Sylvain Laurent

Promotion 2002

Rapport de stage ingénieur

Intégration d'Applications d'Entreprise & Gestionnaire de Transactions Distribuées pour le framework LEAF

Stage effectué du 23 Juillet au 21 Décembre 2001 dans la division architecture de la société ELCA

TECHNOLOGY CONSULTING INNOVATION

ELCA

ENST
46 rue Barrault
75634 Paris Cedex

http://www.enst.fr

ELCA
Avenue de la Harpe 22-24
1000 Lausanne 13 / Suisse

http://www.elca.ch

Résumé

J’ai effectué mon stage du 23 juillet au 21 décembre 2001 au sein de la société ELCA, dont les bureaux principaux se trouvent à Lausanne. Ces 5 mois ont été divisés en deux parties de durées à peu prés égale durant lesquelles j’ai étudié deux sujets distincts.
La première moitié du stage a porté sur le domaine de l’intégration d’applications d’entreprise (EAI, Enterprise Application Integration). J’avais pour but d’acquérir une meilleure compréhension de la problématique liée à l’EAI, de partager mes découvertes avec un ingénieur d’ELCA pour ainsi décider de l’orientation à donner au développement d’outils d’aide à la réalisation de solutions d’intégration pour les clients d’ELCA. Cela a notamment abouti à la création d’un framework facilitant le développement de «!connecteurs » entre applications.
La suite de mon travail au sein d’ELCA a été consacrée à la conception et au développement d’un module de gestion des transactions distribuées au sein du framework de développement d’applications distribuées d’ELCA, LEAF (Lean Extensible Architecture Framework).

Table des matières

Résumé 2

Table des matières 3

Introduction 5

1.1 Présentation de ELCA Informatique SA 5

1.2 Missions réalisées durant le stage 5

2 Intégration d'applications d'entreprise (EAI) 6

2.1 Problématique 6
2.1.1 L'intégration d'applications d'entreprises 6
2.1.2 Objectifs 7
2.1.3 Travail effectué 7

2.2 Eléments techniques de compréhension de l'EAI 8
2.2.1 Ouverture des applications vers l'extérieur 8
2.2.2 Intégration peer-to-peer, centralisée, en bus 9
2.2.3 Localisation des données 10
2.2.4 XML!: le nouveau Graal 10
2.2.5 Moteur de workflow 11
2.2.6 Communication unidirectionnelle ou bidirectionnelle 12
2.2.7 Modèles synchrones et asynchrones 12
2.2.8 Transactions 13

2.3 Présentation d'un logiciel d'EAI!: WebLogic Application Integration 14
2.3.1 Application Integration 14
2.3.2 Data Integration 15
2.3.3 Business Process Management 15

2.4 La solution xconnector 15
2.4.1 Objectifs de xconnector 16
2.4.2 Principe de fonctionnement 16
2.4.3 Scénarios de déploiement 17

3 Gestionnaire de transactions pour le framework LEAF 19

3.1 Présentation de LEAF 19
3.1.1 LEAF et le modèle EJB 19
3.1.2 LEAF au-delà du modèle EJB 19
3.1.3 Les différents processus exécutés par LEAF 20

3.2 Problématique 21
3.2.1 Nécessité des transactions 21
3.2.2 Transactions distribuées 22

3.3 Présentation des différentes étapes suivies 22
3.3.1 Compréhension de l'existant 22
3.3.2 Recherche de solutions 23
3.3.3 Développement 24
3.3.4 Tests et documentation 24

3.4 Résultats obtenus 24
3.4.1 Le gestionnaire de transactions 24
3.4.2 Choix du niveau d'interopérabilité 25
3.4.3 Invocation du container EJB 25
3.4.4 Invocation du container extension 26

3.4.5 Récupération de transactions en cas d'erreur 27
3.4.6 Tests effectués 30
3.5 Perspectives d'évolution 31
3.5.1 Amélioration du modèle d'invocations du cext. 31
3.5.2 Passage à IIOP 31
4 Conclusions 33
Remerciements 33
Bibliographie 34

Introduction

1.1 Présentation de ELCA Informatique SA

ELCA est une société de service et d'ingénierie en informatique basée à Lausanne en Suisse romande. Créée en 1968 sous le nom de «!Electro-Calcul!», la société a évolué au fil des années d'un métier centré sur l'électronique de contrôle pour l'industrie vers la haute technologie en informatique d'entreprise et Internet.
Ses clients son souvent de grands comptes suisses comme les CFF (Chemins de Fer Fédéraux, l'équivalent de la SNCF), les banques et les assurances. D'autres mandats plus petits sont aussi confiés à ELCA, notamment des sites Web comme récemment celui du Paleo Festival de Nyons.
ELCA connaît depuis une dizaine d'année une croissance rapide, se développant notablement vers la Suisse allemande par l'ouverture de bureaux à Bern et Zürich, mais aussi à Genève et Neuchâtel. Des bureaux de représentation à Londres et Paris ainsi qu'une branche entière située à Hô-Chi-Minh ville contribuent à l'ouverture internationale de la société. Cette rapide expansion a permis d'atteindre un chiffre d'affaire pour l'année 2000 de 42 millions de francs suisses, soit environ186 MF. Le personnel de l'entreprise a lui aussi fortement augmenté, atteignant environ 340 personnes fin 2001.

Le stage que j'ai effectué durant le deuxième semestre 2001 s'est déroulé dans les bureaux d'ELCA à Lausanne, mais mon travail m'a aussi amené à me rendre plusieurs fois à Bern afin de rencontrer d'autres personnes travaillant sur le même projet.

1.2 Missions réalisées durant le stage

Mon stage s'est articulé en deux missions distinctes!: tout d'abord un travail sur l'intégration d'applications d'entreprise puis un travail sur un framework de développement d'applications distribuées propre à ELCA, LEAF.
La première partie du stage m'a conduit à m'intéresser au domaine de l'EAI (Enterprise Application Integration), à étudier de manière globale la problématique de l'intégration d'application, puis faire un tour d'horizon des solutions proposées, évaluer un produit commercial, et enfin développer un prototype de framework de développement de solutions d'intégration en Java. Ce travail m'a permis de couvrir une large partie de la mise en œuvre d'un projet puisque ma tâche a pu commencé très en amont et se poursuivre jusqu'au développement d'une solution.
Cette première partie du stage a été suivie par Bernhard Rytz, chef de projet du bureau de Bern, ce qui m'a donné une première occasion de faire l'expérience du travail «!à distance!». D'autre part, la recherche de cas concrets d'intégration d'applications m'a aussi permis de rentrer en contact avec différentes personnes appartenant à d'autres divisions de la société.

La deuxième moitié de mon stage a été consacrée à l'étude et l'amélioration du framework LEAF (Lean Extensible Architecture Framework) d'ELCA au niveau de la gestion des transactions au sein d'applications distribuées. Pour cela j'ai rejoint l'équipe de développement de LEAF, constituée d'environ 5 personnes sur Lausanne et 3 à Zürich. Le projet LEAF est coordonné par Christian Gasser, mon responsable de stage, et mon travail a été principalement suivi par Philippe Oser travaillant à Zürich. Toutefois mes contacts les plus fréquents ont été avec les trois ingénieurs de Lausanne ayant développé le cœur du framework, ceux-ci m'apportant une aide précieuse dans la compréhension du logiciel.

2 Intégration d’applications d’entreprise (EAI)

2.1 Problématique

2.1.1 L'intégration d'applications d'entreprises

Aujourd'hui une des premières ressources de nombreuses entreprises est, après son personnel, l'information. A tous les niveaux d'une entreprise, des quantités importantes d'informations sont échangées, manipulées, traitées chaque jour. L'amélioration des performances d'une entreprise, tant au niveau financier qu'au niveau de la satisfaction de sa clientèle, passe souvent par une meilleure intégration de ses différents services.
Prenons par exemple une société de vente par correspondance. Les clients passent commande par divers moyens tels que téléphone, Internet, fax, courrier… Déjà à ce niveau il est préférable d'uniformiser le suivi des clients quel que soit leur moyen de commande, on peut alors par exemple attribuer facilement un numéro de client unique quelque soit le type de commande passée, si bien qu'une future commande pourra être passée par un moyen différent sans que le client n'ait à ressaisir ses coordonnées. Une fois la commande passée, l'entreprise doit l'honorer en expédiant le produit demandé. Là encore un ou plusieurs systèmes d'informations interviennent, ceux-ci permettant de gérer les stocks, les commandes aux fournisseurs, l'édition des bordereaux d'envoi… En parallèle la facture doit être éditée et envoyée. Une fois la commande honorée, il est assez courant que le service clientèle s'assure de la satisfaction de la clientèle et, aidé du service marketing, essaie de proposer de nouveaux produits ou services susceptibles de fidéliser le client. De tels systèmes sont baptisés e-CRM pour "electronic Customer Relationship Management". Il est encore possible d’imaginer d’autres systèmes d’informations pour l’analyse financière, le service après-vente…

Intégrer ces différents systèmes entre eux permet alors une meilleure fluidité de l’information au sein de l’entreprise, et élimine les risques d’erreurs liés à la saisie manuelle d’informations.
En fait on peut distinguer deux approches à l’uniformisation de l’information dans l’entreprise. La première, qui était promue encore par certains consultants il y a quelques années, consiste à créer un logiciel essayant de répondre à un maximum des besoins de l’entreprise. On a alors créé des logiciels immenses et complexes, difficiles à gérer et déboguer. Cette approche a certes pu répondre pendant un temps aux besoins de l’entreprise, mais la volonté d’intégrer de plus en plus de systèmes auparavant totalement déconnectés a montré les limites d’une telle approche.
La deuxième approche fait face à cette complexité croissante en conservant les systèmes actuels et en les reliant entre eux de manière à ce qu’ils puissent communiquer données et événements. On réalise alors au passage des économies substantielles à plusieurs niveaux. Les logiciels étant déjà achetés, il n’est pas nécessaire de les changer complètement, même si des mises à jour mineures peuvent être nécessaires. Il n’est pas non plus nécessaire de former à nouveau les personnes habituées au système en place, ce qui est très important dans la mesure où l’apprentissage d’un logiciel peut être assez long et coûteux. Enfin, ne pas se baser sur un logiciel «!à tout faire!» permet de minimiser la dépendance vis à vis du fournisseur de ce logiciel. En effet, la première approche à l’uniformisation du système d’informations de l’entreprise impliquait que le même logiciel soit utilisé dans les différents services de l’entreprise, aussi celle-ci se rendait «!prisonnière!» du fournisseur de la solution (sauf cas de développements internes), si bien que ce dernier pouvait refuser de faire des mises à jour, ou présenter des tarifs exorbitants, ou encore tout simplement disparaître du marché.

2.1.2 Objectifs

ELCA a déjà réalisé plusieurs projets d'intégration de logiciels d'entreprise (EAI) auprès de clients ayant des systèmes d'informations très spécifiques, dans le monde des assurances, de la banque, etc. Toutefois, pour chacun de ces projets, une approche très pragmatique du problème a été prise, si bien qu'en première approche on peut considérer que l'intégration a nécessité une réflexion et un développement original adapté à chaque cas. L'objectif de mon travail était justement d'étudier les opportunités qu'aurait ELCA de systématiser cette intégration. L'idée de départ est motivée par le fait que pour plusieurs projets, on peut constater que des problématiques semblables apparaissent. La transformation de données, la gestion de la sécurité et des transactions sont par exemple souvent présentes dans les projets d'intégration.

En fait les objectifs à atteindre dans le cadre de mon stage étaient assez flous dans la mesure où il fallait d'abord que je comprenne le domaine de l'EAI, puis que j'essaie de proposer, sous l'impulsion de Bernhard Rytz travaillant à Bern, diverses orientations à mon travail. Selon les résultats, cela pouvait aller de décider de ne pas chercher à rationaliser l'approche à l'intégration, jusqu'à développer un logiciel d'intégration, en passant par l'utilisation de logiciels d'intégration du marché.

2.1.3 Travail effectué

Compréhension du domaine, détermination de cas concrets

La première partie de mon travail a consisté à étudier le domaine de l'intégration d'applications d'entreprise. Concrètement cela représente de nombreuses lectures de documentations disponibles sur Internet, d'articles de journaux spécialisés, de «!livres blancs!» (white papers) d'éditeur de produits…
En parallèle de cette activité de documentation, il était nécessaire de recenser les divers projets d'intégration menés par ELCA ces dernières années afin de comprendre les besoins des clients habituels de la société. Cela m'a d'ailleurs permis de contacter et rencontrer plusieurs personnes au sein de la société, afin de les «!interviewer!» sur les projets d'intégration auxquels elles ont participé.
Ces cas concrets ont alors servi de base à la réflexion sur l'orientation à donner à mon travail, dans la mesure où les besoins techniques de ces projets ont déterminé le choix de développer un framework d'intégration décrit plus loin au paragraphe 2.4.

Etude des éléments techniques

Une fois la problématique de l'intégration un peu mieux connue, j'ai pu commencer à m'intéresser aux aspects techniques de la discipline. Le but était de comprendre les besoins logiciels liés à l'intégration. Les résultats de ce travail seront présentés au paragraphe 2.2.
Le but de cette partie était de déterminer quelles sont les technologies que maîtrise actuellement ELCA et qui pourraient être appliquées dans le cadre de l'intégration d'applications d'entreprise. Par exemple de nombreux ingénieurs connaissent Java, aussi une plate-forme d'intégration basée sur Java peut être intéressante car le savoir-faire est déjà très bon au sein de la société. Et, toujours pour donner un exemple de problématique, dans le cadre de la plate-forme Java il est nécessaire de comprendre comment les différentes technologies et spécifications liées à Java permettent d'aider à résoudre les problèmes que présente l'EAI.
Dans cette optique là, il a été fort instructif de réaliser un tour d'horizon du marché des logiciels d'EAI. En effet, ELCA n'est pas la seule entreprise à vouloir améliorer son approche

des problèmes d'intégration, et certaines sociétés en ont d'ailleurs fait leur raison de vivre. Afin de réellement comprendre ce qu'apportent de telles solutions logicielles, j'ai tout particulièrement évalué l'offre de BEA, WebLogic Application Integration. Un rapide tour d'horizon des possibilités de ce logiciel est fait au paragraphe 2.3.

Développement d'un prototype de framework d'intégration

De la partie précédente de mon travail, il est ressorti une meilleure compréhension des problèmes techniques que présente l'EAI. Mais en fait, si les solutions du commerce sont effectivement attrayantes, elles présentent un défaut inhérent dans leurs coûts et dans la non maîtrise de leurs évolutions dans le temps. C'est pourquoi un prototype de framework facilitant le développement de solutions d'intégration a été conçu et développé durant une partie de mon stage. Ce framework sera présenté au paragraphe 2.4.

2.2 Eléments techniques de compréhension de l'EAI

2.2.1 Ouverture des applications vers l'extérieur

Lorsqu'on parle d'intégration d'applications, il est évident que les applications à intégrer doivent être capables de communiquer entre elles par un moyen ou par un autre. Un prérequis est donc que chaque application présente une interface de communication avec le monde extérieur. Il faut en fait préciser que l'on recherche non pas une interface utilisateur mais une interface plus programmatique, que l'on est capable de piloter à partir d'un autre programme.
Le premier type d'une telle interface est l'API!: Application Programming Interface. C'est le terme générique pour désigner un ensemble de fonctions du logiciel appelables depuis un programme écrit dans un langage de programmation spécifique. On parle ainsi souvent d'API C/C++ pour un logiciel par exemple, ce qui signifie que le fournisseur du logiciel propose aux développeurs C/C++ une interface de pilotage de son application par l'intermédiaire d'une librairie utilisable depuis ces langages.
Un autre type d'interface met en jeu la communication par messages. C'est par exemple le cas de certains logiciels qui répondent à des commandes envoyées sur des connexions TCP/IP. C'est en fait le modèle sous-jacent d'une technologie très «!en vogue!» en ce moment!: SOAP (Simple Object Access Protocol). Ce protocole permet d'envoyer des appels de procédures à distance au travers d'internet en se basant sur des messages XML standardisés, transportés sur http, qui lui même utilise TCP. Plus anciens et plus proches du système, les RPC d'Unix par exemple sont aussi un moyen pour une application de présenter une interface utilisable de manière programmatique. Enfin, souvent basés sur les sockets TCP, les systèmes de queue de messages permettent à une application de recevoir des ordres d'autres processus de manière asynchrone (cf §2.2.7). Dans la plupart des cas, les systèmes basés sur une communication par TCP ont l'avantage d'être indépendants du système d'exploitation et du langage dans lequel l'application pilote est écrite. Avec SOAP, une application écrite en C sur Unix peut par exemple transmettre des requêtes à un service SOAP écrit en Java et exécuté sur Windows.

Si aucune interface n'est clairement proposée pour utiliser une application de manière automatisée, il peut être parfois nécessaire de recourir à des astuces spécifiques. De très vieilles applications par exemple ne connaissent que le pilotage par une interface utilisateur en mode texte. C'est le cas sur de vieux terminaux de type IBM ou autre. Il est alors nécessaire de simuler les actions que ferait un utilisateur sur ce terminal afin de faire exécuter des commandes par l'application. Tout un travail de reconnaissance des écrans (parsing de la sortie du terminal) et d'envoi des caractères «!tapés!» permet ce genre de manipulations. Le problème est que si une mise à jour de l'application entraîne le changement de quelques

caractères dans un écran, il est nécessaire de mettre aussi à jour le module d'émulation de terminal pour l'adapter. Ce n'est donc clairement pas une solution idéale, mais elle est parfois nécessaire si l'on ne peut pas se passer d'une application.
D'autres applications présentent la particularité de travailler en «!batch!», par exemple en scrutant régulièrement la présence de fichiers à traiter dans un répertoire donné. L'interface d'un tel processus batch peut alors consister à lui soumettre son travail sur demande d'une autre application en copiant un fichier dans le répertoire cible.

2.2.2 Intégration peer-to-peer, centralisée, en bus

Une fois qu'on est donc capable d'interfacer une application avec «!le monde extérieur!», il faut la relier aux autres applications. Pour cela plusieurs topologies existent, analogues aux topologies d'un réseau de machines!: en étoile, en bus, fortement connexe…

Topologie fortement connexe

Avoir une topologie fortement connexe signifie que chaque application est reliée à toutes les autres. On a donc un pont entre chaque paire d'applications, portant à $C_n^2 = \frac{n(n-1)}{2}$ le nombre de liens à développer pour n applications. Si un grand nombre d'applications est en jeu, on voit donc que la complexité de l'intégration est trop contraignante puisque dans le cas d'une modification d'une des applications, il faudra mettre à jour n-1 des «!ponts!».

Topologie en étoile

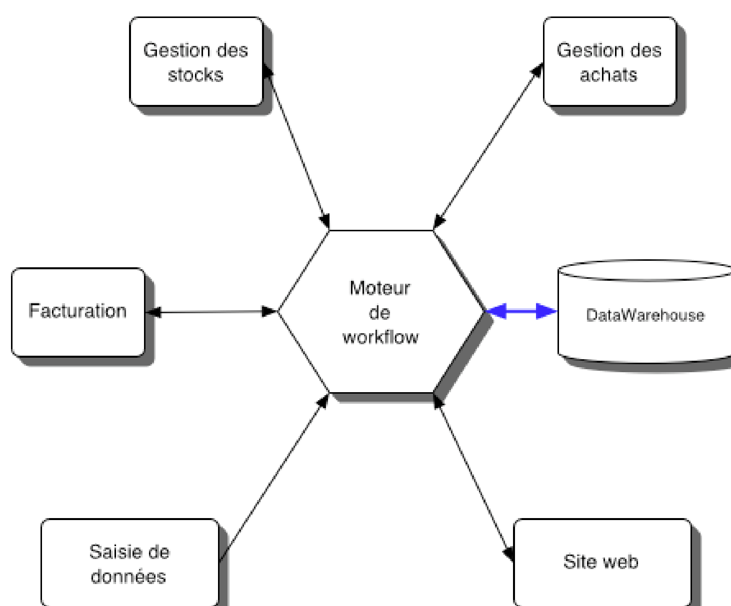
Une topologie en étoile permet de réduire le nombre de connections à effectuer entre les applications, rendant le système beaucoup plus gérable.
La topologie en étoile place souvent en son centre un serveur d'applications associé à un moteur de workflow (cf. §2.2.5). Il est alors plus aisé de construire des connections entre chaque application et le centre, et les applications se retrouvent plus indépendantes puisqu'elles n'ont pas à connaître la présence des autres applications pour fonctionner. Le serveur central de la topologie en étoile permet aussi d'effectuer des transformations sur les données qui transitent par lui.
Une application de saisie de documents par exemple enverra les documents saisis au serveur central, celui-ci aura été programmé pour le traiter (envoi à une autre application, stockage dans une base de données…), mais l'application de saisie n'a pas besoin de connaître l'application qui traitera ensuite les documents envoyés.

Topologie en bus

Tout comme la topologie en étoile, la topologie en bus réduit le nombre de liens à développer pour faire communiquer toutes les applications entre elles. Sur un bus, les applications envoient des messages (ou des requêtes) à des destinataires spécifiés (sauf en cas de diffusion). Chaque application (ou du moins le module de liaison qui la relie au bus) doit donc savoir à qui elle doit envoyer ses messages sur le bus. On a donc un modèle moins centralisé mais qui du coup implique plus de complexité puisque le changement d'une application peut nécessiter de remettre à jour la manière dont les autres applications lui envoyaient des messages. De plus, comme on le verra au paragraphe suivant, le bus se prête moins bien à la transformation des données des messages qui y circulent.
Un exemple courant de bus est celui matérialisé par CORBA (Common Object Request Broker Architecture) qui permet l'invocations d'objets à distance.

2.2.3 Localisation des données

Une fois résolus les problèmes de communication pure entre les différentes applications, se pose la question cruciale des données. En effet, avant l'intégration on dispose de plusieurs systèmes d'information gérant chacun leurs propres données puisqu'ils n'ont pas forcément été écrits dans l'optique de coopérer avec d'autres applications. Deux possibilités se présentent alors!: soit chacune des applications garde ses données, et celles-ci doivent être répliquées et synchronisées entre les différentes applications. Soit un dispositif central est chargé de gérer les données, on parle dans ce cas d'entrepôt de données, ou datawarehouse.
La première solution est assez difficile à mettre en œuvre dans la mesure où la synchronisation des différentes données est coûteuse en termes de performance machine et réseau. De plus, il est courant que certaines applications n'aient besoin que d'une partie seulement des données. Donc si le modèle de données décrivant par exemple un compte client a beaucoup d'attributs, chaque application n'a besoin que d'une partie seulement de ces attributs.
La deuxième solution est la plus employée dans le cadre de l'intégration d'applications d'entreprise. Elle est d'ailleurs généralement inscrite dans une topologie en étoile, car cela s'y prête très bien. Le schéma suivant montre un exemple d'une telle architecture!:



Ici les applications envoient les données au moteur de workflow qui se charge de les stocker dans le datawarehouse. Si une application a besoin de données (par exemple le site web pour afficher à un client l'état d'une commande), elle les demande au moteur de workflow qui les récupère dans le datawarehouse. Grâce à cette architecture les données sont facilement transformable par le moteur de workflow ou une couche logicielle entre celui-ci et l'application cible.

2.2.4 XML!: le nouveau Graal

Outre la localisation des données qu'on vient de voir, la transformation de celles-ci est très importante pour la bonne intégration des applications. Les applications gèrent en effet les données sous un format particulier, souvent propriétaire. Lorsque les données sont échangées entre les applications, il faut donc être capable de les traduire d'un format dans un autre. Une application peut par exemple stocker ses données en format texte séparant les enregistrements avec des retours chariot, et une autre stocker ses données en écrivant directement les

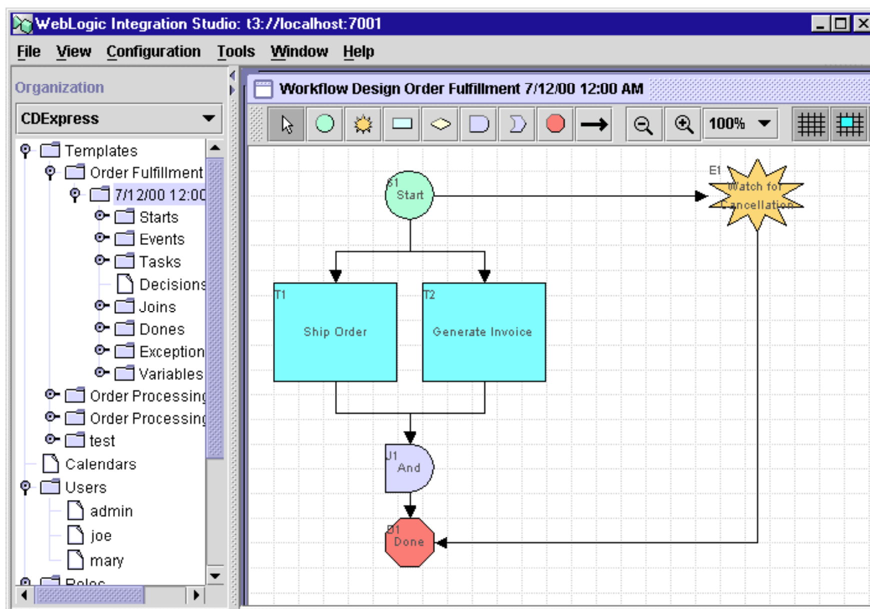
structures C dans un fichier, introduisant une dépendance vis à vis de la plate-forme matérielle sous-jacente (big ou little endian, entiers sur 2 ou 4 octets…).
Afin de faciliter l'écriture de filtres convertisseurs de données, il existe des moteurs de transformation de données qui, à partir des données initiales et de règles de transformation, génère les données sous un autre format. Le travail de l'intégrateur d'applications, en ce qui concerne les données, reviendra donc à l'écriture de ces règles de transformation.

Mais là encore on se retrouve devant le même problème que pour le choix de la topologie!: soit on crée un convertisseur capable de convertir n'importe quel format en n'importe quel autre, soit on adopte un format intermédiaire servant de pivot à chacune des transformations. Ce format intermédiaire doit alors être suffisamment riche pour ne pas écarter d'informations d'un des formats propriétaires. Actuellement plusieurs produits d'intégration (dont WLAI présenté au § 2.3) ont choisi le format XML (eXtensible Markup Language) pour jouer ce rôle.
XML permet de stocker sous forme arborescente n'importe quel type de données, le tout en ayant l'apparence d'un fichier texte. D'éventuelles données purement binaires (comme une image) doivent d'abord être transformées pour être insérées dans un fichier au format XML (l'algorithme base64 est souvent utilisé pour cela).
L'intérêt d'XML ici est qu'un grand nombre d'outils existent pour faciliter sa lecture, son écriture et sa transformation car XML est standardisé par l'organisme W3C qui gère déjà le standard HTML du Web.

XML permet donc une plus grande facilité pour le développeur, mais il faut néanmoins savoir que cela a un coût en termes de performances. En effet, la lecture (parsing) et la sérialisation (écriture) d'un fichier XML demande un temps processeur non négligeable. Et bien que le travail sur un arbre XML une fois en mémoire (on parle alors d'arbre DOM pour Document Object Model) soit facilité car le parcours de l'arborescence est simple à programmer, cela reste très coûteux en espace mémoire. Le traitement de messages XML de plusieurs mega-octets n'est aujourd'hui envisageable qu'à condition d'avoir un matériel assez puissant. Et le passage à l'échelle (en cas d'augmentation du trafic d'un site web utilisé par les clients par exemple) sera alors problématique.

2.2.5 Moteur de workflow

Le terme de moteur de workflow (workflow engine en anglais) a été plusieurs fois évoqué précédemment sans toutefois expliquer de quoi il s'agissait. Un moteur de workflow permet de décrire un enchaînement de tâches à exécuter de manière à accomplir un processus métier. Les tâches et leur enchaînement sont décrites de manière visuelle, puis une fois validée le moteur effectuera les traitements adéquats lors de l'arrivée de certains messages ou à des moments précis de la journée.
Prenons l'exemple d'une commande qui est remplie sur le site web puis transmise au moteur de workflow. Celui contient une suite d'actions à effectuer en cas d'arrivée d'une commande. Il faut par exemple générer la facture et expédier le produit commandé. Ces deux tâches peuvent être faites simultanément dans le cas d'un paiement à la livraison par exemple. Un tel processus métier peut-être facilement modélisé dans le moteur de workflow comme le montre la capture d'écran suivante (extraite de WLAI)!:



On peut voir ici que le processus métier ne sera considéré comme terminé que si les tâches d'envoi de la commande et de facturation sont terminées. Sur la partie droite, un événement «!asynchrone!» est prévu pour pouvoir interrompre le processus métier en cas d'annulation de la commande au cours de son traitement.
Le rôle du moteur de workflow dans l'intégration d'applications est de gérer les tâches définies (les rectangles bleus dans l'image ci-dessus), celles-ci faisant appel à des applications externes, ici peut-être la gestion des stocks (pour notifier du déstockage d'un produit) et la facturation. Et en basant l'ensemble des informations sur un format commun comme XML, le moteur de workflow peut lui-même avoir accès aux données afin d'influencer ses décisions.

2.2.6 Communication unidirectionnelle ou bidirectionnelle

Lors de l'élaboration d'une solution d'intégration, il convient d'analyser le sens des flux de données ou commandes entre les applications. Ainsi certaines applications ne peuvent que recevoir des ordres et données (pour affichage par exemple), d'autres qu'en émettre (application de saisie), et d'autres peuvent faire les deux.
Cela est assez important dans la mesure ou il arrive parfois que les interfaces d'entrée soient totalement décorrélées des interfaces de sortie. Une application ancienne peut par exemple avoir une émulation de terminal pour entrée et la génération de fichiers pour sortie. Il convient alors que le moteur de workflow ou le bus reliant les applications soit capable de gérer cette asymétrie des interfaces.
Dans une topologie en étoile, le «!hub!» lui-même (basé sur un moteur de workflow par exemple) peut présenter une asymétrie dans ses interfaces d'entrée-sortie. Par exemple une solution basée sur un serveur EJB (Enterprise Java Beans) permet de multiples possibilités de sortie, mais peu d'entrée (les invocations se font soit par RMI-IIOP soit sous forme de messages asynchrones JMS).

2.2.7 Modèles synchrones et asynchrones

La communication entre les applications se fait, comme on l'a vu précédemment, par l'envoi de messages ou l'invocation de méthodes au travers des interfaces que les applications présentent au monde extérieur. Mais selon le type de communication envisagé, on aura affaire soit à un modèle synchrone soit à un modèle asynchrone. Dans le premier une application qui envoie un message ou une commande restera bloquée jusqu'à ce qu'une réponse lui soit

donnée. Dans le deuxième, l'application envoie son message ou sa requête, et continue son exécution immédiatement après sans attendre de réponse. Si une réponse doit arriver, elle arrivera en différé par un autre canal.
Prenons quelques exemples de ces différentes types de communication. L'envoi d'une requête http, une invocation de RPC ou encore l'exécution d'une fonction d'une librairie native sont des communications synchrones. Par contre l'envoi d'un message dans une queue de messages ou l'écriture d'un fichier dans un répertoire scruté par une autre application correspondent à des émissions asynchrones.

Le choix entre un mode ou un autre est souvent déterminé par les capacités des applications à intégrer. La plupart du temps, seul le mode synchrone est géré car c'est le plus facile à implémenter. Toutefois il peut poser des problèmes de passage à l'échelle car en cas de surcharge d'un des liens, cela peut causer le ralentissement des autres applications. Et si l'application cible est en panne, cela va bloquer les applications qui lui envoient des messages.
Le mode asynchrone semble donc plus approprié dans le cadre de l'EAI. Le principal outil pour implémenter l'asynchronisme est le serveur de queue de messages. Dans ce domaine il est nécessaire de se baser sur un produit unique car il n'y a pas de normes permettant d'utiliser plusieurs serveurs de messagerie asynchrone dialoguant entre eux. Sur plate-forme Java, la spécification JMS (Java Message Service) standardise l'API d'utilisation du serveur par les clients, mais ne propose pas encore de standard pour l'interopérabilité des serveurs. Néanmoins, il est possible grâce à un tel serveur de faire communiquer les applications entre elles en leur proposant de s'envoyer des messages de manière sûre et asynchrone.

Lié au modèle asynchrone, le mode de signalement des événements est important dans une architecture d'intégration. En effet, si l'envoi de messages sur une queue est assez direct, la réception des messages peut se passer de deux manières différentes!: soit l'application interroge régulièrement la queue pour voir si il n'y a pas de nouveaux messages pour elle, soit le serveur de messagerie signale directement un événement à l'application cible.
Dans le premier cas on parle de «!polling!» de la part de l'application. Cela consomme nécessairement du temps CPU et éventuellement des ressources réseau. On peut paramétrer l'intervalle d'interrogation afin d'obtenir les performances voulues et le temps de réaction moyen à l'arrivée d'un message. Le cas d'une application qui scrute régulièrement un répertoire du système de fichiers est aussi un exemple de polling même si il n'y a pas de queue de message en jeu.
Pour l'autre mode de réception de messages, on parle de «!pushing!» parce que le serveur de messagerie pousse littéralement le message vers l'application cible. Dans cette catégorie on peut distinguer deux types de push!: le callback et l'attente bloquante. Avec le premier, l'application donne les moyens au serveur de la rappeler lorsque des messages arrivent à son intention. Le serveur va par exemple utiliser l'interface de pilotage de l'application. Dans le deuxième cas, l'application demande au serveur de recevoir un message et reste bloquée jusqu'à ce qu'un message soit effectivement disponible. Dés qu'il arrive l'application est débloquée et obtient donc son message sans délai par rapport au polling. En réalité on ne bloque pas toute l'application mais seulement un thread de l'application afin qu'elle puisse toujours continuer son travail normal.

2.2.8 Transactions

La communication de données entre des applications n'est pas toujours suffisante pour réaliser l'intégration de systèmes d'information. Il faut en effet pouvoir garantir que certains échanges se fassent de manière transactionnelle. Si un client passe commande, il faut par

exemple que l'ensemble des actions de facturation, envoi, débit de carte bancaire se fassent de manière atomique. Sans quoi le client pourrait par exemple être facturé sans que le produit ne soit jamais expédié. On doit donc résoudre des problèmes de transactions distribuées sur lesquels la deuxième partie de mon stage a d'ailleurs porté.
En fait le problème des transactions est relié au mode de communication des applications entre elles. En effet, dans le cas d'une communication synchrone, il est beaucoup plus facile d'assurer le bon déroulement d'une transaction que dans un modèle asynchrone. Dans le cas synchrone, le client qui passe commande sur le Web sera par exemple en attente jusqu'à ce que les systèmes de facturation et de gestion des envois aient enregistré de manière persistante la commande. Si par exemple le système de facturation refuse la transaction car le numéro de carte bancaire est invalide, alors le système de gestion des envois ne prendra pas en compte la commande et le client sera immédiatement notifié de l'échec de sa commande.
Il existe toutefois des transactions qui peuvent durer très longtemps, des heures ou même des jours. Dans ces cas là, il est rare que la communication entre les applications se fasse de manière synchrone car cela voudrait dire qu'elles restent bloquées (ou au moins un thread) jusqu'à ce que la transaction se termine. On doit donc disposer d'un gestionnaire de transactions assez évolué, capable de gérer des transactions très longues.

2.3 Présentation d'un logiciel d'EAI!: WebLogic Application Integration

Afin d'avoir un meilleur aperçu de ce que représentent les éléments «!théoriques!» présentés au paragraphe 2.2, voici un bref aperçu des possibilités qu'offre WebLogic Application Integration, logiciel d'intégration que j'ai particulièrement évalué durant mon stage.
WLAI est un logiciel d'intégration basé sur le serveur d'applications EJB de BEA, WebLogic Server, et sur un moteur de workflow. La réalisation d'un projet d'intégration avec WLAI correspond donc à une topologie en étoile comme expliqué précédemment. WLAI se compose essentiellement de 4 parties!: Application Integration, Data Integration, Business Process Management et B2B Integration. Ce dernier module traite plus particulièrement de l'intégration d'applications inter-entreprises et je n'ai pas eu l'opportunité de l'étudier en profondeur. C'est pourquoi je ne le présenterai pas ici.

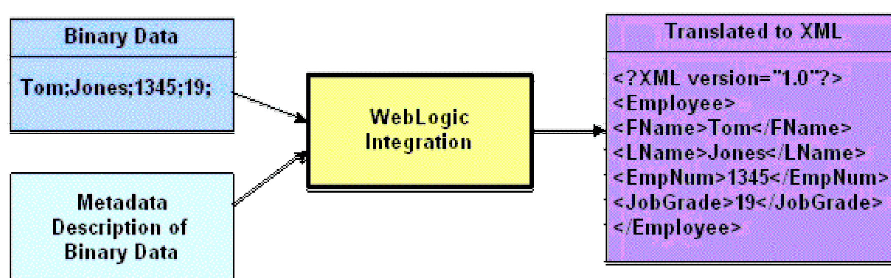
2.3.1 Application Integration

Cette partie de WLAI est celle qui permet justement la communication entre les différentes applications et le moteur de workflow. Elle se base essentiellement sur la spécification JCA (Java Connector Architecture) de Sun. Cette spécification permet à des composants de type EJB de s'interfacer avec des applications extérieures. Mais dans sa version actuelle, JCA est assez asymétrique dans la mesure où les composants EJB peuvent initier une communication avec des applications extérieures mais pas l'inverse. Cela signifie qu'un bean peut par exemple interroger une application, mais qu'une application ne peut normalement pas signaler à un bean qu'un événement vient de se produire.
Pour pallier à cette limitation de la spécification JCA, BEA a créé au dessus de JCA et de son serveur WebLogic un module de réception d'événements basés sur des messages asynchrones. Pour cela le serveur JMS (Java Message Service) de BEA est utilisé pour fournir l'infrastructure de messagerie.

Un kit de développement d'adaptateurs (ADK, Adapter Development Kit) est fourni pour que les développeurs créent des modules d'interfaçage avec des logiciels extérieurs. En fait un marché est en train de se créer autour de ces adaptateurs, et certaines sociétés proposent déjà des modules pour se connecter à des systèmes comme SAP, PeopleSoft, CICS, Siebel…

2.3.2 Data Integration

Ce module de WLAI permet de créer des règles de transformations de données depuis et vers le format XML. Pour cela le développeur crée un fichier décrivant les données à traduire (metadata file), et la correspondance en un format basé sur XML. La figure suivante montre un exemple de transformation depuis un format propriétaire vers XML.



2.3.3 Business Process Management

Le module BPM correspond en fait au moteur de workflow de WLAI. C'est là que les tâches à effectuer sont modelées graphiquement. Ce module est en fait très dépendant des deux précédents puisque c'est grâce au module Application Integration que le moteur de workflow peut recevoir des messages à traiter (événements) et adresser des messages aux applications extérieures. En fait les messages qui arrivent au niveau du moteur de workflow sont en XML, si bien que l'enchaînement des tâches peut être déterminé par le contenu des messages à traiter. Grâce à la spécification Xpath, on peut facilement accéder à un nœud de l'arbre XML, pour faire des tests conditionnels par exemple.
Lorsqu'on veut ensuite envoyer un message à une application extérieure, on le fait sous forme XML, qui sera éventuellement transformée en un format plus compréhensible par l'application cible.
C'est aussi le module BPM qui gère les transactions longues, sous la forme de liste de tâches à exécuter, et qu'un opérateur manuel peut aussi consulter afin de vérifier l'avancement.

2.4 La solution xconnector

L'étude des généralités du domaine de l'EAI a permis de mieux comprendre ce domaine, et de le mettre en relation avec les ambitions d'ELCA vis à vis de ses clients. En fait, si les possibilités offertes par des logiciels spécialisés dans l'intégration comme WLAI sont alléchantes, elles posent le problème de la répercussion du coût de la licence sur le client. En effet, ces logiciels sont proposés à des tarifs très élevés (aucun tarif officiel n'est disponible car cela dépend des clients, mais on parle par exemple de plusieurs dizaines de milliers de dollars pour MQ Series Integrator d'IBM), si bien qu'il peut devenir difficile de proposer des solutions à des coûts compétitifs aux clients.
C'est pour cela que sous l'impulsion de Bernhard Rytz j'ai concentré mon attention vers le projet open-source OpenAdaptor qui permet de développer des solutions d'intégration à moindre frais. En fait ce qui nous intéressait dans ce projet étaient les aspects techniques et l'évolutivité de l'architecture. De nombreux enseignements ont pu en être tiré, mais nous en sommes en fait arrivé à la conclusion qu'il était préférable d'essayer de développer notre propre framework d'intégration répondant à certains cas de figures recensés parmi les mandats déjà effectués, tout en sachant que cette solution serait de toutes manières moins complète que celle proposée dans des produits spécialisés.

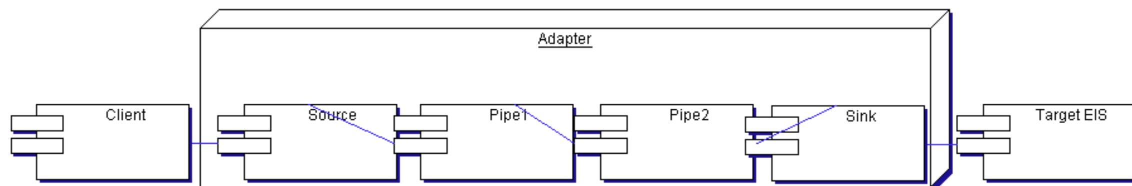
J'ai donc conçu et développé un prototype de framework d'intégration d'applications d'entreprise que l'on a baptisé «!xconnector!».

2.4.1 Objectifs de xconnector

En termes de fonctionnalités, xconnector n'ambitionne certainement pas de rivaliser avec un WLAI ou un MQSI, mais plutôt de répondre à des besoins simples d'intégration entre quelques applications.
Il ne dispose pas de moteur de workflow, mais permet par contre de créer des «!ponts!» entre une application et un tel moteur. On peut aussi créer des «!ponts!» entre deux applications directement.

2.4.2 Principe de fonctionnement

Xconnector n'est pas un logiciel à part entière mais plutôt un framework permettant le développement de solutions d'intégration. Il est orienté vers la mise en relation des applications plus que vers la gestion des processus métiers. Pour ce faire, il propose une architecture de pipeline à travers lequel des messages peuvent transiter et être traités. La figure suivante montre un tel pipeline.



Le pipeline commence par un composant appelé «!source!» qui est chargé de faire l'interface entre une application cliente et le pipeline. Un exemple de composant source serait un serveur http, un scruteur de répertoire, ou une queue de messages asynchrones.
Viennent ensuite plusieurs «!pipes!» ou tuyaux. Ceux-ci permettent d'appliquer des traitements sur les messages, comme la transformation des données, la vérification d'éléments de sécurité…
Au bout du pipeline, on trouve le «!sink!» ou robinet par lequel sortent les messages à destination de l'application cible. Ce sink est le pendant de la source et sait donc communiquer avec l'interface native de l'application.

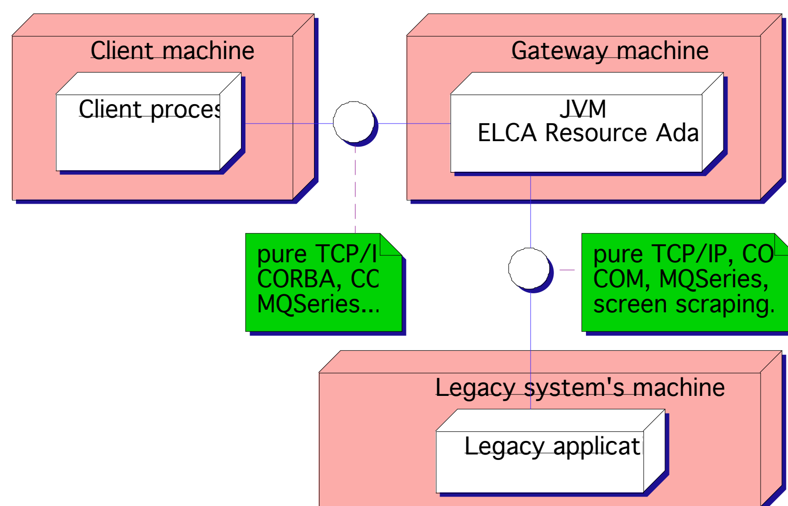
Le type de données qui transite à travers le pipeline est en fait indépendant de l'architecture du framework. Concrètement les pipes se passent un objet entre eux, mais à chaque niveau le format des données dans l'objet peut être différent. En fait un format intéressant est le format XML puisqu'il est standardisé. Pour profiter de XML, il faut donc que le composant source interprète les données de l'application source et les transforme en XML. Ensuite l'arbre DOM est transmis de pipe en pipe. Il existe un type de pipe qui est par exemple chargé d'appliquer une transformation sur l'arbre DOM à partir d'un fichier XSL. Ce genre de transformation est assez courante et très pratique puisqu'il existe des moteurs de transformation disponible gratuitement comme Xerces d'Apache.

Au niveau des transactions, un petit gestionnaire de transaction est présent, afin de gérer la démarcation des transactions lors de l'arrivée d'un message. En fait celui-ci a permis d'expérimenter des transactions au niveau web services en créant des messages SOAP démarquant les transactions. Le problème des transactions pour les web services est d'ailleurs encore loin d'être réglé, et les expériences faites avec ce prototype ont tout juste permis de

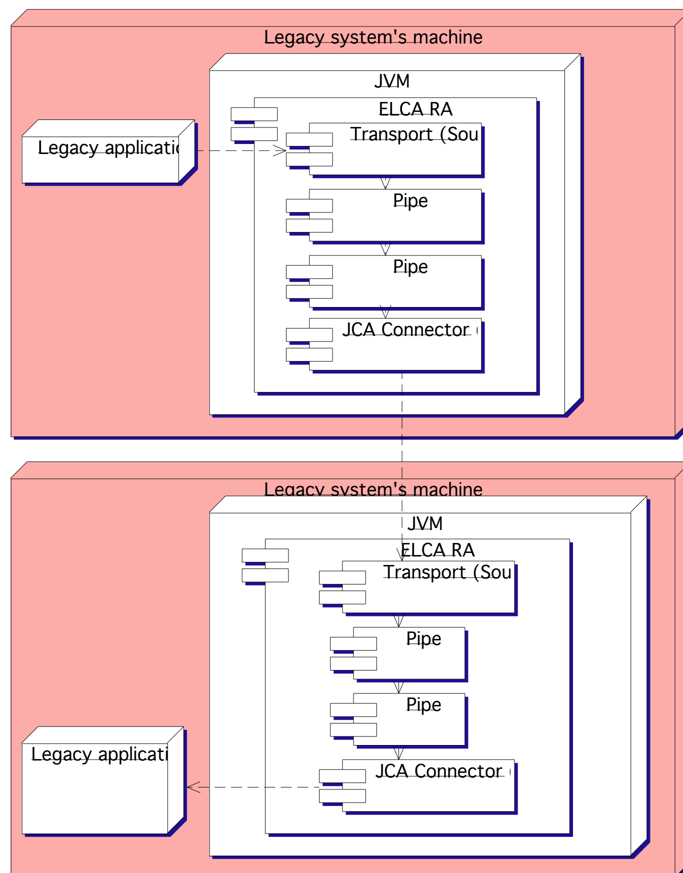
mettre le doigt sur les problèmes à ce niveau, et il est probable que le W3C travaille sur une spécification de web services transactionnels.

2.4.3 Scénarios de déploiement

Lors de la conception d'une solution d'intégration d'applications, il est important de savoir ce que l'on peut faire ou pas pour faire communiquer les applications entre elles. En effet, selon le type de machine auquel on a affaire, il n'est parfois pas possible ou du moins difficile ou pénalisant d'y ajouter une brique logicielle trop grosse. Par exemple xconnector est écrit en java, si bien qu'il est nécessaire de disposer d'une machine virtuelle pour exécuter une instance d'un pipeline. Donc si l'on doit intégrer une application tournant sur une machine ancienne pour laquelle aucune machine virtuelle n'existe, il faut développer un module d'interfaçage spécifique qui communiquera avec le pipeline situé sur une machine différente comme illustré sur le diagramme suivant!:



Par contre pour interfacer deux applications exécutées sur des machines distinctes mais supportant une machine virtuelle, on peut installer une instance de xconnector sur chacune des machines!:



En fait le nombre de scénarios possibles est assez grand et il serait inutile de tous les exposer ici. On aura néanmoins pu apercevoir que dans le domaine de l'intégration il faut souvent agir au cas par cas, et qu'un développement spécifique est toujours nécessaire. Le prototype de framework qu'est xconnector permet d'aider à la construction d'une solution d'intégration.
ELCA n'ayant pas de projet d'intégration immédiatement applicable, il était difficile de continuer le développement de xconnector sans avoir de repères en termes de besoins des clients. C'est pourquoi on m'a proposé de m'intéresser à un autre sujet, les transactions dans le framework LEAF. Même si un petit gestionnaire de transactions a été fait pour xconnector, le domaine des transactions m'était assez étranger et je désirais le connaître mieux.

3 Gestionnaire de transactions pour le framework LEAF

3.1 Présentation de LEAF

LEAF est l'acronyme de Lean Extensible Architecture Framework. Il s'agit d'un framework de développement d'applications distribuées pour la plate-forme Java. LEAF utilise la plate-forme J2EE (Java 2 Enterprise Edition) de Sun en la complétant et en introduisant des abstractions du modèle de composants utilisé par J2EE, particulièrement le modèle EJB (Enterprise Java Beans).

3.1.1 LEAF et le modèle EJB

Les composants EJB sont des composants logiciels hébergés par un logiciel serveur d'applications fournissant à ces composants les services nécessaires aux applications d'entreprise!: gestion du multitâche, des sessions, de la persistance, de la sécurité, des transactions… Le développement d'applications serveur est ainsi facilité et le développeur peut se concentrer sur les spécificités du service qu'il veut fournir au travers de ce composant.
Il existe sur le marché plusieurs serveurs d'applications capables d'héberger ces EJB, de tels serveurs sont appelés des containers EJB. Les noms de serveur les plus connus sont WebLogic de BEA, WebSphere d'IBM, iPlanet de Sun. Quelques produits «!open-source!» existent aussi, jBoss étant sans doute le plus abouti. Tous ces serveurs doivent suivre les spécifications EJB émises par Sun, mais chacun essaie de se différencier en apportant des améliorations et en comblant les lacunes des modèles EJB et J2EE.
C'est à ce niveau que LEAF apporte un premier bénéfice, en créant une abstraction des différences entre les serveurs. Ainsi, des composants basés sur LEAF pourront être facilement déployés dans des containers EJB différents avec un minimum d'effort. Pour cela LEAF redéfinit la notion de composant telle qu'introduite dans la spécification EJB, en créant des «!beans LEAF!» qu'on peut considérer dans une première approche comme des composants encapsulés dans des Enterprise Java Beans (EJB). Développer une application pour LEAF consiste donc à développer des composants utilisant l'architecture de LEAF, ces composants seront alors transformés pour être vus comme des EJBs, pouvant être hébergés par un container EJB.

3.1.2 LEAF au-delà du modèle EJB

Le modèle EJB tel qu'il se présente dans sa version actuelle, la plus récente fin 2001 étant la 2.0, prévoit 3 types de composants EJB!: les stateless beans (sans état)!; les statefull beans (avec état) et les entity beans (composants persistés dans une base de données). Ces composants sont des objets possédant des méthodes qui sont invocables par des clients extérieurs au serveur d'application ou par d'autres composants de type EJB. Le modèle d'invocation imposé par la spécification EJB est un modèle synchrone, ce qui entraîne certaines restrictions dans le comportement des EJB.
Par exemple, un EJB ne peut pas effectuer un travail en tâche de fond suite à une requête d'un client puisque le client est bloqué durant son invocation d'une méthode de l'EJB. Il n'est pas possible non plus pour l'EJB de créer des threads afin de faire un tel traitement en tâche de fond après être retourné de l'invocation du client. Le modèle EJB se révèle donc totalement inadapté à des traitements de type «!batch!», pouvant durer des heures.
LEAF propose de remédier à cette limitation en fournissant un container abritant des composants développés au-dessus de LEAF, ceux-ci ayant le droit de faire des traitements en

tâche de fond, de créer des threads… Dans la terminologie de LEAF, un tel container est appelé le cext pour container extension, puisqu'il étend en quelque sorte le container EJB.

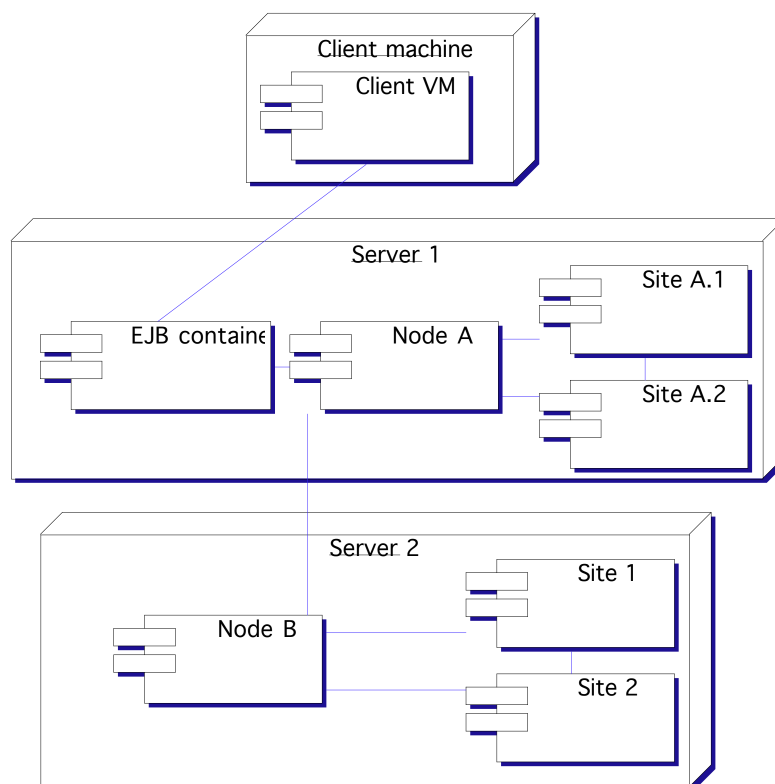
Un des aspects remarquables du cext de LEAF est le fait qu'il soit potentiellement réparti en plusieurs processus tournant sur une ou plusieurs machines. Ce système permet à l'heure actuelle d'obtenir une certaine tolérance aux pannes dans le système distribué. Une extension future lui permettra aussi de répartir la charge entre les différents processus qui composent le cext.

Enfin, on peut rapidement décrire quelques fonctionnalités de LEAF qui lui permettent de justifier de sa valeur ajoutée par rapport au modèle EJB. LEAF est très extensible, dans la mesure où il est possible de rajouter des couches logicielles par lesquelles chaque invocation de composant de LEAF (service LEAF) passera avant d'atteindre le composant cible. Ces couches logicielles sont organisées en une chaîne d'!«!invokers!» s'appelant les uns à la suite des autres. De tels mécanismes d'extensibilité permettent ainsi de fournir des services de mesure des performances (mesure de la durée des invocations), de sécurité (vérification de l'authenticité de l'appelant et de ses autorisations sur l'objet cible), de traçage des invocations, de transaction (ce sur quoi mon travail a porté)…

3.1.3 Les différents processus exécutés par LEAF

La description faite de LEAF jusqu'ici s'attachait à décrire de manière abstraite le positionnement de LEAF vis à vis du modèle EJB, et à présenter le modèle de composants adopté dans ce framework. Toutefois, il me semble intéressant de descendre plus en détail dans le fonctionnement de LEAF afin d'avoir une idée de son architecture relativement aux divers processus mis en jeu vis à vis du système d'exploitation.
Il faut tout d'abord rappeler que LEAF est un framework écrit en langage Java et destiné à développer des applications distribuées en Java. Cela entraîne donc que chaque processus de LEAF est en fait une machine virtuelle Java. Un client accédant aux composants de LEAF est aussi potentiellement une machine virtuelle à part entière. Le container EJB sur lequel se base LEAF est aussi exécuté au sein d'une machine virtuelle. On omettra donc dorénavant de préciser qu'un processus au sens système d'exploitation est en fait une machine virtuelle, et on ne parlera donc plus que de machine virtuelle ou VM.

Dans ce système distribué matérialisé par LEAF, chaque VM comporte une instance de LEAF. Un client LEAF est en lui-même une instance de LEAF, puisque afin d'être capable de faire des invocations sur un container, le client a besoin d'une partie de la logique de LEAF. Le container EJB déploie aussi une instance de LEAF, puisque les beans LEAF sont encapsulés dans des beans EJB, l'encapsulation étant le fait du framework LEAF. Enfin, le container extension est lui-même matérialisé par plusieurs VM. En effet, afin de supporter la tolérance aux pannes, deux types d'instance de LEAF pour le cext sont définis!: les Node et les Site. Un Node est une instance de LEAF déployée dans sa propre VM et permet de créer des Sites sur la même machine physique que lui même. Pour une seule machine physique, il n'existe à tout moment qu'un seul Node, et potentiellement plusieurs Sites. Chaque Site est en fait une instance de LEAF exécutée dans sa propre VM. Le système de tolérance aux pannes est tel que chaque Node surveille les Sites qui lui sont associés, et si l'un d'eux vient à ne plus répondre, le Node relancera un nouveau Site.
Afin de clarifier la description précédente, on peut schématiser un ensemble de VM déployées sur plusieurs machines.



Comme on peut le voir sur ce schéma, les différentes instances de LEAF, donc les différentes machines virtuelles, communiquent entre elles, faisant des invocations distantes. Le client appelle par exemple une méthode sur un bean hébergé dans le container EJB. Ce bean peut à son tour utiliser des services déployé dans le cext, lançant par exemple des processus batch…

3.2 Problématique

La précédente description de LEAF, et notamment de la pluralité des processus en jeu, permet maintenant d'introduire la problématique réelle de cette deuxième moitié de stage.

3.2.1 Nécessité des transactions

Comme on peut le voir sur le schéma précédent, différentes instances de LEAF communiquent entre elles, s'envoyant des invocations de méthodes sur des objets distants. Or, dans un système d'information, la notion de transaction est très importante puisqu'elle permet de garantir la consistance des informations. Dans un système client/serveur, le client peut manipuler des données par l'intermédiaire du serveur, et il est possible d'effectuer des modifications sur ces données. Dans le cadre d'un processus métier, il peut être nécessaire d'effectuer plusieurs modifications séquentiellement, mais il est impératif de réaliser l'ensemble des modifications de manière «!atomique!». Les transactions sont justement faites pour garantir les propriétés suivantes aux manipulations du système d'information!: Atomicité, Consistance, Indépendance, Durabilité. On parle alors des propriétés ACID d'une transaction.
L'atomicité garantit que la mise à jour des données se fait en une unité indivisible de travail, c'est à dire qu'aucune autre modification de ces données ne sera faite durant l'accès à ces données.
La consistance d'une transaction signifie que lors d'une transaction, si des données sont mises à jour soit l'ensemble des mises à jour est appliqué, soit aucune mise à jour n'est appliquée. Pour prendre un exemple courant, on peut imaginer la transaction qui fait passer de l'argent

d'un compte en banque A à un compte B. Les modifications de données concernent le solde des comptes, et celles-ci doivent être faite dans le cadre d'une transaction afin de garantir la consistance de l'état global des comptes. Si par exemple le compte A est débité de 100€, le compte B sera crédité de 100€. La consistance de la transaction garantit que ces deux opérations sur les soldes seront soit appliquée toutes les deux, soit aucune. En effet, si ce n'était pas le cas et que seule la première opération était effectuée, le compte A serait ainsi débité de 100€ sans pour autant que le compte B soit crédité. 100€ auraient donc disparu dans l'opération, et grâce à Lavoisier on sait bien que ce n'est pas possible…

3.2.2 Transactions distribuées

Dans le cadre de LEAF, le modèle client/serveur est étendu à un modèle n-tiers, puisqu'un client accède à un serveur qui peut lui-même accéder à d'autres serveurs, à de multiples bases de données… Par exemple, le client invoque une méthode d'un bean, causant la modification de données dans une base de données A, puis un appel à un service hébergé par le cext, celui-ci accédant alors à une base données B. Garantir que l'appel du client sur le bean est fait de manière transactionnelle revient de manière simplifiée à dire que si tout se passe bien, les deux bases de données A et B ont été modifiées de manière atomique, mais que si quelque chose se passe mal, l'appel du client retourne avec une erreur et aucune des deux bases de données n'a été modifiée.

Jusqu'à récemment, la gestion des transactions dans LEAF était assez limitée!: un composant pouvait utiliser une seule base de données à la fois dans la même transaction, et la transaction restait confinée à l'instance de LEAF dans laquelle elle était initiée. Cela signifiait que si par exemple un service du cext utilisait une base de données et faisait appel à un bean dans la même transaction, l'appel au bean ne se faisait pas de manière transactionnelle. Aussi si le bean modifiait des données à son tour, il y avait potentiellement une inconsistance des données puisque le bean pouvait réussir à modifier les données alors que le service échouait par exemple.

Le but de cette seconde partie de stage était donc de faire en sorte que les transactions soient propagées entre les différentes instances de LEAF (Nodes, Sites, client, container EJB), et par la même occasion pouvoir utiliser plusieurs SGBD dans le cadre d'une même transaction.

3.3 Présentation des différentes étapes suivies

Afin de mener à bien ma «!mission!», j'ai procédé à différentes étapes menant au choix et à l'élaboration d'une solution au problème posé. Je présente ici de manière synthétique le cheminement que j'ai suivi durant ce projet, les résultats concrets de mon travail étant présentés au paragraphe 3.4.

3.3.1 Compréhension de l'existant

Un prérequis évident face à un tel problème est de pouvoir comprendre l'architecture et le fonctionnement du logiciel existant afin d'être capable de juger des opportunités d'extension, des besoins de modifications, de proposer des améliorations annexes… Il n'est heureusement pas indispensable de comprendre la totalité du logiciel dans ses moindres détails pour pouvoir commencer à concevoir une solution au problème posé, et de nombreux points de détails pourront être vus par la suite.
Afin de mieux comprendre le fonctionnement du logiciel, j'ai tout d'abord essayé de le faire marcher en tant qu'utilisateur «!standard!» de celui-ci. S'agissant en fait d'un framework de développement, être utilisateur signifie plutôt développeur d'applications basées sur ce

framework. Implémenter le classique «!Hello World!!!» version distribuée avec LEAF a été un premier pas, et non des moindres, puisque l'installation de l'environnement de travail est parfois fastidieuse pour les non-initiés.
Pour comprendre l'architecture globale de LEAF, un certain nombre de documents était à ma disposition. La lecture de ceux-ci a été fort instructive, mais l'assimilation d'une telle architecture a néanmoins nécessité quelques explications de la part des ingénieurs ayant développé l'architecture de LEAF. D'autre part, le but de mon travail étant de réellement développer une amélioration de l'existant, il était nécessaire pour moi de comprendre l'organisation du code source afin de pouvoir me repérer dans les diverses parties du framework. Une bonne partie du travail de lecture du code source a consisté à trouver et comprendre comment les concepts globaux de l'architecture se matérialisent dans le source de LEAF. Pour cela plusieurs outils m'ont aidé dans cette démarche!: tout d'abord la lecture des «!Javadocs!» qui sont générés à partir du code source du logiciel. Ce type de documentation donne des explications très proches de l'implémentation des concepts décrits dans la documentation de plus haut niveau. D'autre part j'ai pu comprendre de nombreux points qui m'échappaient grâce à l'utilisation d'un débogueur afin de suivre le flot d'appel lors de certaines parties critiques du code, articulant divers modules entre eux.

3.3.2 Recherche de solutions

La recherche de solutions au problème posé nécessite en fait encore une fois d'acquérir de nouvelles connaissances sur les possibilités offertes pour résoudre le problème. Ainsi mon problème relevant des transactions, il était indispensable que je comprenne mieux les problèmes liés à celles-ci, les protocoles de 2 phase commit (2PC), de récupération en cas de problème… D'autre part la plate-forme sous-jacente étant J2EE, la lecture des spécifications relatives aux transactions (JTA, JTS) faisait partie intégrante du travail de recherche. Certains points obscurs ont pu être éclaircis par l'étude du code source du container EJB jBoss disponible sous licence GPL (une des licences open-source).
Enfin, un aspect non négligeable de la recherche de solutions a été l'évaluation de produits existants afin de prévoir les possibilités d'interopérabilité entre LEAF et les containers EJB sur lesquels il s'appuie. Ce travail d'évaluation consistait en l'étude des documentations des produits comme WebLogic ou WebSphere, complétée par l'élaboration de tests mettant réellement en jeu certaines fonctionnalités des produits.

Si l'émergence des idées de solution venait «!naturellement!» dans la tête, l'élaboration de nombreux diagrammes schématisant les idées permettait de franchir une première étape de validation. Ainsi j'ai pu découvrir et ensuite tirer parti des diagrammes UML (Unified Modeling Language), notamment les diagrammes de déploiement et les diagrammes de séquence. Les premiers m'ont permis de mieux décrire les scénarios faisant intervenir les transactions distribuées, en montrant de manière claire les besoins de propagation de transaction. Les diagrammes de séquence quant à eux permettent de décrire un enchaînement d'appels entre divers modules du logiciel. L'élaboration des diagrammes de séquence m'a souvent permis de repérer les points flous de mes propositions de solutions, et parfois d'en rejeter certaines en constatant des impossibilités techniques.

Afin de présenter mes idées et déterminer un choix validé par les ingénieurs en charge du projet, j'ai réalisé de petits documents reprenant ces diagrammes UML, et expliquant les avantages et inconvénients de chaque solution.

3.3.3 Développement

Une fois l'idée de solution retenue, le développement proprement dit a pu commencer. Face à l'ampleur de la tâche, il est apparu nécessaire de se fixer des étapes dans le développement. Ces étapes correspondent généralement à des sous-parties distinctes du travail, mais il est arrivé que certaines étapes soient seulement une portion d'un module, une étape ultérieure apportant des améliorations par la suite.
L'avancement de mon travail a été suivi spécifiquement par Philippe Oser auquel je faisais des comptes rendus de l'évolution toutes les une à deux semaines. Ces réunions permettaient aussi de faire des choix sur des détail techniques survenant au cours du développement.

3.3.4 Tests et documentation

La réalisation des tests et de la documentation est souvent considérée comme la dernière étape d'un projet. En fait les tests doivent être faits le plus tôt possible, le mieux étant de programmer les tests durant la phase de développement afin de ne pas oublier de tester des parties de code. Cela permet aussi d'être sûr de la validité de certains composants avant de développer les suivants.
On distingue deux types de tests!: les tests unitaires et les tests d'intégration. Les tests unitaires permettent de tester les composants du logiciel de manière isolée, tandis que les tests d'intégration permettent de valider les interactions entre ces composants. Suivant les recommandations de l'équipe de LEAF, les tests sont aussi automatisés que possible, afin de pouvoir les lancer régulièrement pour vérifier la non régression du logiciel. Cela est très important lorsque l'on travaille en équipe sur un même code source, car les modifications faites par une personne peuvent avoir des répercussions sur des parties écrites par une autre personne.

La documentation quant à elle a été écrite autant que possible au fur et à mesure du développement, car il est très difficile de revenir ultérieurement sur des parties de code anciennes pour les documenter. LEAF étant écrit en Java, le moyen naturel de documenter le code est d'écrire des commentaires dans les fichiers sources, puis de les transformer en pages HTML grâce à l'outil JavaDoc. Cela permet de documenter précisément chaque classe et chaque méthode. On complète la documentation par l'écriture de quelques pages HTML décrivant de manière plus globale les fonctionnalités d'un module, ces pages étant aussi intégrées à la documentation javadoc.
Si il est bien sûr préférable d'écrire la documentation au fur et à mesure du développement, cela n'est pas toujours très facile à faire car au tout début du développement certaines parties peuvent être modifiées plusieurs fois, rendant à chaque fois obsolète la documentation. L'approche que j'ai tentée a consisté à attendre que le code se stabilise un peu avant d'écrire la documentation. Il a néanmoins été indispensable de vérifier et compléter la documentation lorsque mon stage touchait à sa fin et que le développement proprement dit était terminé.

3.4 Résultats obtenus

3.4.1 Le gestionnaire de transactions

Le gestionnaire de transaction développé durant le stage joue un rôle central dans la solution adoptée. En effet, même si chaque container EJB possède son propre gestionnaire de transactions, il est indispensable que LEAF ait aussi le sien, ne serait-ce que pour pouvoir gérer les transactions dans les instances de LEAF clientes ou du cext. Dans ces derniers cas, le container EJB peut être totalement ignoré, et il est nécessaire de gérer les différentes ressources utilisées dans une instance de LEAF.

Ce gestionnaire de transaction fonctionne de la manière suivante!: chaque fois qu'une ressource est allouée à un service (par exemple une connexion à une base de données), cette ressource s'enregistre auprès du gestionnaire de transaction afin de le notifier de son appartenance à la transaction courante. Une transaction est associée à un thread, et il est possible d'avoir plusieurs transactions simultanément, chacune opérant indépendamment. Il est du ressort de chaque ressource de synchroniser les transactions entre elles, par exemple avec un système de verrou, approche classique des bases de données.

Concrètement le gestionnaire de transaction est capable de gérer des ressources de type XA, qui gèrent donc le commit à 2 phases, et au plus une ressource non XA. Avec au plus une ressource non XA, il est possible de faire un presque 2PC, comme décrit ultérieurement au paragraphe 3.4.3.
Les ressources XA avec lesquelles j'ai pu tester mon gestionnaire de transaction étaient principalement des bases de données (Oracle, Cloudscape), mais aussi avec un «!connecteur!» suivant les spécifications JCA (Java Connector Architecture).

3.4.2 Choix du niveau d'interopérabilité

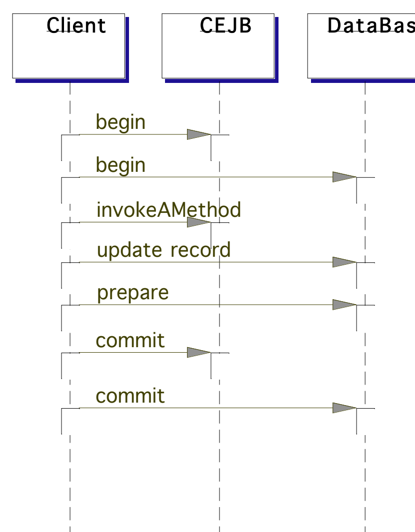
La phase de réflexion et de lecture de documentation a abouti à arrêter un choix quant au niveau d'interopérabilité entre les différentes instances de LEAF, particulièrement vis à vis de l'interopérabilité entre le container EJB et le cext. Le problème était le suivant!: La propagation de transactions entre plusieurs containers nécessite que ceux-ci présentent une interface de type XA (spécification de l'Open Group permettant notamment de gérer le 2PC) utilisable depuis «!l'extérieur!» du container. Or, d'après la spécification EJB actuelle l'interopérabilité entre containers doit se faire par l'utilisation du protocole IIOP (Internet InterOrb Protocol) défini par l'OMG (Object Management Group) et est à la base de CORBA (Common Object Request Broker Architecture). IIOP gère l'interopérabilité au niveau des invocations principalement, mais en ce qui concerne les transactions, ce sont les spécifications OTS (Object Transaction Service) qu'il faut suivre.
Malheureusement si les spécifications EJB rendent obligatoire l'utilisation de IIOP pour pouvoir déclarer un container conforme à EJB 2.0, le support de l'interopérabilité des transactions reste optionnel dans ces mêmes spécifications. En conséquence, très peu de containers EJB la gèrent et à l'heure actuelle il semble que seuls WebSphere d'IBM et l'implémentation de référence de Sun (J2EE RI) gèrent cela.
D'autre part, même si l'utilisation d'IIOP pour l'interopérabilité entre le cext et le container EJB semblait la solution la plus «!propre!», cela impliquait d'avoir un ORB de type CORBA au sein du cext afin de pouvoir transmettre les invocations par IIOP. L'écriture d'un tel ORB n'était vraiment pas envisageable, et l'utilisation d'un ORB existant comme celui intégré au JDK 1.3 posait des problèmes car il recèle de nombreux bogues connus.

Je me suis donc orienté vers une solution très différente de ce qui serait «!la voie normale!». Elle consiste à considérer des cas distincts selon qu'une transaction est démarquée par le container EJB ou par une instance de LEAF différente (client ou service hébergé dans le cext). Avant de rentrer dans les détails de la solution retenue au paragraphe suivant, il faut souligner que le niveau d'interopérabilité offert par la cette solution est moindre, et nous étions bien conscients de ce problème en choisissant cette solution. Ce choix a été fait en fonction des besoins de systèmes réels auxquels LEAF avait pour ambition de répondre.

3.4.3 Invocation du container EJB

Un bean hébergé par le container EJB peut être utilisé soit par un client «!normal!», soit par un service hébergé par une instance de LEAF composant le cext. Dans le modèle EJB, les

clients invoquent des méthodes d'un bean au travers d'un stub (souche). Ce dernier est généré spécifiquement pour chaque bean par des outils de pré-compilation fournis par le container EJB (ou dynamiquement pour jBoss). Une instance cliente de LEAF, ou un service du cext va utiliser ce stub pour appeler des méthodes, mais aussi éventuellement pour démarquer la transaction, c'est à dire la faire commencer puis commettre (commit) ou abandonner (roll back).
Le problème de l'utilisation de cette interface cliente est que l'interface de démarcation des transactions ne permet que de faire des transactions «!locales!», c'est à dire un commit à une phase (1PC). Le seul moyen de passer à un 2PC est justement d'utiliser IIOP et OTS, ce qui a été écarté pour les raisons évoquées plus haut. Or il peut être nécessaire pour le client d'accéder durant une même transaction à la fois à un bean et à une base de données. Face à des ressources transactionnelles multiples, il faut appliquer le protocole de commit à 2 phases (2PC), ce qui nécessite que les ressources le gèrent. Heureusement, bien que l'interface d'accès au container EJB ne gère pas le 2PC, il est possible d'appliquer presque la totalité du 2PC malgré cette ressource «!spéciale!».
Pour cela, il faut appeler *commit* sur la ressource 1PC après que toutes les ressources 2PC ont répondu positivement à une demande de préparation de la transaction. En effet, une réponse positive à *prepare* garantit qu'il sera possible par la suite de commettre la ressource. Donc si l'appel à commit sur le container EJB se déroule bien, il ne reste plus qu'à commettre les autres ressources, et dans le cas contraire il faut abandonner la transaction.
Le diagramme de séquence suivant illustre le cas où un client utilise un bean et une base de données et où la transaction se passe bien.



3.4.4 Invocation du container extension

Un service du container extension (cext) peut être appelé soit par un EJB soit par un autre service du cext, hébergé par la même instance de LEAF ou non. Dans le cas où l'appel ne change pas d'instance de LEAF, les choses sont plus faciles puisqu'il «!suffit!» d'avoir un gestionnaire de transactions dans l'instance de LEAF, et que celui-ci gère la synchronisation des ressources. L'appel de services entre 2 instances du cext représente un peu plus de difficultés dans la mesure où il faut propager les identifiants de transactions, à savoir un identifiant global et un identifiant de branche. Même si le développement semblait délicat, le

concept semblait valide et réalisable car s'agissant du cext, j'avais totale maîtrise du source et pouvait faire des modifications pour l'adapter à mes exigences.
En fait la difficulté, du point de vue de la conception de la solution, venait de la propagation de la transaction entre le container EJB et le cext. En effet, il faut que le container EJB coopère afin de prendre en compte une nouvelle ressource transactionnelle que constitue le cext. Heureusement la récente spécification JCA (Java Connector Architecture) de Sun permet justement cela. A l'origine JCA a d'ailleurs été développée pour permettre l'intégration d'application «!legacy!» au sein d'un système J2EE.
La création d'un connecteur JCA a donc permis de faire prendre en compte au container EJB l'existence d'une ressource transactionnelle nouvelle dans le cas où un bean accède à un service du cext. Dans ce cas, la gestion de la transaction est laissée au gestionnaire de transaction fourni par le container EJB, et le connecteur reçoit les ordres de démarrage, préparation et achèvement de la transaction. Le connecteur propage alors ces demandes au cext, dont le gestionnaire de transaction prendra en compte cette démarcation et gèrera à son tour les ressources transactionnelles (comme une base de données) que les services du cext utilisent dans le cadre de cette transaction.

Une des particularités de la spécification JCA est d'avoir prévu la possibilité d'utiliser des connecteurs JCA en dehors d'un container EJB. Cela m'a donc permis de réutiliser le connecteur écrit pour la communication CEJB-CEXT dans le cadre de la communication CEXT-CEXT, c'est à dire entre deux Nodes ou Sites. Ainsi pour le gestionnaire de transaction d'un site A, une instance B de LEAF n'est rien d'autre qu'une nouvelle ressource de type XA (donc capable de faire un 2PC).

3.4.5 Récupération de transactions en cas d'erreur

Maintenir les propriétés ACID d'une transaction est une tâche ardue car il faut prévoir toutes les possibilités de crash d'un des participants à la transaction!: le gestionnaire, le client, une ressource, ou tout simplement le réseau… Pour expliquer cela, comparons ce qui se passe lors d'un crash dans le cas du commit à une phase et du commit à 2 phases.

Cas du 1PC

Dans le cas du 1PC, il n'y a qu'une seule ressource en jeu. Etant donné que le client et le gestionnaire de transactions ne font qu'un seul et même processus, on est donc en présence de seulement deux processus!: le client (avec son gestionnaire de transactions) et la ressource transactionnelle (une base de données par exemple). Si l'un des deux plante avant que «!commit!» ait été appelé, la transaction sera perdue. En effet, si il s'agit d'un crash du client, celui-ci, lorsqu'il sera relancé, aura totalement oublié la transaction et la base de données au bout d'un certain laps de temps abandonnera d'elle-même la transaction. Si au contraire il s'agit de la base de données qui a subi un plantage, lorsqu'elle sera relancée elle nettoiera d'elle même les verrous éventuellement laissés et la transaction aura donc finalement été abandonnée. Les propriétés ACID sont donc bien respectées et les données sont toujours consistantes.

Cas du 2PC

Prenons maintenant le cas du commit à 2 phases avec un client et au moins deux ressources impliqués dans la transaction. Il faut envisager deux cas!: un crash du client et un crash d'une ressource.
Si tout se passe normalement, le client conclue la transaction en appelant «!prepare!» sur chacune des ressources et celles–ci répondent positivement (soit OK, soit READ ONLY, ce

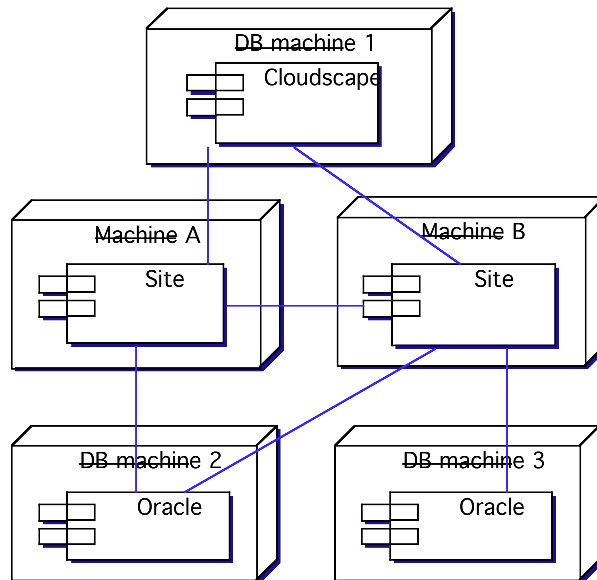
qui est une optimisation pour éviter au gestionnaire de transaction d'appeler «!commit!»). Alors le gestionnaire de transactions appelle «!commit!» sur chacune des ressources. La transaction est alors terminée, les données sont consistantes puisque les deux ressources ont été modifiées.
Mais si le client subit un crash entre deux appels à «!commit!», on trouve là une situation de non consistance des données. En effet, une ressource a été commise tandis qu'une autre a juste été préparée mais pas encore commise. Lorsque le gestionnaire de transactions sera relancé, il ne saura pas qu'une ressource n'a pas encore été commise. Il est absolument nécessaire qu'il la commette pour garder la consistance des données.

Solution!: la table des décisions

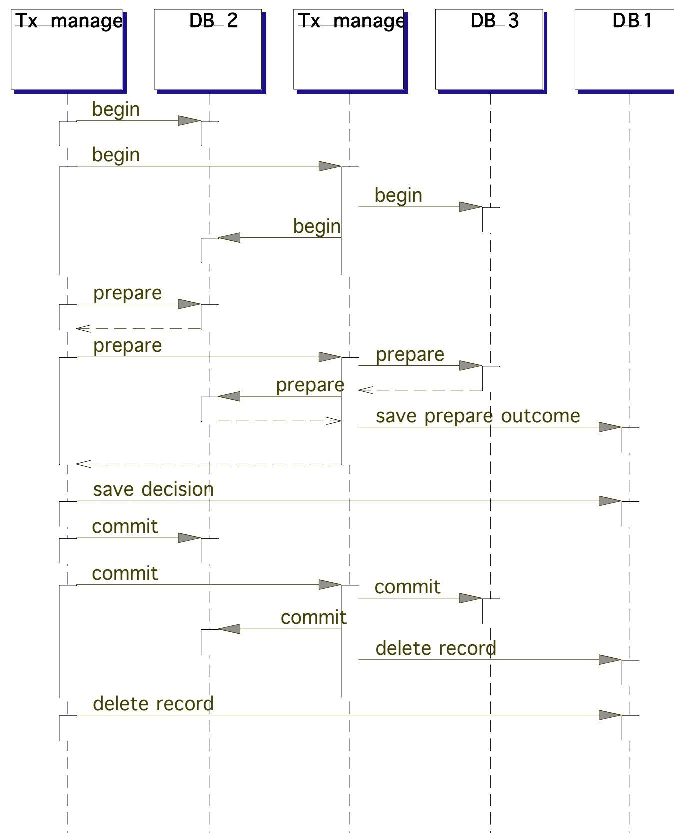
La solution à ce problème consiste à enregistrer la décision que prend le gestionnaire de transactions de manière persistante. En fait il suffit d'enregistrer uniquement les décisions de commettre, une telle décision étant prises lorsque toutes les ressources impliquées dans la transaction ont répondu positivement à la commande «!prepare!». Le gestionnaire de transaction que j'ai développé enregistre cette décision dans une base de données, stockant les numéros d'identification des transactions dans une table spécifique. Il y a un enregistrement par ressource et par transaction préparée mais non encore commise. Une fois que le gestionnaire de transaction a commis une ressource, il supprime l'enregistrement correspondant de la table des décisions. En cas de plantage, lorsqu'il est relancé il accède à cette table et sait alors quelles sont les transactions qu'il lui reste à commettre.
Si après préparation de toutes les ressources l'une d'elles avait répondu négativement, le gestionnaire de transaction doit appeler «!roll-back!» sur les ressources qui avaient répondu positivement. Si un crash survient au gestionnaire de transaction avant qu'il puisse dire aux ressources d'abandonner les transactions, il faut qu'il puisse le dire lorsqu'il sera relancé. Pour cela, il interroge, lors de son lancement, chaque ressource pour savoir si elle a des transactions préparées mais non commises. En comparant la réponse des ressources au contenu de la table des décisions, le gestionnaire de transactions sait si il doit demander à la ressources de commettre ou d'abandonner la transaction en attente.

Cas de plusieurs instances de LEAF

Les problèmes décrits précédemment montrent des cas où il y a une instance de LEAF accédant à plusieurs ressources, telles que des bases de données.
Mais dans le cas où plusieurs instances de LEAF sont impliquées dans une même transaction, il faut pouvoir enregistrer les décisions de chacune sachant qu'en fait il n'y a qu'une seule instance qui prend la décision. Cette dernière instance est celle qui a initié la transaction et qui la démarque donc. Le problème est que l'instance de départ (A) n'est pas consciente de l'ensemble des ressources utilisées dans la transaction puisqu'une autre instance de LEAF peut utiliser une nouvelle ressource dans la transaction. Le diagramme de déploiement suivant illustre un scénario où l'instance de LEAF A utilise la base de données Oracle sur la machine 2 et fait appel à un service sur l'instance de LEAF B. Ce dernier service utilise non seulement la base 2, mais aussi une base 3. La base Cloudscape 1 sert à stocker les décisions suite à «!prepare!».



La solution mise en œuvre permet à l'instance A de LEAF de considérer l'instance B comme une ressource, au même titre que la base de données 2. De ce fait, un appel à «!prepare!» sera fait sur l'instance 2. Cette dernière propage l'appel en invoquant «!prepare!» sur les bases 2 et 3. Si ces dernières répondent positivement, le gestionnaire de transaction de B va enregistrer une décision mais en précisant que la décision n'est pas définitive puisqu'il n'est pas l'initiateur de la transaction. Il retourne alors une réponse positive au gestionnaire A. Celui-ci peut alors prendre sa décision et l'enregistrer dans la base 1. Dans la base 1, on a alors des enregistrements correspondant à la base 1 (2 fois) et à la base 3. Toutefois deux de ces enregistrements sont marqués comme n'étant pas une décision définitive. Ces enregistrements peuvent être reliés entre eux car l'identifiant de transaction globale est le même, seul l'identification de branche de transaction diffère. Le diagramme de séquence suivant montre l'enchaînement des appels entre les différents composants impliqués dans l'exemple de transaction précédent.



En cas de crash d'un des gestionnaires de transaction, n'importe quelle instance de LEAF est capable de récupérer une transaction inachevée car en fait la configuration du cext est globale dans le sens où les ressources utilisables par une instance de LEAF sont déclarées auprès de l'ensemble des instances de LEAF.

3.4.6 Tests effectués

Après avoir exposé le fonctionnement des points intéressants de la solution à laquelle je suis arrivé, il convient d'évoquer maintenant rapidement les tests auxquels j'ai soumis mon gestionnaire de transaction.
En ce qui concerne les containers EJB, j'ai seulement pu tester avec jBoss 2.4.3 et WebLogic 6.1 SP1. Ces logiciels se sont avérés les seuls à ma connaissance à supporter la spécification JCA nécessaire à mon projet. En ce qui concerne les bases de données, j'ai pu tester uniquement avec Cloudscape (une base de données écrite entièrement en java) et Oracle 8.1.6.
Mes tests n'ont toutefois pas pu être poussés jusqu'au bout pour diverses raisons. J'ai tout d'abord été confrontés à plusieurs bogues des divers logiciels avec lesquels le gestionnaire de transaction interagissait. La toute dernière version de WebLogic en comporte un certain nombre d'assez gênants, et la version d'Oracle que j'avais à ma disposition aussi. Une version plus récente existe et est censée les corriger, mais je n'ai pas pu me la procurer.
Outre cela, mes tests n'ont porté que sur les fonctionnalités et pas du tout sur les performances. Je n'ai pas eu le temps matériel d'effectuer des tests mettant en jeu plusieurs machines à la fois afin de tester en conditions réelles la distribution des transactions et leur performances.

Il ressort néanmoins de cette période de tests que le module transactionnel de LEAF marche et permet maintenant d'utiliser plusieurs bases de données, ainsi que de propager les transactions entre plusieurs instances de LEAF.

3.5 Perspectives d'évolution

3.5.1 Amélioration du modèle d'invocations du cext.

A l'heure actuelle, il est impossible dans LEAF de réaliser des invocations provoquant un «!bouclage!» entre instances de LEAF. C'est à dire qu'un service d'une instance A peut appeler un service d'une instance B, mais ce dernier ne peut rappeler un service de l'instance A dans le cadre de la même transaction. Par contre le service sur B peut appeler d'autres services de B ou d'une troisième instance C et ainsi de suite…
Cette limitation est due au fait que lorsqu'une invocation arrive sur une instance de LEAF, elle est prise en charge par un nouveau thread. Donc dans le cas ou on aurait A->B->A, A ferait son appel avec un thread A.1, B traiterait l'appel avec un thread B.1 et invoquerait A qui traiterait l'appel avec un thread A.2. Or cela casse le modèle de transaction adopté qui associe une transaction à un thread.
En fait au-delà du problème au niveau transactionnel, le système actuel d'invocation peut potentiellement entraîner des inter-blocages (deadlocks) puisque le thread A.1 peut verrouiller une ressource à laquelle le thread A.2 voudra accéder. Or en fait il n'y a pas de raison d'avoir un blocage puisque A.2 est en fait indirectement une invocation de A.1. La solution serait de modifier la manière dont les invocations sont faites pour que le thread A.1 soit réutilisé lorsque B invoque A à la suite d'une invocation de A.

3.5.2 Passage à IIOP

Le fait d'invoquer le container EJB par son interface client entraîne plusieurs problèmes au niveau de la propagation des transactions.
Tout d'abord on retrouve le problème évoqué ci-dessus où une boucle dans les invocations se fait. Si par exemple une transaction est initiée dans le container EJB, puis qu'un bean appelle un service du cext et que ce dernier invoque une méthode sur un bean, on a un bouclage où les deux beans sont invoqués par deux threads différents.
A l'inverse, si un service du cext utilise une base de données et un bean qui utilise à son tour la même base de données, la base de données ne sera pas utilisée avec les mêmes identifiants de transaction globale car l'interface cliente du container EJB ne permet pas de donner un identifiant de transaction. On pourra donc éventuellement avoir un blocage au niveau de la base de données puisqu'elle considère qu'il y a deux transactions différentes menées en parallèle.
Enfin, le quasi-2PC tel que décrit au paragraphe 3.4.3 pose un problème potentiel de consistance des données. En effet, comme décrit précédemment , on appelle «!prepare!» sur toutes les ressources capables de faire un 2PC, puis «!commit!» sur la ressource non-XA (incapable de faire le 2PC, comme le container EJB via l'interface cliente) et enfin «!commit!» sur les ressources préparées. Mais comme expliqué en 3.4.5 il est nécessaire d'enregistrer la décision que l'on prend afin de pouvoir finir des transactions en cas de plantage. Avec le quasi-2PC, la décision ne peut-être prise qu'après l'appel à «!commit!» sur la ressource non-XA. Or si le crash du gestionnaire de transaction a lieu entre le commit de cette ressource et l'écriture de la décision, lorsqu'il sera relancé il abandonnera les transactions en attente sur les ressources XA puisqu'il n'avait pas mémorisé sa décision. On pourrait penser à certaines solutions qui mettrait en jeu un bean chargé d'écrire la décision pour le gestionnaire de transaction afin que la réussite du commit du container EJB implique

de manière atomique l'enregistrement de la décision. Un tel système se révèlerait assez lourd à mettre en œuvre et il a été décidé de se contenter d'un système imparfait pour l'instant.

En fait tous les problèmes évoqués dans ce paragraphe auraient pour solution le passage au protocole IIOP pour l'invocation du container EJB. Il faut reconnaître que la décision prise au début d'éviter IIOP aurait peut-être était différente si les problèmes évoqués avaient été connus dés le départ. En fait l'expérience du domaine des transactions me manquait ainsi qu'à l'équipe de développement de LEAF. Aussi on peut dire que le travail effectué durant mon stage aura au moins permis d'apporter quelque chose à l'entreprise au niveau de la connaissance des problèmes liés aux transactions.

4 Conclusions

Travailler au sein d'ELCA pendant plus de 5 mois m'a permis d'acquérir à la fois de nouvelles connaissances techniques, mais aussi et surtout de découvrir le milieu de l'informatique professionnelle et la gestion des projets de développement.
Participer à la réflexion très en amont sur les besoins d'ELCA en matière d'EAI et continuer jusqu'au développement d'un prototype de solution a été pour moi l'occasion de voir comment gérer un projet notamment lorsqu'il est inscrit dans un cadre assez ouvert. La fixation régulière d'objectifs à court terme et les comptes-rendus faits à mon encadrant ont été déterminant pour la bonne marche du projet.
Du point de vue technique, la deuxième partie de mon stage a été la plus riche en enseignements, les transactions distribuées étant un domaine important en informatique d'entreprise, mais finalement peu souvent abordé. Les liens que j'ai pu entretenir avec l'équipe de développement de LEAF, tant à Lausanne qu'à Zürich, m'ont initié au travail et au développement en équipe, ce qui constitue une expérience professionnelle fort instructive pour ma prochaine entrée dans le monde du travail.

Remerciements

Je tiens à remercier vivement ELCA de m'avoir accueilli si chaleureusement durant ces 5 mois de stage et d'avoir fait que tout se soit aussi bien passé. Merci à Bernhard Rytz, Christian Gasser et Philippe Oser de m'avoir dirigé et empêché de perdre le fil de mon travail tout au long du stage. Une grande reconnaissance aussi aux développeurs de LEAF, Eric Castan, Alain Borlet-Hotte et Yves Martin, qui ont eu la patience de m'expliquer beaucoup de choses et avec lesquels j'ai pu avoir de nombreux échanges. Enfin je tiens à exprimer toute ma sympathie pour mes compagnons de bureau Felix Jaeger, Paul Puech et Olivier Cathala qui ont découvert ELCA en même temps que moi.

Bibliographie

EAI

The EAI journal!: http://www.eaijournal.com/
Liste d'articles sur EAI journal!:
http://www.eaijournal.com/Department.asp?DepartmentID=5
Brève histoire de l'EAI!:
http://www.eaijournal.com/Article.asp?ArticleID=119&DepartmentId=5
EAI et JCA!: http://www.eaijournal.com/Article.asp?ArticleID=348&DepartmentId=5
CrossWorlds (fournisseur d'EAI)!: http://www.crossworlds.com
BEA WLAI!: http://edocs.bea.com/wlintegration/v2_1/index.html
A service broker!: http://www.theserverside.com/resources/article.jsp?l=Service-Broker
EAI, a crucial technology in large scale systems!:
http://ejbinfo.com/features/00/09/07/2335214.shtml
Advice for selecting EAI tools!: http://eai.ebizq.net/str/sholler_1.html
B2B integration using IBM MQSeries and MQSI!:
http://www.redbooks.ibm.com/pubs/pdfs/redbooks/sg246010.pdf
OpenAdaptor : http://www.openadaptor.org

XML et Web Services

XML!: http://www.w3.org/XML/
EbXML et web services!:
http://www.webservicesarchitect.com/content/articles/irani03.asp
Practical considerations in implementing web services!:
http://www.webservicesarchitect.com/content/articles/irani01.asp
Apache SOAP!: http://xml.apache.org/soap/index.html
Apache Xerces : http://xml.apache.org/xerces2-j/index.html

LEAF

LEAF introduction!:
http://www.elca.ch/Home/Competencies/Distributed_Systems/398.html
LEAF Data Sheet (PDF) : http://www.elca.ch/ressources/others/395.pdf

Transactions

Spécifications JTA!: http://java.sun.com/products/jta/index.html
Spécifications JTS!: http://java.sun.com/products/jts/
Spécifications XA!(payantes) : http://www.opengroup.org/pubs/catalog/c193.htm

Technologies Java

Spécifications J2EE!: http://java.sun.com/j2ee
Spécifications EJB!: http://java.sun.com/products/ejb/
Spécifications JCA!: http://java.sun.com/j2ee/connector/index.html