



**Ecole Nationale Supérieure d'Informatique
et de Mathématiques Appliquées de Grenoble**



Rapport de Projet de fin d'études

Interopérabilité JOnAS et CORBA

**Patrick Bruneton
Marc Dutoo**

EVIDIAN

Réalisé à:
BULL Echirolles
1, rue de Provence
B.P 208
38432 Echirolles Cedex

EVIDIAN
A Groupe Bull Company

26 février – 26 juin 2001

Remerciements

Nous tenons à remercier toutes les personnes de l'équipe JOnAS, en particulier notre responsable de stage Gérard Vandome, pour l'accueil qu'elles nous ont réservé, ainsi que pour les nombreux conseils qu'elles nous ont fournis au cours du projet.

Notre gratitude va également aux membres de l'équipe Jonathan, François Horn et Bruno Dumant, qui ont apporté une aide sans faille à notre prise en main de leur produit.

Sommaire

REMERCIEMENTS	1
SOMMAIRE	3
INTRODUCTION.....	5
1. CONTEXTE	7
1.1. BULL ET EVIDIAN.....	7
1.2. TECHNOLOGIES UTILISEES.....	8
1.3. PRODUITS.....	10
2. OBJECTIFS	13
2.1. INTEROPERABILITE.....	13
2.2. REFORMULATION DES OBJECTIFS.....	15
3. CHOIX D'UN PROTOCOLE	17
3.1. BESOINS	17
3.2. LES PROTOCOLES TESTES.....	17
3.3. PERFORMANCES.....	18
3.4. INTEROPERABILITE.....	19
3.5. LA SOLUTION SUN	20
3.6. MULTI-PROTOCOLE	22
4. INTEGRATION DE JONATHAN 3.0	25
4.1. BESOINS	25
4.2. RECEPTION DE JONATHAN 3.0	25
4.3. INTEGRATION.....	25
4.4. INTEROPERABILITE CORBA	28
5. TRANSACTION ET SECURITE.....	31
5.1. INTRODUCTION.....	31
5.2. DIFFERENTES ARCHITECTURES POSSIBLES.....	32
5.3. IMPLEMENTATION D'UNE COUCHE OTS AU-DESSUS DU TM	33
5.4. IMPLEMENTATION D'UN OTS COMPLET.....	34
5.5. IMPLEMENTATION DE LA COOPERATION ENTRE TM ET OTS	36
5.6. MISE EN ŒUVRE DE LA COOPERATION ENTRE TM ET OTS	37
5.7. MISE EN ŒUVRE DE LA SECURITE.....	38
6. BILAN.....	39
BIBLIOGRAPHIE ET LIENS	41
ANNEXE A : EVALUATION DE PROTOCOLES	43

Introduction

En tant qu'élèves ingénieur en troisième année d'étude à l'ENSIMAG, nous avons effectué notre projet de fin d'études en binôme dans l'entreprise Evidian au sein de l'équipe du projet JOnAS. La problématique en est l'étude et la mise en œuvre de l'interopérabilité entre CORBA et le serveur d'EJB JOnAS.

Dans un premier temps, nous présenterons le contexte afin d'y familiariser le lecteur, avant d'opérer une reformulation des objectifs énoncés dans le pré-rapport pour tenir compte des nouveaux éléments entrant en jeu. Nous étudierons ensuite le portage de JOnAS au-dessus protocole RMI/IIOP et l'interopérabilité avec CORBA sous les divers angles du choix de l'implémentation de RMI/IIOP à utiliser, de son intégration basique à JOnAS et enfin des services transactionnel et de sécurité. Un bilan final technique et humain conclura notre propos.

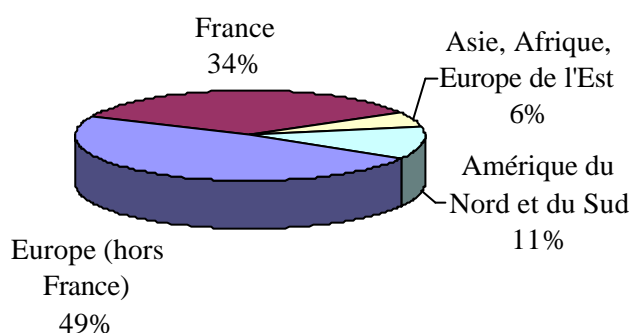
1. Contexte

1.1. Bull et Evidian

1.1.1. Bull, un groupe informatique international

C'est en août 1921 que Fredrik Rosing Bull construit et livre sa première machine à cartes perforées à une compagnie norvégienne d'assurances, afin de résoudre le problème de l'automatisation du traitement des statistiques. Emergera par la suite une entreprise de droit français à capitaux suisses et belges qui s'installera à Paris sous le nom de H.W.Egli Bull et qui deviendra Bull... Aujourd'hui Bull est un groupe informatique international qui emploie 18 358 personnes dans plus de 100 pays. En 1999, il a réalisé un chiffre d'affaires de 3,8 milliards d'euros dont plus de 65% hors de France, son pays d'origine. Les activités du groupe sont de dimension internationale comme le montre le schéma suivant :

Une base de clientèle internationale



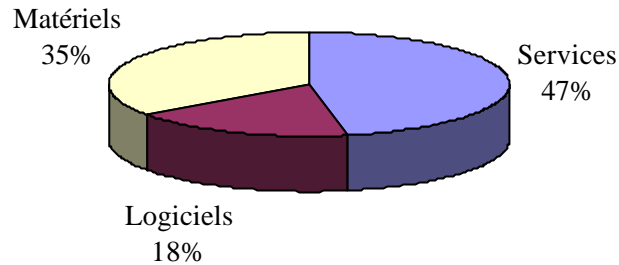
Répartition du C.A. par zone géographique en 1999

Dans le cadre du flux croissant d'informations, Bull travaille essentiellement sur une nouvelle génération de grands serveurs ouverts capables d'exécuter les services ou applications Internet les plus exigeants en termes de performances. Un gros effort d'innovation a été fait, en particulier sur la « *scalabilité* » et la haute disponibilité de l'architecture de ces serveurs, sur leur capacité à accueillir simultanément des environnements hétérogènes, comme Unix, NT, Linux, ainsi que sur des solutions d'archivage en réseaux. Dans le cadre de la miniaturisation et de la mobilité, Bull invente le concept du plus petit serveur web transportable et personnel. Avec la technologie iSimplify!, la carte à microprocesseur devient un nœud personnalisé du réseau dès qu'elle est introduite dans un lecteur approprié. Dans le cadre de la sécurité, au sein de la filiale Evidian, des nouvelles fonctionnalités ont été apportées aux logiciels Openmaster et Accessmaster. Ces innovations visent les grands portails, le commerce électronique, les services de banque à domicile ou d'enchères en ligne qui constituent des applications critiques : il faut garantir un fonctionnement sans perturbations, une permanence de leurs accès ou encore la sécurité des transactions qui leur sont associées. Les nouvelles fonctions des logiciels Openmaster et Accessmaster offrent dès maintenant une totale sécurité des services Internet.

1.1.2. Le site d'Echirolles

A Echirolles, près de Grenoble, se trouve le plus important Centre de Recherches mondial du Groupe Bull spécialisé sur Unix. Cette plate-forme contient de nombreuses divisions de Bull regroupées autour de multiples thèmes (plate-forme AIX, Logiciel, Service, Technologie Objet ...). Les activités de la région Rhône-Alpes sont une importante source de revenus pour le groupe en France :

Répartition du chiffre d'affaire par type de vente au 31/12/1999



C.A région Centre-Rhone-Alpes 31/12/1999 : 718MF

1.1.3. La filiale Evidian

Evidian, une subdivision du groupe Bull, est une entreprise axée sur le développement de logiciels internet et sur les stratégies e-business. Basée à Billerica, Mass, USA, et de dimension internationale, Evidian déploie ses centres de recherches, ses succursales et ses opérations commerciales sur les USA, l'Europe et l'Asie. Au travers de la croissance sur les nouvelles technologies Evidian se situe dans le développement et la commercialisation de solutions sécurisées pour les grands systèmes informatiques.

Evidian est le résultat de deux années d'efforts pour faire de Bullsoft une entité indépendante. Cette « Software- Company » affiche un revenu annuel de plus de 600MF avec 400 employés. Son principal axe de R&D reste le développement de grands systèmes informatiques (AccessMaster, OpenMaster et OpenMaster pour Telecom). C'est dans ce cadre que se situe le Projet Orchidée à l'intérieur duquel est développé le serveur d'EJB Open Source JOnAS.

1.1.4. La communauté ObjectWeb

Evidian est à l'origine, en partenariat avec France Télécom R&D et l'INRIA de l'initiative ObjectWeb (www.objectweb.org), une communauté ouverte de logiciel libre, créée fin 1999, dont le but est de développer des composants logiciels « intermédiaires » (*middleware*), sous licence logiciel libre (*open-source*). Ces composants sont utilisés pour construire des plates-formes réparties adaptables aux différents besoins des applications. ObjectWeb vise à fournir des implantations innovantes, sous licence libre, des principales spécifications du domaine, qu'elles soient issues du JCP (*Java Community Process*) de Sun Microsystems ou des travaux de l'OMG (*Object Management Group*).

La base de code initiale d'ObjectWeb constitue une plate-forme libre de qualité industrielle. Cette base de code est déjà largement utilisée, à la fois pour des applications opérationnelles et pour l'expérimentation de services et d'applications réparties avancées. Par exemple, le serveur EJB JOnAS a déjà été téléchargé plus de 40000 fois et est utilisé par des milliers d'utilisateurs. Enfin, JOnAS est intégré à la plate-forme J2EE Open Source Enhydra (www.enhydra.com), soutenue par Lutris Technologies (www.lutris.com).

1.2. Technologies utilisées

1.2.1. EJB

1.2.1.1. EJB framework

Les EJB (*Enterprise Java Bean*) visent à simplifier la manière de construire des applications réparties selon l'architecture 3 tiers (interface client, traitement, persistance). Le but est de rendre une application facile à développer, déployer et administrer et ce indépendamment de la plate-forme permettant son exécution.

Les spécifications EJB décrivent un cadre de travail (*framework*) serveur dans lequel des composants serveurs (les *Enterprise Beans*) peuvent être déployés pour former une application. Ce *framework* facilite

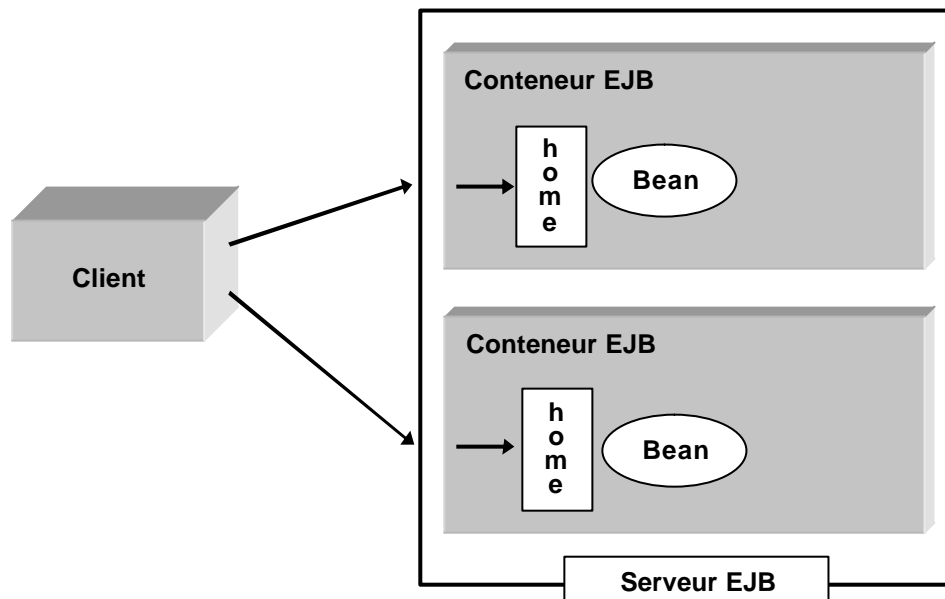
grandement la gestion des ressources de l'application, du cycle de vie des composants serveur, de la sécurité et de l'identification, des transactions, des services de nommage...

L'architecture EJB identifie les éléments suivants :

- les composants logiciels (*Enterprise Bean*)
- les conteneurs
- les serveurs
- les clients

Les conteneurs servent à héberger les beans au sein d'un serveur et permettent de les isoler du client et d'une implémentation spécifique d'un serveur. Le serveur implémente des mécanismes de bas niveau tels que la persistance, la sécurité, les transactions, la gestion mémoire et permet une utilisation simple de ces services par les beans.

Les clients peuvent accéder aux Beans en utilisant leur interface Home qui est enregistré dans un service de nommage (via JNDI) que les clients utilisent pour les retrouver.



1.2.1.2. Entreprise Bean

Il existe deux classes de Bean : les Entity Bean et les Session Bean :

Les Sessions Beans sont non persistants (ils sont perdus lorsque le serveur est arrêté). On distingue les Stateless Session Bean qui n'ont pas de données internes et peuvent être partagés par plusieurs clients, et les Statefull Session Bean qui possèdent un état interne.

Les Entity Bean représentent les données d'une base de données. Ils sont persistants et survivent à une panne du serveur. La persistance peut être gérée par le bean lui-même ou de façon implicite par son conteneur. Les Entity Bean peuvent être accédés de manière concurrente par différents clients (grâce aux transactions).

1.2.1.3. Protocole de communication

EJB 1.0 est basé sur le protocole RMI/JRMP spécifique au monde java. Cependant, EJB 2.0 a mis l'accent sur l'interopérabilité et prône désormais le protocole IIOP (*Internet Inter-ORB Protocol*, issu du monde CORBA) pour les échanges entre les EJB et entre les EJB et CORBA.

1.2.2. CORBA

1.2.2.1. Présentation

CORBA (*Common Object Request Broker Architecture*) est un standard ouvert développé et maintenu par l'OMG (*Object Management Group*), association qui compte aujourd'hui plus de 850 membres. CORBA est

apparu au début des années 90 et est aujourd'hui la plus répandue des infrastructures d'informatique distribuée. Son modèle de composant côté serveur est le CCM (*CORBA Component Model*).

Le bus CORBA est un intermédiaire qui permet à des objets de dialoguer. Ses caractéristiques sont les suivantes :

- Indépendance vis à vis des langages de programmation : utilisation de l'IDL (*Interface Definition Language* qui peut être *mappé* vers différents langages tels que C++, Java, etc.)
- Les requêtes aux objets sont transparentes pour les clients. Ces derniers ne savent pas où sont situés les objets appelés.
- Activation automatique et transparente des objets : un objet n'est en mémoire que lorsqu'il est utilisé. Cependant les clients ont l'illusion que tous les objets sont en mémoire.
- Invocations statiques (contrôlées à la compilation) ou dynamiques (construites et contrôlées à l'exécution).
- Interopérabilité entre bus : un protocole générique de transport des requêtes (GIOP pour *General Inter-ORB Protocol*) a été défini permettant l'interconnexion de bus CORBA provenant de fournisseurs distincts.

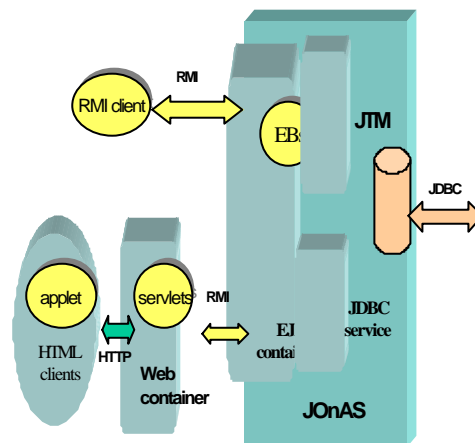
1.2.2.2. Architecture serveur CORBA et EJB

Les architectures serveur CORBA et EJB sont compatibles et même complémentaires, notamment au niveau du protocole de communication (IIOP). L'implémentation de référence de la J2EE fournie par Sun présente même un serveur EJB construit selon les normes CORBA : présence d'un POA (*Portable Object Adapter*), d'un JTS (*Java Transaction Server*), le *mapping* Java de l'OTS (*Object Transaction Server*) de CORBA...

1.3. Produits

1.3.1. JOnAS

La plate-forme JOnAS est une implémentation Open Source de EJB 1.1 (donc écrite en Java), qui est actuellement en cours d'évolution vers le respect des spécifications EJB 2.0 (évolution à laquelle doit participer ce projet). Une nouvelle version majeure de JOnAS a été sortie pendant la période couverte par notre projet : il s'agit de JOnAS 2.3, dont la principale nouveauté est le support des Message Driven Beans.



Les principaux composants de JOnAS sont:

- Conteneurs EJB fournis par un ensemble de classes Java et un outil pour générer les classes d'interposition;
- Un JTM (*Java Transaction Manager*) offrant le support des transactions, la coordination des transactions distribuées et l'initialisation de transactions par le client ;
- Un service JDBC permettant la gestion des connexions aux bases de données;
- Un ensemble d'outils pour développer, déployer et gérer les EJB ;
- Une implémentation Open Source du JMS (*Java Messaging Service*) nommée JORAM (*Java Open Reliable Asynchronous Messaging*) est aussi intégrée à JOnAS, permettant l'emploi de Message Driven Beans.
- Un service de sécurité implémentant le principe des certificats (*rôles*).

Actuellement, il existe deux versions de JOnAS reposant sur des protocoles différents pour la communication entre les clients (qui peuvent être une application indépendante, un applet, un Bean, etc.) et le serveur :

- La version RMI, qui utilise le protocole propriétaire Sun : JRMP;
- La version Jérémie ; Jérémie est la personnalité RMI (mais non compatible RMI/IIOP !) d'un *Object Request Broker* (ORB) appelé Jonathan, qui a également une personnalité CORBA.

1.3.2. Jonathan

Jonathan est un *Object Request Broker* (ORB) entièrement écrit en Java lancé par France Télécom, actuellement maintenu et renouvelé par sa filiale Kelua (www.kelua.com). Il fait partie de l'offre de la communauté Open Source Objectweb, et se caractérise par une architecture ouverte disposant de deux personnalités :

- Jérémie, qui est une implémentation de RMI au-dessus de GIOP ;
- David, basé sur les spécifications CORBA de l'OMG.

Ses principales fonctionnalités sont :

- MOA (*Minimal Object Adapter*) et SOA (*Simple Object Adapter*)
- Serveur de nom pour chacune des personnalités
- Architecture de services CORBA
- Event channel
- Utilitaires : compilateur de stubs pour chacune des personnalités

Jonathan en est à sa version finale 2.0.5, et Jonathan 3.0 a été récemment sorti en version alpha. La personnalité David de cette pré version suit les spécifications CORBA 2.3 et autorise le modèle de programmation RMI/IIOP. Cette dernière caractéristique est d'un intérêt majeur dans le cadre de notre projet. Par ailleurs, la configuration de Jonathan 3.0 repose désormais sur un fichier XML de sémantique spécifique, et non plus sur des fichiers de propriétés Java.

2. Objectifs

2.1. Interopérabilité

2.1.1. Les spécifications EJB

2.1.1.1. Présentation

Dans un premier temps, nous avons examiné les spécifications EJB afin d'en extraire les informations susceptibles de guider notre démarche. D'une part, il y a plusieurs niveaux de conformité aux spécifications EJB, qui sont autant d'objectifs possibles à atteindre. Chacun de ces objectifs a un coût de réalisation spécifique au cadre JOnAS, et des avantages plus ou moins importants sur le plan de l'interopérabilité CORBA ou du portage vers RMI/IIOP. D'autre part, des hypothèses d'implémentation y sont souvent formulées et commentées.

Tout particulièrement, la version 2.0 des spécifications introduit les EJB au-dessus du protocole RMI/IIOP, lequel est issu du monde CORBA, et le support de l'interopérabilité sous forme de trois cas d'utilisation selon le type de client du serveur d'EJB :

- Le client est un composant J2EE (JSP, servlet, EJB d'un vendeur éventuellement différent de celui du serveur d'EJB)
- Le client est une application tierce (c'est le cas de base d'utilisation des EJB)
- Le client est un client CORBA utilisant l'ORB d'un vendeur quelconque.

Outre la mise en œuvre de ces trois cas, l'accent est mis sur l'absence de contraintes supplémentaires sur le développeur d'applications et l'utilisation de standards reconnus au niveau de l'interopérabilité dès que possible. Une bonne part de ces standards étant issus (ou commun) au monde CORBA, il est normal que l'interopérabilité entre composants J2EE et entre composants EJB et client CORBA soit traitée dans le même chapitre. De manière générale, la lecture des spécifications EJB 2.0 fait clairement apparaître que EJB et CORBA sont désormais compatibles et même complémentaires.

On se concentre sur les aspects qui ont trait à l'interopérabilité CORBA et plus spécifiquement le cas d'utilisation 3 (un client CORBA accède et invoque un EJB), mais aussi aux aspects ayant trait au portage de JOnAS sur RMI/IIOP (et donc au cas 2). L'analyse qui suit est tirée des spécifications EJB.

2.1.1.2. Invocation distante de méthodes

Pour assurer l'interopérabilité, les invocations distantes de méthodes doivent être faites au moyen du protocole IIOP 1.2. La traduction des types Java en messages IIOP doit être effectuée par le serveur d'EJB suivant les spécifications « Java to IDL mapping ». Cette traduction est implicite dans le cas des interfaces Java Remote si RMI/IIOP est utilisé pour leur invocation, et explicite dans le cas d'une invocation CORBA pure (typiquement, un client CORBA). La traduction des paramètres transmis par valeur dans les méthodes distantes RMI fait usage des objets par valeur CORBA.

Les exceptions système Java levées par le conteneur d'EJB sont traduites en leurs équivalents CORBA avant leur renvoi sous forme de message IIOP. La manipulation inverse est effectuée par l'ORB du côté client.

2.1.1.3. Transactions

La propagation du contexte transactionnel se fait suivant le mécanisme de propagation implicite décrit par les spécifications CORBA OTS 1.2 (*Object Transaction Service*, le système de gestion de transactions CORBA). Notamment, le format du contexte transactionnel est donné sous forme d'IDL, format obligatoire pour qui veut supporter l'interopérabilité transactionnelle.

Le côté client du système transactionnel est spécifié par JTA (Java Transaction Architecture), comme dans EJB 1.1.

Le conteneur d'EJB doit suivre le protocole de validation à deux phases. Un composant J2EE ne doit pas utiliser les transactions imbriquées dans les scénarii d'interopérabilité.

L'interopérabilité avec un conteneur sans support transactionnel (ou en tout cas sans support de l'interopérabilité transactionnelle) est aussi définie (propagation d'un contexte null), de même que le comportement du conteneur et les politiques transactionnelles à ce niveau.

2.1.1.4. Nommage

Le service de nommage doit permettre l'accès interopérable aux références des Home Beans. C'est un service *CosNaming* (spécifications du CORBA Interoperable Naming Service) au-dessus de IIOP.

Les applications J2EE clientes doivent utiliser un fournisseur JNDI pour accéder au service *CosNaming* afin d'accéder aux références des Home Beans.

Comme le *CosNaming* ne peut stocker que des objets CORBA, il est normal que d'autres services de nommage soient utilisés pour stocker les autres ressources (on peut penser ici aux connexions aux bases de données, par exemple).

Toutes les interfaces distantes des Beans, lorsqu'elles sont récupérées par un client, doivent être contraintes au moyen d'un « narrow » (méthode issue de CORBA).

2.1.1.5. Sécurité

L'interopérabilité en matière de sécurité est aussi spécifiée ; de manière générale, c'est encore une fois les spécifications CORBA qui sont reproduites, notamment concernant le contexte de sécurité et sa propagation.

2.1.2. L'exemple Inprise

2.1.2.1. Présentation

William Edwards et Salil Deshpande décrivent dans le « livre blanc » Intégration de CORBA et d'EJB (datant de janvier 2000) le résultat du travail effectué sur l'intégration et l'interopérabilité entre CORBA et EJB avec Inprise Application Server. Après une première présentation de CORBA, de l'unification CORBA/Java et des modèles de composant serveur CORBA et EJB, l'architecture de l'interopérabilité EJB – CORBA au sein d'IAS est révélée puis appliquée dans une série d'études de cas.

IAS semble implémenter toutes les spécifications EJB 2.0 et y avoir une conformité maximale sur le plan de l'interopérabilité. La raison en est évidente : IAS est bâti sur la noyau de l'ORB Inprise Visibroker. En conséquence, tous les cas d'interopérabilité EJB – CORBA sont possibles ; leur mise en œuvre est d'ailleurs envisagée dans de multiples cas utilisant des clients EJB, CORBA C++ ou CORBA Java. L'architecture IAS effectue tout de même des choix intéressants qu'il est bon de commenter.

2.1.2.2. Analyse des choix spécifiques à IAS

Le premier de ces choix est celui du renoncement au protocole RMI/JRMP classique (l'unique protocole préconisé par la précédente version des EJB), choix inévitable dû à l'architecture IAS au-dessus de CORBA (qui implique donc le protocole IIOP).

Les choix suivants, pour la plupart, sont la poursuite logique du premier, et prouvent dans la pratique la possibilité de superposer les spécifications EJB et CORBA. Ainsi, c'est l'implémentation d'OTS par Inprise qui est utilisée comme serveur de transactions ; par conséquent, les besoins transactionnels des spécifications EJB sont couverts. Il en va de même pour la sécurité, elle aussi basée sur les fonctionnalités sécurité de l'implémentation de CORBA Visibroker.

Dans le cas du nommage, l'existence d'un autre serveur de noms (a priori nécessaire pour stocker les ressources autres que les références distantes aux Beans) que le *CosNaming* n'est pas mentionné, mais une implémentation spécifique du JNDI est fournie au-dessus du *CosNaming*, implémentation dont il est facile de concevoir (étant spécifique) qu'elle puisse remplir tous les besoins des EJB et notamment fournir l'accès à (voire fédérer) des serveurs de noms adaptés aux ressources autres que les objets CORBA ou RMI/IIOP.

Cette architecture EJB au-dessus de CORBA permet la mise en œuvre de nombreux cas d'interopérabilité : un client Java RMI/IIOP, Java CORBA ou C++ CORBA (sur CORBA Visibroker Java ou C++) accède et utilise des EJB, ou un EJB accède et utilise des objets EJB (Java RMI/IIOP), Java CORBA ou C++ CORBA (sur CORBA Visibroker Java ou C++), et ceci simplement modulo les bonnes projections IDL et conversions de types.

2.1.2.3. Enseignements

Même si le cas JOnAS diffère considérablement du cas IAS (car JOnAS était au départ une implémentation RMI/JRMP de EJB 1.1, et non RMI/IIOP au-dessus de CORBA comme IAS), des enseignements utiles peuvent être tirés de l'interprétation que IAS fait des spécifications EJB concernant RMI/IIOP et l'interopérabilité J2EE et CORBA. Notamment, l'intention manifeste des concepteurs des EJB d'arriver à une complémentarité CORBA – EJB a été traduite par l'implémentation d'une solution EJB par API fines au-dessus d'une implémentation CORBA. Cette solution est certes excellente lorsque l'on part d'un ORB et de services CORBA, puisqu'elle apporte une souplesse inégalable en matière d'interopérabilité avec le monde CORBA, mais pas forcément si on a à la base un serveur d'EJB sur RMI/JRMP que l'on souhaite en premier lieu porter sous RMI/IIOP.

2.2. Reformulation des objectifs

2.2.1. Buts et intérêts de l'interopérabilité

CORBA est un ensemble de spécifications middleware (ORB et services de nommage, transactions, sécurité...) plus ancien qu'EJB et bien implanté en entreprise, puisque ce fut longtemps la seule solution pour développer des applications réparties en plusieurs langages. Dans un tel cadre, il est vraisemblable que des applications EJB nouvellement développées soient appelées par des applications clientes plus anciennes écrites en CORBA (donc vraisemblablement écrites dans un autre langage que Java, par exemple C++), mais intégrées dans le fonctionnement de l'entreprise. Ce cas se révèle être fréquent, et donne à l'interopérabilité EJB – JOnAS un intérêt crucial.

Le cas d'un client Java CORBA n'est pas à ignorer, puisqu'il peut permettre de réutiliser ou d'adapter une application CORBA Java préexistante (rappelons que Java CORBA est plus ancien de plusieurs années que le nouvellement spécifié Java RMI/IIOP) pour lui permettre l'accès à une nouvelle application EJB JOnAS sans sortir du monde CORBA.

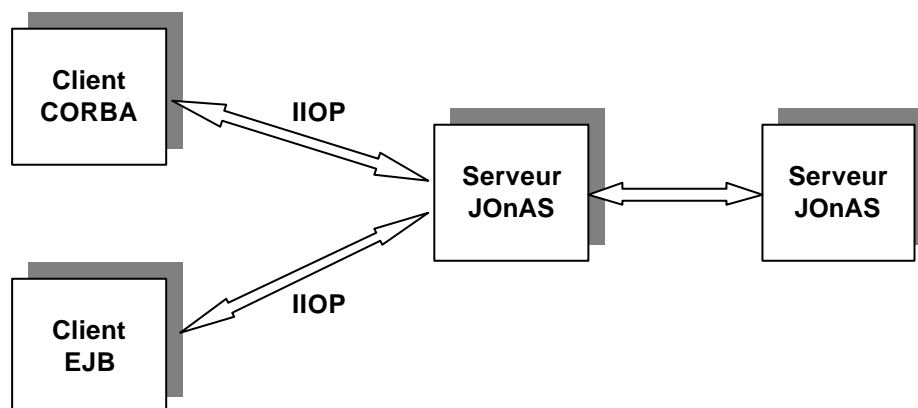
Il est enfin possible qu'un EJB ait besoins des fonctionnalités offertes par une application CORBA (Java ou non, dans des cas similaires à ceux décrits plus haut). Le travail à fournir pour permettre l'interopérabilité n'est à ce moment-là plus au niveau du serveur d'EJB, mais du Bean, et est donc plutôt à la charge du fournisseur de l'EJBean (*Bean Provider*). Ce travail n'est pas très différent de celui nécessité par des appels CORBA en Java standard (hors les contextes de services) ; nous nous contenterons donc de le survoler.

Rappelons que la compatibilité des implémentations CORBA, et donc aussi l'interopérabilité avec RMI/IIOP, est un point loin d'être acquis puisque les vendeurs considèrent rarement l'utilisation de produits concurrents ; de plus, ils implémentent souvent des fonctionnalités spécifiques qui peuvent entrer en conflit les unes avec les autres

2.2.2. Cas d'utilisation

Dans le cadre de l'interopérabilité CORBA – JOnAS, le cas d'utilisation considéré sera celui où un client CORBA appelle et utilise des EJB hébergés sur un serveur JOnAS RMI/IIOP.

Dans le cadre du portage de JOnAS vers une version RMI/IIOP (indispensable à l'interopérabilité CORBA selon EJB 2.0), les cas d'utilisation classique d'une application EJB seront également considérés.



Ces cas d'utilisation impliquent une interopérabilité JOnAS – CORBA à différents niveaux, qui sont décrits et explicités ci-après.

2.2.3. Protocole

Le premier niveau est l'interopérabilité des protocoles, qu'EJB 2.0 résout en requérant le protocole RMI/IIOP. Un premier objectif sera donc de choisir une implémentation du protocole RMI/IIOP après évaluation de ses fonctionnalités, de ses performances et de son interopérabilité effective avec d'autres ORBs CORBA courants. Le protocole choisi devra en effet être capable de supporter l'interopérabilité aux niveaux supérieurs (nommage, transaction, sécurité), avoir des performances décentes. Enfin, une application CORBA devra pouvoir voir les beans en tant qu'objets CORBA et appeler leurs méthodes de même. L'intégration à JOnAS du protocole choisi est le point final de cette étape.

2.2.4. Nommage

Le premier service à devoir être intégré à JOnAS RMI/IIOP est le service de nommage, qui est indispensable à la mise en œuvre des cas d'interopérabilité les plus simples. Ce sera bien sûr un *CosNaming*. Il faudra également mettre en place une couche JNDI au-dessus du *CosNaming* et trouver le moyen de stocker les ressources autres que les références de beans.

2.2.5. Transactions

L'interopérabilité transactionnelle implique l'intégration à JOnAS RMI/IIOP d'un système transactionnel compatible CORBA (par exemple en adoptant l'une des solutions envisagées dans les spécifications EJB 2.0) et le support du contexte transactionnel défini par le module CORBA CosTransactions. Les objectifs comprennent les transactions démarrées par le client, avec un client RMI/IIOP aussi bien que CORBA.

2.2.6. Sécurité

Il nous faut préserver les fonctionnalités actuelles de JOnAS, tout en permettant leur utilisation dans le cadre des cas d'interopérabilité avec le monde CORBA.

2.2.7. Prise en compte de l'existant

La prise en compte des versions existantes de JOnAS est un objectif à part entière. En effet, maintenir une seule source de JOnAS (tant au niveau du code que des scripts de lancement et de tests) et garder la cohérence des modifications apportées à JOnAS avec l'architecture existante est un point important de notre projet, ce qui rend la réalisation de la non régression des versions préexistantes plus complexe.

3. Choix d'un protocole

3.1. Besoins

L'adoption d'un protocole RMI/IIOP comme protocole d'invocation des méthodes distantes des EJBs est spécifiée par EJB 2.0 pour réaliser l'interopérabilité EJB – CORBA. Le choix d'un tel protocole RMI/IIOP afin de mettre en œuvre l'interopérabilité au niveau protocole entre un client CORBA et un objet distant (*Remote*) RMI/IIOP s'imposait donc comme la première étape de notre projet.

Nous avons évalué les protocoles candidats (Sun RMI/IIOP et David RMI) et les avons comparés aux protocoles de référence actuellement utilisés par JOnAS (Sun RMI/JRMP et Jérémie). Le protocole retenu est celui dont l'intégration avec JOnAS, les caractéristiques et les performances s'accordent le mieux.

3.2. Les protocoles testés

3.2.1. Présentation

- RMI/JRMP Sun JDK 1.3 est l'implémentation Java RMI de Sun, utilisant le compilateur de stubs RMIC de la JDK 1.3. C'est la nouvelle version du compilateur RMIC v1.1 utilisé aujourd'hui dans JOnAS, et son évaluation peut fournir des arguments en faveur de la mise à jour de JOnAS RMI.
- RMI/JRMP Sun JDK 1.3 mode 1.1 utilise le compilateur RMIC avec l'option `-v1.1` ; c'est le protocole utilisé par la version actuelle de JOnAS, d'où la nécessité de son évaluation.
- RMI/IIOP Sun 1.3 est l'implémentation RMI/IIOP de référence (par Sun et IBM). Ce protocole est RMI/IIOP et permet donc l'interopérabilité avec CORBA : c'est un candidat pour notre évaluation.
- RMI Jérémie est la personnalité RMI de l'ORB Jonathan de France Télécom et est utilisé dans la version courante de JOnAS, ce qui rend son évaluation d'un intérêt certain.
- RMI/IIOP David est l'implémentation de RMI/IIOP au-dessus de David, qui est la personnalité CORBA de Jonathan. Ce protocole est l'une des innovations de Jonathan 3.0, et est l'autre candidat à l'intégration de RMI/IIOP avec JOnAS.

3.2.2. Caractéristiques

Les protocoles évalués ont des caractéristiques diverses (protocole sous-jacent, vendeur...) qui leur donnent divers avantages et désavantages. Voici une description des principales caractéristiques qui les différencient, et le tableau de leur présence ou non dans chacune des implémentations :

- Garbage Collector (GC) distribué : Le Garbage Collector (ramasse-miettes) récupère l'espace mémoire occupé par des objets qui ne sont plus utilisés (vers lesquels ne pointe plus aucune référence) et revient donc à une désallocation mémoire automatique. L'un des avantages de l'implémentation Java RMI originelle est qu'elle maintenait cette fonctionnalité de Java dans le cadre de la programmation distribuée (ce qui est loin d'être trivial) : c'est le garbage collector distribué.
- Objects By Value (OBV) : c'est l'une des principales nouveautés de CORBA 2.3, qui permet le passage d'objets par valeur (et non plus seulement par référence), et qui fait partie de l'interopérabilité CORBA – RMI/IIOP. Son intérêt est évident.
- Passage de paramètres implicite : cela consiste à passer dynamiquement lors de l'invocation de toute méthode distante des paramètres supplémentaires, typiquement les contextes des différents services (contexte transactionnel, de sécurité...) en CORBA. Cette fonctionnalité est quasi indispensable pour le support de la sécurité et des transactions dans JOnAS.
- Intégration à JOnAS : il s'agit de la facilité de la configuration et de l'intégration de l'ORB RMI/IIOP à JOnAS. Comme la personnalité Jérémie de Jonathan est déjà intégrée, celle de sa personnalité cousine David est rendue plus aisée.
- Optimisation locale : elle consiste à optimiser l'appel de méthode distante lorsque l'appelant et l'appelé sont en réalité dans la même JVM.

	GC distribué	Params. implicites	OBV	Intégration à JOnAS	Optimisation locale
RMI/JRMP Sun 1.1	oui	si stubs patchés	pas de sens	réalisée	non
RMI/JRMP Sun 1.3	oui	non	pas de sens	non	non
RMI/IIOP Sun 1.3	non	non	oui	non	oui
RMI Jérémie	non	oui	pas de sens	réalisée	oui
RMI/IIOP David	non	oui	oui	facilitée	oui

3.3. Performances

3.3.1. Programme d'évaluation

3.3.1.1. Description

Nous avons comparé les performances en invocation de méthodes distantes des différents protocoles. Pour cela, nous avons conçu un programme de mesure de performances simple qui permet de mesurer les temps d'appel de méthodes distantes avec en paramètre un échantillon significatif des trois grandes catégories d'éléments susceptibles de participer à une invocation distante : les références d'objets distants (i.e. les implémentations de l'interface *java.rmi.Remote*), les objets par valeurs et les valeurs (i.e. les entiers...). Nous dédions un cas d'étude spécial à la mise en évidence de l'optimisation intra-JVM des protocoles Jonathan. Ce programme sera lancé au cours de plusieurs campagnes d'évaluation de performances suivant divers scénarii.

3.3.1.2. Méthode de mesure

Les mesures sont de l'ordre de quelques dizaines de microsecondes. Cependant, le chronomètre de la JDK est connu pour ne pas être fiable dans de telles conditions. C'est pourquoi nous l'avons remplacé par un chronomètre natif, plus précis (développé par Guillaume Rivière dans le cadre d'un stage Evidian).

Il faut s'attendre à ce que certaines de nos mesures soient plus ou moins erronées, du fait d'autres processus systèmes, de la surcharge (*overhead*) du réseau et même de la surcharge générée par notre propre programme. Afin d'obtenir des résultats pertinents, il est ainsi nécessaire d'utiliser des outils statistiques afin d'éliminer les valeurs trop peu vraisemblables.

3.3.1.3. Appels distants mesurés

Les invocations distantes des méthodes suivantes sont mesurées:

- Void op() : sans paramètre
- Void op(int)
- Void op(long)
- Void op(short)
- Void op(string)
- Void op(Bar) : Bar est une référence d'objet distant (*Remote*)
- Void op(Vector) : paramètre objet
- Void op(array(int)) : "gros" paramètre
- Void op(int, string, Vector, array(int), Bar) : paramètres de plusieurs types

La méthode distante qui permet d'évaluer l'optimisation intra JVM est la suivante:

- void opBar(int n)

Elle appelle n fois la méthode distante op() (qui ne fait rien) de la référence d'objet distant Bar, située dans la même JVM que le serveur. Deux valeurs de n sont utilisées: n = 10 pour une valeur "réaliste" et n = 1000 pour mettre davantage en évidence l'optimisation en question.

3.3.2. Résultats

Nous n'avons eu à notre disposition la version 3.0 de Jonathan (et donc le protocole David RMI/IIOP) qu'à partir de la deuxième semaine d'avril. Ce retard a gravement nuit au déroulement du projet, et nous a obligé à trouver des solutions temporaires ou à nous orienter vers des sujets indépendants du protocole (qui sont rares, malheureusement). Voilà pourquoi les tests ci-dessus ont été effectués dans un premier temps sur tous les protocoles sauf David RMI, et que le protocole utilisé dans une première période comme protocole RMI/IIOP de choix pour la suite des opérations fut celui de Sun.

Les résultats détaillés sont disponibles en annexe, nous n'en présentons ici que l'analyse. Globalement, trois groupes de performances sont à distinguer, qui correspondent aux trois spécifications de protocole utilisées dans les versions testées du modèle de programmation RMI, à savoir RMI/JRMP, Jérémie et RMI/IIOP.

- RMI/JRMP a les meilleurs résultats sur les types non distants (typiquement, tout objet java non distant, ainsi que les paramètres non objet: entiers, chaînes de caractères – dont le statut est un peu spécial – et même tableaux). Cela n'est pas étonnant, puisque c'est le protocole "natif" du modèle de programmation RMI, mais n'a visiblement pas d'optimisation intra-JVM. On notera que la version 1.1 de ce protocole est meilleure avec les paramètres non objet, tandis que la version 1.3 prend le dessus dès lors que les paramètres sont pur objet (non distant).
- Les performances de Jérémie l'amènent ensuite, à mi-chemin entre les protocoles JRMP et les protocoles IIOP. Mais ses performances en passage de références d'objets distants en paramètre sont de manière

étrange les plus mauvaises de toutes. Toutefois, son optimisation intra – JVM est un puissant argument que nos benchmarks mettent parfaitement en évidence.

- Les protocoles au-dessus de IIOP, quant à eux, prennent l'avantage lors du passage en paramètre de références d'objets distants, mais sont en queue de peloton dans le cadre des objets par valeur (ce qui paraît logique : la version originelle de ce protocole ne les prenait en effet pas en compte).
- L'optimisation intra JVM des deux personnalités de Jonathan apparaît très nettement, et fait son office.

Jonathan 3.0 David RMI montre des performances similaires à l'implémentation RMI/IIOP de la JDK. Même si cette dernière a l'avantage d'être l'implémentation de référence et donc d'avoir une plus grande compatibilité en pratique avec d'autres CORBA et protocoles RMI/IIOP, les fonctionnalités de David RMI enfoncent le clou en faveur de la solution Jonathan : outre l'optimisation lors d'invocations locales, le passage de paramètres implicites prévu d'origine nous sera très utile lors de la mise en œuvre de l'interopérabilité transactionnelle et de la sécurité. L'intégration actuelle de son cousin Jérémie avec JOnAS est enfin un argument de poids ; et dans l'hypothèse où JOnAS au-dessus de David RMI remplacerait JOnAS Jérémie, il est bon de savoir que les performances n'en seraient pas sacrifiées pour autant.

Toutefois, Jonathan 3.0 souffre d'un désavantage majeur, dont nous n'avons pas connaissance à l'avance : il n'a pas été disponible avant la deuxième semaine d'avril, et encore fut-ce en version alpha, dont nous fûmes les testeurs. C'est pourquoi la solution Sun fut dans un premier temps utilisée comme protocole RMI/IIOP de choix, et que les travaux se sont orientés vers une partie indépendante du protocole : le développement du multi-protocole.

3.4. Interopérabilité

3.4.1. Tests effectués

A terme, l'objectif est d'obtenir un client CORBA appelant un serveur JOnAS fonctionnant avec un protocole RMI/IIOP. Il est donc judicieux de tester sur des exemples simples l'interopérabilité des protocoles RMI/IIOP à notre disposition avec différentes implémentations de CORBA. Puisque JOnAS est toujours en mode serveur, nous pouvons nous contenter de tester des clients CORBA appelant des serveurs RMI/IIOP.

Les protocoles RMI/IIOP testés sont :

- RMI/IIOP de Sun ;
- La personnalité RMI de David : David RMI.

Les implémentations de CORBA testées sont les suivantes :

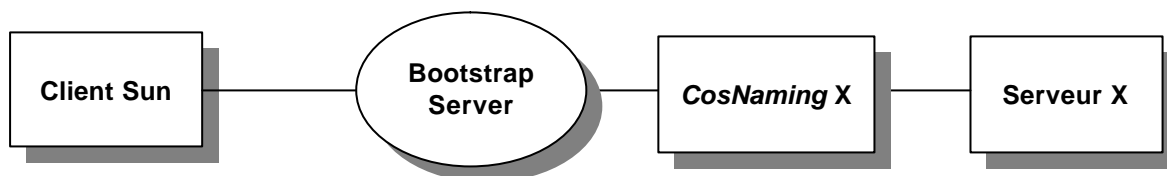
- JavaIDL : c'est l'implémentation de Sun qui est livrée avec la JDK standard ;
- David ;
- Orbacus 4 : une implémentation payante de CORBA (en essai libre) qui permet l'utilisation de deux langages : Java et C++. Nous n'avons utilisé que Java.

Le programme utilisé pour passer les tests d'interopérabilité est très simple: il s'agit en effet du classique « Hello World ! » adapté à l'invocation de méthodes distantes. Un code source unique est utilisé côté serveur, seules les propriétés passées à la JVM changent selon l'implémentation de RMI/IIOP désirée. Il en est de même pour CORBA côté client.

La méthodologie suivie pour créer les tests est calquée sur celle qui sera utilisée lors de l'écriture d'un client CORBA pour JOnAS :

1. écriture de l'interface Remote et de son implémentation du côté RMI/IIOP et compilation.
2. génération des sources des classes d'interposition RMI/IIOP à l'aide du Sun `rmic -iiop` et compilation.
3. génération de l'IDL CORBA correspondant à l'interface Remote RMI/IIOP par Sun `rmic -idl`.
4. génération du code source des classes d'interposition CORBA à l'aide du compilateur IDL du vendeur CORBA et compilation.

Le *CosNaming* utilisé pour chaque test est de préférence celui du serveur, c'est à dire soit celui de Sun soit celui de David. En théorie le choix du *CosNaming* n'importe peu car ils sont censés être standards et interchangeables. En pratique nous nous sommes aperçus qu'un *BootstrapServer* devait être utilisé pour que le client Sun JavaIDL puisse opérer avec le *CosNaming* de David. Le *BootstrapServer* se place entre un *CosNaming* d'un tiers et le client JavaIDL, à qui il permet de connaître les services CORBA non Sun disponibles :



D'autre part, Orbacus ne sait pas utiliser le *CosNaming* de David, mais heureusement David fonctionne avec celui d'Orbacus.

3.4.2. Résultats

Le tableau suivant présente les résultats de l'exécution du programme de test décrit ci-dessus dans différents cas d'interopérabilité. La colonne de gauche (resp. 1^{ère} ligne) indique l'ORB qui est utilisé du côté client (resp. serveur). La valeur « ok » indique que le test s'est déroulé correctement.

Client \ Serveur	Sun RMI/IIOP	David RMI/IIOP
Sun RMI/IIOP	ok	ok
Sun Java IDL	ok	ok
Orbacus	ok	Echec
David RMI/IIOP	ok	ok
David CORBA	ok	ok

Le serveur Sun RMI/IIOP a parfaitement fonctionné avec tous les clients CORBA testés. En revanche un client Orbacus ne parvient pas à appeler un serveur David RMI : l'appel distant commence correctement mais ne se termine jamais. Ceci peut être gênant car cela signifie qu'une version de JOnAS sur David RMI ne peut pas être appelé par des clients Orbacus.

3.5. La solution Sun

3.5.1. Intégration

3.5.1.1. Besoins

Les raisons de l'intégration de la solution Sun en attendant Jonathan 3.0 sont de trois ordres : tout d'abord, le programme d'évaluation de performances a été écrit dans l'environnement de programmation JOnAS, d'où la nécessité d'intégrer à JOnAS tous les protocoles que l'on veut tester. Ensuite, elle nous permettait d'aborder les problèmes spécifiques aux spécifications RMI/IIOP, par exemple : RMI/IIOP impose l'utilisation de *PortableRemoteObject.narrow()* au lieu d'un *cast* Java avant de pouvoir utiliser l'objet distant. Enfin, l'intégration d'un protocole RMI/IIOP était tout de même l'étape zéro du projet, sans laquelle bien peu de pans de l'interopérabilité pouvaient être abordés.

3.5.1.2. Modifications de JOnAS

Pour réaliser l'intégration de RMI/JRMP et de Jérémie au sein d'une même source dans sa version actuelle, JOnAS utilise le préprocesseur JPP, qui permet notamment les commandes *#ifdef*, *#define* et *#endif* du préprocesseur du langage C. Nous l'utilisons de même pour l'intégration du protocole RMI/IIOP de Sun.

Le moyen le plus courant de créer une classe distante (pouvant être connectée à l'ORB) dans le modèle de programmation RMI est de la faire hériter d'une classe particulière :

- *java.rmi.UnicastRemoteObject* en RMI/JRMP
- *javax.rmi.PortableRemoteObject* en RMI/IIOP
- *org.objectweb.jeremie.libs.binding.moa.UnicastRemoteObject* en Jérémie.

Les sources de JOnAS font appel à une classe *RemoteObject* spécifique, qui utilise celle de RMI/JRMP ou de Jérémie selon la version de JOnAS à générer à l'installation, choisie par la valeur d'une variable d'environnement, *OBJECTWEB_ORB*. Nous avons en conséquence modifié *RemoteObject* pour qu'il utilise le *RemoteObject* de RMI/IIOP dans le cas où *OBJECTWEB_ORB* prend la nouvelle valeur *RMIIOP*. Il y a aussi quelques *Makefiles* prenant cette même variable en compte à modifier (essentiellement pour lancer la compilation des stubs avec le compilateur associé au nouveau protocole s'il est choisi), et le script de lancement du serveur de nom.

3.5.1.3. Premières évaluations de l'intégration

En premier lieu, il faut compiler JOnAS et l'installer en version RMI/IIOP. Cette manipulation génère une arborescence d'installation qui comprend les scripts de lancement, les exemples standards et un fichier jar RMIIOP_jonas.jar avec toutes les classe de JOnAS. Cette première étape, nécessaire pour disposer de l'environnement JOnAS en mode RMI/IIOP, a été franchie avec succès.

Ensuite, le programme d'évaluation de performances et son utilisation constituent un bon test, lui aussi réussi, comme on l'a vu plus haut. A ce niveau, l'intégration du protocole RMI/IIOP de la JDK 1.3 de Sun avec l'environnement de programmation JOnAS est donc réalisée de manière effective.

3.5.2. Evaluation des fonctionnalités réelles

3.5.2.1. Besoins

Il s'agit désormais de savoir précisément ce que l'on peut faire avec le RMI/IIOP de Sun, et ce que l'on ne peut pas. Un certain nombre de questions restent en suspend, notamment au niveau du passage implicite de paramètres et de la gestion du multi-protocole par cette implémentation. En effet, même si elle ne dispose pas officiellement du passage implicite de paramètres et des intercepteurs qui s'y associent, il est possible que des classes internes et non documentées fournissent des moyens (*hooks* et *callbacks*) de le mettre en œuvre ; le fait que le protocole IIOP prévoit cette fonctionnalité (il permet en effet d'insérer des contextes de services dans les paramètres d'invocation) plaide dans ce sens. A ces fins, nous avons entrepris des recherches dans le code source de l'implémentation de référence de J2EE (*Java 2.0 Enterprise Edition*, dont les EJB font partie) D'une manière générale, l'utilisation que la J2EE fait du protocole RMI/IIOP nous renseigne sur ses possibilités et sur la manière d'en exploiter les fonctionnalités.

Nous ne décrivons pas les recherches elles-mêmes, mais seulement leurs résultats.

3.5.2.2. Intercepteurs et contextes de services

Après recherches, il s'avère que la J2EE réimplémente ou rajoute bien des choses en comparaison de l'implémentation de RMI/IIOP faite dans la J2SE. L'un de ces ajouts est la présence d'un POA (*Portable Object Adapter*, composant CORBA côté serveur qui gère les objets CORBA) et une nouvelle implémentation de l'ORB RMI/IIOP qui le prend en compte : *com.sun.corba.ee.internal.POA.POAORB* (rappelons que le préfixe *com.* désigne les classes internes chez Sun). L'ORB utilisé par le serveur EJB de la J2EE est sa classe fille *com.sun.enterprise.iiop.POAEBORB*. Dans son source, un certain nombre des lignes d'import de classes sont commentées par la ligne suivante : « *Ces classes sont des API internes à l'implémentation du POA RMI/IIOP, parce qu'il n'y a pas pour l'instant d'API publiques pour les intercepteurs* ». Ainsi, nous voilà fixés quant aux fonctionnalités de passage de contextes « officielles » de l'implémentation de référence de RMI/IIOP, fut-ce celle fournie par la J2EE. Quant à la possibilité d'utiliser directement le POAORB et ses méthodes, elle impliquerait de tirer à soi une bonne partie des classes internes de la J2EE (en effet, le POAORB utilise à lui seul toutes les classes du package *com.sun.corba.ee.internal.POA*). Or ceci nous est prohibé, ne serait-ce que parce que la distribution des classes de la J2EE avec JOnAS est interdite (par licence) et que le client EJB devrait utiliser une J2EE et non plus une J2SE. En conséquence, l'utilisation des intercepteurs de la J2EE pour accéder au passage implicite des contextes de services est abandonnée.

A l'extrême limite, le patchage automatique des stubs afin de permettre le passage de paramètres implicites est envisageable (c'est fait ainsi dans la version RMI de JOnAS), mais reste lourd et peu flexible (on peut difficilement traiter le cas de plusieurs contextes de services par appel).

3.5.2.3. Gestion des protocoles

La J2EE peut-elle supporter le multi-protocole ? Comment gère-t-elle la possibilité d'utiliser d'autres protocoles que RMI/IIOP ? Après recherches dans le code source, il s'avère que la J2EE dispose d'un système de gestion de protocoles RMI au-dessus de n'importe quel protocole sous-jacent, permettant de les interchanger dès lors que l'interface *com.sun.enterprise.ProtocolManager* est convenablement implémentée. Cette interface est implémentée par la classe *com.sun.enterprise.iiop.POAProtocolManager* pour le protocole RMI/IIOP de la J2EE. Toutefois, il y a un unique ProtocolManager par serveur d'EJB J2EE, et donc pas plus d'un protocole supporté simultanément ; cela est démontré par la structure de la classe *com.sun.enterprise.Switch*, qui référence les principaux composants de la plate-forme J2EE de Sun et qui ne pointe que vers un unique ProtocolManager.

Enfin, nous avons la certitude que le multi-protocole n'est pas supporté dynamiquement par l'implémentation de référence de la J2EE, et que cette fonctionnalité serait un plus indéniable pour JOnAS. Le support d'autres protocoles RMI est convenablement géré, mais différent de celui de JOnAS.

3.6. Multi-protocole

L'intégration d'un nouveau protocole dans JOnAS peut s'envisager de deux manières : la première, statique est simple à mettre en œuvre, alors que l'autre, plus dynamique, offre une plus grande souplesse qui se paye par sa complexité d'implémentation.

La version actuelle de JOnAS peut utiliser deux protocoles pour la communication entre les clients (qui peuvent être une application indépendante, un applet, un bean, etc.) et le serveur :

- RMI, qui utilise le protocole propriétaire Sun : JRMP ;
- Jérémie: la personnalité RMI de l'*Object Request Broker* Jonathan.

Cependant, un seul protocole peut être utilisé dans un serveur JOnAS, soit RMI soit Jérémie et le choix doit être effectué au lancement. L'ajout du support d'un protocole IIOP peut donc logiquement être effectué selon le même principe : on obtiendrait une version de JOnAS pouvant être lancée selon trois modes différents correspondants aux protocoles supportés. Cependant cette solution offre une interopérabilité limitée : un serveur JOnAS configuré pour utiliser le protocole IIOP permettrait à des clients CORBA de s'y connecter mais les clients RMI/JRMP ou Jérémie ne le pourraient pas.

C'est pourquoi nous avons envisagé une interopérabilité plus forte visant à obtenir une version du serveur JOnAS auquel des clients utilisant des protocoles distincts et non compatibles entre eux peuvent se connecter. Pour des raisons détaillées en fin de partie, nous avons finalement abandonné cette seconde solution et privilégié la première. Cependant nous présentons ci-après les pistes que nous avons étudiées pour réaliser cette interopérabilité (Il s'agit d'un résumé de l'ébauche des spécifications que nous avons élaborées).

3.6.1. Eléments de JOnAS concernés

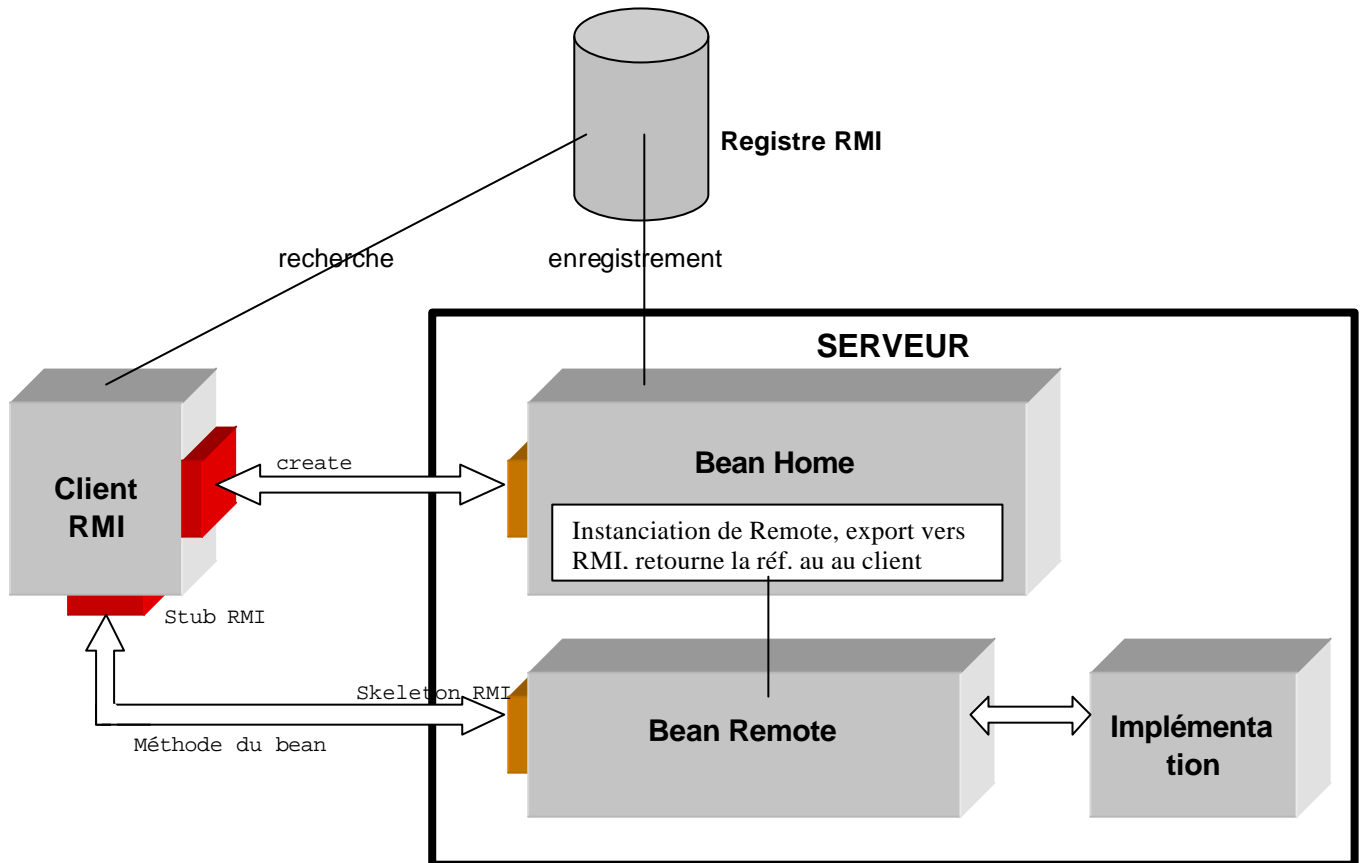
Cette partie établit un état des lieux des éléments de JOnAS, et plus généralement d'un serveur d'EJB, concernés par le support simultané de plusieurs protocoles.

- **Le service de nommage** : le support simultané de plusieurs protocoles implique la présence de leurs serveurs de noms associés en tant que *providers* JNDI. Le service de nommage (d'une manière générale, c'est le JNDI) est utilisé dans JOnAS :
 - par les clients pour établir un premier contact avec le serveur ;
 - par les beans qui souhaitent accéder à leur environnement ou obtenir diverses ressources (connexions a des bases de données par exemple) ;
 - par des composants de JOnAS tels le service d'administration distante qui s'enregistre dans le service de noms.

Les objets précédemment enregistrés dans un seul serveur de noms doivent désormais l'être (en règle générale) dans le serveur de nom du protocole associé. Ceci implique une révision de la gestion du nommage dans JOnAS.

- C'est la **publication des beans vers le monde extérieur** qui est la plus concernée par le multi-protocole simultané. Il convient de rappeler comment, dans la version actuelle de JOnAS, un bean est créé par le fournisseur de bean, déployé au sein de JOnAS et accédé par le client, ceci afin de mieux appréhender ce que le support de plusieurs protocoles modifie dans le schéma actuel.
 - Le fournisseur de bean (*Bean provider*) doit fournir plusieurs classes pour créer un bean :
 - L'interface Remote du bean qui contient la définition de toutes les méthodes métier (*Business methods*) du bean.
 - L'interface Home du bean qui définit les méthodes permettant la création ou la recherche de beans.
 - Une classe qui contient l'implémentation des méthodes métier et des méthodes de création de et de recherche de bean.
 - Avant de déployer un bean dans JOnAS il faut utiliser l'utilitaire GenIC qui génère à partir des classes du bean deux classes d'interposition qui serviront d'intermédiaires entre le client et le bean.
 - La première classe générée par GenIC est une implémentation de l'interface Remote du bean (on dit la classe Remote du bean). Elle reçoit les appels des méthodes métier par le client et les délègue (après pré traitement) à la véritable implémentation du bean.
 - La seconde classe générée est une implémentation de l'interface Home du bean. Son rôle est, entre autres, la gestion de la création et destruction des instances de la classe Remote du bean, et elle constitue un point d'entrée pour le client. Pour un bean donné cette classe est instanciée une seule fois et enregistrée dans le serveur de noms.

- Le **client** accède à un bean grâce au service de nommage du serveur qui lui permet d'obtenir une référence sur la Home générée par GenIC. Ensuite il appelle une méthode `create()` sur l'interface Home obtenue pour créer une nouvelle instance du bean. Le schéma suivant résume comment un client accède à un bean en supposant que le serveur JOnAS a été configuré pour utiliser le protocole RMI :



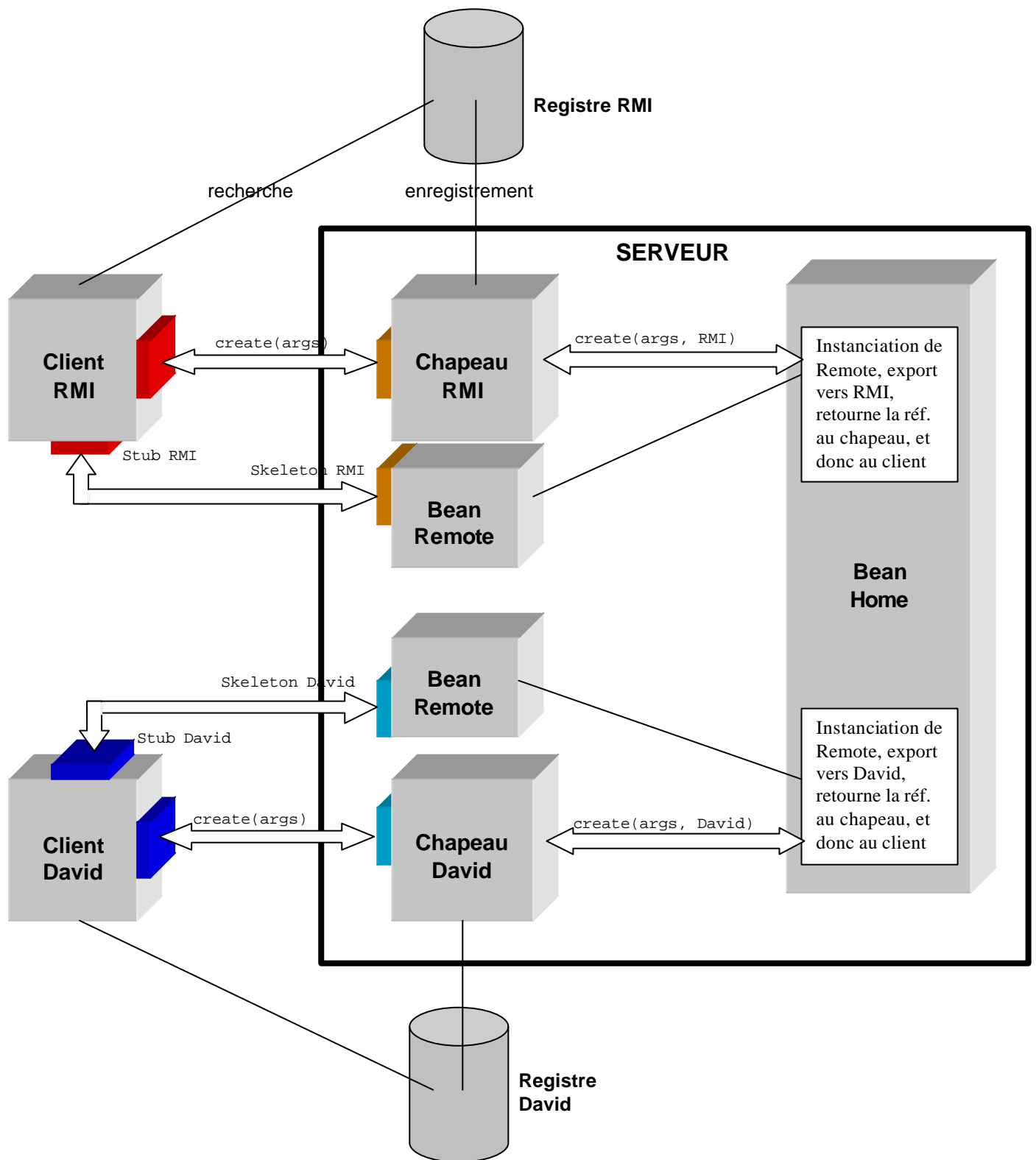
3.6.2. Etude d'un cas : export multiple d'un bean

Afin d'avoir une gestion optimale du support du multi-protocole, les ressources sont dynamiquement allouées en fonction des protocoles des clients :

- Lorsque la Home est contactée pour créer un bean, elle doit retourner une instance de l'interface Remote du bean qui supporte le protocole utilisé par le client pour communiquer avec la Home (et uniquement ce protocole).
- Lorsque la Home doit renvoyer un bean existant, elle doit lui ajouter si nécessaire le support du protocole client.

La classe Home des beans doit donc être capable de déterminer le protocole des clients qui l'accèdent. Pour ce faire, il suffit d'interposer des « chapeaux », un pour chaque protocole, entre le client et la Home. La Home n'est plus un objet distant (elle n'est plus exportée sur les ORB) et n'est plus enregistrée dans le JNDI. Ce sont les chapeaux qui jouent ce rôle : chaque chapeau est exporté sur l'ORB de son protocole et est enregistré dans son service de nommage avec le nom de la Home. Chaque chapeau se contente de déléguer les appels de méthodes à la véritable Home en lui transmettant son protocole en paramètre supplémentaire. Ainsi, lorsque la Home reçoit un appel, elle sait quel protocole est utilisé et peut donc agir en conséquence : si par exemple elle doit créer un bean elle l'exportera sur l'ORB du client.

Le schéma suivant illustre ces mécanismes dans le cas où un client RMI et un client David créent un bean en utilisant sa Home.



3.6.3. Abandon du multi-protocole

Nous n'avons pas terminé l'étude du multi-protocole simultané dans JOnAS car, après avoir étudié notre ébauche de spécification, l'équipe JOnAS a jugé sa mise en œuvre trop complexe et ne correspondant pas à la philosophie JOnAS : la simplicité. Il est vrai qu'un certain nombre de points techniques non triviaux restaient à résoudre (notamment l'appel en cascade de beans) et continuer dans cette voie nous aurait éloigné du cadre du stage tel qu'il était prévu. Nous nous sommes donc orientés vers une intégration simple de David dans JOnAS, ce qui n'était pas possible lorsque nous étudions le multi-protocole puisque nous ne disposions pas encore de David. C'est d'ailleurs cette absence de David qui nous a poussé à étudier le multi-protocole, ce que nous n'aurions sans doute pas fait dans le cas contraire.

4. Intégration de Jonathan 3.0

4.1. Besoins

Nous avons vu que l'intégration d'un protocole RMI/IIOP à JOnAS est la première étape indispensable à l'interopérabilité avec le monde CORBA. Maintenant que le protocole David RMI de Jonathan 3.0 a été choisi, il nous faut donc créer une nouvelle version de JOnAS au-dessus de David RMI. A cette fin, et après réception de Jonathan 3.0, il nous faut étudier les aspects techniques des spécificités RMI/IIOP et du service de nommage, avant de valider à la fois la nouvelle version mais aussi la non régression des anciennes.

4.2. Réception de Jonathan 3.0

Jonathan est loin d'être étranger au monde JOnAS, puisque tous deux font partie de la même communauté Open Source ObjectWeb et que JOnAS dispose déjà d'une version au-dessus de la personnalité Jérémie (implémentation originale de RMI au-dessus de GIOP) de Jonathan 2. Cependant, Jonathan 3.0 introduit plusieurs nouvelles fonctionnalités qui n'ont plus rien à voir avec ses versions précédentes. Il s'agit bien sûr du support du protocole RMI/IIOP par sa personnalité CORBA nommée David, mais aussi de son nouveau système de configuration à base de fichier au format XML spécifique.

Ce système permet de charger en mémoire des instances de n'importe quelles classes disposant d'une *factory* et d'en fournir les paramètres d'instanciation. La totalité de l'initialisation du noyau de Jonathan tient ainsi dans un même fichier XML : aussi bien les ORBs David et Jérémie que les *marshallers* CORBA ou la configuration de la gestion des services pour David comme pour Jérémie s'y retrouvent. La prise en main de ces possibilités n'est pas aussi évidente que celle du précédent (un simple fichier de propriétés Java), et on verra qu'il n'est pas exempt de bugs. Mais sa très grande souplesse d'utilisation (que nous mettrons à l'épreuve plus loin avec succès) et sa véritable simplicité une fois que son principe a été compris en font un des atouts de Jonathan.

Il faut enfin signaler que la réception des binaires de Jonathan 3.0 n'eut lieu que début avril, parce qu'elle n'était pas prête avant ; quant aux sources, dont l'utilité nous fut majeure pour la poursuite et la correction des bugs, elles ne nous furent livrées que début mai. Ces retards ont été très préjudiciables au bon déroulement du projet, car l'évaluation et l'intégration de David RMI à JOnAS en étaient réellement l'étape zéro. Par ailleurs, les multiples bugs rencontrés au cours de nos utilisations de David aussi bien dans son ORB, dans son compilateur de stubs que dans son système de configuration, inévitables dans une version alpha, ont mobilisé nos énergies. Mais la disponibilité des programmeurs de Kelua ne s'est jamais démentie, ni leur promptitude à corriger les bugs que nous avons isolés.

4.3. Intégration

L'intégration de la nouvelle version de Jonathan dans JOnAS s'est déroulée en plusieurs étapes. Dans un premier temps, nous avons modifié l'environnement de développement afin de produire une troisième distribution de JOnAS comme nous l'avions déjà fait pour l'intégration de RMI/IIOP (voir section 3.5.1). Nous souhaitons obtenir rapidement une version JOnAS sur David qui fonctionnait pour les services de base (Appel de bean par des clients essentiellement) afin de faciliter ensuite l'intégration des services plus complexes comme les transactions et la sécurité. Pour ce faire il a fallu revoir la gestion du nommage dans JOnAS afin de s'adapter au serveur de noms de David (C'est un *CosNaming* CORBA) qui n'est pas compatible avec les serveurs de noms utilisés jusqu'alors par JOnAS.

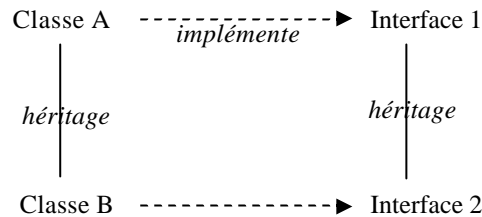
4.3.1. Intégration du protocole

Comme décrit dans la section 3.5.1 nous avons modifié des scripts de compilation afin d'avoir la possibilité de produire une troisième distribution de JOnAS utilisant David comme *middleware* sous-jacent. L'intégration du protocole lui-même est simple à réaliser puisqu'il suffit de modifier le fichier *RemoteObject.jpp* qui définit la classe de base pour tous les objets distants, pour que le serveur repose sur David pour communiquer avec ses clients.

Lors de la compilation de JOnAS tout objet *Remote* est fourni à un programme qui génère des classes d'interposition (des *stubs* et des *skeletons*) requise par l'ORB. Dans la version de JOnAS sur RMI/JRMP le générateur de stubs utilisé est *rmic* de sun, alors que pour Jérémie il s'agit d'un programme livré avec Jonathan. La version 3.0 de Jonathan inclut un compilateur de stubs pour David RMI et c'est donc logiquement ce dernier que nous avons adopté pour générer les stubs des classes *Remote* de JOnAS. Malheureusement ce programme s'est avéré défectueux ce qui nous a nouveau fait perdre un temps précieux. A chaque bug découvert nous

devions le signaler à France Télécom et attendre leur correction, plusieurs semaines ont été nécessaires pour obtenir une version fonctionnelle.

Nous avons donc essayé d'utiliser un autre compilateur de stub, mais le seul disponible, `rmic` avec l'option `-iiop`, ne fonctionnait pas non plus (Les stubs produits par `rmic -iiop` sont compatibles avec David). En effet, dans la configuration suivante, `rmic -iiop` génère une erreur (on souhaite générer les stub de la classe B) :

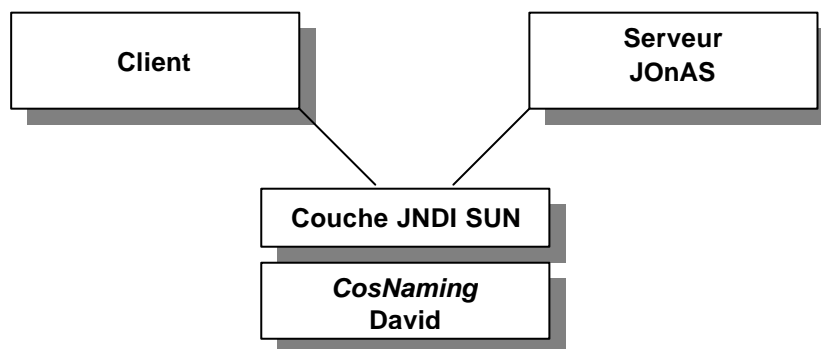


Il s'agit d'un bug identifié de la JDK 1.3 dont la correction est prévue dans la JDK 1.4. Malheureusement ce cas est plusieurs fois présent dans JOnAS et il n'est pas possible de le supprimer

Lorsque nous avons obtenu une version sans bugs du compilateur de stubs de David nous avons pu compiler JOnAS entièrement.

4.3.2. Intégration du service de nommage

L'emploi de David RMI en tant que *middleware* pour JOnAS implique des modifications dans le service de nommage. David RMI est une couche RMI au-dessus d'une implémentation de CORBA, son service de nommage est donc lui-même une implémentation de celui de CORBA, le *CosNaming*. Dans JOnAS le service de nommage est accédé grâce à l'interface standard JNDI, or il se trouve que le *CosNaming* de David ne possède pas de couche JNDI permettant un tel accès. Heureusement nous sommes parvenu sans trop de difficultés à utiliser la couche JNDI que Sun met à disposition dans la JDK pour fournir un accès JNDI à son propre *CosNaming*. Ceci est possible car les interfaces d'un *CosNaming* et d'un fournisseur JNDI sont standards.



Cependant cet « assemblage » n'a pas fonctionné dès les premiers essais, il a fallu que France Télécom modifie légèrement les sources de son *CosNaming* pour le rendre totalement compatible avec la couche JNDI de Sun (La couche Sun fait des suppositions étranges sur le format des IOR).

Toutefois le *CosNaming* de David ne peut pas se substituer entièrement au registre RMI utilisé dans les versions RMI et Jérémie de JOnAS car il ne peut référencer que des objets *Remote*. Or JOnAS utilise le registre RMI pour y stocker des objets non *Remote*, d'une part pour fournir aux Beans un moyen d'accéder à des ressources (Base de données, Messaging Service,...) et d'autre part pour permettre aux clients d'initier des transactions. D'un autre côté le registre RMI n'est pas capable d'enregistrer des objets *Remote* David (ce sont des objets CORBA). C'est pourquoi il est nécessaire de combiner l'utilisation de deux registres pour lancer un serveur JOnAS avec le protocole David RMI:

- Dans le *CosNaming* sont enregistrés les Beans, le service d'administration et le gestionnaire de transaction (Ce sont des objets *Remote*).
- Dans un registre RMI (on prendra celui de la JDK) sont enregistrées les références sur les ressources des beans et une référence sur le Current (Le Current permet l'initiation de transactions côté client).

L'utilisation de deux services de nommages complique légèrement l'écriture d'un client, car ce dernier peut en effet être amené à rechercher des objets dans chacun des deux registres, et doit donc créer un accès JNDI sur le *CosNaming* et sur le registre RMI. Pour créer un accès à un service de nommage via le JNDI il faut créer un contexte initial qui offre des méthodes d'exploration :

```
InitialContext ictx = new InitialContext() // Création du contexte
Object obj = ictx.lookup("nom") // Recherche d'un objet
```

On remarque qu'aucune information concernant le service de nom à accéder n'est fournie au constructeur de *InitialContext* : c'est parce qu'elles sont situées dans un fichier nommé *jndi.properties* qui doit être stocké dans un répertoire référencé dans le Classpath du programme. Par exemple si l'on souhaite créer un accès JNDI sur un registre RMI qui tourne sur la machine toto et qui écoute sur le port 2001, le fichier *jndi.properties* doit contenir :

```
java.naming.factory.initial    com.sun.jndi.rmi.registry.RegistryContextFactory
java.naming.provider.url       rmi://toto:2001
```

Cette méthode utilisée jusqu'à présent par les clients JOnAS ne peut pas fonctionner lorsque l'on doit utiliser deux registres puisque le fichier *jndi.properties* ne peut en décrire qu'un seul. Il existe cependant une solution à ce problème qui consiste à passer les informations de *jndi.properties* au constructeur de *InitialContext*(). Ainsi, il est possible de créer plusieurs contextes différents. On remarque qu'un client JOnAS doit nécessairement accéder au *CosNaming* (pour trouver les beans) et optionnellement au registre RMI (pour démarrer des transactions). C'est pourquoi nous conseillons aux clients JOnAS de procéder de la manière suivante pour gérer les services de nommages :

- Créer un fichier *jndi.properties* qui pointe sur le *CosNaming* du serveur JOnAS. Ainsi, pour rechercher des beans le code client se contente de créer un contexte initial sans passer d'arguments au constructeur.
- Lorsqu'ils souhaitent initier une transaction, ils créent un contexte en fournissant explicitement l'adresse du registre RMI :

```
java.util.Properties prop = new java.util.Properties();
prop.put("java.naming.factory.initial",
"com.sun.jndi.rmi.registry.RegistryContextFactory");
prop.put("java.naming.provider.url", "rmi://localhost:2001"); // par exemple
InitialContext ictx = new InitialContext(prop); // pointe sur rmi registry
```

Le même problème se pose du côté serveur car lui aussi doit accéder aux deux registres, et il utilisait jusqu'à présent un fichier *jndi.properties*. Nous avons adopté la même démarche que pour le côté client, mais le registre RMI a été privilégié par rapport au *CosNaming* car cela impliquait moins de changements dans JOnAS et sa configuration. Cependant d'après les spécifications EJB, un bean doit avoir accès aux autres bean (à leur home en fait) et ce en créant un contexte initial sans paramètres. Or un tel contexte créé au sein d'un serveur JOnAS pointe sur le registre RMI, qui ne contient pas les beans. La solution consiste à enregistrer dans le registre RMI des références sur les beans qui, lors de leur résolution, font appel au *CosNaming* (grâce au mécanisme de *factory* de JNDI).

4.3.3. Validation

À ce niveau d'intégration nous avons bénéficié de toute la batterie de tests développés par l'équipe JOnAS afin d'effectuer une première validation de JOnAS sur David RMI. Il a toutefois été nécessaire de les adapter pour qu'ils fonctionnent avec toutes les distributions de JOnAS, et ce à différents niveaux :

- Les scripts de lancement ont été modifiés car David RMI requiert le passage de plusieurs paramètres à la JVM.
- Les tests eux-mêmes ont dû être révisés car *PortableRemoteObject.narrow()* n'était jamais utilisé alors que le protocole RMI/IIOP requiert son emploi. Le code permettant d'initier des transactions côté client est différent selon les distributions de JOnAS et il a été isolé dans une classe séparée afin de rendre les tests plus génériques.

Ces tests nous ont été très utiles car ils nous ont permis d'identifier des problèmes auxquels nous n'aurions jamais pensés compte tenu de notre modeste connaissance du code JOnAS. Par exemple dans une des classes génériques de JOnAS dont héritent tous les Entity Bean, il existe une méthode nommée *isIdentical()*, dont le but est de comparer deux beans. Celle-ci se contentait de comparer les objets, ce qui ne fonctionne plus en RMI/IIOP car un *narrow* préalable des objets à comparer est nécessaire.

Après debugage, les tests ne faisant pas appel aux services de transaction ou de sécurité ont été passés avec succès. Il reste quelques tests qui échouent mais nous n'avons pu identifier les problèmes faute de temps.

Tous ces tests ont montré que le nouveau serveur JOnAS sur David RMI pouvait être correctement appelé par des clients RMI/IIOP, et nous a permis d'étudier son interopérabilité avec CORBA.

4.3.4. Non régression

Toutes les distributions de JOnAS étant issues d'un code commun, il est nécessaire de vérifier que les modifications apportées à JOnAS pour intégrer David n'ont pas affecté les versions antérieures, à savoir la version RMI/JRMP et la version Jérémie.

4.3.4.1. RMI

A ce stade, les sources de la version RMI n'étaient quasiment pas affectées, et les tests de JOnAS cités plus hauts se sont déroulés correctement, ce qui montre que les modifications des scripts de JOnAS et de la gestion des services de nommage pour David, ont été réalisées correctement.

4.3.4.2. Jérémie

Les sources de la version Jérémie ne sont pas plus affectées que celles de RMI, mais l'intégration à JOnAS de Jonathan 3.0 pour le mode RMI/IIOP implique pour une question de cohérence le passage à la nouvelle version de Jérémie. Il est vrai qu'à ce stade, il n'est pas encore décidé si la version Jérémie de JOnAS sera conservée une fois la version David RMI en place ; mais le travail effectué à ce niveau ressemble beaucoup à celui qui devra être réalisé pour la version David RMI : configuration et mise en œuvre des services de transactions et de sécurité, mais en bénéficiant d'un environnement déjà intégré à JOnAS dans son ancienne version.

La principale différence avec la version précédente de Jérémie est le nouveau système de configuration de Jonathan par fichier au format XML, qui permet à l'utilisateur de maîtriser entièrement le chargement du noyau du *middleware*. Par exemple, l'utilisateur peut rajouter les services CORBA de son choix, choisir ou réimplémenter toute l'architecture de sérialisation/désérialisation, voire modifier intégralement la configuration de l'ORB.

Notre travail s'est résumé à configurer les services CORBA de support des transactions et de la sécurité pour Jérémie. Un service CORBA est représenté par son *Handler* spécifique, qui référence les deux intercepteurs (envoi et réception) et sérialise et désérialise à leur intention leur contexte de propagation privé. Le *Handler* du service transactionnel de Jérémie est déjà configuré dans le noyau ; il n'y a donc qu'à l'enregistrer comme le service CORBA pour Jérémie d'identificateur zéro comme l'exige la convention, et à lui spécifier les intercepteurs de JOnAS pour Jérémie, dont l'initialisation doit être réécrite pour l'occasion ; elle en devient d'ailleurs plus simple. Et tout cela ne prend que trois lignes de XML. Par contre, le *Handler* de la sécurité n'est pas fourni ; il faut donc le réécrire en partant de la version actuelle de celui de JOnAS et lui fournir une *factory* avant d'écrire les lignes XML qui expliquent comment Jonathan doit l'instancier. Puis, là aussi, il faut l'enregistrer comme service CORBA pour Jérémie et lui spécifier ses intercepteurs, réécrits pour l'occasion à partir de la version actuelle.

Enfin la nouvelle version de Jérémie a été testée avec la batterie de tests JOnAS dédiés. Un problème inattendu est apparu ici : certains identificateurs de services CORBA étaient incompatibles les uns avec les autres dans Jonathan 3.0 ! Une fois isolé et signalé, ce bug fut rapidement corrigé par l'équipe de Kelua. Les tests ont dès lors tous été passés avec succès.

4.4. Interopérabilité CORBA

Disposant d'une version fonctionnelle de JOnAS utilisant le protocole RMI/IIOP, nous avons pu envisager d'appeler ce serveur par des clients pur CORBA. Pour ce faire il est nécessaire de commencer par rappeler le processus habituel du développement d'un client et d'un serveur CORBA :

1. Dans un premier temps on écrit l'interface des objets distants que l'on souhaite héberger dans le serveur CORBA. Ces interfaces sont écrites dans le langage IDL (*Interface Definition Language*) spécifié par CORBA.
2. Ensuite on projette ces interfaces côté serveur dans un langage cible (ex: Java, C++, ...) afin de générer des classes de base, qu'il faut étendre pour implémenter les méthodes définies dans l'interface IDL.
3. On projette ces mêmes interfaces IDL pour le côté client vers un langage éventuellement différent de celui utilisé pour implémenter les objets côté serveur. On obtient des classes d'interposition et d'« aide » qui rendent possible l'écriture d'un client.

5. Transaction et sécurité

5.1. Introduction

5.1.1. Les spécifications EJB

5.1.1.1. Architecture

Le support des transactions distribuées est une fonctionnalité majeure des EJB, qui permet la mise à jour selon les principes ACID (atomicité, cohérence, isolation, durabilité) de plusieurs bases de données réparties sur des sites éventuellement distants, ceux-ci pouvant disposer de serveurs d'EJB de vendeurs différents.

Le fournisseur de Beans a le choix entre deux méthodes de gestion des transactions : les transactions gérées par le Bean (*Bean-managed transaction demarcation*) ou par le conteneur (*container-managed*). Dans le premier cas, le fournisseur du Bean doit implémenter lui-même la démarcation des transactions au moyen de l'interface *javax.transaction.UserTransaction* (interface du Java Transaction API disposant des classiques *begin()*, *commit()* et *rollback()*). Dans le deuxième, c'est le conteneur de Beans qui démarque les transactions selon la politique que lui assigne le développeur de l'application finale au moyen du script de déploiement, et qui peut être d'exécuter le travail du Bean dans une transaction initialisée préalablement par le côté client, dans une nouvelle transaction initialisée par le conteneur, ou sans transaction.

EJB spécifie l'API JTA comme interface entre un gestionnaire de transaction et les autres éléments impliqués dans un système de traitement des transactions distribuées, qui sont les programmes applicatifs, les gestionnaires de ressources et le serveur d'application. JTA forme donc le côté client transactionnel ; le côté serveur transactionnel (coordination distribuée des transactions) n'est pas spécifié. Dans les deux cas de gestion des transactions, c'est au conteneur et au serveur d'EJB d'implémenter les protocoles transactionnels de bas niveau nécessaires, comme le passage du contexte transactionnel ou le protocole de validation à deux phases distribué.

5.1.1.2. Interopérabilité CORBA

CORBA possède également un service de transaction, l'*Object Transaction Service* (OTS), dont l'interopérabilité JOnAS / CORBA doit tenir compte. Les spécifications EJB ne détaillent pas l'interopérabilité des EJB avec CORBA sur le plan des transactions mais elles présentent deux conditions qu'un serveur d'EJB souhaitant l'interopérabilité transactionnelle doit satisfaire :

- La propagation du contexte transactionnel doit se faire suivant le mécanisme de propagation implicite décrit par les spécifications CORBA OTS ;
- Le format du contexte transactionnel est donné par l'IDL *CosTransactions::PropagationContext*.

Ces deux points important méritent quelques explications : lorsqu'un client démarre une transaction, il doit fournir au serveur des informations sur cette transaction. Celles-ci sont stockées dans une structure appelée contexte transactionnel, transmise au serveur lorsque le client invoque une méthode distante. Les données du contexte sont transmises en plus des données de l'invocation et ce de manière invisible pour le client (on parle donc de transmission implicite). Après traitement le serveur renvoie le contexte transactionnel, éventuellement modifié, au client, et ce selon le même mécanisme.

Pour qu'un client CORBA et un serveur EJB puissent coopérer transactionnellement, il est nécessaire que les contextes transactionnels qu'ils s'échangent aient un format commun et que l'échange lui-même soit standard. C'est pourquoi les spécifications EJB imposent aux serveur EJB interopérables CORBA d'utiliser le contexte transactionnel spécifié par CORBA et de le transmettre en utilisant les mécanismes standards du protocole IIOP (EJB est plus récent que CORBA c'est donc à lui de s'adapter).

Les spécifications EJB n'introduisent pas d'autre contraintes quant à l'interopérabilité transactionnelle. En particulier les serveurs d'EJB interopérables ne sont pas obligés d'avoir un serveur transactionnel qui respecte les interfaces de l'OTS (dont la projection en Java est appelée *Java Transaction Service* ou JTS), et ils sont libres dans la mise œuvre de l'interopérabilité. Les spécifications évoquent quelques pistes comme une implémentation non Java de l'OTS, un pont entre un gestionnaire transactionnel existant et le protocole OTS ou encore un adaptateur (*wrapper*) OTS autour d'un gestionnaire transactionnel existant.

5.1.2. Les API JOnAS TM et CORBA OTS

Cette partie a pour but de présenter les API du service transactionnel de JOnAS (que l'on appellera désormais JOnAS TM) et de le comparer à l'OTS afin de mieux appréhender les modifications à apporter pour supporter l'interopérabilité.

5.1.2.1. API JOnAS TM

Le côté client transactionnel du service transactionnel JOnAS respecte l'interface JTA comme cela est requis par les spécifications EJB. On a en particulier les classes suivantes :

- **Current** : cette classe est utilisée directement par le client ou les Beans pour initialiser, valider ou annuler une transaction.
- **Transaction** : c'est la représentation locale d'une transaction éventuellement distribuée. Elle dispose d'un identificateur, d'un Subcoordinator et éventuellement d'un Coordinateur distant.
- **SubCoordinator** : cette classe a pour but la gestion (validation, annulation) des ressources locales d'une transaction. Elle est enregistrée comme une ressource distante dans son coordinateur distant s'il existe.

Le côté serveur transactionnel se caractérise par les classes suivantes :

- La classe **Control** est le coordinateur et terminateur distant d'une transaction distribuée. Elle enregistre des ressources (ex : des subcoordinators) qu'elle peut valider ou annuler à la demande.
- **TransactionFactory** qui crée le **Control** d'une nouvelle transaction.

Le contexte transactionnel de JOnAS transmet l'identificateur de la transaction en cours et son **Control**, s'il existe (s'il a déjà été créé par le serveur d'EJB).

5.1.2.2. Comparaison entre TM et OTS

Il existe plusieurs différences entre le TM JOnAS et l'OTS et ce à plusieurs niveaux :

- Dans les versions actuelles de JOnAS, le TM utilise les protocoles RMI/JRMP ou Jérémie, alors que l'OTS utilise exclusivement IIOP (puisque'il est formé d'objets CORBA).
- Du côté client transactionnel, le TM utilise les interfaces JTA (**UserTransaction** pour les clients, **TransactionManager** dans le conteneur), l'OTS utilise le pseudo objet **Current** (qui ressemble à une fusion de **UserTransaction** et **TransactionManager**).
- Du côté serveur transactionnel, le TM utilise des interfaces spécifiques à JOnAS très proches de celles du JTS. Notamment, les noms des méthodes sont exactement les mêmes. Mais les types des paramètres et des exceptions diffèrent, puisqu'ils sont dans JOnAS des types et des exceptions spécifiques à JOnAS et non ceux spécifiés par l'OTS, même s'il est évident que leurs rôles sont les mêmes. On peut observer le même phénomène du côté client du gestionnaire de transactions, avec l'existence d'un **Current** spécifique à JOnAS et qui n'est pas celui de l'OTS. En conséquence, l'architecture transactionnelle JOnAS n'est pas telle quelle compatible avec OTS, et il faudra éventuellement l'adapter.
- Le TM utilise une version non standard de *org.omg.CosTransactions.PropagationContext* : une interface vide, implémentée par un contexte transactionnel spécifique à JOnAS. Sa propagation se fait par le biais de stubs patchés dans la version RMI et d'intercepteurs standards CORBA au niveau de l'ORB Jérémie dans la version Jérémie. L'OTS utilise le *org.omg.CosTransactions.PropagationContext* standard (une classe finale) et le propage comme un contexte de service CORBA par les intercepteurs standards CORBA au niveau de l'ORB IIOP.

5.1.2.3. Les cas RMI/JRMP et Jérémie

Il faut noter que si l'on implémente l'API OTS dans JOnAS, totalement ou en partie, cette implémentation ne pourra pas être présente dans les versions RMI/JRMP ou Jérémie de JOnAS puisqu'elle implique des objets CORBA et donc un ORB IIOP. La seule exception sont les classes « localement contraintes » (le **Current**) ou qui ne sont pas des objets CORBA (l'identificateur de transactions **otid_t** et le **PropagationContext**). Mais ces classes ont souvent des objets CORBA distants dans leurs attributs ou dans les signatures de leurs méthodes, ce qui rend impossible leur utilisation hors du cadre RMI/IIOP ou CORBA. Pour arriver à des versions RMI – Jérémie et RMI/IIOP différentes, on peut se servir du système de preprocessing JPP déjà utilisé dans JOnAS, ou encore des méthodes statiques de la classe **Env** qui donnent la version de JOnAS à l'exécution.

5.2. Différentes architectures possibles

Une architecture transactionnelle JOnAS RMI/IIOP compatible avec CORBA vérifie

- Le cas client applicatif CORBA appelant un serveur d'EJB JOnAS, avec initialisation de la transaction côté client ou côté serveur d'EJB;

- Les spécifications de l'interopérabilité de EJB 2.0 (utiliser le contexte de propagation CORBA et le propager sur IIOP avec les mécanismes d'envoi implicite de contextes de services CORBA standards) ;
- Eventuellement le cas Bean d'un serveur JOnAS en cours de transaction appelant une application CORBA.

La partie serveur de transactions d'une telle architecture transactionnelle peut être conçue sur la base de plusieurs concepts différents, que l'on retrouve évoqués de manière similaire dans les conseils d'implémentation fournis par EJB 2.0 :

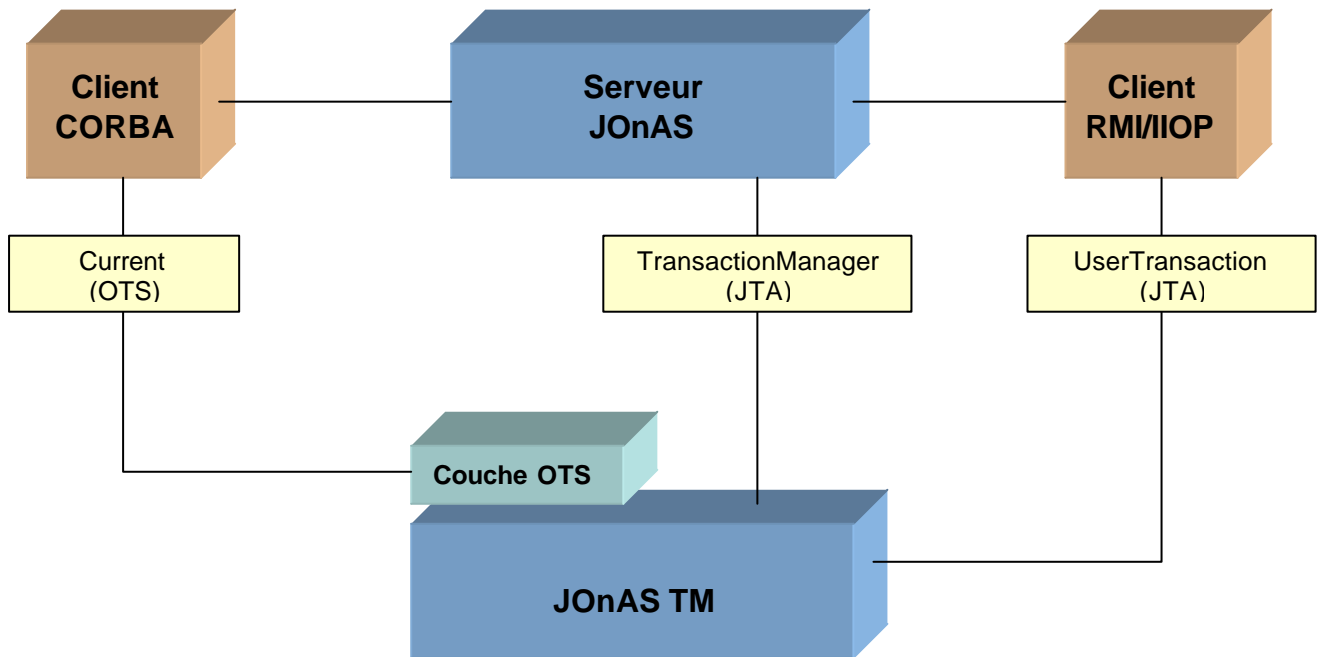
- Implémenter l'API JTS comme une couche fine au-dessus du TM JOnAS ;
- Faire une implémentation totalement CORBA du JTS et l'intégrer à JOnAS ;
- Rendre le TM JOnAS interopérable avec le mécanisme transactionnel de l'OTS.

L'API de la partie client, quant à elle, est spécifique au client utilisé: c'est l'API JTA dans le cas d'un client RMI/IIOP (bean ou application finale) comme le spécifie EJB 2.0, et c'est l'API CosTransactions::Current spécifiée par l'OTS dans le cas d'un client CORBA.

5.3. Implémentation d'une couche OTS au-dessus du TM

5.3.1.1. Présentation

Cette solution (schématisée ci-dessous) permet la réutilisation de l'API JOnAS TM existante pour l'écriture d'un OTS en implémentant l'API JTS comme une couche fine au-dessus du gestionnaire de transactions JOnAS. En pratique, le monde JOnAS (les clients RMI/IIOP ou les serveur d'EJB JOnAS) utilisera l'API JOnAS TM pour générer et démarquer les transactions, alors que le monde CORBA (un client CORBA ou des objets distants CORBA accédés dans le code des Beans) utilisera la couche OTS.



Les avantages de cette solution sont :

- Réutilisation de l'API JOnAS TM et de son implémentation actuelle ;
- Il n'y a pas à modifier de classe commune aux autres versions de JOnAS, mais seulement à en rajouter ;
- Obtention d'un OTS complet que l'on pourra à terme sortir de JOnAS et empaqueter séparément, à côté non négligeable fort apprécié par l'équipe JOnAS qui en souhaite un ;
- Compatibilité CORBA totale (les trois conditions peuvent être vérifiées)
- En pratique, le fonctionnement du mécanisme reste le même, y compris en CORBA, si ce n'est le passage éventuel mais transparent dans la couche JTS.

Au vu de ces nombreux avantages (la bonne intégration à JOnAS et la compatibilité totale n'en sont pas les moindres), nous nous sommes orientés dans un premier temps d'un commun accord avec l'équipe JOnAS vers

l'étude de cette solution. Le principe de son implémentation est que chaque classe CORBA du côté serveur transactionnel de l'OTS est implémentée comme un *wrapper* d'un unique objet JOnAS du type correspondant, auquel les appels à ses méthodes sont délégués. Les principaux problèmes d'implémentation issus de notre étude sont présentés ci-dessous.

5.3.1.2. Aspects techniques

Côté serveur transactionnel :

- La couche fine CORBA amène une part importante de traduction d'exceptions JOnAS en exceptions CORBA du côté serveur. Cette traduction est faite selon les règles spécifiées par EJB 2.0.
- Tous les types de paramètres des méthodes de la couche JTS doivent être convertis en leurs types JOnAS correspondants. Parmi ces paramètres se trouvent des objets OTS de contrôle distant de la transaction (Control, Coordinator et Terminator) qui sont traduits en les objets JOnAS TM équivalents qu'ils *wrappent*. Ceci est permis en référant l'équivalent JOnAS dans son *wrapper* OTS.
- De plus, les objets de contrôle distant JOnAS TM de la transaction retournés par les méthodes de l'API JOnAS TM doivent pouvoir être retournés par les objets JTS qui ont appelé les dites méthodes, et donc traduits en leur *wrapper* JTS associé. Cela n'est possible que par deux moyens : faire des tables de traductions enregistrant toutes les relations objet *wrapper* OTS - objet *wrapper* JOnAS TM, mais au prix d'une grande lourdeur et du problème de la destruction distribuée des relations inutiles (lors de la fin d'une transaction); ou rajouter des méthodes de renvoi du *wrapper* associé dans l'API des objets JOnAS TM. Cette dernière solution a été préférée car moins complexe.
- Le contexte de propagation est propagé indifféremment dans le monde CORBA et dans le monde JOnAS; il doit donc être compatible CORBA, c'est-à-dire être la classe finale `CosTransactions::PropagationContext`. Cette classe référence les objets JTS de contrôle distant de la transaction en cours, identifie la transaction en cours par un objet `otid_t` et dispose d'un champ `impl_specific_data` de type `any` dont l'usage est à la disposition du développeur du système transactionnel. De plus, le même contexte transactionnel doit fournir au monde JOnAS les interfaces JOnAS TM et au monde CORBA les interfaces OTS de contrôle de la transaction en cours, c'est-à-dire référencer le Control JOnAS en plus des Coordinator et Terminator CORBA. Ce peut être fait en mettant le Control JOnAS dans son champ `any` comme un objet distant CORBA (chose possible puisqu'il est RMI/IIOP). L'unification du contexte transactionnel de propagation implique l'unification des identificateurs de transaction : JOnAS en a deux (Otid dans le JOnAS TM et Xid dans le JTA, côté client) différents de celui de l'OTS (`otid_t`). Il faut garder le Xid pour l'interface JTA côté client, mais partout ailleurs (propagation et JTS) on utilise l'`otid_t` CORBA. La conversion entre les deux est facilitée par leur grande ressemblance.

Côté client transactionnel :

- L'implémentation du côté client transactionnel CORBA est aisée mais l'API (interface `Current` du JTS) empêche l'optimisation en cas de transaction locale : il faut créer un contrôleur distant de la transaction dès son début dans le client, et donc avoir accès à la `TransactionFactory` distante créatrice de tels objets `Control`.
- La version RMI/IIOP est la même que celle de la version JOnAS RMI (au contexte de propagation près).

5.3.1.3. Conclusion

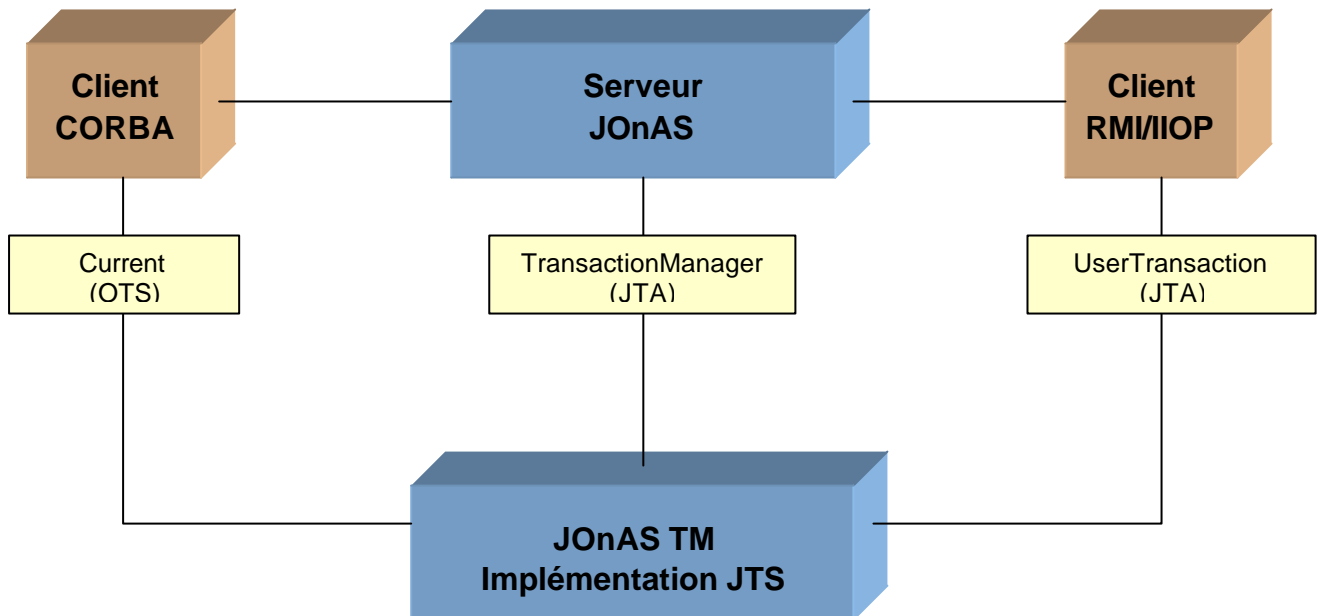
L'implémentation de cette solution a été quasiment terminée, mais s'est heurtée à la non disponibilité d'un compilateur de stubs sans bug (que ce soit `rmic -iiop` ou `jrmi` pour David), qui a empêché le test de la version de JOnAS réalisée même si la plupart des techniques spécifiques employées ont été testées avec succès. Nous avons poursuivi en parallèle le traitement des bugs de David RMI et de `jrmi`.

Toutefois, cette solution est trop complexe (traductions multiples d'exceptions et de types) et n'a finalement pas la meilleure intégration à la version actuelle de JOnAS (modifications des API du TM). De plus, ses performances et sa robustesse souffriraient de son architecture hybride. C'est pourquoi cette solution, riche d'enseignements toutefois, est abandonnée au profit de l'étude de la suivante.

5.4. Implémentation d'un OTS complet

5.4.1.1. Présentation

Afin d'éviter la complexité de la solution hybride, nous avons décidé d'étudier la solution de l'implémentation d'un OTS complet à partir de l'API JTS. En pratique, le gestionnaire de transactions présente une unique interface OTS utilisée de manière similaire par l'architecture client transactionnel spécifique à CORBA (i.e. CosTransactions::Current) et par celle spécifique à EJB 2.0 (i.e. l'API JTA) pour gérer la transaction courante. Le schéma suivant résume ces mécanismes :



Les avantages de cette solution sont :

- Plus simple que la solution hybride, moins de traductions requises (seuls sont à traduire les identificateurs de transactions entre l'OTS et le JTA pour un client RMI/IIOP) ;
- Grâce à la similarité extrême des API JOnAS TM et JTS, le code existant peut être réutilisé quasiment tel quel pour l'implémentation ;
- Obtention d'un OTS complet ;
- Interopérabilité CORBA totale (tous les cas d'utilisation couverts) ;
- Il n'y a plus à utiliser le champ spécifique à l'implémentation dans le contexte transactionnel CORBA (plus grande compatibilité CORBA).

Certains désavantages sont intrinsèquement liés à ces choix (moins de partage de code avec la version JOnAS RMI qui ne peut pas traiter des objets CORBA, intégration à JOnAS à considérer). Mais ces premières considérations semblant prometteuses, il fut décidé de l'étudier plus avant.

5.4.1.2. Aspects techniques

Le principe de l'implémentation de cette solution est que tous les mécanismes de JOnAS sont réutilisés moyennant leur adaptation aux API JTS. Voici les principaux problèmes qui sont issus de notre étude :

Côté serveur transactionnel :

- Le même raisonnement que dans la solution précédente est fait sur le contexte transactionnel (son unification est indispensable), mais ici le champ de type any n'est plus nécessaire.
- L'adaptation se résume la plupart du temps à de simples correspondances de types.
- Toutefois, il n'y a aucune possibilité de garder du code commun avec la version RMI de JOnAS à cause de l'incompatibilité totale entre les API du JOnAS TM et du JTS.
- L'intégration du serveur de transactions OTS à JOnAS est problématique, obligeant à de nombreuses modifications du code source d'autres parties de JOnAS, notamment lors du démarrage du service. Cela multiplie le nombre de fichiers utilisant le préprocesseur, ce qui est à éviter.

Côté client transactionnel :

- L'implémentation du côté client transactionnel CORBA est aisée, comme dans la solution précédente.
- L'implémentation JTA du côté client RMI/IIOP doit être réécrite en utilisant l'API JTS et non plus JOnAS TM. Cela impose un certain nombre d'adaptations, mais une certaine part de code commun avec la version RMI de JOnAS peut être préservée avec une architecture convenable.

5.4.1.3. Conclusion

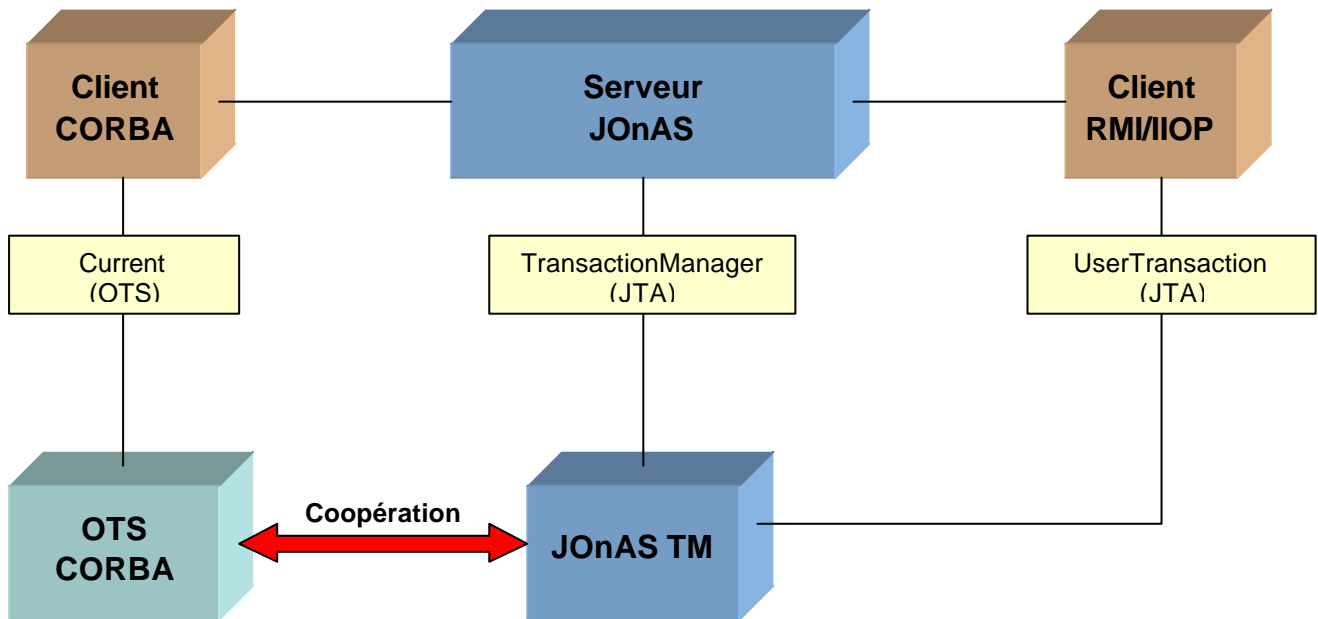
L'implémentation de cette solution a été terminée, mais pas son utilisation par JOnAS, qui n'aurait de toute façon pas pu être testée à cause des bugs du compilateur de stubs. Son intérêt va au-delà du sujet de l'interopérabilité, puisqu'elle aurait fourni à la communauté ObjectWeb la base d'un gestionnaire de transactions compatible OTS pouvant être utilisé séparément.

Mais l'absence de code commun avec la version RMI rend le suivi du JOnAS TM problématique, et l'intégration à JOnAS oblige des modifications de la version RMI de nombreux autres modules que le gestionnaire de transaction proprement dit. Ces deux points allant à l'encontre de la philosophie de JOnAS, l'équipe JOnAS a décidé avec nous d'écarter la solution d'un OTS complet, pour privilégier la dernière solution qui y semble a priori plus conforme, et qui est présentée dans les parties suivantes.

5.5. Implémentation de la coopération entre TM et OTS

5.5.1.1. Présentation

Afin d'obtenir une meilleure intégration au code existant de JOnAS et à sa version RMI, nous abandonnons l'idée que le JOnAS TM contrôle les transactions sur l'ensemble des deux mondes JOnAS et CORBA et considérons à présent la solution de l'implémentation de la coopération entre le JOnAS TM existant et un monde transactionnel CORBA contrôlé de manière externe. En pratique, une transaction initialisée dans le monde JOnAS (dans un client applicatif RMI/IIOP, dans le conteneur d'EJB ou dans un Bean) est contrôlée par le gestionnaire de transactions de JOnAS de la même manière qu'aujourd'hui, et une transaction initialisée par le monde CORBA (i.e. par un client CORBA dans le cas d'interopérabilité étudié) est contrôlée par un OTS spécifique au monde CORBA. Ceci est résumé par le schéma suivant :



5.5.1.2. Aspects techniques

Les avantages de cette solution sont :

- Réutilisation maximale de l'API, de l'implémentation et du mécanisme actuels du JOnAS TM
- Pas besoin d'écrire le côté client transactionnel CORBA (c'est à la charge de l'OTS externe)

Cependant, cette solution amène aussi une plus grande difficulté de mise en œuvre, puisqu'il faut disposer d'un OTS externe en plus du JOnAS TM et le mettre en œuvre. Mais la très bonne intégration à JOnAS en fait une solution à considérer de près. Le principe de son implémentation est que, dans le cas d'une transaction initialisée par un client CORBA, l'objet de contrôle de la transaction dans le monde JOnAS est enregistré comme ressource de l'objet de contrôle de la transaction dans le monde CORBA ; ainsi, la validation ou l'annulation de la transaction par le client CORBA se répercuteront sur les ressources du monde JOnAS qui y sont impliquées. Voici les principaux problèmes d'implémentation qui sont issus de notre étude :

- Le même raisonnement que dans la première solution est fait sur le contexte transactionnel (son unification est indispensable) et l'usage fait du *any* est similaire, à la différence que les champs dédiés aux objets OTS de contrôle de la transaction référencent le contrôle racine de la transaction dans le monde CORBA gouverné par l'OTS externe, alors que le champ *any* référence l'objet de contrôle par JOnAS de la seule partie de cette transaction qui recouvre le monde JOnAS.
- Afin de faire pouvoir enregistrer le Control JOnAS TM dans le Control OTS, on utilise une couche fine (*wrapper*) qui implémente l'interface OTS *Resource* et délègue l'exécution de ses méthodes au Control JOnAS, lequel peut par ce biais être enregistré comme une ressource OTS par le Control OTS.
- L'enregistrement du Control JOnAS comme une ressource du Control OTS se fait au moyen d'une fonction *recreate* de l'API JOnAS TM, jusqu'alors non utilisée par JOnAS, et qui est parfaitement adaptée à notre cas puisque ses spécifications (données par celles du JTS) précisent qu'elle peut être utilisée pour importer une transaction d'origine externe. Elle est appelée en lieu et place de sa fonction sœur *create* lors de l'arrivée (interception) d'un contexte transactionnel qui spécifie un contrôle de la transaction par un OTS externe mais encore aucun contrôle par le JOnAS TM de la transaction sous-jacente couvrant le monde JOnAS.
- Il n'y a effectivement rien à faire côté client transactionnel.
- Toutefois, l'interopérabilité transactionnelle dans le cas où un Bean appelle un objet CORBA n'est envisageable que dans le cas où le Bean démarque une transaction autour d'appels CORBA (qui couvre tout de même nombre de cas d'utilisation en pratique), mais du fait de notre utilisation « hors spécifications CORBA » que nous faisons du contexte de propagation transactionnel, le passage du monde JOnAS vers le monde CORBA ne peut pas se faire de manière transparente. La solution est de laisser le monde CORBA en aval gérer ses propres transactions, et d'en récupérer le contrôle au retour du contexte. Afin de signaler au monde CORBA l'existence préalable d'une transaction dans le monde JOnAS, le Bean doit utiliser un Current CORBA spécifique à JOnAS (afin qu'il soit compatible avec les intercepteurs JOnAS installés sur le serveur d'EJB) auquel des propriétés auront spécifié le gestionnaire de transactions OTS distant qui gère le monde transactionnel CORBA.

L'implémentation de cette solution a été terminée (sauf le cas d'appel vers CORBA) et les techniques spécifiques employées ont été testées avec succès. La simplicité de son mécanisme et son excellente intégration à la version RMI de JOnAS font d'elle le meilleur choix pour l'interopérabilité transactionnelle. Sa mise en œuvre et sa validation sont vues dans la section suivante.

5.6. Mise en œuvre de la coopération entre TM et OTS

Cette mise en œuvre comprend d'abord la mise en place et la configuration du service transactionnel pour David RMI, puis l'implémentation des solutions techniques particulières à notre cas, ensuite les tests standards de la version de JOnAS avec support des transactions ainsi réalisée et finalement les tests des cas d'interopérabilité CORBA – JOnAS.

Le service transactionnel pour David RMI se configure de manière similaire à celui pour Jérémie (voir section 4.3.4.2), sauf que le *Handler* transactionnel n'est pas présent dans Jonathan. Son écriture est directe en utilisant le patron standard fourni, à condition de prendre en compte le besoin d'une sérialisation CORBA IIOP. Toutefois, son instanciation par l'ORB David RMI requiert d'autres classes qu'il nous faut donc configurer correctement au préalable. Il s'avère que l'une d'elles est l'ORB IIOP, ce qui donne lieu à une tentative d'initialisation croisée qui rend impossible la poursuite de l'initialisation. Ce bug, une fois isolé et signalé, a été corrigé par Bruno Dumant de Kelua. De notre côté, nous avons mis en place une correction temporaire afin de poursuivre les tests. Le reste de la configuration du support transactionnel est identique à celle de Jérémie nouvelle version.

Le principal point technique de la mise en œuvre de notre solution est l'utilisation du champ du contexte transactionnel de type CORBA *Any* pour référencer l'objet de contrôle par le JOnAS TM de la transaction dans le monde JOnAS. Plusieurs solutions peuvent être envisagées. Celle retenue consiste à enregistrer le stub du Control JOnAS comme un objet distant CORBA (ce qu'il est, puisqu'il est RMI/IIOP) dans ce champ. Le test de cette méthode a dans un premier temps échoué, à cause d'un bug de Jonathan qui empêche l'enregistrement de *null* dans un *Any*. Nous avons soumis notre correction du code source à l'équipe Jonathan qui l'a intégrée.

Les batteries de tests standards vérifiant la non régression à JOnAS ont toutes été passées avec succès. Il s'agit ici de l'accomplissement d'un de nos objectifs les plus importants: la réalisation et la validation d'une version RMI/IIOP de JOnAS avec support des transactions.

Les tests d'interopérabilité avec CORBA ont demandé plus de travail. Il a fallu avant tout trouver une implémentation d'OTS afin de lui donner le contrôle des transactions dans le monde CORBA. Nous avons déjà pensé à celui d'Orbacus, disponible en Java et en C++; mais OOC, le concepteur d'Orbacus, s'est fait racheter par Iona Technologies et ne distribue plus son OTS. En conséquence, et faute de la compatibilité des services de

nommage d'OpenORB et de David, nous nous sommes rabattus sur celui de Jacorb (ORB tout Java, et donc d'intérêt moindre dans notre cas). La lecture de son code source montre qu'il utilise aussi le champ *Any* du contexte transactionnel, mais de manière redondante aux champs «standards», et que la gestion locale des transactions n'y accédait pas si elle en est l'initiatrice. Notre solution peut donc théoriquement fonctionner avec Jacorb, mais nous n'avons pas pu le vérifier faute de temps.

5.7. Mise en œuvre de la sécurité

La sécurité dans JOnAS ne concerne que l'aspect droits d'appel des méthodes de Beans (fondée sur des certificats). C'est certes un point important, mais la sécurité EJB (et aussi CORBA, dont elle s'inspire pour l'interopérabilité) comprend également de nombreux autres aspects : confidentialité, authentification... Nous n'allons cependant pas implémenter ces nouvelles fonctionnalités dans JOnAS, mais plutôt essayer de reproduire celles existantes avec le portage RMI/IIOP et de les intégrer harmonieusement à l'interopérabilité avec un client CORBA. La propagation du contexte de sécurité est primordiale ainsi que son intégration en tant que service CORBA, mais son format pourra être adapté aux besoins relativement légers de JOnAS, afin de préserver la simplicité et la bonne intégration avec la version Jérémie de la solution actuelle.

C'est pourquoi, plutôt que d'implémenter les multiples niveaux de sécurité spécifiés par CORBA et leurs interfaces *Replaceable* (pour la plupart par des «valeurs par défaut»), bien trop complexes pour les besoins actuels de JOnAS, nous allons conserver l'architecture existante et mettre en œuvre son utilisation par le monde CORBA. Ceci se fait en plusieurs étapes : d'abord configurer le service de sécurité CORBA de David RMI pour qu'il utilise celui de JOnAS et le tester avec JOnAS version RMI/IIOP, puis implémenter le côté client sécurité CORBA et enfin tester les cas d'interopérabilité CORBA.

Nous n'avons eu le temps de réaliser qu'une adaptation spécifique au monde Java RMI/IIOP, ce qui se fait en reprenant la configuration de la sécurité obtenue pour la version Jérémie de JOnAS telle quelle et qui passe effectivement les tests de sécurité standards de JOnAS.

Ce support de la sécurité n'est pas compatible CORBA hors Java, car le contexte de sécurité JOnAS est un objet Java et non basé sur une définition IDL CORBA, et est en conséquence sérialisé au moyen de la sérialisation Java et non de la sérialisation IIOP. Pour que ce soit possible, il faudrait adapter la version RMI/IIOP de JOnAS à un contexte de sécurité qui implémente en Java un DL équivalent, et utiliser un *marshaller* CORBA comme dans la configuration du service transactionnel (au lieu de celui de Jérémie).

6. Bilan

Conformément aux attentes exprimées dans nos objectifs, nous avons réussi à effectuer un portage de JOnAS sur RMI/IIOP et à le valider, tout en préservant une cohérence maximale avec les versions RMI et Jérémie de JOnAS ainsi que leur non régression. Les divers niveaux de fonctionnalités du protocole, du nommage, des transactions et de la sécurité y sont inclus. Un objectif secondaire atteint est le passage de Jonathan 2.0 à Jonathan 3.0 pour la version Jérémie de JOnAS.

Nous avons également rempli notre objectif de réalisation de l'interopérabilité JOnAS – CORBA, et ce non seulement au niveau protocole, mais encore aux niveaux plus élevés du nommage et des transactions. L'aspect sécurité a été mis en œuvre sous une forme privilégiant son effectivité à sa conformité aux spécifications CORBA.

Les perspectives en sont larges, ne serait-ce que parce que la version RMI/IIOP de JOnAS basée sur nos travaux doit être officialisée. L'interopérabilité avec le monde CORBA sera aisée à y intégrer, grâce à la légèreté des solutions envisagées. C'est pourquoi il est prévu que des membres de l'équipe JOnAS poursuivent nos développements.

L'essence même du sujet de ce projet de fin d'études nous a amené à découvrir, utiliser et finalement maîtriser un vaste domaine de compétences. En effet, non seulement nous nous sommes familiarisés avec le produit que nous avons à faire évoluer (JOnAS) et avec ses méthodes de développement concurrentiel, mais encore nous avons dû acquérir une certaine expertise des spécifications qu'il implémente. Ceci est également vrai pour le monde CORBA et ses aspects ORB, nommage et transactions, que le besoin d'intégrer finement l'ORB Jonathan nous a fait découvrir et comprendre.

Au final, nous avons apprécié la liberté d'action qui nous a été donnée dans un sujet au carrefour des technologies Java et applications distribuées les plus avancées, même s'il aurait peut-être mérité d'être davantage défrichée au préalable par l'équipe JOnAS. Enfin, les dépendances vis-à-vis de projets externes comme la JDK 1.4 ou Jonathan ont été un frein au bon déroulement de notre travail, mais ont multiplié les contacts humains et nous ont enseigné les conditions véritables d'un tel projet en entreprise et dans le monde Open Source. L'enrichissement sur le plan humain et technique qui en découle est par conséquent très positif.

Bibliographie et liens

□ Ouvrages

- Client/Server programming with Java & CORBA 2nd edition, aux éditions Wiley, par Robert Orfali et Dan Harkey
- Entreprise Java Beans, aux éditions O'Reilly, par Richard Monson – Haefel

□ Spécifications

- EJB 2.0 Proposed Final Draft, le 23 octobre 2000, par Linda G. DeMichiel, L. Umit Yalcinalp et Sanjeev Krishnan
- EJB to CORBA mapping 1.1, le 11 août 1999, par Sanjeev Krishnan
- Java to IDL mapping, juin 1999
- IDL to Java langage mapping, juin 1999
- CORBA V2.3, Octobre 1999
- CORBA Transaction Service

□ Tutoriaux

- CORBA : des concepts à la pratique, par Jean-Marc Geib, Christophe Gransart et Philippe Merle, Laboratoire d'Informatique Fondamentale de Lille

□ Articles techniques

- How Jonathan propagates a transactional context, mai 2000, par Kathleen Milsted
- When the Java and CORBA worlds collide, juin 2000, par Salih Ergul
- Intégration de CORBA et d'EJB : intégration et interopérabilité entre CORBA et EJB avec Inprise Application Server, janvier 2000, par William Edwards et Salil Deshpande
- RMI, IIOP et EJB, avril 2000, par Dave Curtis (Inprise Corporation)
- Implantation de la fonctionnalité RMI/IIOP dans l'infrastructure logicielle Jonathan, janvier 2001, par Kathleen Milsted et Bruno Dumant
- Java, Entreprise Java Beans and CORBA, novembre 2000, par Paul Harmon
- A Detailed Comparison of CORBA, DCOM and Java/RMI, 1998, par Gopalan Suresh Raj

□ Liens internet

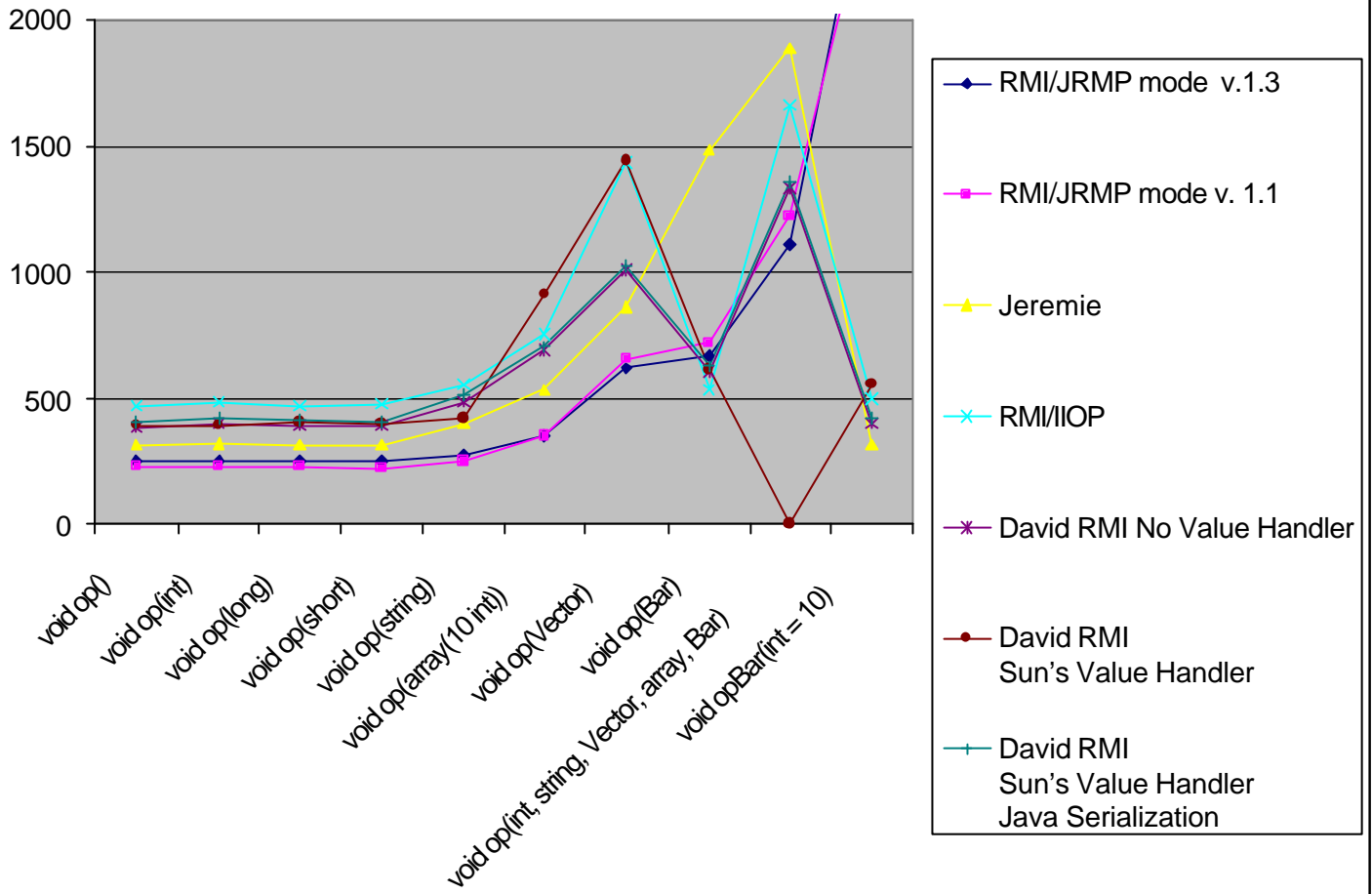
- BULL
 - BULL: <http://www.bull.com>
 - Evidian : <http://www.evidian.com>
- JOnAS
 - ObjectWeb : <http://www.objectweb.org>
 - JOnAS : <http://www.evidian.com/ejb/>
 - Jonathan : <http://www.objectweb.org/jonathan/>
- Api Sun
 - J2SE : <http://java.sun.com/products/jdk/1.3/docs/api/index.html>
 - J2EE : <http://java.sun.com/j2ee/j2sdkee/techdocs/api/index.html>
- Orbacus : <http://www.orbacus.com>
- OMG: <http://www.omg.org>
- Tutoriels
 - RMI/IIOP: http://java.sun.com/j2se/1.3/docs/guide/rmi-iiop/rmi_iiop_pg.html
 - The Java Tutorial : <http://java.sun.com/docs/books/tutorial/>
 - JNDI Tutorial : <http://java.sun.com/products/jndi/tutorial>

Annexe A : évaluation de protocoles

Le tableau suivant présente les résultats des tests performances (Temps d'appel en microsecondes) décrits dans la section 3.3 Il est ensuite représentés sous forme de graphiques.

			RMI/JRMP mode v.1.3	RMI/JRMP mode v. 1.1	Jeremie	RMI/IIOP	David RMI No Value Handler	David RMI Sun's Value Handler	David RMI Sun's Value Handler Java Serial.
Local	Inter JVM	void op()	245 / 415	225 / 395	315 / 520	470 / 640	385 / 605	390 / 600	405 / 605
		void op(int)	250 / 430	225 / 405	320 / 500	480 / 655	395 / 665	390 / 600	415 / 605
		void op(long)	250 / 430	225 / 390	315 / 500	470 / 645	390 / 630	405 / 590	410 / 590
		void op(short)	245 / 490	220 / 395	315 / 480	475 / 620	390 / 650	395 / 580	405 / 580
		void op(string)	270 / 500	250 / 465	400 / 645	550 / 745	480 / 860	420 / 655	510 / 795
		void op(Bar)	670 / 970	715 / 990	1485 / 1780	535 / 725	605 / 870	610 / 815	625 / 830
		void op(Vector)	615 / 930	650 / 995	860 / 1180	1435 / 1765	1010 / 1425	1440 / 1845	1020 / 1400
		void op(array(10 int))	350 / 605	345 / 650	535 / 815	755 / 990	685 / 1055	905 / 1210	705 / 1020
		void op(int, string, Vector, array, Bar)	1105 / 1475	1220 / 1610	1890 / 2155	1660 / 2025	1330 / 1745	ECHEC	1355 / 1740
	Intra JVM	void opBar(int = 10)	2720 / -	2540 / -	315 / -	495 (opt) 440 (no opt)	400 / -	550 / -	420 / -
	void opBar(int = 1000)	253000 / -	237000 / -	610 / -	1795 (opt) 117600 (no opt)	795 / -	785 / -	790 / -	
Remote	Inter JVM	void op()	770 / 1210	750 / 1145	990 / 1270	1115 / 1580	1090 / 1530	1100 / 1460	1100 / 1515
		void op(int)	770 / 1305	760 / 1190	995 / 1280	1090 / 1570	1095 / 1500	1095 / 1590	1100 / 1525
		void op(long)	780 / 1260	760 / 1185	1015 / 1260	1100 / 1595	1100 / 1525	1110 / 1500	1100 / 1555
		void op(short)	785 / 1220	755 / 1235	980 / 1320	1105 / 1550	1100 / 1540	1100 / 1495	1105 / 1580
		void op(string)	825 / 1385	810 / 1330	1085 / 1520	1240 / 1820	1230 / 1880	1210 / 1700	1240 / 1810
		void op(Bar)	1535 / 2170	1560 / 2145	3020 / 3580	1305 / 1855	1375 / 1880	1385 / 2060	1380 / 1830
		void op(Vector)	1405 / 1965	1460 / 2040	1720 / 2446	ECHEC	1940 / 2775	ECHEC	1960 / 2480
		void op(array(10 int))	980 / 1625	970 / 1500	1250 / 1905	1490 / 2080	1485 / 2350	1740 / 2385	1485 / 2110
		void op(int, string, Vector, array, Bar)	2385 / 2960	2470 / 3070	3790 / 4305	ECHEC	2580 / 3350	ECHEC	2615 / 3245
	Intra JVM	void opBar(int = 10)	3230 / -	3050 / -	990 / -	1090 / -	1090 / -	1470 / -	1100 / -
	void opBar(int = 1000)	256500 / -	237500 / -	1450 / -	2450 / -	1460 / -	1460 / -	1480 / -	

Client: Wake, Server: Wake



Client: Sundev, Server: Wake

