'extérieur par le composant. Dans le modèle CCM, chaque composant possède une interface classique équivalent à l'interface associé aux objets Corba traditionnel et la possibilité d'avoir d'autres interfaces chacune ayant une fonction définie par le développeur du composants, il s'agit de regrouper sous une interface des fonctions utiles à un type particulier de client (par exemple administrateur).

* propriétés non-fonctionnelles: Aussi bien dans le modèle EJB, que dans le modèle CCM, il est possible d'associer aux composants des propriétés non-fonctionnelles. Par exemple, il est possible de dire quelles fonctions nécessitent d'être partie intégrante d'une transaction ou encore de définir les données d'un composant qui seront persistantes. Ces propriétés sont décrites dans des fichiers annexes qui complètent les fichiers contenant le code source (ou sa forme binaire). Il s'agit d'une tendance de plus en plus fréquente dans les middleware contemporains.

ADLs

Dans un ADL¹, les composants sont des entités logicielles, qui en prenant part à une architecture plus large, permettent la construction d'une application. Par le mécanisme de la description d'architectures à l'aide de langages spécifiques, les ADLs, on vise à réutiliser ces entités logicielles. Pour cela, l'entité logicielle (ou composant) est l'élément indispensable de tous les ADLs (unité souvent décrite par le mot-clef "component"), mais chaque ADL exprime des contraintes particulières sur ce qu'est un composant. Par exemple, on ne pourra décrire que des applications où les composants sont des exécutable Unix ou dans un autre système où les composants sont des classes héritant d'une classe ou d'une interface imposée. Pour N. Medvidovic dans [Med97], un composant est une unité de calcul ou de stockage, si bien qu'il ne faut pas penser un composant comme étant quelque chose de petit, de simple ou de trivial. C'est pourquoi nous précisons ici à quel type de composants on s'intéresse et quels en sont les principales caractéristiques.

Par réutilisation de composant, on entend généralement, réutilisation d'un module / composant logiciel non-installé. Ce composant est intégré à l'application. éventuellement après quelques « travaux » comme l'écriture d'un *wrapper*. Ensuite, il est installé comme les autres composants de l'application.

Objectifs

Ayant défini le contexte dans lequel nous nous plaçons et précisé quelques notions sur les composants, les middlewares, et les ADLs, nous pouvons maintenant préciser la problématique et les objectifs que nous nous fixons. Nous désirons décrire une application à base de composants sur middleware (ABCM) pour faciliter un certains nombres de tâches liés au déploiement.

Comme le constate P. Bernstein dans [Bern96], les middlewares sont un modèle pour la réalisation de systèmes distribués. Or la répartition, bien que possible, n'est que rarement la première caractéristique mise en avant. Par exemple, [EJB2] des exemples mettant en œuvre plusieurs hôtes sont bien évoqués mais peu de détails sont donnés sur leur réalisation pratique. Si bien que il faudra probablement être un peu expérimenté afin de réaliser une application réellement distribuée. On veut faciliter l'écriture de la répartition à partir des notions locales toujours évoquées. Donc, il est temps de se doter d'outils réellement capable d'aider l'utilisateur à exploiter les possibilités de répartition des éléments constituant son application.

Deuxièmement, on constate le problème de l'hétérogénéité des serveurs. Le rôle de la norme est de fixer le cadre théorique permettant une interopérabilité entre les implémentations qui la respecte. En pratique, l'interopérabilité quand elle st possible, ne signifie pas pour autant que tous les serveurs répondent sur les mêmes ports ou usent des mêmes conventions de nom. Nous proposons que l'outil règle le problème d'interrogation d'un serveur distant quelque soit son implémentation et aussi quelque soit celui sur lequel fonctionne le client. Nous détaillerons plus loin comment faire pour y parvenir.

¹ Architecture Description Language

Troisièmement, on veut pouvoir intégrer facilement un composant nouveau à des éléments existants et installés pour faire une nouvelle application. Il ne faut pas toucher aux applications déjà installées. Dans la suite, nous détaillerons un scénario probable d'utilisation et montrerons comment notre proposition facilite le déroulement du scénario.

Enfin, l'objectif est de proposer un système de "management logiciel" dont la description est un élément de base. Nous proposons donc concrètement de combiner les progrès faits en réseaux (découverte voisinage, administration distante, ...) et les progrès fait en matière de description d'application (personnalisation, installations, vérifications, ...) Pour l'aspect réseau, on s'inspire partiellement du protocoles SNMP (Simple Network Management Protocol) qui permet la gestion d'entités distantes. Ce système, dont on s'attachera à décrire les mécanismes, pourra évoluer vers un système plus complexe de reconfiguration dynamique. La reconfiguration dynamique est un domaine complexe et encore très ouvert dont nous ne ferons qu'aborder certaines notions.

Apports

Ce que tous les ADLs permettent de faire (dans la limite du type de modules logiciels acceptés) c'est de décrire une application dans une configuration statique à l'aide des trois éléments de base des ADLs, à savoir les composants, les connecteurs et les configurations (voir [Med97]).

La configuration décrite est ensuite installée ou vérifiée par des outils relatifs à l'ADL. En effet les buts poursuivis par les ADLs peuvent être classés essentiellement en deux catégories (selon [Iss98]) , ceux qui sont relatifs aux spécifications formelles et ceux qui sont relatifs à de l'implémentation d'applications. Nous nous plaçons dans le cadre des ADLs dont le but est relatif à l'implémentation des applications et non dans le cadre de la vérification formelle. Notre objectif est donc, d'aller plus loin que la description d'une application à installer entièrement à partir d'une situation où rien n'est installé. Notre apport est de permettre de décrire une application qui inclus des composants à installer mais aussi des composants déjà installés, par exemple ceux d'une application en fonctionnement.

Comme dans [Bell99], nous proposons d'utiliser la représentation ADL pour la gestion et les outils d'administration. Cependant, dans cet article les outils et les interfaces associés sont reconnus comme étant à définir. Nous nous proposons donc d'apporter des détails sur ce point précis. De plus, nous nous proposons d'utiliser au maximum les informations que l'on peut qualifier comme étant de niveau ADL dans la mesure où l'une des caractéristiques des composants auxquels on s'intéresse est de posséder des fichiers de descriptions contenant des informations non-fonctionnelles qu'il serait dommage de ne pas réutiliser (on verra plus loin les détails).

Enfin, comme dans Olan [Balter98], on s'attachera à posséder une vision globale du système, c'est à dire la liste de tous les hôtes en faisant partis et disponible pour

l'installation de composants mais on inclura dans notre vision globale les éléments déjà installés pour en tenir compte ultérieurement quand il faudra pouvoir dire si un composant à besoin d'être installé ou pas.

Démarche

Notre but est de décrire une application à base de composants sur middleware pour faciliter un certain nombre de tâches liés au déploiement. Après avoir exposé dans la présente introduction le contexte, les objectifs et les apports, notre démarche sera ensuite de commencer par une étude de la littérature qui permettra de synthétiser l'existant et d'en tirer des leçons et bénéfices pour notre propre proposition. A partir de là, nous détaillerons la proposition et ses mécanismes et montrerons comment elle résout les problèmes évoqués au départ et quels ont été les apports. Enfin, nous conclurons avec quelques remarques générales et donnerons des perspectives pour le futur.

Chapitre 2

ADLs

Le présent chapitre, présente un scénario de déploiement d'applications. A travers ce scénario, nous voulons préciser encore quelques notions relatives aux ADLs et surtout parler de ce que l'on sait faire traditionnellement avec les langages de description d'architecture. I s'agit de rappeler les caractéristiques classiques d'un ADL, caractéristiques presque directement applicables à notre cas. Il ne s'agit pas de faire un survol des ADLs existant (pour cela voir plutôt [Med97, Iss98] où les ADLs sont classés et leurs principales différences étudiés) mais plutôt de voir des caractéristiques qui nous paraissent intéressantes pour la suite. Dans la suite, quand nous détaillerons notre proposition, on montrera comment on s'est inspiré des concepts intéressants passés en revus.

Scénario

Pour illustrer nos propos, nous allons étudier un cas fictif d'architecture logicielle. Après description du cas, nous illustrerons par des exemples, l'utilisation des ADLs connus.

L'exemple est décrit en ayant en tête la philosophie "multi-tiers" du modèle EJB (voir [EJB2]). Ainsi, la notion de couche présentation ou encore la signification de composant est claire.

Supposons que l'on ait d'une part un système de réservation de nuits d'hôtel et d'autre part un système de réservation de billets de train. Chaque système ayant été développé et installé dans un département différent d'une grande entreprise. Les deux systèmes sont assez similaires, ils se composent tous deux d'une base de données, d'une couche présentation (client) et d'un composant serveur le tout installé sur un seul et même hôte.

Une évolution du système est plus tard envisagée. On veut permettre à l'utilisateur de réserver train et hôtel à travers une unique interface client. On envisage donc de rajouter un nouveau composant, faisant ainsi le lien entre les deux systèmes existants et constituant ainsi une nouvelle application dans laquelle les deux anciennes sous-traite du travail.

Dans ce scénario, il n'est pas envisageable de désinstaller les deux applications existantes, pour ensuite décrire une application composée des anciens éléments et du nouveau, le tout à installer. En effet, les deux applications déjà installés sont déjà en fonctionnement et (on la supposé plus haut) elles utilisent des bases de données et pire encore elle sont peut-être en cours d'utilisation et donc leur absence (même peu de temps) serait préjudiciable. Nous ce que l'on désire, c'est faire évoluer le système en douceur, prendre en compte l'existant car une évolution vaut mieux qu'une révolution (qui remplace brutalement l'existant).

On veut donc décrire une application composée d'un composant à installer et de deux composants déjà installés sur des hôtes bien définis. On suppose évidemment, que les éléments installés sont du même type que ceux que l'on rajoute (EJB, CCM, ou autre mais du même type).

Il y n'a aucun intérêt à utiliser un ADL, pour déployer séparément trois applications chacune étant constitué d'un seul composants serveur. Dans ce cas, les outils de déploiement habituels livrés avec les middleware suffisent. Là où l'utilisation d'un ADL prend de l'intérêt, c'est pour déployer (c'est ce qui nous intéresse ici) une application composée de plusieurs composants ayant des interactions. En effet, on peut alors vérifier la cohérence des liens entre les éléments (interfaces correctes), générer du code spécifique ou encore configurer.

A notre connaissance, aucun ADL n'inclus encore de support pour les EJBs. Mais en supposant qu'un ADL ai un support EJB, il permettrait, sans doute, au minimum de décrire une configuration statique et de l'installée. Par exemple en schématisant, on aurait pour nos trois applications, composées en une:

```
COMPONENT reservtrain
Package="reservtrain.jar"
Host="train.essi.fr"

COMPONENT reservhotel
Package="reservhotel.jar"
Host="hotel.essi.fr"

COMPONENT glue
Package="glue.jar"
Host="tout.essi.fr"

CONNECTOR default_connector
...

CONFIGURATION
bind glue, reservhotel using default_connector
bind glue, reservtrain using default_connector
```

On ne détaille pas les connexions fonctions par fonctions comme dans la plupart des ADLs, on s'en explique plus loin et la raison principale est que ce niveau de détail ne paraît pas pour l'instant nécessaire. Ici on voit très clairement que pour pouvoir installer la configuration et compte tenu des connexions, il faut installer les trois composants.

Olan et les contraintes de placement.

L'ADL Olan (voir [Balter 98]) nous intéresse à plus d'un titre. Une des caractéristiques intéressantes de Olan est de pouvoir spécifier des contraintes de placement et ceci avec une assez grande précision. La deuxième caractéristique intéressante est la "configuration machine" qui nous a conforté (l'article [Bell99] également) dans l'idée d'avoir un agent local sur chaque hôte (une autre possibilité serais de tout centralisé, nous donnons plus d'explication dans la partie mécanismes du chapitre 4).

Dans Olan, on peut spécifier des contraintes de placement à l'aide d'attributs de gestion ("management attribute") de deux types. Le premier type d'attribut est le Nœud ("node"). La déclaration d'un nœud comprend le nom, l'adresse IP, la plate-forme, le type de système d'exploitation ainsi que des notions de charge en terme d'utilisation du CPU et en terme de nombre d'utilisateurs d'un hôte. Le deuxième type d'attribut est le propriétaire ("owner"). La définition d'un propriétaire comprend le "user name", le "user id" et la liste des groupes dont il fait parti.

Ensuite, l'utilisation de prédicats permet d'établir les règles à vérifier pour l'installation des composants, cela défini (en quelque sorte) une politique de distribution. Ces règles peuvent utilisées tout ou partie des champs des attributs de gestion. Par exemple, on peut ainsi installer un composant sur n'importe quel hôte parmi ceux dont la charge n'excède pas un certain seuil ou encore forcer la co-localisation (sans pour autant définir précisément sur quel hôte) de deux composants qui communiquent fortement, ceci afin d'optimiser, a priori, les performances (Olan n'a pas pour but de simuler une architecture pour en prévoir les performances, l'hypothèse que co-localiser améliorera les performances vient de l'utilisateur).

Dans notre cas, (on verra les détails plus tard), cela nous permet de nous interroger sur l'éventuelle présence de contraintes de placement dans notre proposition. Nous détaillerons plus loin comment nous procédons dans notre cas.

Intéressons nous maintenant à la deuxième caractéristique, à savoir la "configuration machine". Dans Olan, la "configuration machine" est ce qui a en charge d'exécuter les scripts qui contiennent "orders and guidelines for the initial deployment of the application". Dans notre modèle, cette phase s'apparente à ce qui viens après compilation de l'ADL, (à ce stade on a éventuellement déjà fait les taches de modifications du code de certains composants) et à l'utilisation qui en faite par les outils appelés pour l'instant "outils de *management logiciel*".

Darwin et l'instanciation dynamique

Darwin possède un mécanisme de "lazy instanciation" qui permet de décrire des composants instanciés lors de l'exécution au moment où ils deviennent nécessaires.

Cette capacité à exprimer du dynamisme et la caractéristique de Darwin qui permet selon [Cuesta99] de s'adapter à un modèle aussi complexe que Corba. Cependant, l'article laisse volontairement de coté les aspects autres que l'instanciation dynamique dans Corba pour éviter l'intersection de préoccupations variées. Mais, même l'invocation statique de Corba nécessite un certain degré de dynamisme (la fameuse "lazy instanciation" utilisé dans ce cas pour instancier le serveur au moment de l'appel réel, moment où l'on en a réellement besoin).

Sans rentrer dans les détails, on en retient qu'il est difficile de modéliser finement les mécanismes de l'architecture du middleware Corba.

Mais si l'on se limite à cette utilisation de la modélisation, il n'est pas forcément utile que l'on soit " proche du réel. Cela dépend du niveau sur lequel on veut agir pour modifier, vérifier, installer. Si l'on se contente de vérifier la présence des éléments qui constituent le serveur, pour qu'un client puisse toujours s'exécuter correctement, alors le degré de finesse dans la modélisation peut être moindre. Par contre, si l'on veut connaître tous les détails des communications ou instanciations pour pouvoir agir dessus, cela devient nécessaire d'être plus subtil.

Dans notre cas, nous en tirons que la finesse de description que fournira notre ADLs sera proportionnel aux objectifs fixés/aux missions "attribués" au système.

Chapitre 3

ADLs et Middleware

Introduction

Les middleware présentent des spécificités auxquelles il faut s'intéresser de plus près. Certains travaux ont tentés de d'utiliser ces spécificités. Même si les travaux ont une approche "différente" de la notre, et a priori pas directement intéressante, on montrera en quoi cela nous intéresse pour la suite.

Nous nous proposons donc ici de parcourir la littérature qui s'intéresse aux middleware ([ZI98b], [IBS98b], [Pro99]) et où est sont en œuvre des approches utilisant de manière plus ou moins importantes des ADLs. Nous mettrons en évidence, les éléments qui les intéressent dans les middleware, les caractéristiques intéressantes et ce qu'il en font.

Aster

Les middlewares fournissent aux applications qui les utilisent des services comme la gestion des transaction, la concurrence, la persistance ou la sécurité qui sont aussi appelés "propriétés non fonctionnelles" d'une application.

Les travaux sur ASTER, décrit entre autre dans [ZI98b] et [IBS98b] sont orientés sur les propriétés non-fonctionnelles des middlewares en particulier sur la gestion des transactions. L'idée principale est de personnaliser le service de transactions offert par le middleware afin de l'adapter aux besoins de l'application développée. Il s'agit vraiment de travailler sur des propriétés non-fonctionnelles, c'est à dire que on essaye de se détacher de ce qu'offre le middleware (ce qui implique que pour une implémentation, il faudra des versions Corba, EJB, ...) mais plutôt de définir ce qu'on veut et ensuite d'utiliser des outils pour le réaliser en utilisant une combinaison de services du middleware. Cette combinaison dépend évidemment du middleware réellement utilisé. Ces services seront dit "éligibles".

Pour y parvenir, l'approche utilisée est donc de décrire l'application en utilisant un ADL. Cette description porte une attention particulière sur l'aspect transaction ou plutôt supporte la spécification de propriétés transactionnelles. Elle permet donc de décrire finement quelles transactions doivent être utilisées (elle sont définies puis utilisées) et quand (i.e. pour quelles fonctions).

Une partie de la description concerne donc la définition de transactions ou plutôt de styles de transactions. Chaque style de transaction à un nom et des propriétés propres. Par exemple,

```
TRANSACTION_MODEL Tmodel {  
    PROPERTIES: atomic AND isolated;  
}
```

permet de définir un modèle de transaction, où la transaction à la propriété d'être atomique et isolée.

Ensuite, ce modèle peut être requis pour les fonctions de son choix (où plutôt, avec une étape supplémentaire, le modèle est utilisé dans la définition d'un TRANSACTIONNAL où est indiqué aussi les moment clefs, puis le TRANSACTIONNAL est requis pour telle ou telle fonction). Sont également défini, les fonctions appelées par le client et qui correspondent aux moments clefs dans une transaction qui sont les moments BEGIN, COMMIT et ABORT par exemple associé aux (pseudos-)appels t_begin(), t_commit() et t_abort() respectivement , dans le code source du client.

```
DEFINES TRANSACTIONNAL Trans:  
    MODEL = BankTrans; BEGIN = t_begin(); COMMIT = t_commit();  
ABORT= t_abort();
```

Chaque pseudo-appel est remplacé au cours d'une des phases du travail aboutissant à un middleware personnalisé (ou plutôt à une utilisation personnalisée) par le code correspondant aux opérations à faire à cet instant, ce code n'est pas toujours identique, il dépendra bien sûr des propriétés voulues, des modèles de transaction définis/appliqués.

Nous voyons ici un exemple intéressant d'approche où l'on s'intéresse au code source client A partir, de quelque chose de décrit dans l'ADL (en l'occurrence ici les propriétés transactionnelles voulues), on aboutit à une modification du code source du client. La modification de code existant (assez similaire à la génération de code), fait partis des techniques utilisées pour rajouter des possibilités non prévus à l'origine ou pour simuler une certaine sémantique (comme avec les RPCs [RPC] où l'on cherche à faire des appels de procédures distant tout en ayant la sémantique la plus proche possible des appels de fonctions habituels).

En ce qui nous concerne, on envisage de modifier le code source de certains composants (pas les composants décrits comme déjà installés mais plutôt ceux que l'on rajoute) pour atteindre l'objectif de se défaire des contraintes de localisation (attention ça ne veut pas dire que on ne pourra pas exprimer des contraintes de placement "à la Olan") et de tout exprimer de manière simple, aussi simple que si l'on travaillais en non-réparti.

Dans la mesure où les EJBs, sont basés sur RMI (Remote Method Invocation) essayer de conserver la sémantique de programmation en local est déjà en partie le cas sauf pour l'obtention de la référence du composant serveur sur lequel on veut agir. Ce qui explique les mécanismes que nous proposons dans la suite.

Cet article a pour nous plusieurs implications: on retient l'idée d'avoir plusieurs phases, au moins une où on modifie le code, une où on compile (ou plutôt recompile le morceau modifié), une où on déploie. Comme il peut être contraignant d'avoir à on peut envisager que l'utilisateur ne désire pas utiliser la possibilité de se défaire des contraintes d'obtenir le bon contexte en fonction de l'hôte et du type de container hébergeant le composant voulu.

On note au passage que pour eux aussi, le problème des configurations dynamiques reste ouvert.

Ces travaux ont également pour intérêt de montrer un ADL qui s'intéresse au middlewares. Pour certaines opérations, des renseignements au niveau de la description, sur la plate-forme et sur les sources et les langages utilisés sont utiles. Ceci, pour plusieurs raisons. La première étant de pouvoir agir sur le code et il faut savoir sur quel code. Deuxième raison, le renseignement sur le middleware est utile ici parce que les services à utiliser pour faire les modèles de transactions demandés sont "middleware-dependant" (et on voit mal comment cela ne le serait il pas). On en tire, que il faut dans notre modèle, des renseignements sur le type de middleware et voir aussi le type de container (vu que l'on veut s'en affranchir).

SOFA

SOFA¹ est un projet dont le but est de travailler sur les outils et concepts qui permettront de résoudre les problèmes qui empêchent d'aboutir à un marché du composant logiciel parfois appelé OTS pour "Off-The-Shelf" que l'on peut acheter et utiliser quasi-immédiatement, seuls ou combinés avec d'autres éléments dans une architecture existante.

Deux articles relatifs à SOFA ont retenu notre attention. Le premier ([Pro99]) porte sur les transactions et est intéressant dans la mesure où des considérations théoriques sur ces aspects transactionnelles sont faites sans s'embarrasser de la question de l'implémentation dans un middleware.

Le deuxième ([Mencl]), présente DCUP², un mécanisme (complexe) qui doit permettre de faire un système où les composants seront déplaçables, remplaçables, ... cependant je ne vois pas la possibilité d'utiliser des composants déjà installés, s'ils ne sont pas "DCUP specific".

Evidemment, apporter des possibilités nouvelles à un coût. Ce coût, il faut le minimiser, autant que possible. Il atteint son minimum quand de services à rajouter à un middleware, les nouvelles fonctions deviennent des possibilités incluses par défaut dans le middleware considéré.

Dans "*Transaction Models vers. Behavior Protocols*" ([Pro99]), ils s'intéressent aux transactions dans les architectures logicielles basées sur des composants (avec entre autre un intérêt pour Corba, EJB et MTS (Microsoft Transaction Server) pour les composants COM.

De même que dans Aster, il y a la définition de modèles de transaction mais la façon de l'utiliser dans un middleware n'est pas précisément décrite, "*we would like to create a prototype of transactional SOFA node, which will participate in transactions and which will reflect the transactional behavior of a transaction manager*". Cependant, ils disent que l'approche est applicable dans les "*component-based architecture*" en général.

¹ SOFTware Appliance

² Dynamic Component UPdating

Ce qui est intéressant, c'est que il ne parlent des modèles de composant existants que pour dire comment sont les transactions dans ce modèle et pour justifier la nécessité de leur modèle où la définition des transactions par des protocoles est très fine. Donc, ils restent éloignés des détails des middleware. Ce qui les intéressent, c'est un travail sur les transactions.

Ici pour coordonner les transactions, ils font l'hypothèse que il y a un composant particulier, le composant "transaction manager". Mais en réalité, c'est plus compliqué parce que des éléments tels que les *log manager*, *ressources*, *lock manager* font aussi parti du système.

Dans [Menc1], le modèle DCUP est présenté, il est partie intégrante du projet SOFA, il vise à permettre la mise à jour dynamique de composants. Ici par composant, il faut entendre entité logicielle qui encapsule du code. Par exemple, dans l'implémentation pour Java, les composants sont l'encapsulation de code Java. Le modèle est prévu pour pouvoir être implémenté dans d'autre environnement, car il suffit que le support pour ce nouvel environnement soit ajouté au compilateur CDL et que des modules de génération de code soit développé.

Le schéma semble assez complexe et volontairement on ne veut pas être aussi complexe. On veut fournir un cadre pratique fonctionnant avec les EJBs. Un cadre tel que si on voulait écrire l'implémentation, ce ne serait pas si difficile que cela.

Le schéma présenté, oblige à écrire des *DCUP components*, qui sont donc spécifiques. Nous voulons réutiliser des composants déjà installer sans absolument rien toucher. Chaque composant DCUP possède un *component manager* et un *component builder*. Ces deux mécanismes permettent la mise à jour du composant et le déplacement. L'utilisation d'un *wrapper* est donc systématique. Donc puisque c'est systématique, ils proposent de la génération de code à partir de la description d'un composant en CDL (Component Description Language) qui prend son inspiration dans l'IDL de Corba mais y ajoute des construction ou modifie des constructions existantes. Par exemple, une différence notable avec un IDL est la déclaration des fonctions requises (en plus des fonctions fournies).

```
interface TellerInterface {
    /* attributes and methods here */
    ...

    requires:
        BankDemo::datastoreAccess dsReq;
        BankDemo::supervisorAccess supReq;
};
```

Dans "Abstract ADL" ([Bell99]) que nous étudions par la suite, il y a aussi cette notion d'interface fournis et d'interfaces requises.

D'autres notions inspirées d'autres ADLs sont aussi présentes. Par exemple, la notion de propriétés qui sont des paires nom/valeur associé à un composant. Le modèle possède aussi la notion de sous-composant (que nous expliquons plus loin dans l'étude de "Abstract ADL")

Dans ce modèle, tous les composants doivent rentrer dans ce schéma et c'est pour cela que les possibilités prévues (déplacement, mise à jour) peuvent fonctionner. Par contre, il n'est pas possible d'inclure dans cette architecture des composants existants sans devoir les modifier quelque peu.

Abstract ADL

Ayant remarqué que les concepts de composants, de connecteurs et de configuration sont centraux dans les ADLs, l'approche *Abstract ADL* a comme idée de proposer un ADL abstrait à raffiner selon les besoins des processus logiciels voulus. Par exemple, notre besoin sera dans le domaine de l'implémentation, alors que d'autres auront des besoins en matière d'analyse ou de simulation d'architectures.

La syntaxe présentée dans l'article n'est pas déterminante, ce qui importe ce sont les concepts. L'idée importante est de décrire des configurations à l'aide d'un ADL, puis de compiler la description pour en faire une représentation intermédiaire (ou IR). Cette représentation intermédiaire, est ensuite utilisée par des *CASE¹ tools*. Les outils peuvent par exemple être du domaine du déploiement, c'est ce qui nous intéresse plus particulièrement.

Composants

Le composant est le lieu pour l'implémentation des fonctionnalités dans une architecture logicielle. Il peut être vu comme une boîte noire. On peut tout ignorer de son implémentation. D'ailleurs l'implémentation réelle peut mettre à profit des sous-composants, des données et du code. C'est l'élément de base dans une application. Le composant n'expose au monde extérieur que son (ou ses interfaces).

Certains ADLs ne connaissent que les composants primitifs alors que d'autre (par exemple Darwin) autorisent les composants composites, c'est à dire des composants eux mêmes composés d'un ensemble de composants reliés (donc une architecture réutilisée comme un composant). C'est pourquoi un modèle abstrait de composant ("Abstract ADL") doit prendre en compte ces deux possibilités, même si la possibilité d'avoir des composants composites n'est pas utilisée.

Pour chaque composant, il existe au minimum, une interface. Cette interface fournis la liste des "opérations" (au sens large) possible avec ce composant. Il faut aussi connaître les interfaces requises des autres composants. Comme il s'agit ici d'ADL abstrait, il faut prévoir la possibilité qu'il y est de multiples interfaces (par exemple cela est nécessaire

¹ Computer Aided Software Engineering

pour composants CCM ce Corba). C'est aussi pour pouvoir exprimer les interfaces des sous-composants, quand un composant est composite.

Comme un composant est la plus petite unité de description qui encapsule du logiciel, leur assertion est que leur définition contienne des renseignements complets et précis sur le type de code source encapsulé ainsi que sur son emplacement (par exemple pour l'installation d'un composant) ou encore ses besoins en terme de support d'exécution (OS, version, bibliothèques). On pourra remarquer que c'est au moment où on voudra concrétiser cet ADL abstrait, qu'il faudra avoir précisément défini ces informations.

Connecteurs

Les connecteurs permettent l'interconnexion entre composants. Mais il faut encore définir plus tard, les connections entre composants. Ces interconnexions utiliseront l'un des type de connecteurs définis.

Les interconnexions (ou *bindings*) sont des liens entre deux composants en utilisant un connecteur. C'est l'occasion de vérifier les règles de connections. Comme il faudra concrétiser cet ADL abstrait, il peut très bien n'y avoir qu'un seul type de connecteur. Dans notre cas, on prévoit que le connecteur permette de vérifier l'interopérabilité.

C'est aussi l'emplacement pour des propriétés non-fonctionnelles, bien qu'ils avouent que aujourd'hui, leur modèle ne permet pas de définir ce genre de choses. Il est cependant important, d'avoir prévu l'emplacement pour cela.

Configurations

En fait, une configuration, c'est pour eux la liste des instructions, d'interconnexions et d'instanciations de composants.

Finalement, le but est de décrire une configuration et d'en obtenir ensuite une autre représentation dite IR ("Intermediate Representation"). C'est cette représentation qui est ensuite manipulée par les outils. Chaque outils utilise cette représentation comme structure commune.

Pour obtenir l'IR, il faut au départ décrire l'architecture, soit de manière graphique, soit par un ADL. Le but de l'article n'est pas de définir précisément la syntaxe de cet ADL (mais plutôt la hiérarchie d'objets représentant une configuration), ni de définir comment un outil graphique devrait être. Peu importe comment (ADL-compiler, outil graphique ou autre) on l'obtiens, c'est la représentation intermédiaire qui est au cœur du système.

Un reproche que l'on peut faire est que la représentation intermédiaire proposée n'est rien d'autre qu'une hiérarchie d'objets, dont le problème du stockage (et donc de la représentation sur disque) est résolu en utilisant le mécanisme de sérialisation de Java.

Cela a pour conséquence immédiate, que les outils qui partagent cette représentation sont nécessairement écrit en Java, pour pouvoir désérialiser le fichier qui sert à stocker la représentation intermédiaire et en faire quelque chose.

En pratique, lorsque l'on voudra concrétiser cet ADL pour un environnement spécifique. Le langage Java, ne sera pas forcément une approche intéressante ou peut être ne sera tout simplement pas disponible.

Si l'on décrit une architecture logicielle, c'est pour en faire quelque chose ensuite. La représentation intermédiaire est donc un moyen d'interaction entre différents outils ("CASE tools"). Ces outils sont par exemple, des outils de développement, de configuration, d'administration, de déploiement ou encore de reconfiguration.

En ce qui nous concerne, nous détaillerons dans notre proposition, les outils et les mécanismes que l'on propose. Par exemple, nous proposons un ensemble d'outils répartis pour le déploiement d'ABC¹.

¹ Applications à Base de Composants sur Middleware

Chapitre 4

Proposition

L'objectif de ce chapitre est de présenter notre proposition. Cette proposition vise à atteindre les objectifs énoncés dans l'introduction. Elle s'appuie en parti sur les travaux présentés dans les chapitres précédents ou au moins sur des enseignements tirés de travaux présentés précédemment.

Notre proposition s'articule en deux parties principales: une première partie décrivant avec précision les choix de conception de notre ADL et ensuite sa syntaxe et ses fonctionnalités; puis une deuxième partie décrivant les mécanismes de "management logiciel" que nous proposons et comment il utilisent la description de l'architecture.

ADL

Comme dans "Abstract ADL" [Bell99]. notre proposition s'articule autour des trois concepts fondamentaux que sont les composants, les connecteurs et les configuration.

A travers l'étude de ces trois notions, nous détaillons notre proposition et nos choix. L'ADL est volontairement orienté vers la description d'applications à base de composants EJB mais il y aura parfois des réflexions et commentaires au sujet d'autres modèles (CCM essentiellement).

Nous proposons que cette description soit autant que possible générée par un outil à partir de fichiers déjà existant dans les modèles EJB (*deploy descriptor*) et CCM, les informations manquantes seront à écrire par l'utilisateur. Ceci est également valable pour les composants déjà installés, les agents (voir partie mécanismes) se chargeant de récupérer les informations des composants qui sont sur leur hôte.

Composants

Interfaces

Comme on l'a dit/vu dans l'état de l'art avec Abstract ADL, pour pouvoir agir à partir de la description, il faut pour chaque composant un certains nombres de propriétés ("*renseignements complets/précis sur le type de code source encapsulé* "). Mais en premier lieu il faut connaître pour chaque composant les interfaces fournies et requises. La plupart des ADLs listent une par une les fonctions (avec leur signatures) fournies puis les fonctions requises. Cela peut être utile pour les transactions dans ASTER par exemple, mais dans notre cas une autre possibilité est envisageable. Puisque l'on travaille avec les EJBs (et donc sur un modèle du style RMI), l'interface d'un composant est une interface au sens Java du terme. Par exemple,

```
package com.web_tomorrow.interest;

import javax.ejb.EJBObject;
import java.rmi.RemoteException;
```

```

/**
This interface defines the 'Remote' interface for the 'Interest' EJB. Its
single method is the only method exposed to the outside world. The class
InterestBean implements the method.
*/

public interface Interest extends EJBObject
{
    /** Calculates the compound interest on the sum 'principle', with interest rate per
    period 'rate' over 'periods' time periods. This method also prints a message to
    standard output; this is picked up by the EJB server and logged. In this way we
    can demonstrate that the method is actually being executed on the server,
    rather than the client.
    */

    public double calculateCompoundInterest(double principle,
        double rate, double periods) throws RemoteException;
}

```

Interface d'un composant ne fournissant qu'une seule fonction extérieure.

Dans le modèle EJB, les composants possèdent également une interface dite "Home", celle présentée ici est celle dite "Remote". Donc, si le type de composant est EJB, alors l'interface fournie c'est le nom de la classe Java correspondante. Ensuite pour vérifier que le composant x est bien en mesure d'utiliser l'interface y, il faudra trouver un composant qui fournit l'interface y et vérifier dans la configuration qu'un lien (*binding*) existe entre ces deux composants.

Dans l'ADL, nous exprimons donc pour chaque composant les interfaces fournies et requises à l'aide de la syntaxe suivante (nous adoptons une syntaxe XML plus facile à traiter par des programmes) :

```

<component name="filter">
  <interface>
    <provide>
      <home> homeinterface </home>
      <remote> remote interface </remote>
    </provide>
    <require>
      <home> homeinterface </home>
      <remote> remote interface </remote>
      <home> homeinterface </home>
      <remote> remote interface </remote>
    </require>
  </interface>
  .
  .
  .
</component>

```

Dans le cas de notre scénario (ch. 2), les composants déjà installés ne requièrent aucune interface (ou du moins aucune qui soit d'intérêt pour nous dans le cadre de la description de la nouvelle architecture) mais par contre, le composant nouveau fournit une interface (pour d'éventuels clients qui l'on ne décrit pas ici) et requiert les interfaces des deux composants déjà installés. Sans rentrer dans les détails, nous présentons la partie interface de la description des composants "ToutEnU" et "ReservTrain"

```

<component name="ReservTrain">
  <interface>

```

```

        <provide>
            <home>HomeReservationTrain</home>
            <remote>ReservationTrain</remote>
        </provide>
    </interface>
    . . .
</component>
<component name="ToutEnUn">
    <interface>
        <provide>
            <home>HomeToutEnUn</home>
            <remote>ToutEnUn</remote>
        </provide>
        <require>
            <home>HomeReservationTrain</home>
            <remote>ReservationTrain</remote>
            <home>HomeReservationHotel</home>
            <remote>ReservationHotel</remote>
        </require>
    </interface>
    . . .
</component>

```

Propriétés

En dehors des interfaces, il faut connaître d'autres éléments pour chaque composant.

On commence par le middleware, le type de container et la version de la norme supportée. Ceci afin de pouvoir vérifier si un composant peut être installé sur un hôte donné, en effet les composants peuvent posséder des spécificités pour un type de container et donc ne pouvoir s'installer que sur celui-ci. Il faut aussi pouvoir vérifier que l'on sait communiquer entre les types de container cités.

```

<component name="ToutEnUn">
    . . .
    <host>
        <hostname>dea2b.essi.fr</hostname>
        <middleware>ejb</middleware>
        <type>jboss</type>
        <spec_version>1.1</spec_version>
    </host>
    . . .
</component>

```

Ensuite, il nous faut le nom du composant tel que publié dans le JNDI. Ceci est utile pour les mécanismes de modification / génération de code source (L'utilisateur appellera les outils spécifiques qui s'en occupe avec comme paramètres les fichiers sources à modifier et le nom du fichier contenant la description de l'architecture, d'où ce genre d'informations dans la description.). Ce nom est aussi le nom qui désigne le composant sur un hôte.

```

<component name="ToutEnUn">
    . . .
    <jndiname>ejb/Interest</jndiname>

```

```
    . . .  
</component>
```

Il nous faut également le nom complet de l'archive contenant le composant. Ceci afin de pouvoir transmettre et installer n'importe où le composant. Et même de récupérer les infos du deploy descriptor.

```
<component name="ToutEnUn">  
    . . .  
    <archive>application.jar/composant.jar</archive>  
    . . .  
</component>
```

(Le nom donné en exemple, suppose que le fichier de description de l'architecture soit situé au même niveau que "application.jar" dans la hiérarchie des fichiers).

Il nous faut également savoir l'hôte pour dire où est un composant déjà installé ou alors pour dire où l'installer.

Connecteurs

Comme il le font remarquer dans [Iss98] le connecteur est lieu pour spécifier les propriétés relatives au protocole de communication et aussi aux propriétés non fonctionnelles. En pratique ce sont des unités d'interaction qui ne correspondent pas forcément à une entité compilé lors de l'implémentation du système.

Nous désirons ici caractériser les interactions entre deux composants du middleware. Dans un premier temps (voir ensuite la remarque sur les transactions) nous caractérisons cette interaction par le type de container de chacun des composants. Ceci afin faire ensuite les vérifications qui s'imposent en terme de communication possible d'un composant vers un autre. C'est à dire qu'il s'agit de vérifier l'interopérabilité des composants.

Pour nous, le triplet MIDDLEWARE/TYPE/VERSION caractérise le composant en face. La documentation du compilateur ADL donnera la liste des couples de triplets valables pour un connecteur. Le couple (x/x/x , x/x/x) est toujours valable mais le couple (x/y1/x,x/y2/z) pas toujours.

Dans la description, il faut donc nommer puis décrire les connecteurs utilisés. Ces lors de la vérification de la description que l'on dira si oui ou non ces connecteurs sont valables.

```
<connector name="sun_vers_jboss">  
  <client>  
    <middleware>ejb</midlleware>  
    <type>sun-ri</type>  
    <spec_version>1.1</spec_version>  
  </client>  
<serveur>
```

```
<middleware>ejb</midlleware>
<type>jboss</type>
<spec_version>1.1</spec_version>
</serveur>
</connector>
```

Les connecteurs sont à termes extensibles. On pourra les utiliser pour tout ce qui concerne des propriétés non fonctionnelles. Par exemple, la spécification des EJBs indique que "Transaction interoperability between containers provided by different vendors is an optional feature in this version of the EJB specification." Donc, puisque le support de l'interopérabilité des transactions est optionnel, on a des container qui les supportent et d'autre qui ne les supportent pas. Cette information (support ou non des transactions) peut être mises dans le connecteur. Ensuite, on pourra vérifier que l'interaction entre deux composants nécessitant le support des transactions est possible.

Configuration

Dans "Abstract ADL", une configuration, c'est la liste d'instruction d'instanciation et d'interconnexion des composants. Dans Olan, il s'agit de l'instanciation d'interfaces et l'expression des interactions entre composants. Finalement une description ADL c'est lister les composants, lister les connecteurs, et enfin exprimer la configuration en terme de composants reliés par des connecteurs. Pour simplifier, on a par exemple que si A est relié à B et que A requière une interface fournie par B, alors la description est correcte. Sinon, la description est incorrecte puisque, aucun des composants reliés à B ne fournis l'interface requise par A.

La description, ayant au préalable lister et nommer des composants et des connecteurs, une configuration consiste maintenant à exprimer les liens entre composants à l'aide de connecteurs. Par exemple, puisque que chaque connecteur possède deux points de connexion, il faut pour chacun préciser quel composant y est reliés. Voici un exemple, et en même temps la syntaxe utilisée:

```
<bind>
  <connector>nom_du_connecteur</connector>
  <client>ToutEnUn</client>
  <server>ReservTrain</server>
</bind>
```

Ceci résulte dans le fait que le composant ToutEnUn est client du composant ReservTrain. De même, on pourra exprimer par un autre *binding* que le composant ToutEnUn est client du composant ReservHotel

Outils et mécanismes

Pour réaliser les objectifs fixés, un certains nombres de mécanismes doivent être étudiés en détail. Pour pouvoir faire les évolutions d'architectures citées, nous proposons

d'étudier comment la description faite est maintenant est utilisé au cœur du système de déploiement.

Architecture

Pour notre dispositif de déploiement, nous entrevoyons deux possibilités principales d'architecture. La première possibilité serait d'avoir un seul logiciel centralisé qui puisse communiquer avec les serveurs EJB sur les autres hôtes. Une deuxième possibilité serait d'avoir un système distribué où chaque éléments (un sur chaque hôte) participe à l'ensemble du mécanisme.

La première solution est mise en œuvre dans l'outil de déploiement présent dans l'implémentation de référence de Sun ("Sun RI"). Il permet d'installer à distance sur d'autres hôtes. Si ce logiciel possède ce mécanisme, il est possible de faire quelque chose de similaire pour tout type de serveurs. Cependant, si ce mécanisme marche pour Sun RI, il n'est pas dit pour autant que l'on puisse avoir le détail du protocole. D'autre part, Il n'est pas dit que cette solution permettent de remplir l'objectif de s'affranchir du type d'implémentation de serveur (EJBs ou autre). Par exemple, avec le serveur Jboss, on installe en copiant les composants (les fichiers archives) dans un répertoire spécial et le serveur détecte les nouveaux composants arrivés ou enlevés de ce répertoire. Donc a priori, JBoss ne supporte pas d'installation à distance, mais plutôt hôte par hôte.

En conclusion, on retient que pour pouvoir s'adapter à tous les types de serveurs il faut pouvoir communiquer avec le serveur et utiliser sa propre procédure d'installation d'éléments et que pour faire de l'installation distribuée nous retiendrons la deuxième possibilité qui consiste à installer un agent sur chaque hôte et utiliser notre propre protocole de communication inter-agents ceci afin de posséder un mécanisme d'installation/commande à distance.

SMNP

Pour remplir les objectifs, on s'inspire en outre du protocole SNMP¹ pour nos outils. Le protocole SNMP permet, dans le monde des réseaux, la gestion et l'administration distante d'entités du réseau. Par entité, il faut comprendre, tout élément matériel possédant une interface avec le réseau et pouvant être joint par le gestionnaire.

Au cœur du concept de SNMP, il y a le gestionnaire et les agents. Sur chaque entité, un agent est présent pour répondre aux requêtes du gestionnaire ou pour lui envoyer spontanément une information (événement). Dans ce protocole, les requêtes consistent simplement à demander la lecture ou l'écriture de variables. Comme remarqué dans [Case90], ceci a pour conséquences de simplifier considérablement le travail de l'agent et d'éviter d'introduire dans le protocole un nombre toujours croissant de commandes de gestion.

¹ Simple Network Management Protocol

Notre système

Avec un système "à la SNMP" ,on espère collecter et utiliser des informations qui sont connus localement puis après un certain traitement en déduire les actions à effectuer sur chaque hôte. Dans la suite, on regardera donc plus en détail les mécanismes que le gestionnaire et les agents doivent implémenter afin de pouvoir remplir leur mission.

Pour remplir nos objectifs, il faut distinguer clairement les différentes phases et outils qui interviennent Dans un premier temps, les outils qui modifient le code source sont potentiellement utilisés par l'utilisateur quand il désire développer avec les facilités que l'on propose des nouveaux composants. Dans un deuxième temps, l'utilisateur fait appel véritablement au programme de déploiement et s'adresse donc au gestionnaire sur un des hôtes.

Première phase: les outils de modifications de source

Pour faciliter l'écriture de la répartition à partir des notions locales toujours évoqués, nous proposons que l'utilisateur puisse soumettre son code source à des outils qui généreront le code adéquat en fonction du type de client d'un composant, du type de *container* (coté client et coté serveur) utilisé et de l'emplacement du serveur (hostname).

Il faut distinguer deux type de client pour un composant EJB, les clients qui sont eux-mêmes des composants EJB et les clients qui n'en sont pas (par exemple une application Java classique ou une applet).

Dans le premier cas, si le composant référencés est situé sur un hôte différent, il faut vérifier que pour chaque référence à un autre composant, le nom JNDI complet soit indiqué dans le fichier adéquat (comme cela n'est pas quelque chose qui fait partie de la norme c'est souvent indiqué dans un fichier similaire au deployment descriptor standard). Par exemple, avec un serveur de type JBoss, les noms JNDI sont déclarés dans un fichier jboss.xml qui est dans l'archive du composant. A la demande de l'utilisateur, on vérifie que pour chaque référence, le nom complet JNDI correspondant figure. Il suffit que l'outil recherche les balises <jndi-name> et </jndi-name> et insère le nom nécessaire.

```
<jboss>
  <enterprise-beans>
    <session>
      <ejb-name>Bean A</ejb-name>
      <ejb-ref>
        <ejb-ref-name>ejb/myBean</ejb-ref-name>
        <jndi-name>t3://otherserver/application/beanB</jndi-name>
      </ejb-ref>
    </session>
  </enterprise-beans>
</jboss>
```

Pour le cas où le client n'est pas lui même un composant, il faut s'adresser directement au JNDI correspondant au composant voulu et obtenir une référence. Par exemple,

toujours avec JBoss, pour obtenir un contexte sur un composant publié sous le nom ReservTrain sur l'hôte dea2b.essi.fr:

```
...
Properties env = new Properties();
env.setProperty("java.naming.factory.initial",
"org.jnp.interfaces.NamingContextFactory");
env.setProperty("java.naming.provider.url", "dea2b.essi.fr:1099");
env.setProperty("java.naming.factory.url.pkgs", "org.jboss.naming");

InitialContext jndiContext = new InitialContext(env);
Object ref = jndiContext.lookup("reservtrain");
...
```

Une fois le contexte obtenu, nous voyons que l'utilisation est ensuite classique. Pour que l'outil puisse générer le code qu'il faut et à l'endroit qu'il faut, nous proposons que l'utilisateur qui utilise cet outil utilise une directive spéciale dans son code source par exemple une ligne "#GENERATE_CONTEXT_LOOKUP" qui sera remplacée par le code généré.

Deuxième phase: le "management logiciel"

Sur l'un des hôtes, l'utilisateur décrit son application et fournit cette description à un gestionnaire. Cette application utilise des composants déjà installés ou uniquement des composants à installer. Si la description est correcte, alors le gestionnaire peut s'adresser aux agents du système pour aboutir finalement à l'application voulue par l'utilisateur. Il faut vérifier la situation courante par rapport à la configuration décrite, puis on procède aux installations nécessaires.

Chaque agent est un logiciel complet qui peut faire tout à tour le rôle de coordinateur central ou simplement d'agent répondant aux requêtes/demandes de ce coordinateur. Le coordinateur est le premier auquel l'utilisateur s'adresse. Alors que dans SNMP, il existe une asymétrie, car les agents sont très simple, ils se contentent d'écrire ou de lire des informations alors que le gestionnaire est un logiciel plus complexe qui s'adresse à plusieurs agents. En pratique, on peut écrire ces deux fonctions en deux logiciels mais on exige que ces deux fonctions soient présentes sur chaque nœud et donc qu'il n'y ait aucune asymétrie.

Pour que le système fonctionne, il faudra fournir au gestionnaire la liste de tous les agents du système (c'est à dire les hôtes et les ports sur lequel écoute l'agent). Les agents devront également pouvoir répondre à des requêtes d'informations sur le serveur avec qui il communique (middleware/type/version).

A la demande du gestionnaire, chaque agent doit pouvoir fournir la liste des composants installés sur l'hôte qu'il gère et doit fournir la description ADL de ces composants (au moins une description partielle que l'on peut compléter manuellement). Pour cela, l'agent utilisera des informations du "deploy descriptor" où l'on peut récupérer les informations qui nous sont nécessaires. C'est le gestionnaire qui à partir de l'information si un composant est installé ou pas décidera de donner un ordre d'installation.

Le gestionnaire peut ordonner à un agent l'installation d'un composant. Pour cela, il faut d'abord transmettre l'archive correspondant au composant. Le gestionnaire échangera donc avec l'agent l'URL à laquelle l'agent devra récupérer l'archive. Pour se faire, le gestionnaire utilisera un serveur http ou ftp local ou deviendra pour un instant serveur sur un port indiqué dans l'URL transmise. Du côté de l'agent, on doit avoir prévu de pouvoir récupérer un fichier par http ou ftp.

Le gestionnaire peut à tout moment demander à un agent d'installer un composant. Le gestionnaire ne demandera l'installation que des composants qu'il sait avoir transmis à l'agent. Si l'agent ne possède pas le composant, une retransmission est toujours possible. L'agent doit pouvoir dire quels archives de composants il possède.

Chapitre 5

Conclusion

Objectif et démarche de travail

Ces dernières années ont vu émerger beaucoup de nouveaux modèles de middleware et de modèles de composants. Pour autant, le développement d'une architecture logicielle ne se simplifie pas toujours puisque qu'il faut s'adapter aux nouveaux modèles. Le problème de l'hétérogénéité ne se résout pas mieux qu'avant, au contraire même pour un middleware donné la multiplicité des implémentations et des versions compliquent les choses. Nous avons donc voulu proposer un modèle de description, volontairement tourné vers l'installation des composants, qui permet de s'affranchir de ce type de contraintes. De plus, nous avons voulu mettre la description d'architecture au cœur d'un système de *management logiciel* qui permet entre autre de rajouter des éléments de manière rapide et fiable (c'est à dire que l'adéquation du rajout avec l'existant est vérifié et les travaux préalables sont automatisés (installation des composants dont on a besoin s'ils ne sont pas encore présent)).

Le travail a débuté par l'étude des spécificités des modèles de composants et l'étude de langages de description d'architectures. Cela a permis de proposer ensuite un modèle de description qui est extensible, qui est simple et qui réutilise les informations existantes (puisque que l'on s'est aperçu que le code de composant est accompagné d'informations supplémentaires de niveau ADL).

Evaluation

Il aurait fallu se pencher plus sur d'autres modèles de composant que les seuls EJBs. Les CCM de Corba, bien qu'abordés aurait mérités plus d'attention surtout dans la mesure où il existe des liens et des similarités entre les deux modèles de composant. Cependant, il semble que l'un ne soit pas un sur-ensemble de l'autre et inversement, comme on peut le croire parfois, ne serait ce que parce que les deux continuent d'évoluer séparément.

Sinon, les objectifs sont partiellement remplis, au moins en théorie. Il faudrait avoir plus de code développé pour tester la validité du modèle proposé et le fonctionnement correct des mécanismes proposés.

Perspectives

Une extension possible de l'ADL serait descendre à un niveau plus fin de description (il faudrait descendre à un niveau fonction par fonction, comme on l'a vu dans la littérature au sujet des transactions, pour décrire des propriétés non-fonctionnelles. Ou alors, dans un premier temps de vérifier des hypothèses concernant le

support de telle propriétés (puisque le support est parfois optionnel). Auquel cas, ce genre de chose s'apparente de la vérification (comme quand on fait la vérification de type ou autre).

Il serait intéressant de faire plus dans le domaine de la reconfiguration dynamique dans les ABCM. Mais il s'agit là d'un domaine loin d'être trivial. Il serait intéressant de voir comment adapter certaines approches vues dans la littérature aux modèles modernes de composants et en particulier le modèle EJB qui nous a intéressé ici.

A court terme il serait faisable de réaliser une implémentation complète correspondant à la proposition faites. Il faudrait alors affiner les formats d'échange et les protocoles/

Bien que centrée sur un modèle de composant particulier, nous pensons qu'une approche similaire peut être utilisée pour les autres modèles de composant. Mieux encore, il faudrait quand cela est possible pouvoir décrire des liens inter-modèles de middleware (par exemple en travaillant encore sur les connecteurs).

Enfin, rien n'interdit de perfectionner ou d'augmenter la liste des outils utilisant l'ADL comme entrée.

BIBLIOGRAPHIE

- [Balter 98] R. Balter, L. Bellissard, F. Boyer, M. Riveill and J. Y. Vion-Dury, "Architecturing and Configuring Distributed Applications with Olan", *Proc. IFIP Int. Conf. on Distributed Systems Platforms and Open Distributed Processing (Middleware'98)*, The Lake District, 15-18 September 1998
- [Bell99] Tech Report 13. (C3DS Project) Abstract ADL, *Luc Bellissard, Noël De Palma, David Féliot, Maria Serrano, 23 Pages, 1999*
- [Bern96] Philip A. Bernstein. *Middleware --- A Model for Distributed System Services*. Communications of the ACM, 39(2):86-98, February 1996.
- [Blair00] Gordon Blair, Lyne Blair, Valérie Issarny, Petr Tuma, Apostolos Zarras. The Role of Software Architecture in Constraining Adaptation in Component-based Middleware Platforms. *In Proceedings of Middleware 2000 -- IFIP/ACM International Conference on Distributed Systems Platforms and Open Distributed Processing*. April 2000, Hudson River Valley (NY), USA. Springer Verlag, LNCS
- [Cal91] J.R. Callahan and J.M. Purtilo, "A packaging system for heterogenous execution environments," *IEEE Transactions on Software Engineering*, vol. 17, no. 6, pp. 626-635, 1991
- [Case90] RFC 1157. Simple Network Management Protocol (SNMP). J.D Case, M. Fedor, M.L. Schoffstall, C. Davin. May-01-1990. IETF
- [Cuesta99] Cuesta, C.E., De La Fuente, P., and Barrio-Sol' Orzano, M. An architectural perspective of the static invocation in CORBA. *In Electronic Proceedings of First IFIP Conference on Software Architecture* (Feb. 1999).
- [EJB2] Sun Microsystems, *Enterprise JavaBeans Specification*, Proposed Final Draft Version 2.0,
- [Gar95] D. Garlan, R. Allen, J. Ockerbloom, "Architectural Mismatch or Why it's so hard to build systems out of existing parts", *Proceedings of the 17th International Conference on Software Engineering*, April 1995.
- [Hub99] Richard Hubert, "An Annotated Positioning of CORBA Components and EJB", 18 Jan. 99
- [IBS98b] Valérie Issarny, Cristophe Bidan, Titos Saridakis. Achieving Middleware Customization in a Configuration-Based Development Environment: Experience with the Aster Prototype. *In Proceedings of the 4th International Conference on Configurable Distributed Systems*, pages 207-214, May 1998, Annapolis, Maryland, USA.

- [Iss98] Issarny V., Saridakis T., Zarras A., *A survey of Architecture Description Languages*, C3DS Deliverable A3.1, ESPRIT LTR Project N24962, pp.8-28
- [Jndi] Sun Microsystems, *Java Naming and Directory Interface , Application Programming Interface (JNDI API)*, Version 1.2, July 14, 1999
- [Kron76] *Programming in-the-Large Versus Programming in-the-Small*, by Frank DeRemer and Hans Kron, *IEEE Transactions on Software Engineering*, Vol SE-2 , pp80-86, June 1976.
- [Marv99] Raphaël Marvie, CORBA Components : la proposition unifiés "*Du modèle d'objets au modèle de composants*", *GDR Programmation*, Nantes, France, 19 May 1999, <http://corbaweb.lifl.fr>
- [Med97] N. Medvidovic and R.N. Taylor. *A Framework for Classifying and Comparing Architecture Description Languages*. In Proceedings of the Sixth European Software Engineering Conference, number 1301 in Lecture Notes in Computer Science, pages 60--76, New York, September 1997. SpringerVerlag.
- [Mencl] Mencl, V.: *Component Definition Language*, Master thesis, Charles University, Prague, 1998.
- [O'reil99] Caroline O'Reilly B.A., "BeanBag An Extensible Framework for Describing, Storing and Querying Components", A dissertation submitted to the University of Dublin, Master of Science in Computer Science, September 99
- [Pro99] Marek Prochazka, Frantisek Plasil: *Transaction Models vers. Behavior Protocols*. Presented at the Week For Doctoral Students, Prague, June 10, 1999
- [RPC] RFC 1057. RPC: Remote Procedure Call Protocol Specification Version 2, Sun Microsystems Inc., June 1988. IETF
- [Senart00] Aline Senart, "Aspect Dynamiques dans les Architectures Logicielles en Environnement Réparti", Rapport de DEA, 20 Juin 2000.
- [Snmp++] Peter Erik Mellquist, "SNMP++, An Object-Oriented Approach to Developing NETWORK MANAGEMENT APPLICATIONS", Hewlett-Packard Professional Books, A Prentice Hall PTR title
- [ZI98b] Apostolos Zarras, Valérie Issarny. A Framework for Systematic Synthesis of Transactionnal Middleware. In *Proceedings of MIDDLEWARE'98*, pages 257-272, September 1998, The Lake District, England.