Université de Nantes- Faculté des Sciences
2, rue de la Houssinière
44322 Nantes Cedex 03 - France

Projet de DESS

CorbaTrace

Outils d'observation pour application répartie utilisant CORBA

Faculté des Sciences et Techniques
Université de Nantes
Année 2001/2002

étudiants : Florian Champalle
Audrey Jaccard
Etienne Juliot
Nicolas Lemoullec
Antoine Parra del Pozo

responsables : Christian Attiogbé
Philippe Lamarre

Université de Nantes- Faculté des Sciences
2, rue de la Houssinière
44322 Nantes Cedex 03 - France

Rapport de projet

Projet CorbaTrace

Groupe de projet		
Qualité	**Nom**	**Visa**
Développeurs	Florian Champalle	
	Audrey Jaccard	
	Etienne Juliot	
	Nicolas Lemoullec	
	Antoine Parra del Pozo	
Encadrants	Christian Attiogbé	
	Philippe Lamarre	

Auteur	Tous
Date de création	16/03/2002
Date de dernière modification	20/03/2002
Version	1.0

Diffusion	
Qualité	**Nom**
Groupe de projet	Tous les développeurs
Enseignants	M. Attiogbe, M. Lamarre
Directeur du DESS	M. Royer
Site Web	http://corbatrace.tuxfamily.org

Versions

Version	Date	Parties modifiées	Commentaires
1.0			Création du document

Remerciements

Nous tenons à remercier messieurs Philippe Lamarre et Christian Attiogbé pour tous leurs conseils tout au long du projet et leur disponibilité.

Nous tenons aussi à remercier les précurseurs du projet Vincent Tricoire et Frédéric Breton pour avoir initialisé le projet et pour nous avoir guidé dans les premiers pas de CorbaTrace.

Enfin nous remercions Anthony Gautier pour sa réactivité et son support technique, ainsi que Yann Tanguy, le doyen de la Faculté des Sciences de l'Université de Nantes pour avoir accepté de mettre le projet sous licence LGPL.

Sommaire

1 Introduction 7
1.1 Utilité 7
1.2 Standards 7
2 Présentation 8
2.1 Historique du projet 8
2.1.1 Les intercepteurs portables 8
2.1.2 Les fichiers journaux (ou logs) 9
2.1.3 Génération des diagrammes de séquences 11
2.1.4 Difficultés rencontrées et travail à faire 12
2.2 Description du cahier des charges 13
2.3 Architecture générale 14
3 Mécanisme d'interception 15
3.1 Choix technologique 15
3.2 Principe général 15
3.3 Mise en place dans l'application à observer 17
3.4 Déroulement de l'interception 19
3.4.1 Envoi d'une requête 20
3.4.2 Réception d'une requête 21
3.4.3 Le retour du serveur 22
3.4.4 Recéption du retour par le client 22
3.4.5 Emission d'exception 22
3.5 Utilisation du ServiceContext 23
3.5.1 Utilité 23
3.5.2 Fonctionnement 23
3.6 Architecture des classes d'interception 27
3.6.1 Le flux de sortie 28
3.6.2 Récupération des informations 30
3.6.3 La construction du log 33
3.7 Contenu du log 34
3.7.1 Pourquoi XML ? 34
3.7.2 Explication des balises 35
3.7.3 La balise de fin 35
4 Log2XMI 36
4.1 Introduction 36
4.2 L'architecture globale 37
4.3 Solution pour une reprise à chaud 39
4.4 La structure des messages 40
4.5 Le parsage des logs 41
4.5.1 Le choix de XML 41
4.5.2 Les logs et leur DTD 42
4.5.3 Le parsage des logs par l'outil log2xmi 47
4.6 La fusion des messages 51
4.6.1 Le cas général des messages bien formés 51
4.6.2 Le cas particulier des messages incomplets 52

4.7 La synchronisation des messages 53
4.7.1 Problématique 53
4.7.2 Technique de synchronisation utilisée 54
4.7.3 Implémentation 59
4.7.4 Le cas des messages incomplets 60
4.8 Le système de filtres 61
4.8.1 La Structure objet des filtres 62
4.8.2 Le Fichier des filtres 64
4.9 Le processus de génération XMI 68
5 XMI et diagrammes de séquence 70
5.1 Le standard XMI 70
5.1.1 Qu'est-ce que XMI ? 70
5.1.2 Utilisation de XMI dans notre projet 70
5.1.3 Structure des documents XMI 71
5.1.4 Le problème des extensions 73
5.1.5 Les outils de visualisation choisis 74
5.2 Les APIs XMI 75
5.2.1 Standards utilisés 75
5.2.2 Problèmes rencontrés 75
5.2.3 Structure des APIs 76
5.2.4 Diagrammes de classes 77
5.2.5 Exemple de code 78
5.3 Visualisation des diagrammes de séquence obtenus 80
5.3.1 Avec MagicDraw UML 80
5.3.2 Avec Rational Rose 81
5.4 L'application Xmi2tex 83
5.4.1 Utilité et fondements 83
5.4.2 Fonctionnement 83
5.4.3 Utilisation 83
5.4.4 Fichier XSL 84
5.4.5 Résultat 85
6 Gestion du projet 86
6.1 Travail d'équipe 86
6.1.1 Planning 86
6.1.2 CVS 86
6.1.3 XML 87
6.2 Site Web 87
6.2.1 Hébergeur 87
6.2.2 Conception 88
6.2.3 Statistiques 90
7 Conclusions 92
7.1 Ce que le projet nous a apporté 92
7.2 Un projet finalisé 92
8 Bibliographie 93
8.1 Références Corba 93
8.2 Références Java 93
8.3 Références XMI 93

9 Annexes 94
9.1 Mode d'emploi utilisateur 94
9.1.1 Installation dans votre projet 94
9.1.2 Résultat 96
9.1.3 Captures d'écran 96
9.2 Generer un Diagramme de Sequence UML 99
9.3 Filtres 101
9.4 Mode d'emploi développeur 103
9.4.1 Conditions requises 103
9.5 Compilation 103
9.6 Commandes 104
9.7 Licence 104

1 Introduction

1.1 Utilité

Quand vous développez une application distribuée, il est très difficile de débugger et de trouver précisément d'où viennent les problèmes dans une architecture. Du fait que le programme utilise plusieurs machines, il est plus complexe de maîtriser parfaitement les échanges d'informations et les accès distants.

C'est pourquoi un besoin de créer un outil permettant de tracer facilement les communications entre objets distants et de présenter le résultat graphiquement s'est rapidement fait ressentir.

1.2 Standards

CorbaTrace est basé sur une nouveauté de la norme Corba 2.3 : les intercepteurs portables. Ils permettent de minimiser la quantité de code source à modifier du côté de l'application hôte pour installer CorbaTrace.

Les résultats de l'interception sont sauvegardés sous forme de fichier XML. Puis, il est possible de leur appliquer des filtres pour choisir ce qu'on souhaite visualiser, pour enfin obtenir un fichier XMI, le format choisi par l'OMG pour sauvegarder les diagrammes UML. Ainsi, il devient possible d'utiliser n'importe quel atelier de génie logiciel pour visualiser le résultat sous forme de diagramme de séquences.

Corbatrace n'est basé sur aucun logiciel propriétaire, mais seulement sur les spécifications de l'OMG. Il est écrit en Java (une version C++ a été commencée), et sa licence d'exploitation est la LGPL (la licence libre de la Free Software Fondation). Aujourd'hui, il est basé sur le nouveau Java SDK 1.4.

2 Présentation

2.1 Historique du projet

Dans le cadre de la maîtrise, ce projet, intitulé "outils d'observation pour une application répartie : société BONOM", a été choisi comme sujet de TER par deux étudiants, Vincent Tricoire et Frédéric Breton. Ce projet consistait, d'une part, à créer des outils d'interceptions de messages circulant entre divers objets d'une application répartie, et d'autre part à concevoir des outils de visualisation spécifiques à la société BONOM.

Les applications réparties utilisent la norme CORBA comme norme de communication. Celle-ci permet de faire communiquer des objets implémentés dans différents langages de programmation.

Pour concevoir un outil d'observation d'applications réparties, il faut être en mesure de surveiller les communications qui circulent entre les différents agents d'une application.

Leur premier objectif dans ce projet était de créer des entités capables d'intercepter les communications entre les agents. Des modules permettant de récupérer les messages envoyés et reçus sont donc intégrer à l'application. Une fois interceptés, ces messages seront stockés de manière à créer des fichiers journaux. Ces fichiers journaux constituent un premier outil d'observation.

Leur deuxième objectif était de réaliser un outil permettant de tracer sous forme de diagrammes les informations contenues dans ces fichiers journaux. L'utilisateur pourra ainsi obtenir une représentation graphique des échanges entre différents agents de l'application répartie.

2.1.1 Les intercepteurs portables

Pour observer les communications entre les différents agents d'une application répartie, plusieurs possibilités s'offraient à eux.

Tout d'abord les deux étudiants pouvaient créer des intercepteurs eux-mêmes, et les fixer sur les objets de l'application qu'ils souhaitaient observer. Ces intercepteurs seront donc dépendants du langage dans lequel les objets observés sont programmés et ne pourront donc être utilisés que pour l'interception de messages basés sur le même langage.

La seconde possibilité consistait à utiliser les intercepteurs spécifiés par la norme CORBA. Les communications interceptées sont indépendantes du langage utilisé pour la création des différents objets de l'application répartie, les intercepteurs étant créés sur le bus CORBA. De plus les messages interceptés sont normalisés suivant la norme CORBA. Mieux appropriée, cette solution sera celle retenue par les deux étudiants.

Les intercepteurs portables doivent être déclarés lors de l'initialisation de l'ORB afin de pouvoir être utilisés. Clients et serveurs définissent des politiques d'interception associées à un niveau d'interception. Ils décident ainsi des communications qui pourront être interceptées.

La politique d'interception permet d'autoriser ou non l'interception.

Le niveau d'interception permet, quant à lui, d'indiquer la quantité d'informations extraites des données interceptées.

On différencie les intercepteurs clients des intercepteurs serveurs. Les intercepteurs clients vont intercepter les messages concernant l'activité d'un client, quant aux intercepteurs serveurs ceux d'un objet serveur.

Tout d'abord des objets Clients, Serveurs et Clients-Serveurs ont été créés afin d'être utilisés lors de communications sur un bus CORBA. Ensuite des actions ont été ajoutées afin de pouvoir intercepter les communications : déclenchement de la construction des intercepteurs et des méthodes répondant à leur invocation, puis établissement d'une politique d'interception associée à un niveau d'interception.

La gestion des interceptions s'est déroulait de la façon suivante :

- mise en place de la structure d'interception
- récupération des données nécessaires à la création de fichiers journaux
- création des fichiers journaux

Une fois l'activation des intercepteurs faite, les données de requêtes CORBA vont être extraites. Ces données étant extraites, elles vont être transformées en chaînes de caractères et écrites dans un fichier journal.

2.1.2 Les fichiers journaux (ou logs)

Un fichier journal est créé par objet. Ainsi, dans le cas où plusieurs objets sont déclarés sur le même serveur ou le même client, l'interception des communications entre ces objets est enregistrée dans des fichiers journaux séparés.

L'intercepteur étant le même pour l'ensemble des objets d'un client ou d'un serveur, il doit alors être capable d'identifier l'émetteur et le récepteur d'un message afin d'écrire dans le fichier correspondant.

Le nombre de fichiers journaux est donc égal au nombre d'objets présents dans l'application répartie. L'idée la création d'un fichier journal par objet permet, en plus du fait d'être facile à lire, d'effectuer une recherche plus simple en ne sélectionnant que les fichiers journaux des émetteurs et récepteurs de requêtes qui nous intéressent.

Pour réaliser ces fichiers journaux, plusieurs informations doivent être récupérées lors de l'observation des communications :

- **l'émetteur et le destinataire du message :** permet de définir les agents concernés par le message.
- **un identifiant de message :** cet identifiant permet, ajouté aux deux attributs précédents, d'identifier correctement un message dans deux fichiers journaux et d'en suivre le cheminement d'un bout à l'autre de la communication.
- **la date précise d'interception du message :** cet attribut ne permet pas d'effectuer la correspondance entre le départ d'un message et son arrivée. Le fait de connaître la date de départ d'un message et celle de son arrivée fournit cependant des informations sur la place du message dans la représentation des communications observées.
- **le contenu du message**

Les différents attributs que l'on peut obtenir dans les messages interceptés en fonction des points d'interception appelés ne sont pas toujours les mêmes. Ils dépendent de la nature de l'intercepteur (intercepteur client ou serveur) et de la nature du point d'interception appelé (réponse à une requête, exception, …).

En ce qui concerne l'interception sur les clients, l'identifiant du client n'est donné à aucun endroit. L'idée adoptée consiste alors à demander au client de s'identifier en rajoutant son identifiant comme premier paramètre à toutes ses méthodes déclarées dans les objets du serveur. Celui-ci servira ainsi à identifier un client lors de l'interception d'un message.

De même que pour l'identifiant d'un client, il n'y a pas d'identifiant dans les informations délivrées par les intercepteurs portables de CORBA pour identifier un message. L'idée retenue est donc de demander à l'utilisateur de donner l'identifiant de la requête envoyée comme deuxième argument de ces méthodes.

L'étape suivante consiste à écrire les données interceptées dans le fichier journal de manière à ce qu'elles soient lisibles et à ce qu'il soit facile d'en extraire automatiquement les données pour la réalisation d'un diagramme de séquence.

Voici un exemple d'un fichier journal client :

```
OPEN_FLUX=Friday, June 8, 2001 4:57:26 PM GMT+01:00=992015846253

>>>SEND_REQUEST
DATE=Friday, June 8, 2001 4:57:30 PM GMT+01:00=992015850952
CLIENT_ID=client
```

```
SERVANT_ID=Hello
MSG_ID=502196

OPERATION=say_hello_to
ARGUMENTS=3
ARG=in string(client)
ARG=in string(502196)
ARG=in string(le client du POA 1)

OPTIONS
         request id = 0
         exceptions = (no exceptions)
         response expected = true
END_OPTIONS
….
```

2.1.3 Génération des diagrammes de séquences

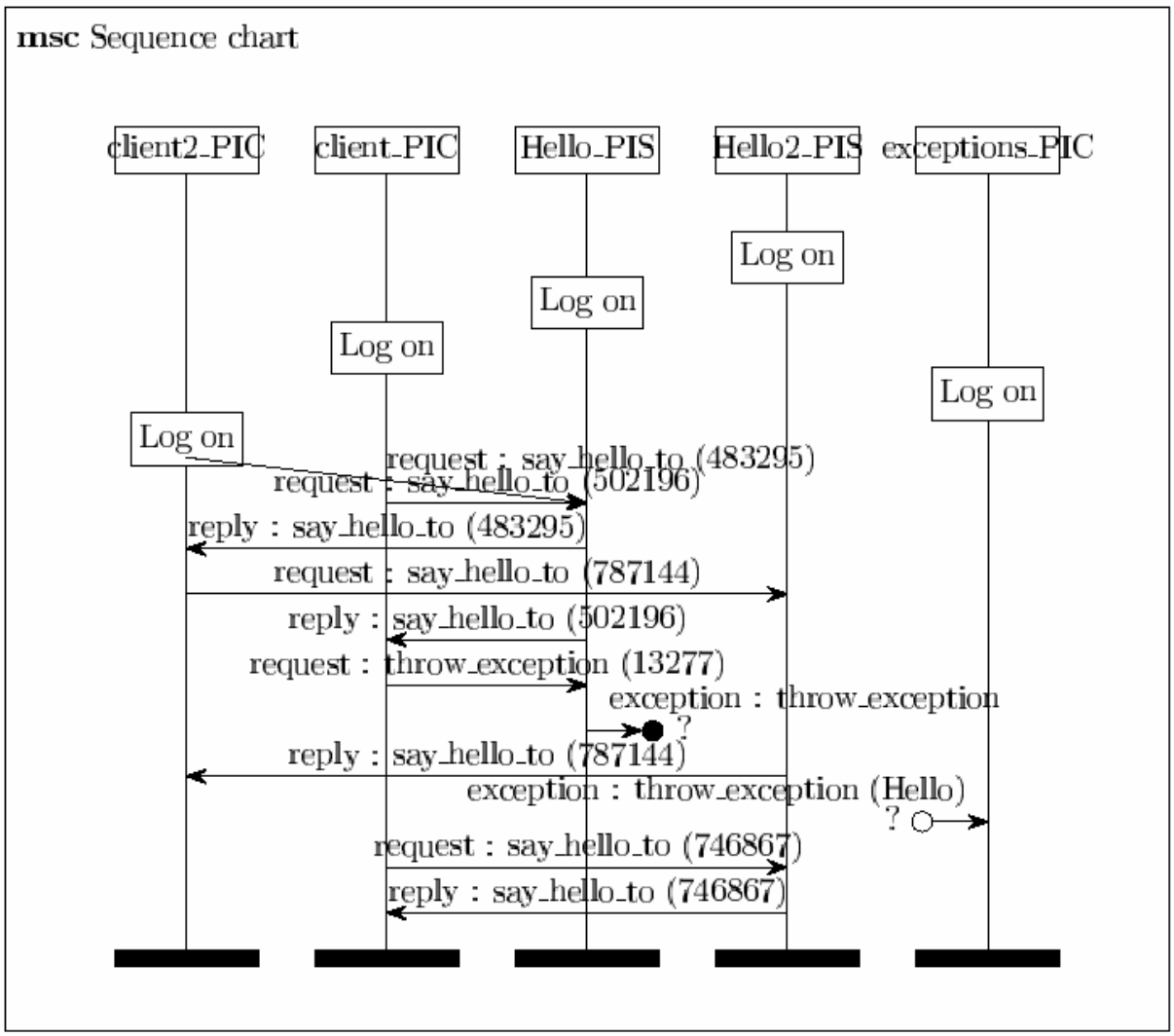
Les informations nécessaires à la création d'un diagramme de séquences sont :

- la date
- le nom du client
- le nom du serveur
- l'identifiant du message
- le nom de l'opération (ou s'il s'agit d'une exception)
- les autres données interceptées qui sont plus ou moins importantes selon le niveau d'interception choisi.

La réalisation des diagrammes de séquence est faite grâce à l'utilisation d'un paquetage Latex. Celui-ci amène des contraintes comme le fait que les communications qui peuvent être représentées ne sont pas basées sur des dates mais sur des décalages et des incréments de niveaux. Cela rend ainsi difficile le juste positionnement des communications les unes par rapport aux autres.

Lors de l'affichage, chaque communication complètes possède un nom comportant le type de la communication, le nom de la méthode appelée et l'identifiant du message. Quant aux communications incomplètes elles sont représentées sous forme d'une flèche interrompue. Elles comportent, pour une exception, le type de la communication, le nom de la méthode qui a levée l'exception et le nom de l'objet serveur mis en jeu.

Voici un exemple de génération d'un diagramme de séquence :



2.1.4 Difficultés rencontrées et travail à faire

Différents problèmes ont été rencontrés lors de l'identification des clients et des messages suite à une interception. La gestion des identifiants de messages et des clients doit être modifiée. En effet ils ne doivent plus être passé en paramètre des différentes méthodes appelées, cette façon de faire étant trop contraignante.

De plus, lors de l'interception d'un message entre un client et un serveur, ces deux objets peuvent avoir une date différente. Il faut donc pouvoir synchroniser les différents messages entre eux. En effet, dans le travail réalisé la date est considérée comme étant la même pour tous, ce qui n'est évidemment pas le cas dans la réalité.

Se pose également le problème de l'exclusion mutuelle pour l'écriture dans les fichiers des données extraites de l'interception. En effet un point d'interception peut-être levé pendant qu'un autre est en action. Ainsi si les deux points interceptés sont destinés à être écrits dans le même fichier, les deux points d'interception se partagent cette ressource critique. L'écriture concurrente dans le fichier entraîne alors un mélange des deux événements et provoque un brouillage des informations enregistrées.

La représentation graphique ne doit plus être réalisée à l'aide du paquetage Latex uniquement. Les fichiers permettant d'enregistrer les informations relatives à une interception doivent être écrit en XML. Suite à cela des filtres peuvent être appliqués, générant un fichier XMI. A partir de ces fichiers XMI il doit être possibles d'obtenir une représentation graphique sous forme de diagramme de séquence UML avec n'importe quel atelier de génie logiciel.

2.2 Description du cahier des charges

Le sujet de ce projet consistait, suite au travail déjà réalisé dans le cadre du TER, à améliorer l'application en mettant en œuvre une interception moins contraignante et une visualisation graphique plus performante. De plus, le code doit être revu et modifié de façon à être plus propre et plus clair.

Notre application, appelée CorbaTrace, est un ensemble d'outils permettant une représentation des communications entre objets Corba.
L'outil devait être générique et fonctionner avec n'importe quelle application utilisant CORBA.

L'interception des messages sur le bus CORBA est une politique locale, chaque objet distribué peut changer de politique à tout instant s'il le désire. Il est donc possible que les informations contenues dans les différents fichiers journaux soient incomplètes. Ainsi un envoi de message peut être présent dans le fichier journal de l'objet A alors que la réception de cette demande n'est pas présente dans le fichier journal de l'objet B.

Les résultats des interceptions doivent être enregistrés dans les fichiers journaux sous forme de fichiers XML. Des filtres sur les informations que l'on souhaite visualiser pourront être possibles, le résultat de ces filtres étant sauvegardé dans un fichier XMI qui est le format choisi par l'OMG pour sauvegarder les digrammes UML. Il est alors possible d'utiliser n'importe quel atelier de génie logiciel pour visualiser le résultat sous forme de diagrammes de séquence.

2.3 Architecture générale

Pour répondre à ce cahier des charges, nous avons décidé de décomposer en trois modules :

- InterceptoreCore : le cœur de l'interception. Ses classes se placent sur l'ORB et produisent le log
- Log2XMI : le parsage des logs et leur filtrage
- XMIGenerator : la génération des fichiers XMI suivant l'atelier de génie logiciel

Le fonctionnement générale est le suivant (cf. :

- Mise en place de classes utilitaires CorbaTrace dans l'application hôte
- Interception des appels de fonctions distantes par le module Interceptor
- Sauvegarde des informations relatives à l'interception dans un log XML
- Récupération des logs par Log2XMI qui choisie les informations pernitentes grâce à un filtre écrit sous forme de fichier de configuration XML
- Filtrage, fusion et synchronisation des messages
- Envoie des messages à représenter au module XMIGenerator
- Ecriture des fichiers XMI pour générer un diagramme de séquence générique, avec des extensions suivant l'AGL.

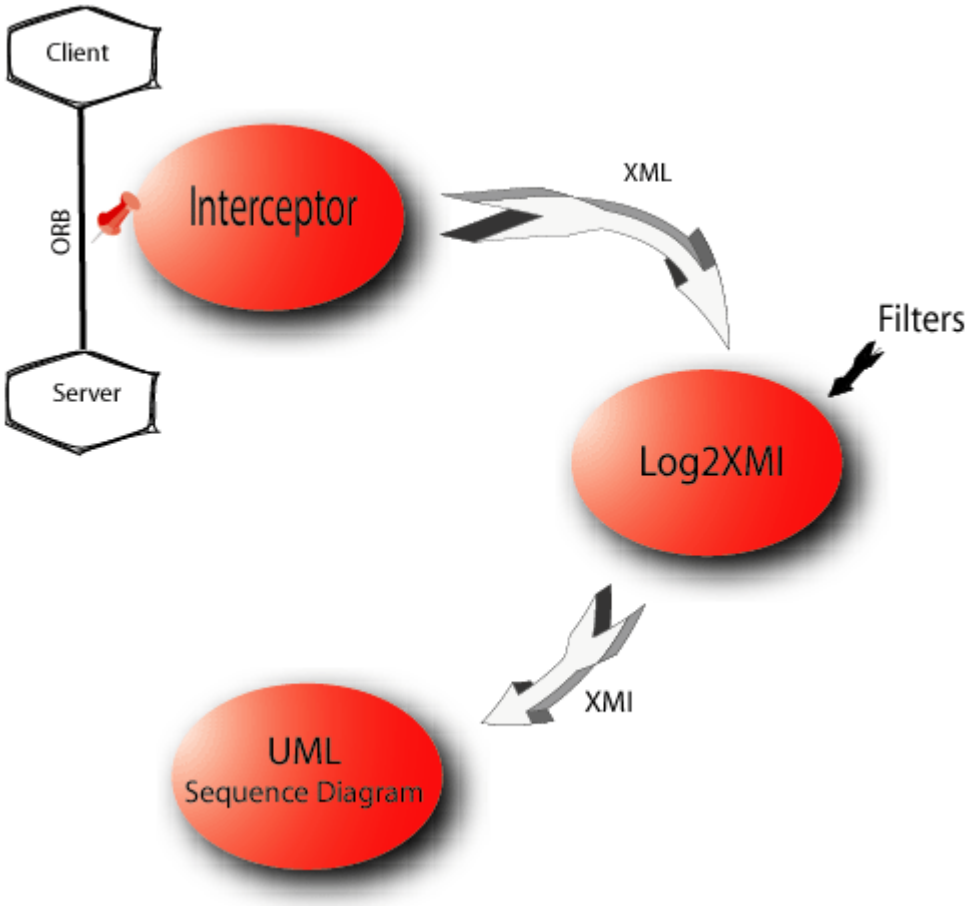


Figure 1 : Dynamique générale de CorbaTrace

3 Mécanisme d'interception

Voici l'explication du mécanisme utilisé par CorbaTrace pour mettre en place des logs cohérents. Il est basé sur les standards de l'OMG sur les intercepteurs.

Nous n'allons pas ici ré-expliquer le fonctionnement de Corba car de nombreux ouvrages traitant ce sujet existent.

3.1 Choix technologique

Dans un ancien projet trouvé sur le web dont le but était également de logger les appels Corba, le choix avait été fait d'intercepter les messages au niveau IP. Ce projet n'a jamais abouti car cette méthode a beaucoup d'inconvénient dont la complexité, le filtrage des paquets IP et la dépendance vis à vis de l'ORB.

Le premier ORB à avoir implémenté les intercepteurs est Orbacus. Nous avons donc choisi de l'utiliser, tout en essayant d'en être indépendant et en respectant scrupuleusement les directives de l'OMG. A moyen terme, nous pouvons espérer que notre projet intéresse du monde et dans ce cas, nous aimerions proposer CorbaTrace comme un élément du projet GNU. La première étape sera donc de le tester sur un ORB GNU et pourquoi pas sur la plate-forme Gnome qui repose sur Corba.

Pour le langage, Java a été le premier choisi grâce à sa facilité de codage, sa grande diffusion et sa portabilité.

Un portage vers le C++ a été commencé. Grâce à notre séparation XML entre l'interception et la transformation des logs, seule la partie d'interception a besoin d'être traduite. Et vu que les objets manipulés sont des objets Corba ou des objets standardisés pour tous les langages, le portage ne révèle pas de grandes difficultés. Par manque de temps, nous n'avons malheureusement pas encore fini le codage. Nous espérons le poursuivre dans les semaines qui suivent (il reste à peu près la moitié du travail).

3.2 Principe général

Depuis à peu près deux ans, l'OMG a standardisé une méthode d'interception portable sur les ORBs avec la mise au point de la norme Corba 2.3. Celle-ci permet d'autoriser une interception lors d'une communication entre deux objets Corba, indépendamment de l'ORB. L'interception peut se faire sur plusieurs points, dont l'émission / réception d'un message ou d'une exception.

Ces intercepteurs ont par exemple été utilisés pour gérer les transactions entre objets Corba. Dans ce cas, le client et le serveur n'ont pas besoin d'être modifié pour prendre en compte ce mécanisme. Il suffit de changer la politique de gestion des logs pour que la transaction se mette en route.

Comme on peut le voir sur la Figure 2, les intercepteurs se placent bien sur le bus Corba. Tout leur mécanisme est caché aux objets métiers.

Par contre, c'est le programme qui met en place ces objets métiers et qui initialise le système qui décide d'activer ou non l'interception. Ainsi, on obtient plusieurs avantages :

- On ne peut pas sniffer une communication Corba sans l'accord de l'application observée
- On peut activer et désactiver dynamiquement l'interception au fur et à mesure des besoins
- On peut régler dynamiquement le niveau d'interception.

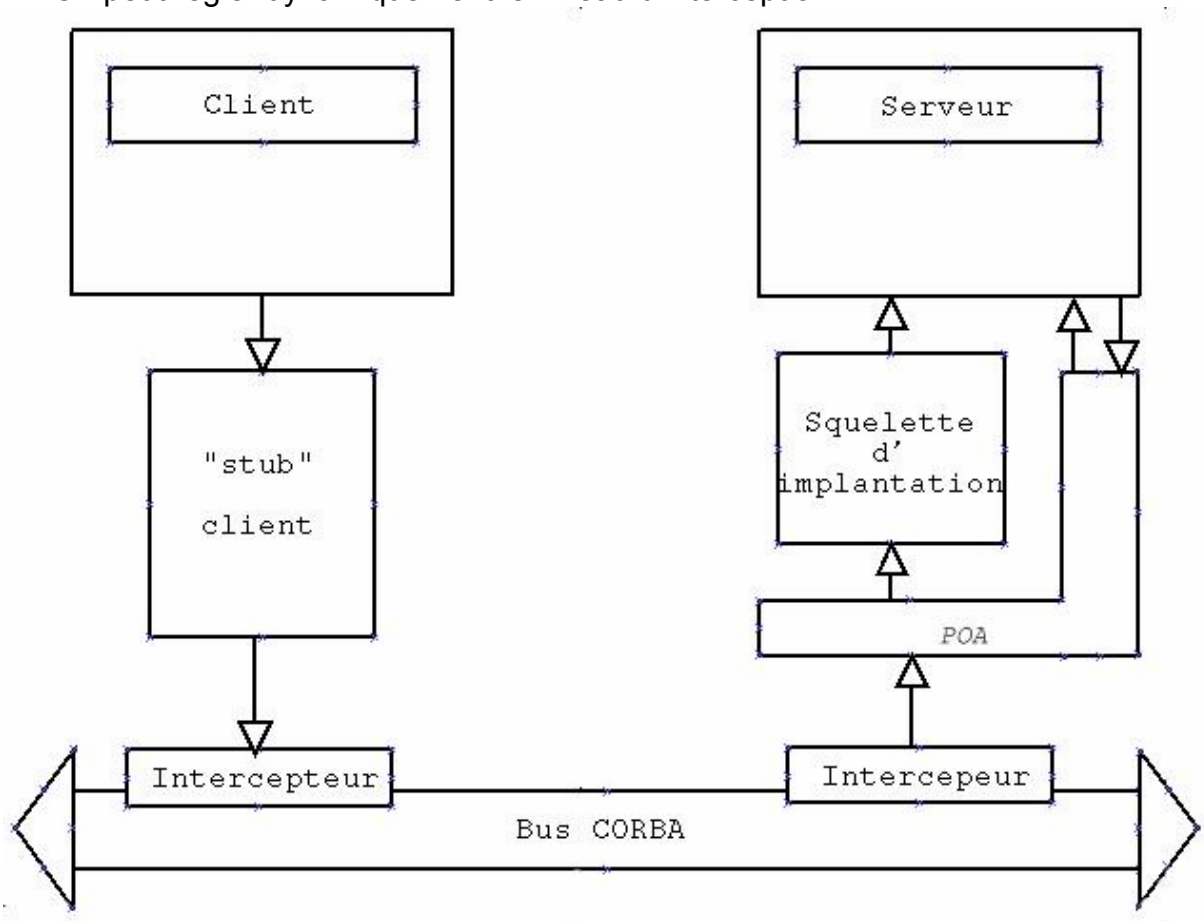


Figure 2 : les intercepteurs vus par l'OMG

Pour permettre de régler la quantité d'informations à intercepter, l'OMG a défini un objet Corba nommé `LoggingPolicy`. Voici son IDL :

```
#include "Policy.idl"

//
// The Logger Policy -- specifies the log level for the
// client and server Request Interceptor loggers
//

//
```

```
// ID for the Logger Policy
//
const CORBA::PolicyType LOGGER_POLICY_ID = 1010;

//
// Definition for the Logger Policy
//
local interface LoggerPolicy : CORBA::Policy
{
    readonly attribute short level;
};
```

Sachant que `LoggerPolicy` hérite de `Policy`, on en déduit donc que l'interception n'est qu'une politique particulière du POA ou de l'ORB, tout comme l'est la persistance, la gestion des cycles de vie ou le contrat transactionnel. Elle permet en outre de stocker une valeur qui représente le niveau d'interception. Ainsi, on décide de la précision et de la quantité des informations à logger. Dans la pratique, un niveau de 1 suffit très largement.

Voici un exemple de mise en place du niveau d'interception pour un POA :

```
Any any = orb.create_any();
Policy[] policies = new Policy[1];
short lglvl = 1

any.insert_short(lglvl);
policies[5] = orb.create_policy(LOGGER_POLICY_ID.value, any);

POA nouveauPOA = poa.create_POA("MonPOAIntercepte", poa.the_POAManager(),
policies);
```

3.3 Mise en place dans l'application à observer

Du point de vue de l'application observée, nous avons limité au maximum la quantité de code à ajouter pour mettre en place CorbaTrace. Pour voir comment procéder exactement, reportez-vous à la documentation utilisateur spécifique. Des classes utilitaires ont été mises à la disposition des applications clientes pour cacher les opérations standardisées à effectuer sur l'ORB pour créer des intercepteurs.

Ces classes sont spécifiques pour un objet Corba client ou serveur. Elles se chargent d'enregistrer les intercepteurs auprès de l'ORB. L'activation est ensuite différente selon le cas client ou serveur, mais la méthode reste semblable.

Pour activer un intercepteur sur un objet, il faut lui passer un niveau d'interception. Celui-ci est standardisé par l'OMG et spécifie la quantité d'information à intercepter. A 0, rien n'est logué. Si aucun niveau n'est spécifié, on le met par défaut à 1, ce qui suffit amplement pour un débuggage classique.

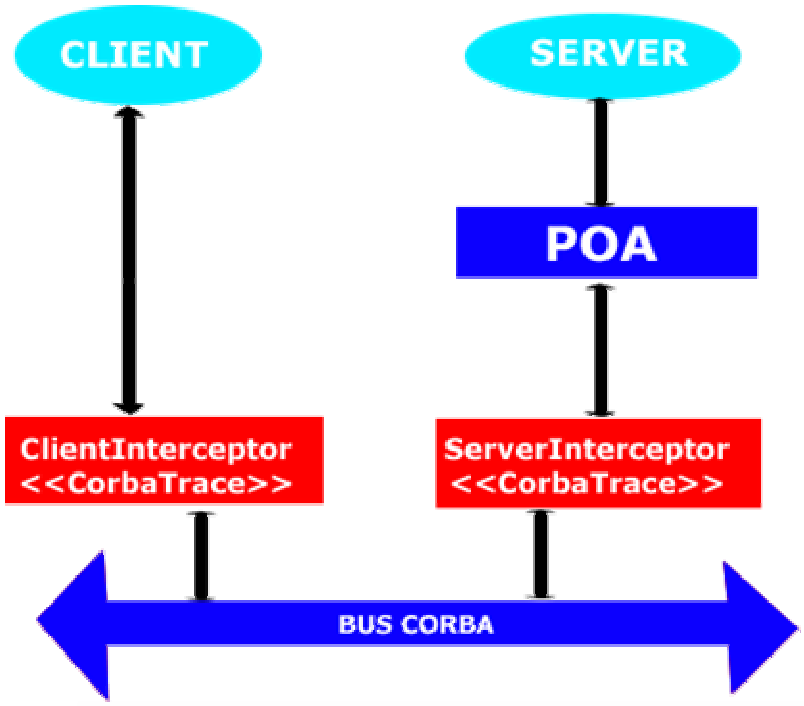


Figure 3 : les intercepteurs vus par CorbaTrace

Ensuite, il faut mettre en place l'intercepteur sur le composant. Du côté client, on le met sur l'ORB, alors que du côté serveur, on le met sur le POA. Dans ce dernier cas, un nouveau POA avec les intercepteurs activés est créé dynamiquement à partir d'un POA existant dans l'application cliente (ou du rootPOA). On permet ainsi d'avoir une vision au niveau composants des communications entre objets. Un débuggage "macro-scopique" semble plus utile qu'un débuggage entre objets d'un même composant.

Pour identifier le composant, il est nécessaire de lui fournir un nom sous forme de chaîne de caractères unique dans l'application. C'est ce nom qui apparaîtra dans le diagramme de séquence. Ce nom est sauvegardé dans un slot du PICurrent (voir plus loin).

Une fois toutes ces initialisations effectuées, il suffit d'utiliser les objets métiers. Dans le cas du client, aucune difficulté ne se présente. Mais pour le serveur, il faut juste mettre le servant (l'objet métier serveur) sur le nouveau POA créer dynamiquement par la classe `IntercepteurServer`.

Il est bien sûr possible de ne logger que le client ou que le serveur ou les deux. Le diagramme de séquence final tiendra compte des manques de certains logs en spécifiant un interlocuteur inconnu.

Pour un mode d'emploi complet, se reporter à l'annexe 9.1.

3.4 Déroulement de l'interception

La Figure 4 permet d'avoir une vision globale du mécanisme d'interception.

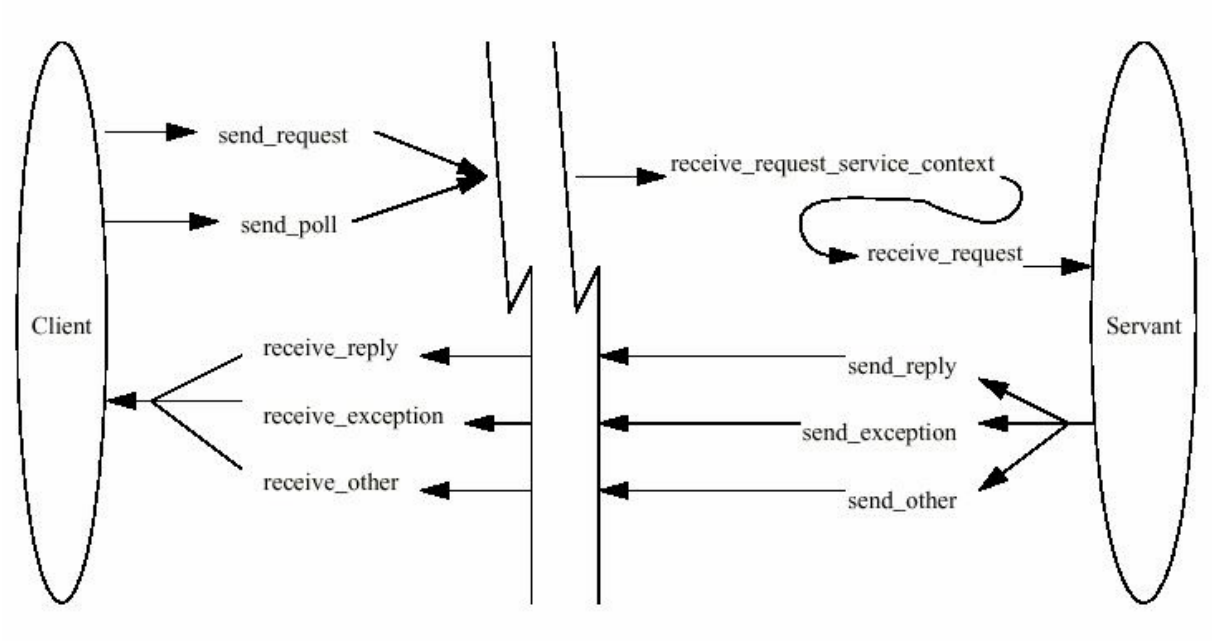


Figure 4 : les points d'interception

Chaque point d'interception a une fonction particulière qui est exposée brièvement ci-dessous avant d'être expliquée plus précisément.

Du côté client :

- **send_request** : point qui va être appelé lors de la détection d'un envoi de requête vers un serveur
- **send_poll** : point appelé lorsqu'un client demande des informations au serveur sur l'état du servant ou sur l'ORB.
- **receive_reply** : point appelé lors de la réception d'une réponse
- **receive_exception** : point appelé lors de la réception d'une exception envoyée par le serveur
- **receive_other** : point appelé lorsqu'on détecte un message envoyé au client qui ne correspond pas à une réponse reçue ou à une exception

Du côté serveur :

- **receive_request_service_context** : ce point d'interception est appelé avant le point d'interception `receive_request` pour récupérer, décrypter et transmettre le `Service Context` dans le `request scope`.

- **receive_request** : ce point d'interception est appelé lorsque l'on détecte une réception de requête.

- **send_reply** : point appelé lors de la détection de l'envoi d'une réponse à une requête

- **send_exception** : point appelé lors de la détection de l'envoi d'une exception levée par une requête

- **send_other** : point d'interception appelé quand on détecte l'envoi d'autre chose qu'une réponse ou une exception

3.4.1 Envoi d'une requête

Lorsque le client appelle une méthode sur un objet distant, le mécanisme d'interception Corba précédemment initialisé se met en route.

Tout d'abord, l'envoi de la requête est intercepté côté client par l'intercepteur dont la classe est `ClientRequestInterceptorLogger`. Le point d'interception est `send_request`. Si les logs sont activés (le niveau d'interception est supérieur à zéro) alors on va vouloir mettre des informations à destination du serveur. Nous avons besoin par exemple de transmettre l'identifiant du client (car le standard Corba 2.3 des intercepteurs ne permet pas de connaître son identité), la date de l'envoi, le numéro de la requête (il en existe un défini par l'OMG, mais il n'est pas certifié unique à travers IIOP).

Ceci est un problème délicat, et nous avons réfléchi à plusieurs méthodes pour y parvenir. Dans la première version du projet, ces données étaient transmises dans les arguments de chacune des méthodes. Cela impliquait de modifier tous les appels de fonction de toutes les classes de l'application, chose inconcevable en réalité. Nous avons ensuite étudié une solution grâce à une solution à base de communications sécurisées où les deux interlocuteurs se connaissent mutuellement.

Enfin, nous avons opté pour une solution normalisée par l'OMG. Celle-ci se déroule en plusieurs étapes :

- Un slot est alloué à la mise en place des intercepteurs
- L'identifiant du composant est stocké dans un objet Corba (avec une IDL) nommé Dataflow
- Cet objet est sauvegardé dans l'objet normalisé "PICurrent", dans le slot précédemment initialisé
- La requête est effectuée
- Automatiquement, l'ORB transfert le contenu du "PICurrent" dans le contexte de la requête

- Le point `send_request` est appelé
- L'objet Dataflow est récupéré dans le `RequestContext`. On lui ajoute la date et l'identifiant unique de la requête
- L'objet Dataflow est sauvegardé dans le `ServiceContext`, pour être reçu par le serveur.

En utilisant l'objet Dataflow, on n'utilise qu'un seul slot. Ce qui limite encore l'influence par rapport à l'application à observer.

Ensuite, la fonction `displayRequestInfo` de `RequestInterceptorLogger` est appelée pour générer le fichier de log en XML, en fonction des informations du contexte d'exécution.

Pour générer ce fichier XML, nous avons choisi de ne pas utiliser DOM car celui-ci demanderait trop de ressources, et ainsi, changerait les caractéristiques du système à observer.

Nous écrivons donc directement dans un fichier et nous utilisons une petite classe qui permet de gérer l'indentation pour le rendre plus lisible.

Pour une explication plus détaillée du contenu des logs, rendez-vous dans le chapitre le fichier de logs en XML.

3.4.2 Réception d'une requête

Après l'émission de la requête du client, celle-ci est transmise par IIOP sur le bus Corba. L'objet serveur (le servant) la reçoit pour répondre à la sollicitation. Mais avant cela, le mécanisme défini par l'OMG défini 2 niveaux d'interception.

Tout d'abord, le premier point d'entrée est `receive_request_service_context`. Il permet de récupérer le `ServiceContext` passé par l'objet `RequestInfo`. L'étape suivante est le décryptage de ce contexte pour récupérer l'objet Dataflow. Pour rendre ses informations disponibles le long des autres étapes de l'interception, on le place dans un slot du `RequestInfo`.

L'étape suivante se nomme `request_intercept`. Ce point d'interception est la réception effective de la requête sur le servant. Il suffit donc de sauvegarder dans le log toutes les informations de la requête comme le nom de l'opération, les valeurs des arguments, les informations passées par le Dataflow, etc.

La requête est ensuite transmise logiquement au servant, et l'opération désirée est exécutée.

3.4.3 Le retour du serveur

Une fois la requête exécutée, deux possibilités sont envisageables :

- la requête est asynchrone (`oneway`) et il n'y a pas de retour.
- la requête est synchrone (cas normal), et on a un retour vers l'objet client, même si la valeur de retour est `void`.

Pour le retour, nous avons réussi à nous contenter des informations à notre disposition sans avoir besoin de transmettre d'informations du serveur vers le client. Ceci peut bien sûr évoluer en fonction des besoins.

3.4.4 Recéption du retour par le client

Enfin, l'ORB du client intercepte le retour de la réponse par l'élément `request_response`. Encore une fois, les logs sont sauvegardés dans un fichier XML. C'est à ce moment que s'est posé le problème de l'identifiant unique de requête (cf. chapitre correspondant).

L'objet Client reprend ensuite la main et continue sont travail.

3.4.5 Emission d'exception

La norme des intercepteurs Corba définit deux types d'exceptions.

La première concerne les exceptions utilisateurs. Elles sont émises dans le corps du code source du servant à cause d'une raison propre à l'application elle-même. Elle peut être générée par l'utilisateur, comme par exemple si une chaîne de caractères n'a pas le format voulu par le servant. Ou elle peut être générée par le langage hôte (exemple d'une division par zéro).

L'envoi d'une telle exception remplace alors l'appel au point d'interception `send_response` par l'appel à `send_exception`. Bien entendu, certaines informations deviennent ainsi indisponibles telle que l'objet retourné (vu qu'il n'y en a plus).

Le deuxième type d'exception est une exception Corba. Si par exemple, une erreur se produit dans l'intercepteur ou dans le cheminement à travers le réseau, une telle exception est déclenchée. Son fonctionnement est différent de la précédente car les informations concernant cette exception sont stockées dans un argument du `RequestInfo` nommé `sending_exception`. On y trouve par exemple la cascade d'exceptions qui ont été déclenchées.

3.5 Utilisation du ServiceContext

3.5.1 Utilité

Pour relier les requêtes stockées dans les logs des clients et des serveurs, il nous faut pouvoir identifier les interlocuteurs. Nous avons donc besoin d'un ID unique pour chaque composant. Chaque requête doit également être identifiée uniquement. L'ORB en fournit toujours un (`request_id`) qui peut nous être utile mais qui ne peut suffire car elle n'est pas unique à travers IIOP. De plus, il serait très intéressant de transmettre la date de réception d'une requête à son correspondant.

La solution trouvée anciennement avait été de passer toutes ces informations comme arguments des opérations métiers. Il était donc nécessaire de modifier tous les appels de fonctions de l'application hôte. Cette solution n'était donc pas du tout viable.

Une solution envisagée a été d'utiliser le service de sécurisation des appels distants (basé sur ssh). Ce service connaît les deux interlocuteurs car il procède à leur authentification. On impose par contre son utilisation à l'utilisateur.

Notre solution repose en fait sur un mécanisme normalisé par l'OMG depuis peu. Celui-ci permet de faire passer à la fois des informations de l'application hôte aux intercepteurs, de l'intercepteur du client à l'intercepteur du serveur et vice-versa.

Nous en avons déjà parlé dans l'explication des points d'interception, donc nous n'allons pas rentrer dans les détails, mais plutôt nous attarder sur son utilisation générale et son cycle de vie.

3.5.2 Fonctionnement

Le mécanisme général est décrit sur la figure ci-dessous.

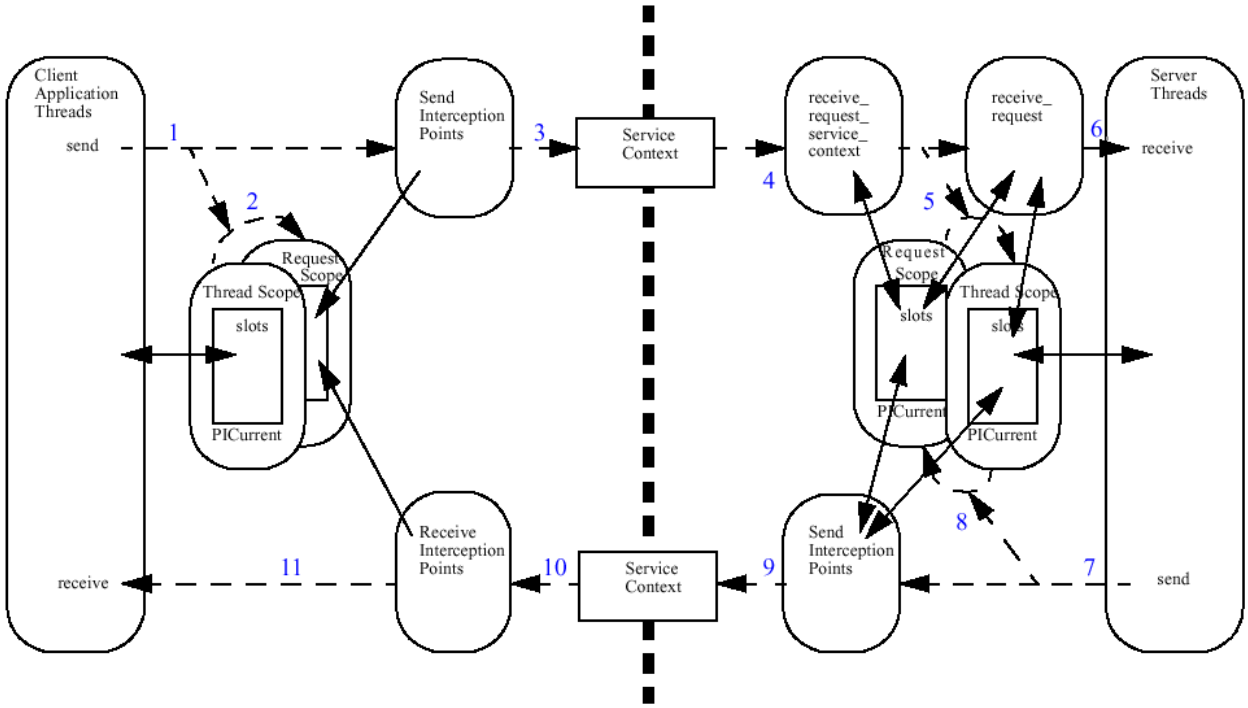


Figure 5 :cycle de vie du Service Context

Il existe deux types de contexte : le thread scope et le request scope.

- Le request scope contient des informations qui ne sont disponibles que le temps d'une requête. Seul ce contexte est accessible aux points d'interceptions.
- Le thread scope sert à transmettre des informations entre l'application hôte et les intercepteurs ou du client au serveur (sans passé par un appel de fonction).

Le request scope peut par exemple être utilisé pour transmettre des informations d'un point d'interception à un autre. Sachant qu'il ne faut pas utiliser les variables d'instance de nos classes intercepteurs (car c'est l'ORB lui-même qui gère leur création et la concurrence d'accès aux instances), c'est même la seule manière de faire transiter de l'information.

Avant d'appeler la méthode distante, le client peut accéder au Thread Scope, via un objet Corba nommé PICurrent. Cet objet possède une collection de slots. Chaque slot est identifié par un entier unique et peut contenir un objet de type Any. Cela signifie qu'il est donc possible de transmettre dans un slot toute information que l'on désire.

Le PICurrent possède est caractérisée par une IDL très simpliste

```
module PortableInterceptor {
     typedef unsigned long SlotId;
     exception InvalidSlot {};
     local interface Current : CORBA::Current {
          any get_slot (in SlotId id) raises (InvalidSlot);
          void set_slot (in SlotId id, in any data) raises (InvalidSlot);
     };
};
```

Pour que le client remplisse un slot du PICurrent, il lui suffit d'exécuter le code suivant :

```
any myData = ...; // get data from Transaction's Current
PortableInterceptor::Current pic =
orb.resolve_initial_references ("PICurrent");
pic.set_slot (mySlotId, myData);
```

Pour éviter d'utiliser plusieurs slots, nous avons créé une classe Dataflow qui stocke toutes les informations à faire transiter dans le contexte. Cette classe ne sert qu'à stocker de l'information, une simple structure IDL lui suffit donc.

```
struct Dataflow {
    string client_id;
    long long date_send;
    long request_id;
  };
```

Quand l'application active les logs sur l'ORB du client, il appelle la fonction `activate_log` avec en paramètre l'identifiant du composant. Celle-ci crée le Dataflow et l'initialise avec cet identifiant.

```
public static void activate_log(ORB orb, String name) {
      org.omg.PortableInterceptor.Current pic;
      org.omg.CORBA.Object objPic;
      try {
          objPic = orb.resolve_initial_references("PICurrent");
          pic = org.omg.PortableInterceptor.CurrentHelper.narrow(objPic);

          Any anyPic = orb.create_any();
          Dataflow flow = new Dataflow();
          flow.client_id = name;
          DataflowHelper.insert(anyPic, flow);

          pic.set_slot(RequestInterceptorLogger.slotId, anyPic);
      }
      catch(org.omg.CORBA.ORBPackage.InvalidName ex) {
          throw new RuntimeException();
      }
      catch(org.omg.PortableInterceptor.InvalidSlot ex) {}
      catch(org.omg.CORBA.BAD_PARAM ex) {}
}
```

Une fois les objets initialisés, le client appelle la méthode distante et déclenche le mécanisme d'interception.

Voici pas à pas la suite des transformations effectuées pour la transmission du Dataflow (cf. Figure 5) :

1. Le client appelle une méthode distante sur le stub du servant.
2. Le contenu du PICurrent est automatiquement copié dans le ClientRequestInfo.

3. Le point d'interception `send_request(ClientRequestInfo ri)` (ou `send_poll`) est appelé.
   Si les logs sont activés, il récupère le Dataflow du slot du ClientRequestInfo par l'appel à `getDataflow` de la superclasse RequestInterceptorLogger.

```
protected Dataflow getDataflow(RequestInfo info)
      throws org.omg.CORBA.BAD_OPERATION {

      try {
          Any any = info.get_slot(mySlotId_);
          if( any == null )
            throw new org.omg.CORBA.BAD_OPERATION();
          Dataflow flow = DataflowHelper.extract(any);
          return flow;
      } catch(org.omg.PortableInterceptor.InvalidSlot e) {
          throw new org.omg.CORBA.BAD_OPERATION(e.getMessage());
      }
}
```

Une fois récupéré, il suffit de lui spécifier la date de l'interception ainsi que le request_id courant, et enfin de le transformer en Any.

```
Any any = createAny();
Dataflow flow = getDataflow(ri);
flow.date_send = new Date().getTime();
flow.request_id = ri.request_id();
DataflowHelper.insert(any, flow);
```

Sachant qu'on utilise IIOP pour transmettre les informations, une étape supplémentaire est nécessaire pour encoder l'objet Any précédemment créé. Enfin, il ne reste plus qu'à créer le Service Context avec ces informations et à l'ajouter à la requête.

```
// Encode Dataflow
byte[] data = cdrCodec_.encode_value(any);

// Add encoded Dataflow to service context
org.omg.IOP.ServiceContext sc = new org.omg.IOP.ServiceContext();
sc.context_id = REQUEST_CONTEXT_ID;
sc.context_data = data;
ri.add_request_service_context(sc, false);
```

4. L'intercepteur du serveur est activé par l'appel du point d'interception `receive_request_service_contexts(ServerRequestInfo ri)`. La première action est de procéder aux opérations inverses de celles de l'étape 3. C'est-à-dire : décodage, extraction du Dataflow, ajout de celui-ci dans le request scope du server (par l'intermédiaire du ServerRequestInfo) :

```
public void receive_request_service_contexts(ServerRequestInfo ri) {
      // (les exceptions ne sont pas retransmises par souci de clareté)
      System.out.println("receive_request_service_contexts");
```

```
        if(isLogActivate(ri)) {
            Dataflow flow = null;
            org.omg.CORBA.Any any = null;
            ServiceContext sc= ri.get_request_service_context(REQUEST_CONTEXT_ID);
            any = cdrCodec_.decode_value(sc.context_data, DataflowHelper.type());
            flow = DataflowHelper.extract(any);
            DataflowHelper.insert(any, flow);
            ri.set_slot(mySlotId_, any);
        }
}
```

5. La méthode `receive_exception` est appelée. Il est ainsi possible d'avoir accès à toutes les informations stockées dans le Dataflow qui se trouve dans le ServerRequestInfo. Automatiquement, l'ORB transfert tout le contenu du request scope dans le thread scope du serveur.

6. Le serveur prend la main et réalise la méthode du servant qui a été invoquée. Il peut avoir accès au contenu du Dataflow ou de tout objet passé dans le Service Context par l'intermédiaire de l'objet Corba, PICurrent.

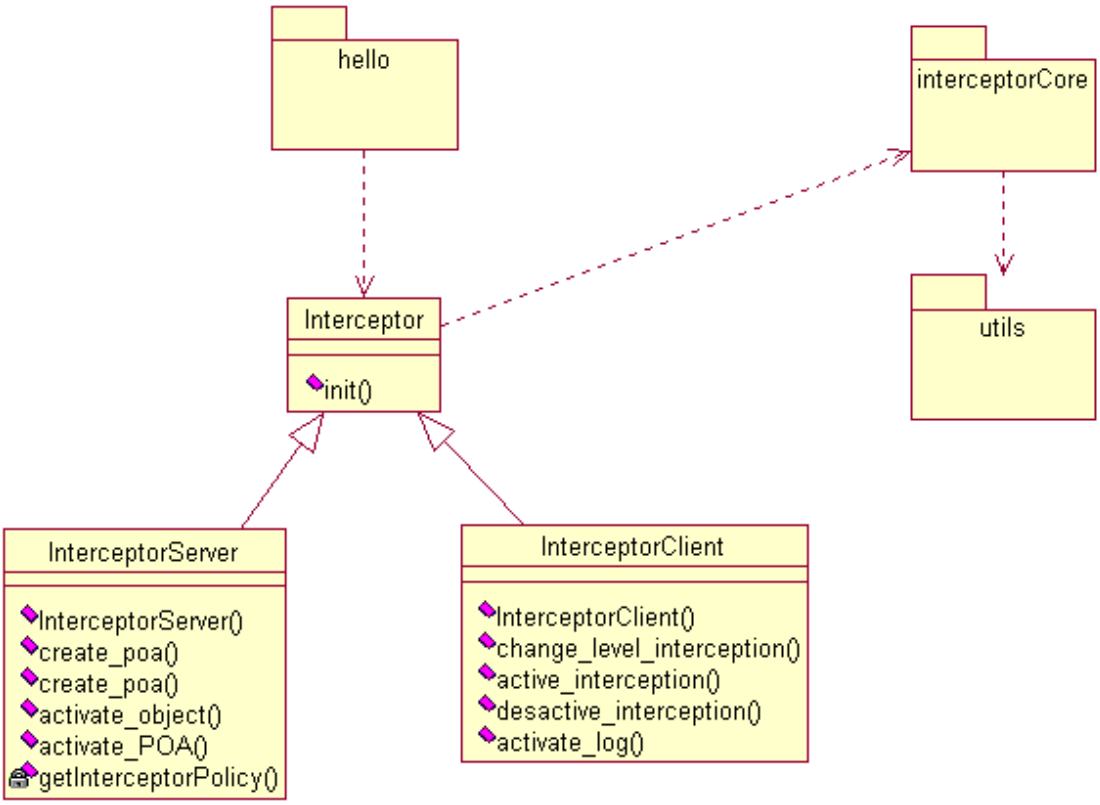
7. à 11. Le processus inverse est déclenché pour le retour de fonction. A chaque fois, les mêmes opérations sont effectuées.

3.6 Architecture des classes d'interception

A la reprise du projet de l'année 2000-2001, l'architecture de CorbaTrace avait été négligée. Le projet s'était concentré sur des problèmes techniques au détriment de la propreté du code. Dès le début, un réaménagement complet des classes a été nécessaire pour clarifier les sources, permettre un débuggage plus rapide et une maintenance bien plus facile.

Le premier travail a été de centraliser la génération du fichier de sortie. Anciennement, chaque classe écrivait son petit bout de log dans le fichier en créant des problèmes de concurrence d'accès au fichier et en rendant très difficile la localisation d'un problème.

La solution envisagée a été de créer le log entièrement en mémoire, et de l'écrire d'un seul coup dans le fichier. On évite ainsi le problème de concurrence d'accès, on gagne en performance et le tout devient plus lisible.



Sachant que le choix du XML comme format de sortie a été retenu, il nous était possible d'utiliser DOM (Document Object Model, de W3C) pour construire le fichier. Mais DOM a un gros problème : il prend beaucoup de place mémoire et il prend du temps machine. Certes, ce n'est pas non plus immense, mais si on veut tracer le comportement d'un programme, il est critique de limiter l'impact de l'interception.

En utilisant DOM, on aurait ainsi de nombreuses millisecondes qui pourraient compromettre l'interprétation des logs. De plus, DOM n'a pas un intérêt pertinent car on a besoin de seulement rajouter des noeuds à l'arbre XML, sans jamais modifier ou supprimer une branche.

Une autre solution (préconisée d'ailleurs par les experts XML pour des applications critiques comme la notre) est de concaténer directement le flux XML dans une chaîne de caractère (une `StringBuffer` en Java). Nous avons opté pour cette solution, tout en créant une classe utilitaire qui nous permet de gérer l'indentation.

3.6.1 Le flux de sortie

Cette classe se nomme `IndentString` et se situe dans le package `corbatrace.utils`. Elle pourrait tout à fait servir dans une autre application que CorbaTrace.

Son unique but est de créer une String tout en fournissant une indentation des balises XML, le tout avec un souci de performance et d'économie de place en mémoire.

Dans notre cas, le fichier XML sert normalement d'entrée au module Log2Xmi, mais il doit également être lisible par un humain car il contient des informations qui se suffisent à elle-même et qui peuvent ne pas être retrouvées dans le diagramme de séquence final.

Pour bien comprendre son fonctionnement, voici un exemple de génération d'un fichier XML.

On veut stocker les caractéristiques d'une personne.
Si on utilisait une simple `StringBuffer`, on obtiendrait en résultat :

```
<personne><age>53</age><name first="Omer" last="Simpsons"/></personne>
```

En imaginant qu'on doivent stocker ses coordonnées, ses fonctions et qu'on a plusieurs milliers de personnes, le fichier XML est illisible pour un humain.

Le résultat a atteindre est donc :

```
<personne>
   <age>53</age>
   <name first="Omer" last="Simpsons"/>
</personne>
```

Pour y parvenir, il faut exécuter le code suivant :

```
 IndentString out = new IndentString();
 out.insert("<personne>");
 out.inc();
 out.insert("<age>53<age/>");
 out.indent();
 out.append("<name");
 out.append(" first="Omer" last="Simpsons"/>");
 out.newLine();
 out.dec();
 out.insert("</personne>");
```

Voici l'explication des méthodes :

- inc : augmente d'un cran le niveau d'indentation
- dec : diminue d'un cran le niveau d'indentation (remonte dans l'arbre XML)
- append : concatène simplement le texte
- newLine : saute une ligne
- insert : indente, insère le texte et passe à la ligne

Ces fonctions peuvent paraître rudimentaires, mais elles ont l'avantage d'être simples et suffisantes (on peut même les utiliser pour écrire autre chose que du XML).

3.6.2 Récupération des informations

Pour récupérer des informations sur le contexte d'interception, l'OMG a défini un objet nommé `RequestInfo` passé en argument à chaque point d'interception dont l'IDL est ci-dessous :

```
local interface RequestInfo {
      readonly attribute unsigned long request_id;
      readonly attribute string operation;
      readonly attribute Dynamic::ParameterList arguments;
      readonly attribute Dynamic::ExceptionList exceptions;
      readonly attribute Dynamic::ContextList contexts;
      readonly attribute Dynamic::RequestContext operation_context;
      readonly attribute any result;
      readonly attribute boolean response_expected;
      readonly attribute Messaging::SyncScope sync_scope;
      readonly attribute ReplyStatus reply_status;
      readonly attribute Object forward_reference;
      any get_slot (in SlotId id) raises (InvalidSlot);
      IOP::ServiceContext get_request_service_context (in IOP::ServiceId id);
      IOP::ServiceContext get_reply_service_context (in IOP::ServiceId id);
};
```

Suivant le type de ce point d'interception et le type de l'intercepteur, différentes informations sont accessibles. Pour être le plus exhaustif et pour ne pas faire nous même le choix des informations pertinentes, nous loggons le maximum d'informations à chaque fois. Le choix des informations à garder se fera au moment du parsage du fichier de log par le module Log2Xmi ou par une application tiers.

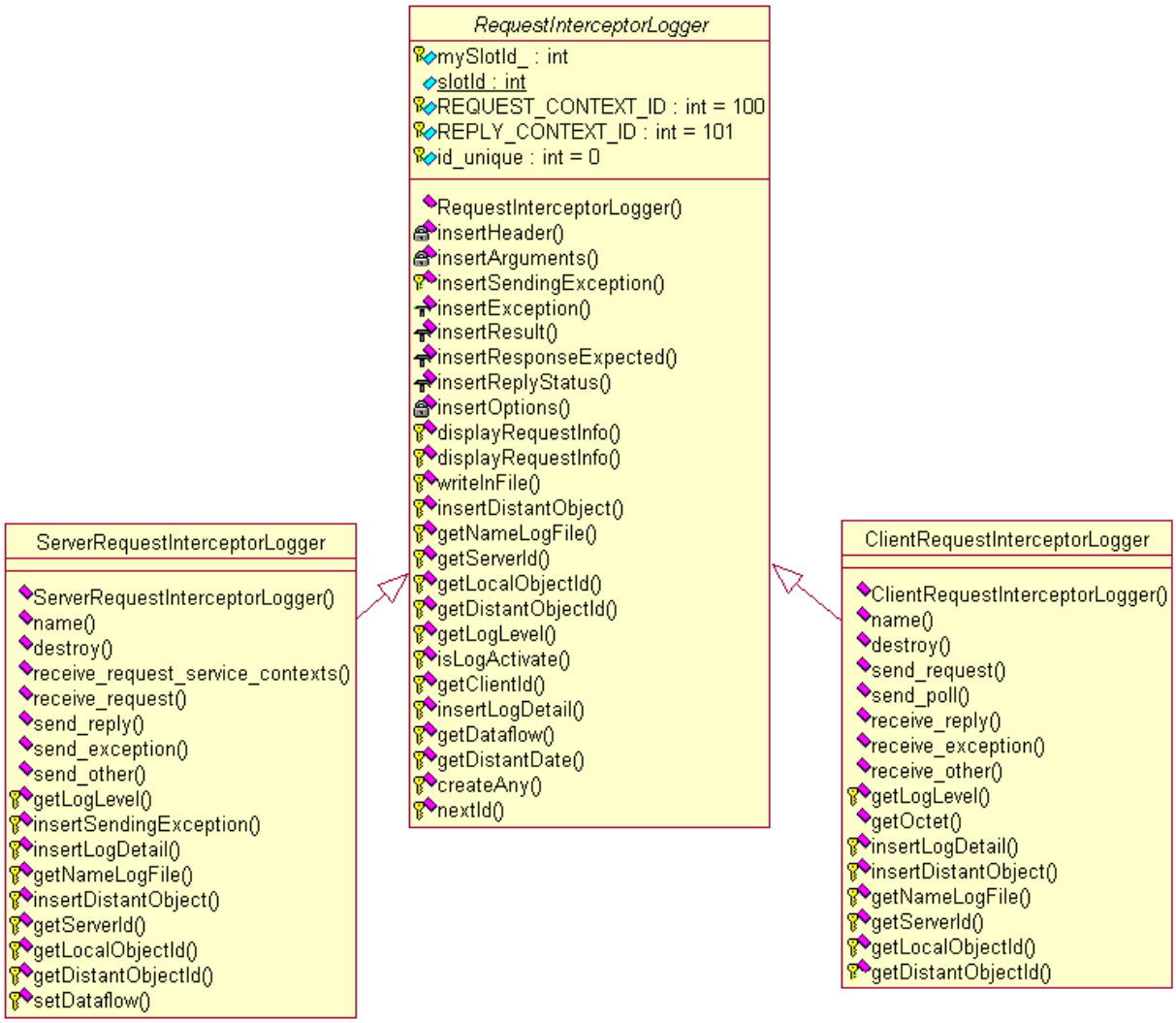
	receive_request_ service_contexts	receive_request	send_reply	send_exception	send_other
request_id	yes	yes	yes	yes	yes
operation	yes	yes	yes	yes	yes
arguments	no	yes_1	yes	no_2	no_2
exceptions	no	yes	yes	yes	yes
contexts	no	yes	yes	yes	yes
operation_context	no	yes	yes	no	no
result	no	no	yes	no	no
response_expected	yes	yes	yes	yes	yes
sync_scope	yes	yes	yes	yes	yes
reply_status	no	no	yes	yes	yes
forward_reference	no	no	no	no	yes_2
get_slot	yes	yes	yes	yes	yes
get_request_ service_context	yes	yes	yes	yes	yes
get_reply_ service_context	no	no	yes	yes	yes
sending_exception	no	no	no	yes	no
object_id	no	yes	yes	yes_3	yes_3

Figure 6 : informations disponibles pour l'intercepteur serveur 1/2

	receive_request_ service_contexts	receive_request	send_reply	send_exception	send_other
adapter_id	no	yes	yes	yes_3	yes_3
target_most_ derived_interface	no	yes	yes	yes_3	yes_3
get_server_policy	yes	yes	yes	yes	yes
set_slot	yes	yes	yes	yes	yes
target_is_a	no	yes	yes	yes_3	yes_3
add_reply_ service_context	yes	yes	yes	yes	yes

Figure 7 : informations disponibles pour l'intercepteur serveur 2/2

Entre l'intercepteur client et l'intercepteur serveur, de nombreuses informations sont communes. Il nous a donc paru tout à fait légitime de faire hériter les deux classes d'interceptions d'une même classe `RequestInterceptor`.



Celle-ci a comme point d'entrée la méthode `displayRequestInfo` qui permet de dispatcher la récupération d'information. Toutes les fonctions communes aux serveurs et aux clients sont traitées dans cette superclasse. Dès qu'une information est spécifique au type de l'intercepteur, on utilise une fonction abstraite qui est redéfinie dans les sous-classes. Cette méthode permet une grande souplesse et une factorisation du code maximum.

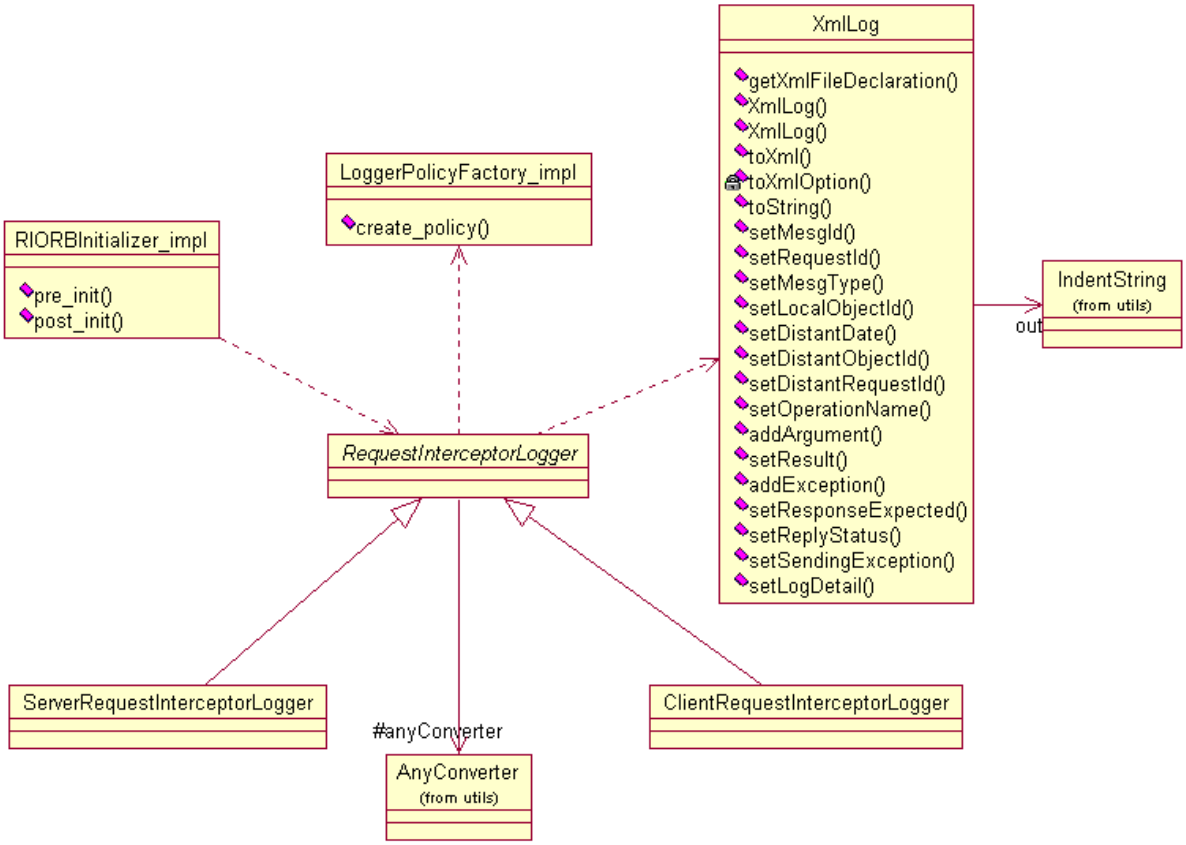
3.6.3 La construction du log

Dans les premières versions de CorbaTrace, le fichier XML était généré au fur et à mesure de la récupération des informations de l'interception. On utilisait la classe `IndentString` présentée plus haut pour directement décrire les balises XML et leur contenu. Voici un exemple du mélange entre XML et informations du RequestInfo. On s'attache ici à logger les informations sur l'identité de l'objet local et de l'objet distant et l'identité et la date de la requête.

```
private void insertHeader(IndentString out,
                          RequestInfo info,
                          String type) {
      // get current date and converts it as String using a defined date format
(df).
      Date dt = new Date();
      String laDate = df.format(dt);

      // insert the message type, the date and time.
      out.insert("<message mesg_id=\"" + nextId() + "\" " +
                "request_id=\"" + info.request_id() + "\" " +
                "type=\"" + type + "\">");

      // add an indent level.
      out.inc();

      out.insert("<local_object id=\"" + getLocalObjectId(info) +
                "\" date=\"" + laDate + "\"/>");

      insertDistantObject(out, info);
      out.newLine();
    }
```

Dès qu'une modification de la DTD avait lieu, il fallait retrouver dans les 3 classes d'intercepteurs où se situait la ligne concernée, ce qui n'était pas aisé. De plus, si on voulait utiliser plusieurs formats de log, il fallait refaire tout le code.

Bien sûr, ces problèmes étaient loin d'être critiques, mais leur résolution permet d'optimiser l'architecture globale de l'interception.



Une solution envisagée a donc été de séparer la récupération spécifique à Corba des informations à logger, de la génération du XML.

Ainsi, la classe XmlLog a été mise au point pour centraliser la génération du XML. Elle possède des méthodes du type `setXX` qui servent à stocker les informations récupérées de l'interception. Elles ne prennent en paramètre que deux types de variables : une `String` ou une collection de `String`. Ainsi, la classe XmlLog n'a pas à s'occuper de conversion qui de toute façon ne la concerne pas.

Une fois tous les méthodes d'affectation des données appelées, il suffit d'exécuter la méthode `toString()` qui va renvoyer le fichier XML. Toute la génération est ainsi centralisée dans une seule fonction, ce qui permet une modification du log de sortie très facile. Il peut également être très facile de créer une autre classe de log qui utilise les mêmes méthodes mais qui produit un log différent.

3.7 Contenu du log

3.7.1 Pourquoi XML ?

En reprenant le projet, les logs étaient stockés sous forme de fichiers en texte brut, avec un semblant de syntaxe propriétaire. Le parsage de ces logs était très fastidieux.

Le choix de XML a donc été immédiat grâce à sa facilité de parsage (avec SAX), de génération et de manipulation (avec DOM), grâce à son universalité et grâce à sa bonne intégration avec Java.

De plus, le format de présentation des communications Corba choisi est le XMI. Ce format utilise XML comme format de sauvegarde. Notre choix nous permet ainsi une architecture cohérente et homogène.

XML a encore d'autres avantages qui sont loin d'être négligeables :

- Possibilité de relecture humaine des logs
- Indépendance des plate formes et des langages
- Séparation entre les différents modules (et donc meilleure organisation du travail en équipe)

3.7.2 Explication des balises

Chacune des balises va être décrite précisément dans le chapitre suivant, où l'on décrit le parsage des logs.

3.7.3 La balise de fin

Un problème récurrent à tous les systèmes d'écriture de flux constants XML est la balise de fin XML. En effet, on écrit régulièrement dans le fichier de log en concaténant à chaque fois les nouvelles informations à sauvegarder (la balise `<message>` et ses sous-balises) avec les informations déjà écrites.

Pour la balise de début et la déclaration XML, le problème ne se pose pas car on peut l'écrire à la création du fichier. Pour la balise de fin, nous avions pensé pour contourner le problème à rajouter la balise de fin par un tout petit script shell ou par l'application récupérant les logs sur les différents ordinateurs.

Mais cette méthode aurait été une transgression du principe de cloisonnement entre émetteur et récepteur de flux XML. Nous avons donc opté pour une solution plus compliquée mais qui à l'avantage de produire en sortie de l'interception un fichier XML bien formé.

La solution consiste à écrire à la suite des nouveaux logs, la balise de fermeture `</log>`. A la prochaine ouverture du fichier pour un nouveau log, on effacera cette balise avant d'écrire les nouvelles informations.

4 Log2XMI

4.1 Introduction

Cette partie s'intercale comme le montre le schéma entre la partie de l'application « interception des messages » et la partie « visualisation des diagrammes de séquence ».

Sa finalité est de transformer les différents fichiers d'interception des messages (les fichiers de logs) qui sont au format XML (avec une DTD qui nous est propre), en un fichier unique au format XMI, une DTD standardisée par l'OMG, appropriée pour les représentations en diagrammes de séquences.

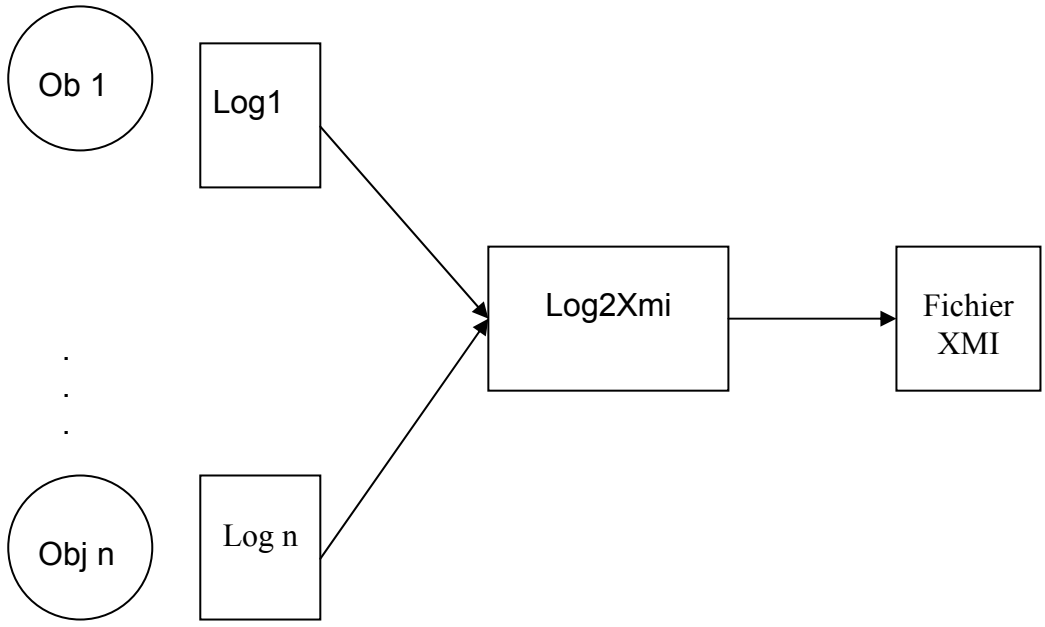


Figure 8 : Application répartie

Pour cela, nous avons décomposé le travail en 5 étapes distinctes :

- **L'analyse des différents fichiers de logs** (parsage). Ces fichiers ne contiennent des informations que sur des demi-messages, ou plus précisément sur l'envoi d'un message ou sur la réception d'un message.
- **La fusion des demi-messages**. On regroupe les demi-messages correspondants afin d'obtenir un message complet.
- **La synchronisation de tous les messages** complets et incomplets obtenus, afin d'obtenir une séquence chronologiquement correcte des messages.
- **Le filtrage des informations**. Toutes les données interceptées ne sont pas forcément intéressantes, on choisit ce que l'on considère comme le plus pertinent.
- **Création du fichier XMI,** à partir des informations précédentes.

Nous allons maintenant présenter l'architecture globale de ce module, les choix effectués et leurs justifications, puis ensuite présenter les différents types de messages que nous avons besoin, puis présenter en détail les différentes étapes du processus.

4.2 L'architecture globale

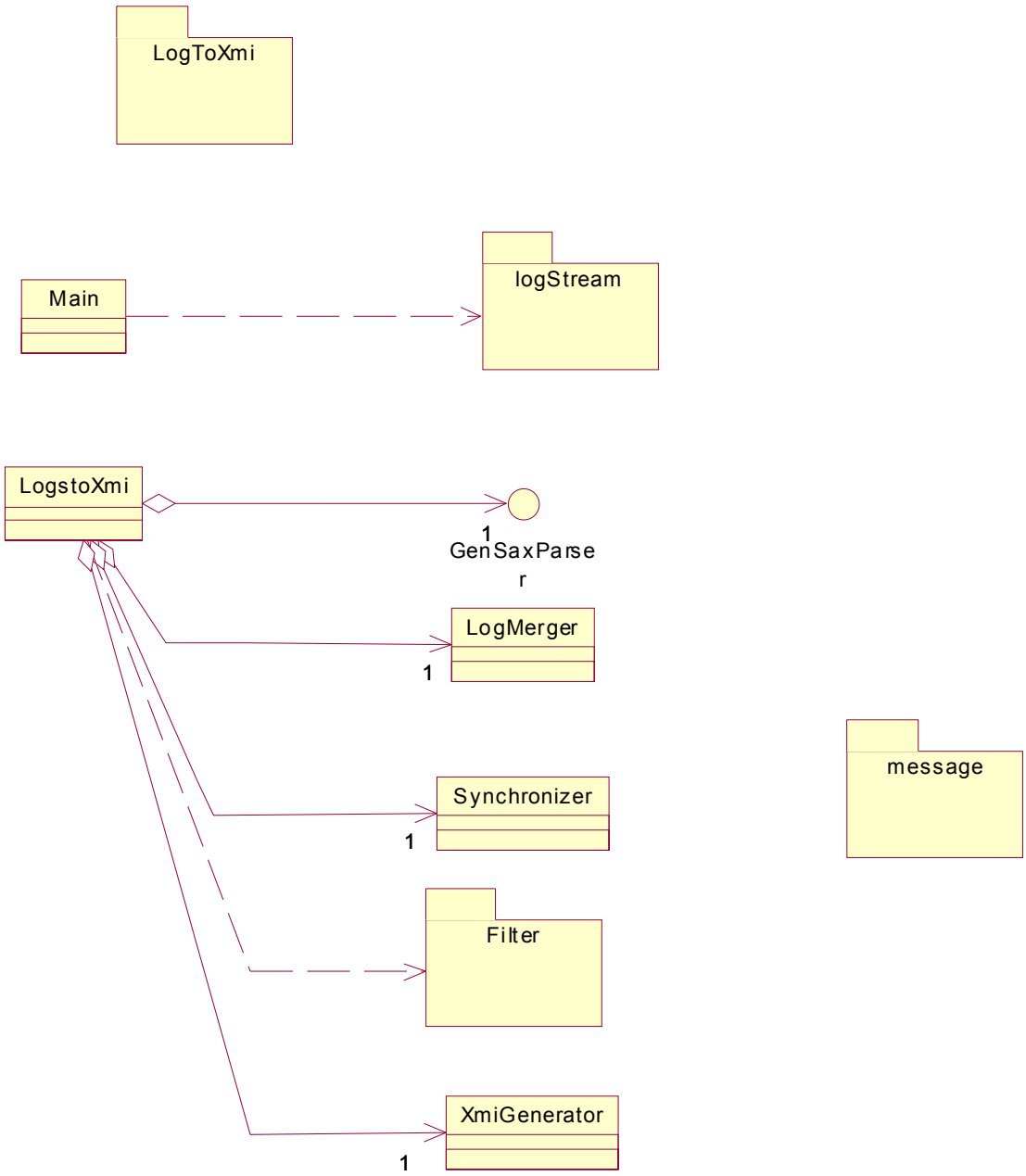
Lorsque s'est posée la question de réaliser l'application, nous avions l'objectif de pouvoir rajouter des fichiers de logs « à chaud ». C'est-à-dire que, après avoir lancer le traitement une première fois sur des fichiers de logs, on voulait s'offrir la possibilité de compléter les logs existants et obtenir le bon résultat sans pour autant relancer tout le traitement. Cela a impliqué de pouvoir sauvegarder, au moins en mémoire, les résultats obtenus et de pouvoir ajouter facilement les compléments d'informations.

Ensuite, puisque nous sauvegardions déjà des résultats en mémoire, nous nous sommes dit que ces résultats pouvaient aussi servir à modifier « à chaud » les options des filtres, en relançant le traitement uniquement à partir de ces sauvegardes. En effet, relancer tout le travail en amont serait futile si seules quelques options de filtres ont changé. C'est d'autant plus judicieux que CorbaTrace est sensé être un outil de débuggage, et que lorsqu'on débugue, on avance généralement par tâtonnement pour situer l'erreur.

De même, pour la création du fichier final, il existe plusieurs outils de visualisation qui possèdent leur format de fichier XMI propre. Même si XMI est normalisé, une partie (les extensions) est laissée sans contraintes pour permettre à chaque outil UML d'y apporter ses informations propriétaires. Les sauvegardes de données permettent la génération des fichiers XMI sous des formats différents, sans forcément ré-effectuer l'intégralité du processus.

L'analyse préliminaire a donc aboutie à une architecture qui réalise les étapes l'une après l'autre (pas de parallélisme de tâches), et qui sauvegarde des données après chaque étape.

Bien évidemment, toutes modifications des données d'une étape relance le traitement sur toutes les étapes en aval.



Le diagramme UML ci-dessus décrit notre architecture globale. On remarque que chaque étape est réalisée par un objet indépendamment des autres étapes. Donc chacun de ces objets a la possibilité de sauvegarder ce dont il a besoin.

4.3 Solution pour une reprise à chaud

Notre système doit permettre de rajouter des logs qui n'avaient pas été pris en compte lors d'un premier parsage. A terme, il pourrait mettre être possible de visualiser en temps réel les interactions entre objets distribués.

Nous avons essayé de procéder à des choix architecturaux nous permettant d'ajouter des logs sans avoir à tout reconstruire à chaque fois. Avec la futur interface graphique, l'intérêt de ce mode de fonctionnement deviendra vite évident.

Pour détailler plus précisément ce qui peut être sauvegardé, repris, ou modifié, nous allons reprendre le séquencement des étapes une à une. Chacune de ces étapes sera décrite plus précisément dans les chapitres qui suivent.

1. La première étape qui concerne le parsage, n'a aucune raison de ré-analyser les fichiers déjà analyser. Nous considérons que lorsqu'un fichier est traité, il l'est une fois pour toute. Les demi-messages qui en sont extraits sont stockés (non pas au niveau de cette étape, mais à l'étape suivante) et ce fichier n'a plus d'utilité. Cela oblige par contre l'utilisateur à ne pas ajouter des logs qui existent déjà, même s'ils ont des informations complémentaires (seules les informations complémentaires nous intéressent). Pour résumé, l'étape de parsage ne prend en compte que les nouvelles séries de fichiers qui ne doivent comporter que des informations nouvelles.

2. La seconde étape, celle qui fusionne les demi-messages pour obtenir des messages complets, mémorise tout les messages complets déjà obtenus. Une fois un message complet obtenu, il n'a aucune raison d'être modifié. Nous identifions nos deux moitiés de messages de telle manière qu'aucune information ne peut remettre en cause ce qui a déjà été fait. La sauvegarde des données nous a semblé ici très évidente à réaliser. Nous sauvegardons aussi à cette étape les demi-messages, obtenu lors d'un parsage, et auxquels nous n'avons put associer leur moitié. Ces demi-messages pourront plus tard être associés avec des nouveaux demi-messages lorsque de nouvelles informations arriveront. Pour résumé, dans cette étape, pour chaque parsage effectué, nous réalisons une opération de fusion uniquement sur les demi-messages (ceux obtenus au cours du parsage, et ceux obtenus au cours d'un traitement antérieur).

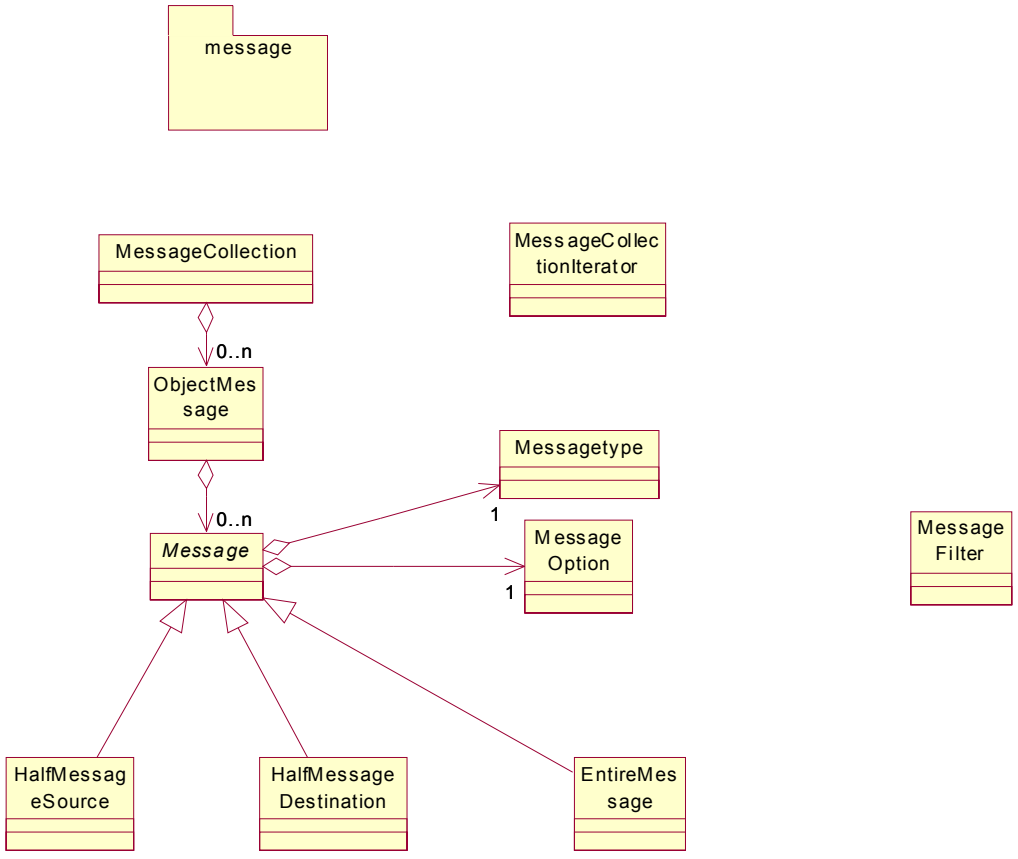
3. La troisième étape traite de la synchronisation des messages. Pour cette étape tout est remis en question. Il est très incertain de synchroniser de nouveaux messages à partir d'une synchronisation déjà existante, cela tout simplement parce que nous pouvons obtenir un message dont la réception intervient avant l'envoi. Ce qui est totalement aberrant. Pour chaque nouveau message, ou lot de nouveaux messages obtenus, nous sommes donc obligé de relancer la synchronisation.

4. La quatrième étape est le filtrage des données. Ces filtrages d'informations sont des sélections des données pertinentes à partir des données obtenues. Par conséquent si les sélections changent, mais que les données ne changent pas, il n'y a pas de raison de relancer l'intégralité du traitement. Nous mémorisons donc ici l'intégralité des messages synchronisés, complets et incomplets, obtenus précédemment,

5. La cinquième étape est la génération, à partir des données filtrées obtenues, d'un fichier XMI pour un outils bien précis. Ici aussi, quelque soit l'outil de visualisation, si les données filtrées ne changent pas, il n'y a pas de raison de relancer l'application. Les différents fichiers XMI peuvent être obtenus à partir des mêmes données en modifiant seulement une option du générateur de fichier.

4.4 La structure des messages

Pour les besoins de notre application nous avons besoin de pouvoir différencier différentes sortes de messages. C'est pour cela que nous avons réalisé l'architecture décrite ci-dessous.



Comme on peut le voir sur ce diagramme UML, nous avons une classe abstraite Message qui se dérive en classes HalfMessageSource, HalfMessageDestination et EntireMessage. Ces différentes classes répondent à nos besoins pour modéliser les différentes sortes de messages auxquelles nous sommes confrontés, c'est–à-dire les messages complets (EntireMessage) et les demi-messages qu'ils soient interceptés à la source (HalfMessageSource) ou à leur destination (HalfMessageDestination). Un message est donc constitué d'un type qui nous indique si le message est une exception, une requête ou une réponse, et s'il est complet. S'il n'est pas complet, c'est donc un demi-message source ou destination. Le cas échéant, ce type peut aussi indiqué que le message est incomplet, mais cette information n'est utile que pour la visualisation du diagramme de séquences.

Comme nous n'avons pas qu'un seul message à manipuler, il nous faut disposer de collection de message. Ces collections existent sous la forme des classes ObjectMessage et MessageCollection. La collection ObjectMessage répertorie tout les messages liés à un seul objet, ces messages sont ensuite classés chronologiquement. La classe MessageCollection n'est ensuite qu'une collection de ces ObjectMessage, qui sont classés par ordre alphabétique des identifiants de message. Ce choix de structure nous permet de classer très précisément les messages dont on dispose, et par la suite de parcourir nos collections très facilement, et sans perte de temps.

Les MessageFilter quant à eux nous permettent de modéliser les messages avec des données filtrées. Ce sont donc des messages avec des informations volontairement épurées.

4.5 Le parsage des logs

4.5.1 Le choix de XML

Comme dit précédemment, nous avons fait le choix de définir des fichiers de logs en XML.

Ce choix se justifie par plusieurs points :

- XML est à l'heure actuelle un format qui s'impose comme le format de manipulation de données le plus utilisé, notamment par sa facilité de lecture.
- Il existe deux standards pour la manipulation des fichiers XML : DOM et SAX. Grâce à cet effort de standardisation, des moteurs de parsage pour XML existent dans la plupart des langages pour SAX et DOM (c'est le cas en Java notamment, avec JAXP, directement intégré depuis le JDK 1.4, mais aussi par des bibliothèques tierces comme Xerces). De fait, le parsage de fichiers XML en est facilité.
- XML par sa simplicité est très adapté à des transformations. Des langages comme XSLT permettent de transformer des fichiers XML en d'autres fichiers XML. C'est le format universel par excellence, souvent utilisé comme format commun pour faire communiquer les données de plusieurs applications entre elles.

- XML définit deux types de documents : les documents bien formés, c'est à dire cohérent vis à vis des syntaxes utilisées par XML (tags d'ouverture, de fermeture, etc.), et les documents valides, plus intéressants car basés sur une DTD. La DTD définit en un sens les règles de production de la grammaire utilisée pour les données XML. Définir une DTD est donc un gage de qualité dans le traitement des fichiers XML.

4.5.2 Les logs et leur DTD

Nous avons donc défini une DTD pour les logs. Nous définissons aussi dans cette DTD les bons formats de filtres (nous y reviendrons plus loin).

Pour définir cette DTD, notre souci est de pouvoir loguer le maximum d'informations interceptées, même si une bonne partie d'entre elles ne nous est pas directement utile pour définir les diagrammes de séquence.

Le principe est d'encapsuler tous les messages sous un tag <log> </log>.
Cela permet de les distinguer d'un filtre <filter> </filter>.

C'est donc le point d'entrée du fichier XML.

Chaque log est donc formé de messages :

```
<!ELEMENT log (message*)>
```

Chaque message est formé d'informations sur l'objet local, sur l'objet distant s'il y en a, sur l'opération du message, son résultat, puis l'ensemble de ses options loguées.
Chaque message possède aussi un identifiant de message, ainsi qu'un identifiant de requête, et un type de message (send_request, receive_request, send_reply, receive_reply, etc.)
Chaque objet, local ou distant, possède comme informations l'identifiant de l'objet, et la date à laquelle l'objet a émis ou reçu le message (en horaire local à l'objet).

Pour la DTD, nous avons fait le choix de plutôt favoriser l'utilisation d'attributs que de sous-éléments XML. C'est généralement un débat entre utilisateurs d'XML. Il nous a semblé intéressant d'utiliser des sous tags lorsqu'il y avait réellement une notion d'encapsulation. Par exemple les arguments d'une opération ou les messages d'un log. Lorsque les informations décrivent davantage un statut ou une valeur unique d'un objet, nous préférons utiliser des attributs pour le tag (c'est le cas par exemple des identifiants de messages ou d'objets).

Tout cela reste évidemment bien subjectif.

Ici, nous avons donc les messages de la DTD sous la forme :

```
<!ELEMENT message (local_object, distant_object?, operation, result?, options?)>
<!ATTLIST message
```

```
          mesg_id CDATA #REQUIRED
          request_id CDATA #REQUIRED
          type CDATA #REQUIRED>

<!ELEMENT local_object EMPTY>
<!ATTLIST local_object
          id CDATA #REQUIRED
          date CDATA #REQUIRED>
<!ELEMENT distant_object EMPTY>
<!ATTLIST distant_object
          id CDATA #REQUIRED
          date CDATA #REQUIRED
          request_id CDATA #REQUIRED>
```

Chaque message est aussi composé d'une opération. Celle-ci est formée d'arguments.

Chaque argument a un type de donnée associé, une valeur, une information pour indiquer la sorte de paramètre (entrée, sortie, ou entrée/sortie), et enfin, accessoirement, un nom d'opération.

Le résultat de l'opération est quant à lui défini par un type de donnée et une valeur de résultat. Le résultat est bien sûr optionnel au niveau du message puisque seuls des messages de réponses (reply) peuvent donner une telle information.

L'opération est ainsi décrite comme suit dans la DTD :

```
<!ELEMENT operation (argument*)>
<!ATTLIST operation
            name CDATA #REQUIRED>

<!ELEMENT argument EMPTY>
<!ATTLIST argument
            inout (in|out|inout) #REQUIRED
            name CDATA #IMPLIED
            value CDATA #REQUIRED
            type CDATA #REQUIRED>

<!ELEMENT result EMPTY>
<!ATTLIST result
            type CDATA #IMPLIED
            value CDATA #IMPLIED>
```

Les options arrivent ensuite. Celles-ci parfois optionnelles. Nous nous sommes basés pour cela sur les spécifications Corba qui décrivent quelles informations les intercepteurs Corba peuvent récupérer, et donc quelles informations sont loguables.

Les options sont les suivantes :

```
<!ELEMENT options (
                    request_id?,
                    exceptions?,
                    contexts?,
                    operation_context?,
                    responseExpected?,
```

```
                    sync_scope?,
                    reply_status?,
                    forward_reference?,
                    slots?,
                    request_service_contexts?,
                    reply_service_contexts?,
                    sending_exception?,

                    object_id?,
                    adapter_id?,
                    target_most_derived_interface?,
                    server_policies?,
                    target_is_a?,

                    target?,
                    effective_target?,
                    effective_profile?,
                    received_exception?,
                    received_exception_id?,
                    effective_components?,
                    request_policies?,
                )>
```

Sans rentrer trop dans le détail, on peut distinguer deux catégories d'options :

- les options qui n'ont qu'une ou plusieurs informations simples. Ce sont des tags avec généralement un attribut ou plusieurs attributs de type chaînes (#CDATA).

  Ainsi, un request_id, received_exception_id, adapter_id, et object_id ont tous un attribut "id", responseExpected et target_is_a une seule valeur (value), “sync_scope” une valeur parmi “sync_none”, “sync_with_transport”, “sync_with_server”, et “sync_with_target” (l'information est obligatoire, d'où le #REQUIRED dans la DTD pour cet attribut de l'option), reply_status a cinq possibilités (successful , etc.).
  L'option “effective_profile” a une information de tag et une de profile (profile_data).
  D'autres valeurs d'attributs courantes sont le type (forward_reference, sending_exception, received_exception, target, effective_target), l'IOR (forward_reference, target, effective_target), ou encore un nom ou une simple valeur.

  La DTD est ainsi définie comme suit pour ce type d'options :

```
<!ELEMENT request_id EMPTY>
<!ATTLIST request_id
            id CDATA #REQUIRED>

<!ELEMENT responseExpected EMPTY>
<!ATTLIST responseExpected
            value CDATA #REQUIRED>

<!ELEMENT sync_scope EMPTY>
<!ATTLIST sync_scope
```

```
            value (sync_none | sync_with_transport | sync_with_server |
sync_with_target) #REQUIRED>

<!ELEMENT reply_status EMPTY>
<!ATTLIST reply_status
            value (successful | system_exception | user_exception |
location_forward | transport_retry) #REQUIRED>

<!ELEMENT forward_reference EMPTY>
<!ATTLIST forward_reference
            type CDATA #REQUIRED
            implementation_name CDATA #REQUIRED
            IOR CDATA #REQUIRED>

<!ELEMENT sending_exception EMPTY>
<!ATTLIST sending_exception
            name CDATA #REQUIRED
            value CDATA #REQUIRED
            type CDATA #REQUIRED>

<!ELEMENT object_id  EMPTY>
<!ATTLIST object_id
            id CDATA #REQUIRED>

<!ELEMENT adapter_id   EMPTY>
<!ATTLIST adapter_id
            id CDATA #REQUIRED>

<!ELEMENT target_most_derived_interface  EMPTY>
<!ATTLIST target_most_derived_interface
            id CDATA #REQUIRED>

<!ELEMENT target_is_a EMPTY>
<!ATTLIST target_is_a
            value CDATA #REQUIRED>

<!ELEMENT target EMPTY>
<!ATTLIST target
            type CDATA #REQUIRED
            implementation_name CDATA #REQUIRED
            IOR CDATA #REQUIRED>

<!ELEMENT effective_target  EMPTY>
<!ATTLIST effective_target
            type CDATA #REQUIRED
            implementation_name CDATA #REQUIRED
            IOR CDATA #REQUIRED>

<!ELEMENT effective_profile EMPTY>
<!ATTLIST effective_profile
            tag CDATA #REQUIRED
            profile_data CDATA #REQUIRED>

<!ELEMENT received_exception EMPTY>
<!ATTLIST received_exception
            name CDATA #REQUIRED
            value CDATA #REQUIRED
```

```
            type CDATA #REQUIRED>

<!ELEMENT received_exception_id EMPTY>
<!ATTLIST received_exception_id
            id CDATA #REQUIRED>
```

- la seconde catégorie d'options est celle des options qui encapsulent une ou plusieurs information(s). L'opération est dans ce cas même si elle est traitée à part. Cette distinction est aussi faite dans notre structure de message, un MessageOption pouvant encapsuler un autre MessageOption et ainsi de suite.

  Nous avons par exemple les exceptions qui sont formées d'une ou plusieurs exception(s), les contextes qui sont formés de plusieurs contextes qui eux-mêmes ont une ou plusieurs propriétés (property), chaque propriété ayant un nom ou une valeur. Un contexte d'opération (operation_context) est lui aussi formé de plusieurs propriétés du contexte (context_property) :

```
<!ELEMENT exceptions (exception)*>
<!ELEMENT exception EMPTY>
<!ATTLIST exception
            name CDATA #REQUIRED>

<!ELEMENT contexts (context)*>
<!ELEMENT context (property)*>
<!ELEMENT property EMPTY>
<!ATTLIST property
            name CDATA #REQUIRED
            string_value CDATA #REQUIRED>

<!ELEMENT operation_context (context_property)*>
<!ELEMENT context_property EMPTY>
<!ATTLIST context_property
            name CDATA #REQUIRED
            string_value CDATA #REQUIRED>
```

De même l'option des slots est composée de plusieurs slots, chacun ayant un id (slot_id), un nom (name), une valeur (value), et un type.
Les options request_service_contexts et reply_service_contexts sont toutes deux composées de service_context, chacun de ces contextes ayant un id, un identifiant de service (service_id), et des données (data). De même les politiques de serveur (server_policies) et de requête (request_policies) sont formées de policy (avec la politique en tant que telle et son type), et enfin les composants effectifs formés de plusieurs composants (avec le tag et les données du composant) :

```
<!ELEMENT slots (slot)*>
<!ELEMENT slot EMPTY>
<!ATTLIST slot
            slot_id CDATA #REQUIRED
            name CDATA #REQUIRED
            value CDATA #REQUIRED
            type CDATA #REQUIRED>
```

```
<!ELEMENT request_service_contexts (service_context)*>
<!ELEMENT reply_service_contexts (service_context)*>
<!ELEMENT service_context  EMPTY>
<!ATTLIST service_context
            service_id CDATA #REQUIRED
            id CDATA #REQUIRED
            data CDATA #REQUIRED>

<!ELEMENT server_policies (Policy)*>

<!ELEMENT request_policies (policy)*>
<!ELEMENT policy EMPTY>
<!ATTLIST policy
            policy_type CDATA #REQUIRED
            policy CDATA #REQUIRED>

<!ELEMENT effective_components (effective_component)*>
<!ELEMENT effective_component EMPTY>
<!ATTLIST effective_component
            tag CDATA #REQUIRED
            component_data CDATA #REQUIRED>
```

Bien sûr, la meilleure façon de cerner le format des logs est de visualiser un exemple de log en soit. Référez-vous à la partie sur les interceptions proprement dite pour avoir un exemple sûrement plus utile à la compréhension.

Ainsi, l'intérêt d'utiliser XML et de définir une DTD pour nos logs est particulièrement bénéfique pour plusieurs points :

- les logs sont faciles à lire, même si très volumineux.
- les logs sont plus aisés à parser grâce aux librairies compatibles avec SAX et DOM définies pour la plupart des langages.
- et surtout, le format étant particulièrement maniable, il peut tout à fait intéresser d'autres développeurs qui souhaitent définir leur propre outil de traitement des logs, quitte à passer par un format de fichiers plus personnel, après avoir effectué une phase de transformation des logs (avec XSLT par exemple).

4.5.3 Le parsage des logs par l'outil log2xmi

4.5.3.1 Le choix de SAX

Pour le parsage des logs, nous utilisons les librairies prédéfinies en Java, et intégrées au JDK depuis sa version 1.4.

Pour le parsage du fichier XML, nous avons le choix entre SAX et DOM :

- avec **DOM**, le fichier est entièrement parsé sous forme d'un arbre. L'ensemble du fichier XML se trouve donc en mémoire. Chaque nœud voisin détermine les attributs et chaque nœud fils les sous éléments (les tags imbriqués). L'avantage est que le parsage se fait de manière automatique, donc il n'y a ensuite plus qu'à parcourir l'arbre et effectuer les opérations souhaitées. DOM est très adapté à des DTD très détaillées, et prend tout son intérêt dès qu'il s'agit de générer à partir de l'arbre un autre format de fichier (xml ou non).

- avec **SAX**, nous retrouvons une utilisation plus classique des outils de parsage comme Yacc ou Cup par exemple : à chaque réduction de règle détectée, il faut définir l'action à réaliser, généralement l'ajout d'un élément dans une structure interne. Pour XML, la notion de règles est simplifiée à son maximum : il s'agit en fait de détecter les tags de débuts et de fin, ainsi que le début et la fin d'un document XML donné. Sax est plus adapté à la construction incrémentale des données en mémoire, au fur et à mesure de la lecture du fichier XML, et sous la forme d'une structure interne plus utile à l'application que ne le sont les arbres de DOM.

C'est pour cette raison que nous avons fait le choix d'utiliser plutôt SAX, puisque notre structure de messages est, selon nous, mieux adaptée aux futurs traitements à appliquer sur l'ensemble des messages (fusion, synchronisation, et filtres).
DOM, par contre est particulièrement utile dès que les structures XML des fichiers à parser deviennent trop lourdes, ce qui n'est pas vraiment le cas de nos logs. DOM est par exemple bien plus adapté à la construction des arbres XMI. Nous y reviendrons beaucoup plus en détail par la suite.

4.5.3.2 SAX et Java

Nos avons utilisé SAX dans sa version 2. Des packages pour SAX2 sont définis depuis très peu de temps dans le JDK (depuis la version 1.4). Le moteur de parsage de la machine virtuelle est connu sous le nom de JAXP. Pour les versions antérieures, on peut lui substituer le parseur Xerces disponible en GPL. D'autres parseurs compatibles avec SAX2 peuvent aussi être utilisés (et peuvent se révéler plus performants). A partir du JDK1.4, il est possible de définir les librairies à utiliser par défaut soit globalement dans la configuration du jdk, soit en passant en paramètre au compilateur le package à substituer à celui par défaut du JDK1.4.

Comme nous avons aussi fait le choix d'intégrer les filtres à la DTD, donc aussi sous forme de fichier XML, nous trouvions intéressant de définir une classe de parsage commune aux deux processus de parsage (les logs et les filtres). Cette classe « *GenSaxParser* » (du package corbaTrace.log2xmi.parser) permet de bien comprendre l'initialisation du parser SAX2.

Pour fonctionner, le parseur nécessite un handler (qui dérive de DefaultHandler). C'est dans cet handler que l'on définit ce que doit faire le parseur lorsqu'il a détecté un nouveau tag d'élément XML. On définit donc par la suite un handler pour les logs (LogHandler) et un autre pour les filtres (FilterHandler) que l'on passe initialement au constructeur et que *GenSaxParser* utilisera à chaque nouveau fichier XML.

Le principe est le suivant :

1) création d'une nouvelle « fabrique » de parsage (SAXParserFactory)

```
SAXParserFactory factory = SAXParserFactory.newInstance();
```

2) définition ou non de l'état de validation du parser. En effet, si on le souhaite, par défaut c'est le cas, mais l'utilisateur peut indiquer une option à la commande log2xmi pour ne pas valider les fichiers XML. Ceci est surtout utile pour les filtres puisqu'ils sont modifiés à la main par l'utilisateur, donc sont susceptibles de contenir quelques erreurs par rapport à la DTD.

```
factory.setValidating(validatedXML);
```

3) un nouveau parseur est alors créé : (saxParser est une donnée membre de la classe, de type javax.xml.parsers.SAXParser)

```
saxParser = factory.newSAXParser();
```

4) Ensuite, dès qu'un nouveau fichier XML est donné à parser, on le passe en paramètre à une méthode startParsingFile() (sous forme de fichier ou d'URL). Celle-ci appelle la méthode parse() du parseur créé précédemment pour ce fichier et pour l'handler correspondant (celui des filtres ou des logs) :

```
saxParser.parse(file, handler);
```

Ensuite, à chaque fin ou début d'élément XML (<element>, </element>), un événement est provoqué par le parseur, cet événement est traduit en un appel de méthode du Handler utilisé.

Chaque Handler doit donc redéfinir les méthodes suivantes :
- *startDocument()* : elle est appelée au début du fichier XML.
- *endDocument()* : elle est appelée en fin de document XML.
- *startElement()* : elle est appelée dès qu'un nouveau tag ouvrant est détecté. On détermine alors le tag en question, et en fonction des actions qu'on souhaite effectuer pour ce tag, on lit chaque attribut du tag à partir du paramètre d'attributs de la méthode (objet de la classe Attributes). A partir des méthodes *getLocalName*(numero_attribut) et *getValue*(numero_attribut), on peut déterminer le nom et la valeur de chaque attribut pour cet élément XML (et exploiter cette valeur).
- *endElement()* : elle est appelée dès qu'un nouveau tag fermant est détecté.

- *characters()* : elle est appelée lorsqu'une chaîne se trouve entre deux tags ouvrant et fermant.

Ainsi que les méthodes d'erreur de parsage : *error()* (appelé à chaque erreur si la validation du fichier à partir de la DTD est activée), et *fatalError()* (problème de document mal formé).

Dans notre cas, chaque erreur de parsage provoque une exception qui est propagée à la méthode appelante (startParsingFile()), qui la propage elle-même à la classe qui demande le parsage, à savoir LogstoXmi, qui se charge alors d'afficher la ligne et le message d'erreur.

Comme chaque information terminale (les chaînes de caractères qui forment l'information, #CDATA dans la DTD) est sous la forme d'attributs d'élément XML, aucune information terminale ne se trouve entre deux tags ouvrant et fermant. La méthode characters() n'est donc pas utilisée.

4.5.3.3 Le parsage des logs

Nous définissons un parseur spécifique pour les logs. Il hérite du parseur générique *GenSaxParser*. Il utilise un handler pour les fichiers de log (LogHandler) et propose deux méthodes d'accès à des Collections de messages (getHalfDestinationMessages() et getHalfSourceMessages()) pour récupérer les messages incomplets lus à partir des différents fichiers parsés.

Le parseur appelle en fait ces mêmes méthodes depuis son handler de logs.

LogHandler définit la manière d'exploiter les données parsées depuis les fichiers de logs. Il conserve deux collections de messages correspondant à l'état actuel des messages lus sans erreur, et correspondent aux messages dont seule la destination est connue (de type receive_request, receive_reply, ou receive_exception), c'est à dire à des HalfMessageDestination, et ce dont seule la source est connue (de type send_request, send_reply, ou send_exception), c'est à dire à des HalfMessageSource.

Le principe général de fonctionnement de LogHandler est le suivant :
- on travaille avec un objet Message temporaire.
- dès qu'il y a une incohérence ou une erreur de parsage, le message temporaire est détruit (car non exploitable).

- à chaque élément lu (méthode startElement() appelée), on détermine quel est le tag en question. Si c'est un <message>, alors on initialise le Message temporaire, soit comme HalfMessageSource, soit comme HalfMessageDestination, en fonction du type de message lu pour l'attribut « type » de l'élément. Ensuite, pour chaque option, on crée un MessageOption temporaire dans le lequel on ajoute les informations nécessaires. Si besoin est, dans le cas d'options encapsulées, on utilise une liste d'options temporaires. C'est lorsque le tag de fin est détecté (endElement()) que la liste des options est définitivement ajoutée pour le MessageOption temporaire, ou que le MessageOption temporaire est définitivement ajouté dans le Message temporaire, ou s'il s'agit d'un message qui se termine, celui-ci est définitivement ajouté dans la liste des messages incomplets (source ou destination, selon le type du message), et le message temporaire est libéré.

  Le cas des opérations est traité comme un MessageOption, puisqu'une opération est en interne un MessageOption (pour un message), mais c'est lors de l'ajout du MessageOption que l'on détecte s'il s'agit d'une opération ou d'une simple option.

Se référer aux sources pour le détail du code (attention, le code assez fastidieux à lire !)

4.6 La fusion des messages

4.6.1 Le cas général des messages bien formés

Le but de cette étape est, on le rappelle, de rassembler les deux moitiés de chaque message. Pour cela nous disposons de deux listes de messages. L'une contenant toutes les sources de messages, et l'autre contenant toutes les destinations.

Pour cette étape, nous faisons remarquer que de manière générale, c'est toujours la partie interceptée par le serveur qui contient le plus d'informations. Par conséquent selon que l'on souhaite rassembler les moitiés d'un message de type request, reply, ou bien exception il faudra considérer la source du message ou sa destination. Pour les messages de type request, il faut commencer par considérer la partie destination du message, et pour les reply et les exception il faut commencer par considérer la source du message.

La fusion des messages se déroule ainsi :

- Nous considérons tout d'abord les messages de type request. Comme nous l'avons dit : c'est le serveur qui a enregistré la plus d'information sur ce message. Nous nous intéressons donc à la liste des parties destinations de messages pour étudier de plus près tout les demi-messages de type RECEIVE_REQUEST. A partir d'un message nous pouvons obtenir l'identifiant de l'objet qui a envoyé ce message, et la date à laquelle a été envoyé ce message.

- Il nous suffit alors de rechercher dans la liste des sources de messages, un demi-message de type SEND_REQUEST qui a été envoyé par un objet ayant l'identifiant que l'on a récupéré et à la date que l'on a récupéré. Si ce message est trouvé, on peut construire le message complet(réunion de toutes les informations des demi-messages) et l'ajouter à la liste des messages complets. Il ne faut pas oublier ensuite de supprimer les demi-messages, qui deviennent inutile maintenan,t dans chacune des listes. Si aucun message n'a été trouvé rien de spécial n'est réalisé. Pas de création de nouveau message, et le demi-message destination reste dans sa liste. Il pourra peut-être complété par des demi-messages provenant de nouveaux fichiers de logs.
- Pour les reply et les exception, on procède à un raisonnement similaire, même si les informations dont ont dispose sont plus limitées. On parcours la liste des parties sources des messages pour analyser tous les messages de type SEND_REPLY et SEND_EXCEPTION et, pour chacun d'entre eux, on récupère l'identifiant de l'objet destinataire. Contrairement à la fusion des précédents messages, ici nous considérons les parties sources des messages et par conséquent, on ne peut pas récupérer la date de la réception du message, ce qui est normal puisque l'on ne la connaît pas encore, mais nous connaissons le request_id du message. Grâce à ces informations on a la possibilité de retrouver l'autre moitié de message dans la liste des parties destination des messages.

4.6.2 Le cas particulier des messages incomplets

Les messages incomplets apportent une information supplémentaire pour l'utilisateur. Ces messages peuvent être incomplets parce qu'il manque un fichier de log (ou que ce fichier est corrompu), mais surtout, et c'est là l'intérêt de les conserver, lorsque le message n'a effectivement pas pu atteindre sa destination.

C'est donc un moyen pour l'utilisateur de connaître les messages qui se sont perdus (par exemple parce que l'objet distant est sur une machine qui a planté ou bien parce que la charge réseau était trop importante).

Ainsi, nous ajoutons les messages incomplets une fois la fusion effectuée afin de pouvoir les afficher. Pour « simuler » un message entier, nous complétons les informations manquantes du message en faisant intervenir un objet « inconnu » (BROKEN_OBJECT), et si besoin est une date par défaut (BROKEN_DATE). Le type du message issu de la « pseudo-fusion » est lui aussi adapté à la situation : un send_reply ou receive_request devient un message de type BROKEN_REQUEST, et de la même manière, des messages BROKEN_REPLY et BROKEN_EXCEPTION peuvent être créés.

Note : nous faisons cette distinction, car comme nous le verrons par la suite, nous n'ajoutons pour la génération XMI que l'opération au niveau de l'objet qui propose réellement l'opération, c'est à dire l'objet qui joue le rôle d' « objet serveur » (c'est à dire que l'opération se trouve sur l'objet émetteur dans le cas d'une réponse, et sur l'objet récepteur dans le cas d'une requête). De cette manière, même si un message est incomplet l'opération sera placée convenablement, soit sur l'objet du serveur en question si c'est la partie connue du message, soit sur l'objet inconnu le cas échéant.

4.7 La synchronisation des messages

Une fois les messages fusionnés, nous devons préparer la génération du diagramme de séquence. Le problème qui se pose est que chaque date au niveau de chaque objet est la date locale à l'objet. On voit aisément que deux objets qui tournent sur des ORB dans des fuseaux horaires différents donneront des décalages de dates si on les garde tels quels. Il faut donc synchroniser les objets, et faute de pouvoir le faire en temps réel, nous le faisons à posteriori une fois tous les messages logués et fusionnés.

Le principe général de toute synchronisation est de choisir un objet comme référence, puis de recalculer les dates locales des autres objets par rapport à la date locale de l'objet référence, devenu la date commune.

4.7.1 Problématique

4.7.1.1 Hypothèse de départ : les messages sont instantanés

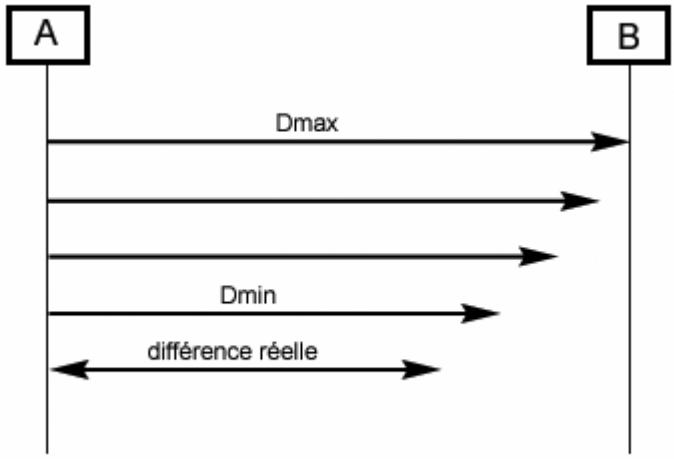
Le diagramme de séquence final en XMI qu'il nous est donné de générer a une grosse limitation : il ne représente pas la durée du message. C'est plus le problème des AGL utilisés en général qui gèrent définissent généralement des diagrammes de séquences dans le seul but de mieux visualiser le déroulement logique du système modélisé (et mieux le comprendre pourrait-on dire).

Il ne se destine pas à l'origine à représenter des cas réels d'utilisations, comme ici. Dès lors, nous ne pouvons envisager de décrire des messages croisés, tout du moins pour la génération finale en XMI. Cela signifie aussi et surtout que nous devons à l'issue de la synchronisation définir un ordre d'exécution des messages. C'est d'ailleurs suivant l'ordre de description des messages dans le fichier XMI que les AGL affichent les messages à l'affichage (il n'y a aucun moyen de définir du vrai parallélisme entre messages).

Partant de ce constat, nous pouvons pour le moment simplifier le problème en considérant que la durée des messages est nulle. En partant de cette hypothèse, la différence de temps entre le départ et l'arrivée d'un message correspond alors à la véritable différence de temps entre les deux horloges locales de chacun des deux objets. Cela reste bien sûr ici très théorique.

4.7.1.2 Le traitement des messages logués

Dès lors, si nous prenons plusieurs messages entre deux même objets. Nous calculons les différences de temps entre les deux dates locales et obtenons des différences variables d_1, d_2, …, d_n. Cela signifie que la différence de temps réelle entre les deux horloges des deux objets varie d'une durée d_{min} à une durée d_{max}. Si nous supposons qu'au niveau local, chaque horloge de l'objet ne se décale pas dans le temps, ni que le serveur de temps sur laquelle se base la machine distante n'a eu de problèmes techniques, cela signifie que la différence de temps réelle entre les deux objets est de d_{min}, et que dans certains cas des critères extérieurs ont retardé l'arrivée du message. Si on supprime cette hypothèse, ce critère extérieur est effectivement la durée du message.



Pour résumer, on peut déterminer à partir des différents messages qui s'échangent entre deux objets, que la différence de temps entre les deux horloges des deux objets est au plus la différence de temps minimale que l'on a calculée entre les dates locales des deux objets. Comme aucune information supplémentaire ne nous permet de définir si oui ou non les objets sont réellement décalés de ce d_{min} dans le temps, au bénéfice du doute, nous considérons que ce d_{min} est une estimation (majorée) du décalage de temps entre les deux objets. De la même manière, on peut aussi dire que la durée minimale du message est la différence entre le temps calculé pour le message ($d_{message}$) moins le temps minimum d_{min}. Dans le cas où $d_{message} = d_{min}$, la durée du message est nulle, et comme dit précédemment, d_{min} correspond au seul décalage entre les deux horloges des deux objets.

C'est ce principe que nous avons utilisé comme base de calcul de la synchronisation.

4.7.2 Technique de synchronisation utilisée

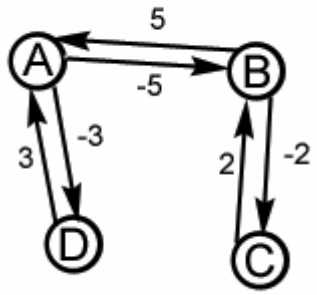
Nous travaillons avec une structure de graphe. Chaque nœud représente un objet, et chaque arc un message entre deux objets. Le poids de chaque arc correspond à la différence de temps minimale entre les deux objets.

La synchronisation se fait en trois étapes.

4.7.2.1 1ère étape : l'ajout des messages logués

Dans un premier temps, tous les messages sont « ajoutés » dans la classe de synchronisation. Au cours de cet ajout, chaque message est évalué comme expliqué précédemment :

La différence de temps entre les deux objets de ce message est calculée et comparée à la différence de temps déjà calculée pour des messages précédents entre ces deux même objets. Si la différence de temps (en valeur absolue) est inférieure à celle déjà calculée, elle tend vers la vraie différence de temps et est donc conservée à la place de la valeur actuelle. Cette valeur est placée dans le graphe comme poids de l'arc entre les deux objets (il y a en fait un sens pour l'arc qui indique la différence de temps entre un objet A et un objet B. Si l'arc de A vers B a un poids positif, B est en avance par rapport à A, et s'il a un poids négatif, B et en retard. L'arc inverse, de B vers A a un poids opposé).



Remarque : chaque nœud est ajouté au besoin, dès qu'un nouvel identifiant d'objet apparaît.

4.7.2.2 2ème étape : l'estimation des décalages d'horloge entre chaque objet

Une fois cette première étape effectuée, chaque arc donne la différence de temps estimée comme la plus proche de la véritable différence de temps entre les horloges de chaque objet (compte tenu des messages qui nous avons à disposition).

L'étape suivante est l'estimation du décalage de l'horloge de chaque objet par rapport à un objet de référence.

Chaque nœud est synchronisé sur un nœud de référence. Le nœud de référence choisi est celui qui a le plus de messages échangés avec ses voisins, donc, à priori, celui sur lequel les objets voisins peuvent se reposer de manière plus fiable que sur d'autres objets.

On parcourt tous les objets du graphe, jusqu'à les avoir tous synchronisés.
Le parcours pour être suffisamment fiable se fait en largeur : chaque nœud voisin du nœud de référence est d'abord synchronisé, puis les voisins de second niveau à partir des nœuds déjà synchronisés, puis de troisième, etc.

C'est à partir du poids de l'arc, estimé comme valeur proche de la réalité, que le poids du nœud est calculé (c'est à dire sa différence par rapport à l'horloge de l'objet de référence).

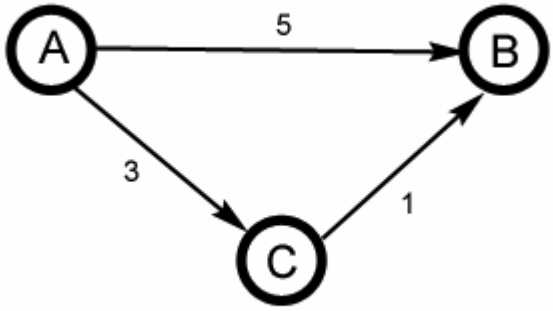
Chaque objet synchronisé est marqué. De cette manière toutes les composantes connexes du graphe sont traitées, avec un objet de référence par composante.

4.7.2.3 Affinage de l'estimation envisagé

Cette 2nde étape est cependant loin d'être parfaite. Nous avons envisagé une amélioration que nous présentons ici (mais par manque de temps ne n'avons pu réussir à l'appliquer directement – nous tenterons de le faire par la suite).

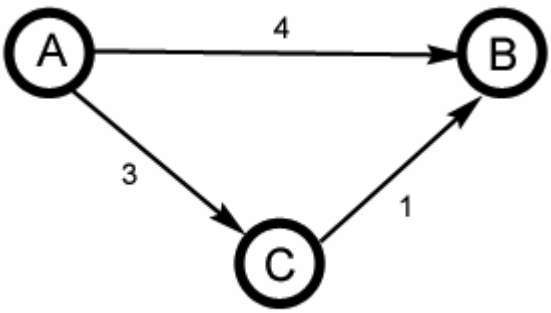
Prenons un exemple simple de graphe orienté vers le haut (ci-dessous) pour comprendre les limites de la méthode précedente :

.



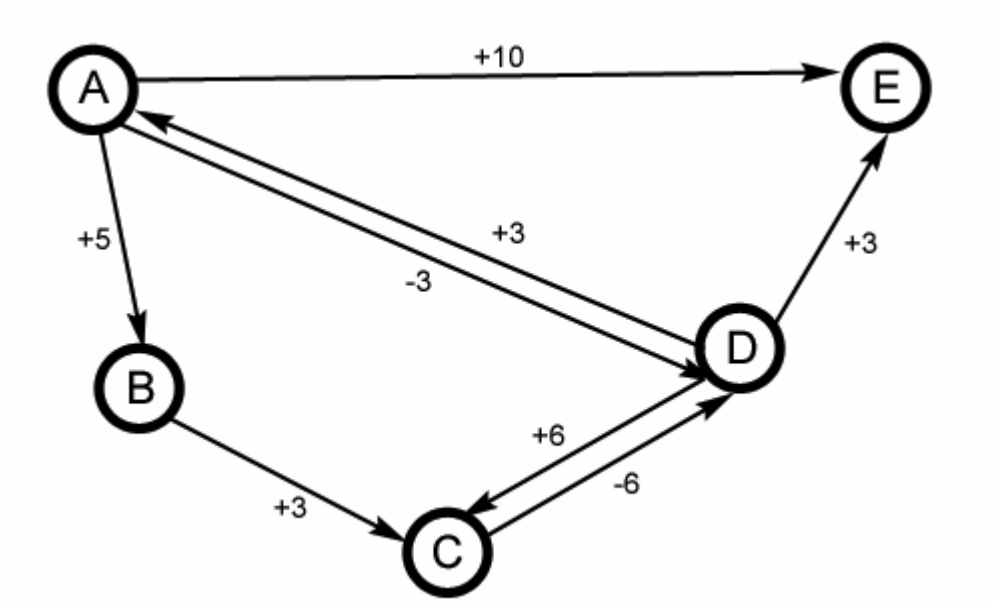
L'arc A→B donne un décalage de 5 unités, et l'arc A→C un décalage de 3 unités. D'après la méthode décrite précedemment, si A est l'objet de référence, seules les valeurs de deux arcs précédents sont utilisées, et ainsi l'objet B est en avance de 5 unités par rapport à A et l'objet C de 3 unités.
On remarque cependant que la différence de temps estimée entre B et C est de 1 unité (de C vers B), alors qu'on s'attend à avoir 2 unités de décalage d'après les poids des arcs A→B et A→C. En déterminant le plus court chemin de A vers B, on obtient les distances 5 et 3+1=4. Donc si l'objet B est décalé de 3 par rapport à A, et B de 1 par rapport à B, B est décalé de 4 au plus par rapport à A, et non de 5 ! De même la distance de A vers C est de 3 ou de 4 : donc de 3 : le décalage par rapport à B est donc bien estimé ici.
Nous avons donc grâce à la recherche du plus court chemin affiné les estimations de décalage entre horloge. La synchronisation est ainsi plus précise.

Le graphe après calcul de la plus courte distance.



Le problème qui se pose ensuite est que les poids peuvent être négatifs (indiquant que l'objet source est en avance par rapport à la destination). N'oublions pas que nous cherchons à obtenir le décalage signé, pour savoir si on doit ajouter ou enlever du temps entre deux objets. Nous devons donc pour chaque message définir un intervalle de décalages. La recherche du plus court chemin (de la plus courte distance en fait, le chemin en soit nous important peut) consiste à affiner de plus en plus l'intervalle. Chaque borne tend alors vers zero.

Prenons l'exemple ci-dessous, à l'issu de l'étape 1 :



Prenons l'objet A comme référence et estimons le décalage avec l'objet E.
Le chemin A-B-C-D-E donne +5+3-6+3, soit un décalage de : $-6 \leq d \leq 11$.
Le chemin A-E donne +10, soit E qui est en avance de 10 au plus. Cela revient à dire que E a un décalage borné de : $0 \leq d \leq 10$. Le résultat est plus précis que le précédent.
Le chemin A-D-E donne : $-3 \leq d \leq 3$. Comme le 1er chemin donne d supérieur à 0, on conserve 0, donc on a la certitude que l'objet E est en avance, de 3 au plus, ou est déjà synchronisé avec A (même horloge, dans le cas où d=0), et le second donne d inférieur à 3, on a pour le moment une meilleure estimation : $0 \leq d \leq 3$.

Les autres chemins n'améliorent pas ce résultat.
Donc l'objet E est en avance de 3 secondes (au plus). Nous conservons ainsi cette valeur.

A la fin, pour estimer le véritable décalage entre les horloges des deux objets, nous ajoutons les deux bornes (d'où nos 3 unités dans l'exemple : 0 + 3). De cette manière, si un décalage est estimé entre –a et +b, comme nous n'avons pas suffisamment d'informations pour savoir si l'objet est en avance ou en retard, on décide arbitrairement en ne conservant des deux valeurs qu'une estimation de la plus élevée d'entre elles (en valeurs absolues). Par exemple, un objet O avaec un décalage compris entre –5 (retarde de 5 unités ou moins) et +10 (en avance) aura une plus chance d'être en avance. Nous considérons que cela revient alors à définir un déclage de +5 unités (-5 + 10 = +5).

Cela reste cependant bien imprécis, faute d'avoir suffisamment d'informations.

L'algorithme utilisé est un algorithme de calcul de la plus courte distance. Elle se décompose en fait en deux calculs : celui de la borne inferieure (la plus longue distance – en ne s'occupant que des arcs de poids négatifs lors du parcours) et celui de la borne supérieure (la plus courte distance – en ne traitant que les arcs de poids positifs). A priori nous utilisons une version modifiée de l'algorithme de Floyd. De cette manière, nous ne nous préoccupons pas encore de l'objet de référence. C'est à la fin qu'on déterminera pour chaque composante connexe un objet de référence, puis que nous mettrons à jour le poids de chaque nœud du graphe de la 1ère étape (le poids indiquant toujours le décalage de l'horloge de l'objet par rapport à celle de l'objet de référence, comme précédemment), à partir des décalages que nous venons juste de calculer (borneInf + borneSup).

4.7.2.4 3ème étape : la génération des messages séquentialisés

Enfin, notre graphe étant synchronisé (ou tout du moins relativement équilibré), il ne reste plus qu'à reprendre chaque message et à recalculer les dates de départ et d'arrivée, puis de renvoyer une liste des messages qui se succèdent, en les triant par date (les dates sont triées selon la date de l'émetteur).

Rappelons que le poids de chaque arc détermine la différence « optimale » de temps entre deux objets, et le poids de chaque nœud donne la différence de temps entre l'horloge de l'objet de référence et l'horloge locale de l'objet correspondant au nœud.

Pour ré-estimer la durée du message, le principe est le suivant :
Pour chaque message, la date d'émission est recalculée en fonction du décalage de temps entre l'objet émetteur et l'objet de référence. La date d'arrivée est ensuite au mieux la même que la date de départ (si la différence de temps avant synchronisation était la plus petit rencontrée), à laquelle on ajoute la durée minimale estimée du message, à partir de la différence de temps évaluée entre les deux horloges locales des deux objets.

Ainsi, pour un objet A décalé de a secondes par rapport à l'objet de référence (supposons A en avance de a secondes). Le message de A vers B logue un départ à depA secondes et une arrivée à depB secondes. La différence entre depA et depB (depB-depA) donne n secondes. D'après les estimations dans le graphe de synchronisation, l'arc de A vers B donne une différence de temps maximale de m secondes. Ainsi, la différence n–m donne la durée minimale du message, et le message part ainsi à depA–a (parce qu'on sait que A est en avance), et l'arrivée du message est à (depA – a) + (n – m) secondes.

Le message est alors placé en fonction de son heure de départ dans la liste croissante des messages.

4.7.3 Implémentation

La synchronisation est définie dans le package "corbaTrace.log2xmi.synchro".

Pour mettre en œuvre cette méthode de synchronisation, nous définissons une structure de graphe :

- la classe « *ObjectGrapheNoeud* » représente le nœud. Il possède comme informations un nom (qui est l'identifiant de l'objet), ses nœuds voisins et les arcs correspondants (pour relier ces voisins). Les arcs sont tous orientés, donc les nœuds voisins sont en fait les nœuds destination. Pour chaque voisin, celui-ci aura dans ses propres nœuds voisins de destination le nœud en question, avec un poids de valeur opposé sur l'arc. Chaque nœud a aussi un poids et un booléen qui permet de le marquer lors d'un parcours.
- un arc est représenté par la classe « *ObjectGraphEdge* ». Il a juste comme information un poids.
- le graphe en lui enfin est simplement formé de tous ses nœuds. Il propose une méthode d'ajout de message qui se charge d'ajouter dans le graphe l'objet (le nœud) et de mettre à jour le poids de l'arc si nécessaire (ou de l'ajouter).

On peut noter que chaque arc est en fait doublé. L'arc est orienté pour déterminer la différence de temps entre un objet A et un objet B. Si le poids de l'arc est positif, l'horloge de B est en avance par rapport à B (et celle de B en retard, donc l'arc de B vers A a un poids opposé, négatif), et négatif si l'horloge est en retard (respectivement en avance de B vers A). De cette manière, l'application de l'algorithme de Dijkstra modifié de l'étape 2 est possible si au moins un message a été rencontré entre deux objets, quelque soit l'emetteur et le récepteur.

Enfin, la classe Synchronizer définit la technique de synchronisation. Elle parcourt le graphe en largeur en marquant chaque nœud déjà rencontré.

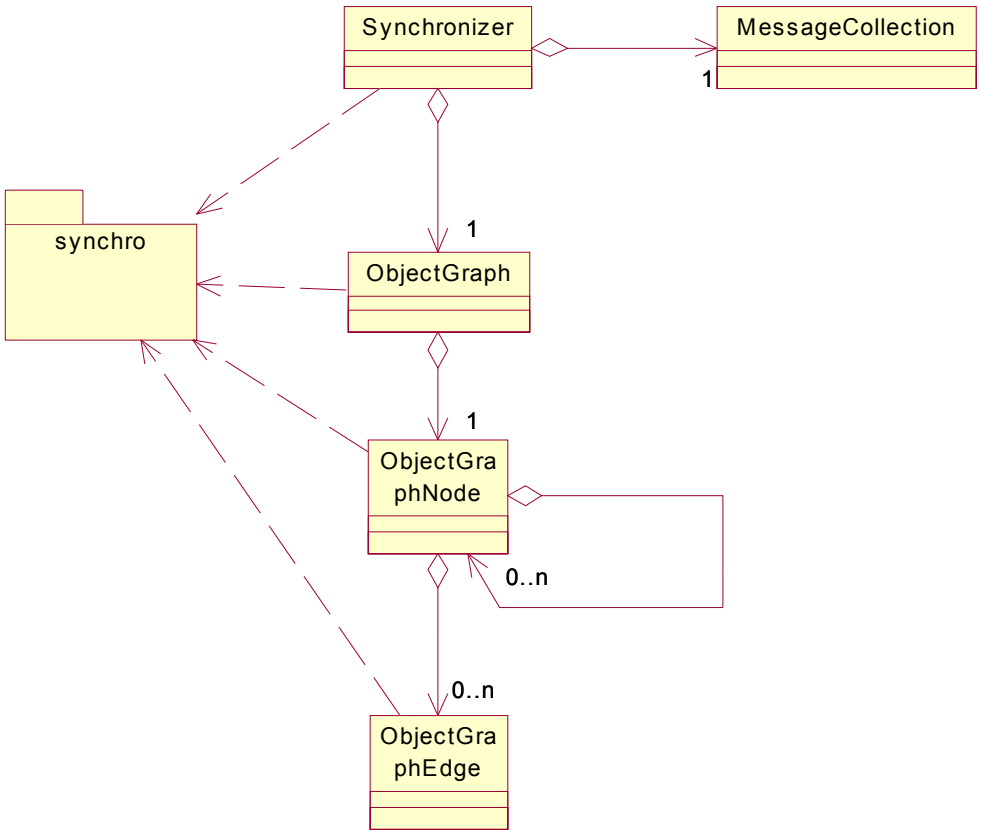


Figure 9 : le paquetage de synchronisation

4.7.4 Le cas des messages incomplets

Les messages incomplets ne peuvent pas intervenir dans la synchronisation puisqu'une des deux dates source ou destination est erronée et donc non fiable (la date BROKEN_DATE de l'objet inconnu est d'ailleurs fixée à une valeur totalement inexploitable). L'objet inconnu (BROKEN_OBJECT) n'apparaît donc jamais dans le graphe de synchronisation.

Par contre, dans la dernière étape, le message doit être placé au mieux dans la liste des messages synchronisés. Pour cela, on recalcule malgré tout la date d'émission ou de réception (celle de l'objet connu) par rapport à l'objet de référence. C'est sur cette date que nous nous basons ensuite pour placer le message au bon endroit.

4.8 Le système de filtres

Le système de filtres est une fonctionnalité quasiment indispensable dans l'outil log2xmi. Sans l'utilisation de filtres, avec des logs qui peuvent rapidement présenter plusieurs centaines de messages, les diagrammes de séquence générés deviennent alors presque impossible à lire, donc à exploiter, étant donné le trop grand nombre de messages affichés. Dès lors, les filtres vont permettre de mieux cibler les messages que l'utilisateur souhaite réellement visualiser

Le filtre est défini par l'utilisateur sous forme d'un fichier. De cette manière le filtre est plus riche et surtout plus facile à utiliser.

Le principe de fonctionnement du filtre est inclusif, c'est à dire qu'il définit quels messages doivent être conservés.

Quatre types d'informations sont filtrables :

- *les types de messages*, comme définis dans la classe MessageType.
  Les types actuellement filtrables sont : REQUEST, REPLY, EXCEPTION, BROKEN_REQUEST, BROKEN_REPLY, et BROKEN_EXCEPTION.
- *les objets* : définis par leur identifant d'objet.
- *les dates* :
  Il y a trois façons de filtrer des dates :
  - *après* une date donnée
  - *avant* une date donnée
  - *entre* deux dates données
- *les opérations* :
  - elles sont définies au moins par leur nom
  - elles peuvent être affinées en définissant des valeurs d'arguments filtrées. Les arguments sont de deux types : soit un type de données et une valeur, soit une position d'argument pour l'opération (de 1 à n) et une valeur.

Nous trouvions intéressant de définir deux niveaux de filtrage :

- au niveau global, c'est à dire sur l'ensemble des messages.
- au niveau d'un objet donné (sachant que l'objet donné s'applique lui au niveau global évidemment).

Que ce soit au niveau global ou au niveau d'un objet, on peut définir des filtres sur des dates, sur des opérations et arguments, et sur des types de messages, comme décrit précédemment.

Les informations filtrables de même type à un même niveau fonctionnent comme des unions, c'est à dire qu'un seul de ces filtres suffit à considérer un message comme vérifiant le filtre. Les informations de types différents (date, objets, types de messages, etc.) fonctionnent comme des intersections, c'est à dire que toutes les conditions doivent être vérifiées.

Par exemple, si nous définissons à un niveau global un filtre sur les dates pour les dates après D ou pour les dates entre D1 et D2, et sur les types de messages 'REQUEST', et sur un objet O avec l'opération op1 et l'argument n°3 à 50 ou l'argument de type String à "abc", alors le message, pour être conservé, doit être obligatoirement être daté après la date D *ou bien* entre D1 et D2, *et* être du type 'REQUEST', *et* porter sur une opération op1 de l'objet O avec *soit* un argument de type String à la valeur "abc", *soit* son 3ème argument à la valeur 50.

Vous pourrez trouvez un exemple de fichier de filtre un peu plus loin.

4.8.1 La Structure objet des filtres

Les filtres se trouvent définis dans le package corbaTrace.log2xmi.message. filter.
Il y a une classe pour chaque type de filtre :

- ObjectFilter, avec un identifiant d'objet
- OperationFilter, avec un nom
- DateFilter, avec une date de début et une date de fin (between), et un type de date (after/before). Dans le cas after/before, seule la date de début est utilisée,
- ainsi qu'une classe spéciale pour les filtres sur les attributs (AttributeFilter), qui est formée d'un type de données, d'une valeur, et d'une position si nécessaire.

Le filtre sur les types de messages est simplement un test d'égalité entre chaînes de caractères et ne nécessite pas à lui tout seul une classe dédiée.

Enfin, la classe MessagesFilter définit le filtre global à proprement parlé.
Ce filtre global définit quatre listes de filtres : une liste de filtres sur les objets, une liste pour les DateFilter, une liste de types de messages (String), et une liste d'opérations.
Le filtre d'objets fonctionne sur le même principe (trois listes de filtres de dates, types, et opérations), en plus de son identifiant. Le filtre d'opérations possède, outre son nom, la liste de ses opérations.

La construction du filtre se fait par appel de méthodes. Par exemple, l'ajout d'un nouvel objet à filtrer addObjectFilter() prends en paramètre l'identifiant et crée un ObjectFIlter dans la liste des filtres d'objets (s'il n'existe pas déjà). Le principe est le même pour les autres filtres. Pour l'ajout d'une opération d'un objet, une méthode du filtre global recherche l'objet en question pour lui ajouter un filtre d'opération. Mais généralement, comme expliqué plus loin pour le parsage, une opération est ajoutée directement sur un objet courant (sans nécessiter sa recherche dans la liste de tous les objets filtrés).
La construction des arguments d'opérations fonctionne sur le même principe que les opérations ajoutées pour un objet donné dans le filtre global (recherche de l'opération puis ajout du filtre d'argument).

Ensuite, chaque filtre définit une méthode isAllowed() qui prend en paramètre un message à tester, et filterIsApplicable() pour le filtre global. Elle renvoie true si le message est autorisé, false sinon.

Par exemple, pour le filtre global, la liste des types est parcourue jusqu'à trouver un type accepté pour ce message, si oui, la liste des filtres de dates est passée en revue : pour chaque date, isAllowed() est appelé, laquelle méthode vérifie si le message correspond au filtre de date. Si au moins une date est acceptée, le filtre des opérations est appliqué (et au moins un argument doit être vérifié, s'il y en a), puis le filtre des objets qui fonctionne sur le même principe que le filtre global (les types, puis les dates, puis les opérations).

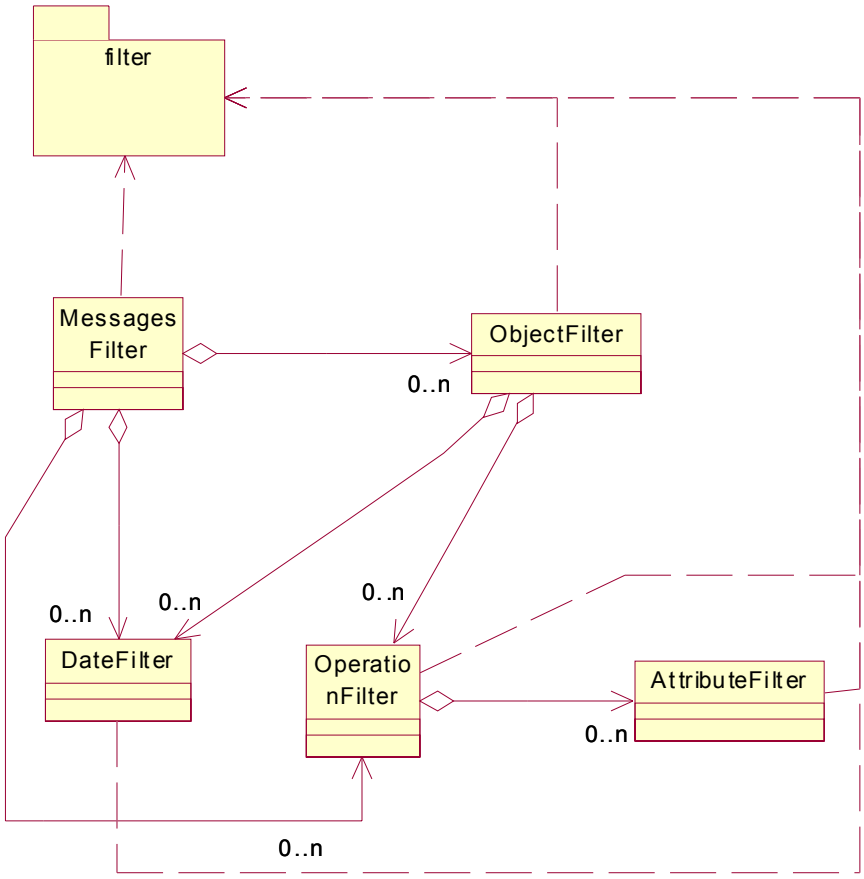


Figure 10 : architecture du système de filtres

A la fin, le message est conservé ou non.
Dans le filtre global la méthode applyFilter() applique un filtre sur un MessageCollection ou une liste de messages. Il consiste à ne conserver dans la collection (ou la liste) seulement les messages étant filtrés correctement. L'intérêt de cette méthode est de pouvoir appliquer un filtre à n'importe quelle étape du traitement des logs, avant la fusion, après la fusion, ou encore après la synchronisation. Cela est particulièrement utile. Dans notre cas, le filtre est toujours appliqué après la synchronisation (à cause du filtrage des dates, plus cohérent si synchronisées).

4.8.2 Le Fichier des filtres

Nous trouvions intéressant d'utiliser aussi XML pour définir nos fichiers de filtres, non seulement pour les raisons déjà évoquées précédemment (simplicité, facilité de transformation, etc.), mais aussi pour étendre la DTD déjà définie pour les logs.

4.8.2.1 La DTD des filtres

Par facilité d'usage, nous souhaitions définir une DTD commune aux logs et filtres. Cependant, ces deux utilisations sont relativement indépendantes, et il nous a paru judicieux de bien les distinguer. Ainsi, si les logs sont encapsulés par des tags <log></log>, les filtres sont eux encapsulés entre des tags <filter></filter>. De cette manière, lorsque le point d'entrée du document est défini dans le fichier XML, selon le choix de la racine « log » ou « filter », nous savons à quel type de documents nous sommes confrontés : un fichier de logs ou bien un fichier de filtres.

La structure du fichier XML, et de la DTD, reprend dans les grandes lignes les mêmes principes évoqués pour la structure objet des filtres :

Un filtre est composé d'éléments de filtre globaux : les types de messages (message_types), les dates, les méthodes de classes (methods), et les objets (objects). C'est l'intersection des quatre éléments (sachant qu'ils sont tous optionnels) qui donne les messages conservés.

```
<!ELEMENT filter (message_types?, dates?, methods?, objects?)>
```

Un message_types (c'est à dire des filtres de types de messages) est formé de plusieurs types. Si un seul de ces types est vérifié, le message est conservé. Chaque type a une valeur qui est parmi celles utilisées (mais des développeurs peuvent très bien étendre le nombre de types acceptés sans remettre en cause, ni la DTD, ni le parsage). Les types réellement utiles sont prédéfinis dans la classe message.MessageTypes.

```
<!ELEMENT message_types (type)*>

<!ELEMENT type EMPTY>
<!ATTLIST type
          value CDATA #REQUIRED>
```

Le filtre des dates est formé de différents critères de sélection de dates. Ces critères sont de trois types :

- les dates « between », pour lesquelles deux attributs « from » et « to » sont définis (et déterminent un intervalle de temps entre deux dates)
- les dates « before », pour lesquelles un attribut date donne la date majorant l'ensemble des dates autorisées.
- les dates « after», pour lesquelles, comme pour les dates « before », un attribut date donne la date minorant l'ensemble des dates autorisées.

Remarque : les dates utilisent le même format que les logs : annee-mois-jourTheures-minutes-secondes-millisecondes (par ex: 2002-03-09T17:00:00.000), comme défini dans la classe corbaTrace.utils.DateUtilities (c'est au lors du parsage que ce format est pris en compte, et non au niveau de la DTD).

Le filtre des dates est ainsi défini comme suit dans la DTD :

```
<!ELEMENT dates (between | after | before)*>

<!ELEMENT before EMPTY>
<!ATTLIST before
          date CDATA #REQUIRED>

<!ELEMENT after EMPTY>
<!ATTLIST after
          date CDATA #REQUIRED>

<!ELEMENT between EMPTY>
<!ATTLIST between
          from CDATA #REQUIRED
          to CDATA #REQUIRED>
```

Le filtre des méthodes (c'est à dire des opérations) est formé de plusieurs filtres de méthode. Chaque méthode a un nom comme attribut et peut optionnellement être formée d'argument(s), de deux types : ArgumentAt et typedArgument. Un ArgmentAt signifie qu'on indique une position d'argument pour l'opération et une valeur pour cet argument. Un typedArgument indique un type de données correspondant à un des arguments de la méthode et une valeur pour cet argument. Il est bien sûr moins précis que l'argument positionné.

```
<!ELEMENT methods (method)*>

<!ELEMENT method (argumentAt | typedArgument)*>
<!ATTLIST method
            name CDATA #REQUIRED>

<!ELEMENT argumentAt EMPTY>
<!ATTLIST argumentAt
            position CDATA #REQUIRED
            value CDATA #REQUIRED>

<!ELEMENT typedArgument EMPTY>
<!ATTLIST typedArgument
            type CDATA #REQUIRED
            value CDATA #REQUIRED>
```

Enfin, le filtre des objets qui définit des objets. Chaque objet a un identifiant (id) et peut définir lui-même des filtres plus poussés sur l'objet : des filtres de types de messages, de dates, ou de méthodes. Ils ont la même syntaxe que vu précédemment mais ne s'appliquent ici qu'à l'objet englobant.

```
<!ELEMENT objects (object)*>
```

```
<!ELEMENT object (message_types?, dates?, methods?)>
<!ATTLIST object
          id CDATA #REQUIRED>
```

4.8.2.2 La parsage du fichier de filtre

Nous avons utilisé le même principe que pour le parsage des logs, en utilisant SAX2. Comme pour les logs, nous définissons un parseur spécifique pour les filtres (*FilterSaxParser*). Il hérite du parseur générique *GenSaxParser* et utilise un handler pour les fichiers de filtre (*FilterHandler*), et propose une méthode getMessagesFilter() pour récupérer une structure complète de filtre (MessagesFilter) à partir du fichier lu.

FilterHandler définit la manière d'exploiter les données parsées depuis les fichiers de filtre. Il construit un filtre complet de messages (MessagesFilter) au fur et à mesure de la lecture du fichier.

La partie de DTD des filtres étant relativement simple, le principe général de fonctionnement de FilterHandler reste très simple :

- on travaille avec un filtre d'objet temporaire, par défaut à vide.
- A tout moment, on sait si on travaille au niveau global ou au niveau objet.
- dès qu'un filtre d'objet est détecté, le filtre d'objet temporaire est créé pour cet objet.
- à chaque élément de filtre lu (méthode, type, date, etc.), on appelle une méthode privée d'ajout de filtre, qui détermine en fonction du niveau actuel de fonctionnement si l'élément de filtre s'applique à un objet (ObjectFilter temporaire) ou à l'ensemble des messages (MessagesFilter temporaire).
  Les méthodes proposées sont :
  - *addDateFilter(DateFilter date)*
  - *addMessageTypeFilter(String type)*
  - *addOperationFilter(String operationName)*
  - *addObjectFilter(String objectID)*
  - *addAttributeFilter(AttributeFilter attF)*

  Chacune d'elle appelle en fait une méthode du même genre sur l'objet ou sur l'ensemble des filtres.
- le cas des méthodes est particulier : dès qu'un élément « méthode » est détecté, un filtre temporaire d'opération est créé et ajouté dans le bon niveau (global ou objet). Puis dès qu'un argument est ajouté, c'est sur cette opération temporaire qu'il est ajouté.

4.8.2.3 Exemple

Prenons l'exemple de filtre suivant :

```
<filter>
      <message_types>
            <type value="BROKEN_REQUEST"/>
            <type value="BROKEN_EXCEPTION"/>
      </message_types>
      <dates>
            <after date="2002-03-09T17:00:00.000"/>
            <before date="2002-03-09T15:00:00.000"/>
      </dates>
      <methods>
            <method name="Operation1">
                  <argumentAt position="3" value="ddd"/>
            </method>
            <method name="Operation2">
            </method>
      </methods>
      <objects>
            <object id="Objet1">
                  <message_types>
                        <type value="BROKEN_REQUEST"/>
                  </message_types>
                  <methods>
                        <method name="Operation1">
                              <argumentAt position="2" value="bbb"/>
                        </method>
                  </methods>
            </object>
            <object id="Objet2">
            </object>
      </objects>
</filter>
```

Pour qu'un message soit conservé par ce filtre il faut que :

- son type de message soit BROKEN_REQUEST ou BROKEN_EXCEPTION
- *et* qu'il soit daté après 17H ou avant 15H (le 9 mars 2002)
- *et* qu'il porte sur une opération Operation1 avec l'argument n°3 égal à "ddd"
  ou bien sur une opération Operation2 (quelque soit ses arguments).
- *et* qu'il porte sur un objet Objet2
  ou sur un objet Objet1
  avec comme type de message BROKEN_REQUEST
  et comme opération Operation1 avec l'argument n° 2 à "bbb".

Ainsi, un message de type BROKEN_REQUEST, daté à 18H, portant sur l'opération Operation1 de l'Objet1 avec l'argument n° à "bbb" et l'argument n°3 à "ddd" est accepté. Par contre ce même message de type BROKEN_EXCEPTION ou bien si le 3ème argument de l'opération est différent de "ddd", il n'est pas accepté.

4.9 Le processus de génération XMI

Par la suite, nous reviendrons plus en détail sur le format XMI, sur la librairie de création de fichiers XMI que nous avons réalisée pour l'application, et sur l'utilisation en elle-même de cette API XMI.

Nous présentons ici très succinctement la phase de génération XMI à partir d'une collection de messages.

Le moteur de génération XMI (classe corbaTrace.log2xmi.XmiGenerator) fonctionne comme suit :

- Un document XMI est créé et un nouveau modèle lui est ajouté, ainsi qu'un nouveau diagramme de séquence. (note : il est tout à fait envisageable de définir plusieurs modèles pour un seul document, même si nous ne le faisons pas en pratique, donc le générateur XMI connaît à tout moment le modèle courant et le document courant).
- des messages supposés complets, synchronisés, et filtrés sont ajoutés un à un dans l'arbre XMI final. En fait, comme les messages ajoutés sont issus des étapes précédentes, généralement juste après synchronisation et application des filtres, une méthode d'ajout d'une collection de messages ou d'une liste de messages est proposée (addAllMessages()), et se charge d'ajouter tous les messages dans l'arbre XMI les uns après les autres. Il est conseillé de passer en paramètre une liste de messages triés dans le temps (du plus récent au plus ancien), sinon le diagramme de séquence généré n'aura pas grand intérêt car il sera désordonné (puisqu'un MessageCollection trie d'abord les messages par objets et non par dates).
  Cette étape est la plus délicate car il faut définir auprès du document XMI toutes les opérations, avec distinction des arguments, de toutes les classes, et de tous les types de données avant de pouvoir ajouter le message dans le modèle XMI du document. Il ne faut évidemment pas définir auprès du document deux fois la même opération. C'est pourquoi le générateur conserve en interne tous les objets, les classes, et les datatypes qu'il a déjà ajoutés. Ce sont des tables de hashages de XmiRole, XmiClass, et XmiDatatype.

  Pour simplifier, nous créons une nouvelle action par message. Il faut cependant récupérer la bonne opération XMI pour créer chaque action, et récupérer les rôles XMI correspondants aux objets source et destination pour pouvoir ajouter le message.
  Le principe est donc le suivant :
  - on récupère les rôles de l'objet source et de l'objet destination, ainsi que les deux classes correspondantes. Si une des classes ou un des rôles n'existe pas, on le crée dans l'arbre et le référence dans la table des classes et objets.

- on tente de localiser l'opération du message dans l'arbre XMI déjà en mémoire. Nous faisons cela parce que l'opération est plus complexe qu'un simple rôle ou classe, notamment parce qu'il faut tester si les attributs coïncident sans quoi l'opération est considérée comme différente. Pour cela il faut au préalable déterminer par quel objet est proposé l'opération. Dans le cas de message de réponse ou d'exception, c'est l'objet source, et dans le cas de requêtes, c'est l'objet distant. Une méthode operationIsAvailableOnSourceObject() dans la classe message.MessageType permet de déterminer cela en fonction du type du message à ajouter. Comme on a localisé la classe XMI correspondante, on récupère toutes les opérations de la classe, puis pour chaque opération, on vérifie si une d'entre elle correspond à l'opération du message à ajouter. Il faut pour cela non seulement vérifier le nom de l'opération et le nombre d'arguments, mais aussi vérifier si chaque argument est exactement le même (même type, même sorte d'entrée/sortie, même nom le cas échéant). Si aucune opération n'est trouvée, on l'ajoute définitivement dans le modèle XMI pour cette classe.
  Enfin, une fois les deux rôles et l'opération localisés, on peut créer une nouvelle action et ajouter ce message dans le diagramme de séquence du modèle XMI courant.
  Remarque : XMI ne permet pas de traiter les messages perdus. Nous créons donc un objet « inconnu » (« unknown ») qui participe au diagramme de séquence. Cela ne change strictement rien pour le traitement des messages générés puisque chaque message de type BROKEN_xxx définit un objet inconnu.

- L'utilisateur place ses options de génération (il peut aussi très bien le faire avant d'importer les messages) : extension Rational Rose, extension Magic Draw, nom du fichier, etc.
- Une fois tous les messages ajoutés et les options définies, une méthode génération du fichier est appelée. Les extensions sont alors définitivement ajoutées et le fichier est généré. Seule les opérations proposées par l'API XMI sont utilisées ici : createRoseExtensions(), createMagicDrawExtensions(), writeToFile().

Le chapitre suivant décrit plus précisément l'utilisation de XMI.

5 XMI et diagrammes de séquence

5.1 Le standard XMI

5.1.1 Qu'est-ce que XMI ?

XMI signifie « XML Metadata Interchange », c'est à dire en français « échange de données méta en XML ». C'est une spécification de l'OMG, l'Object Management Group. Ce format permet d'échanger facilement des données méta entre des outils de modélisation basés sur les spécifications UML de l'OMG. La version actuelle des spécifications XMI, la version 1.2, supporte le métamodèle UML 1.3.

Les fichiers XMI sont des fichiers XML, dont la structure est définie par une DTD. Cette DTD permet de valider les documents XMI avec un parseur XML standard. Ces fichiers permettent donc de sauvegarder et d'échanger des diagrammes UML dans un format standard et très répandu.

XMI est un standard récent. La dernière version des spécifications de l'OMG date du 01/01/2002. L'avenir et le développement de ce format est assuré non seulement par le fait que l'OMG est mondialement connu et a une forte notoriété, mais aussi parce que c'est en fait un consortium d'industriels, qui s'obligent en signant une spécification à en fournir une implémentation dans des délais donnés.

De grands noms tels que Boeing, Daimler-Benz, Fujitsu, IBM, Oracle, Unisys, NCR, Rational, Softeam, Sybase et Xerox ont, parmi d'autres, participé à l'élaboration de ce standard.

5.1.2 Utilisation de XMI dans notre projet

L'objectif global de CorbaTrace est de permettre de tracer et de visualiser graphiquement des communications Corba entre différents composants. L'objectif de trace est atteint en mettant en place la stratégie d'interception. Mais ce procédé permet seulement de récupérer toutes les données relatives aux communications et de les stocker dans des fichiers de log au format XML, un pour chaque composant Corba.

Ces données sont précises et permettent déjà de déboguer une application Corba, mais l'objectif était d'aller plus loin, vers une visualisation graphique des échanges d'informations. Ainsi, nous avions donc besoin de développer un outil permettant de rassembler les différents fichiers de log, de faire correspondre les informations côté client et côté serveur pour chaque échange de message, et enfin de générer une sortie qui permette une visualisation graphique. C'est le rôle de l'application Log2xmi.

Le but était d'obtenir un diagramme de séquence UML, car c'est une méthode de modélisation idéale pour des envois de messages entre des objets. UML est un standard adopté par quasiment tous ceux qui font de l'orienté objet, et Corba s'inscrit dans cette catégorie. De plus, il existe de nombreux outils permettant de faire de l'UML.

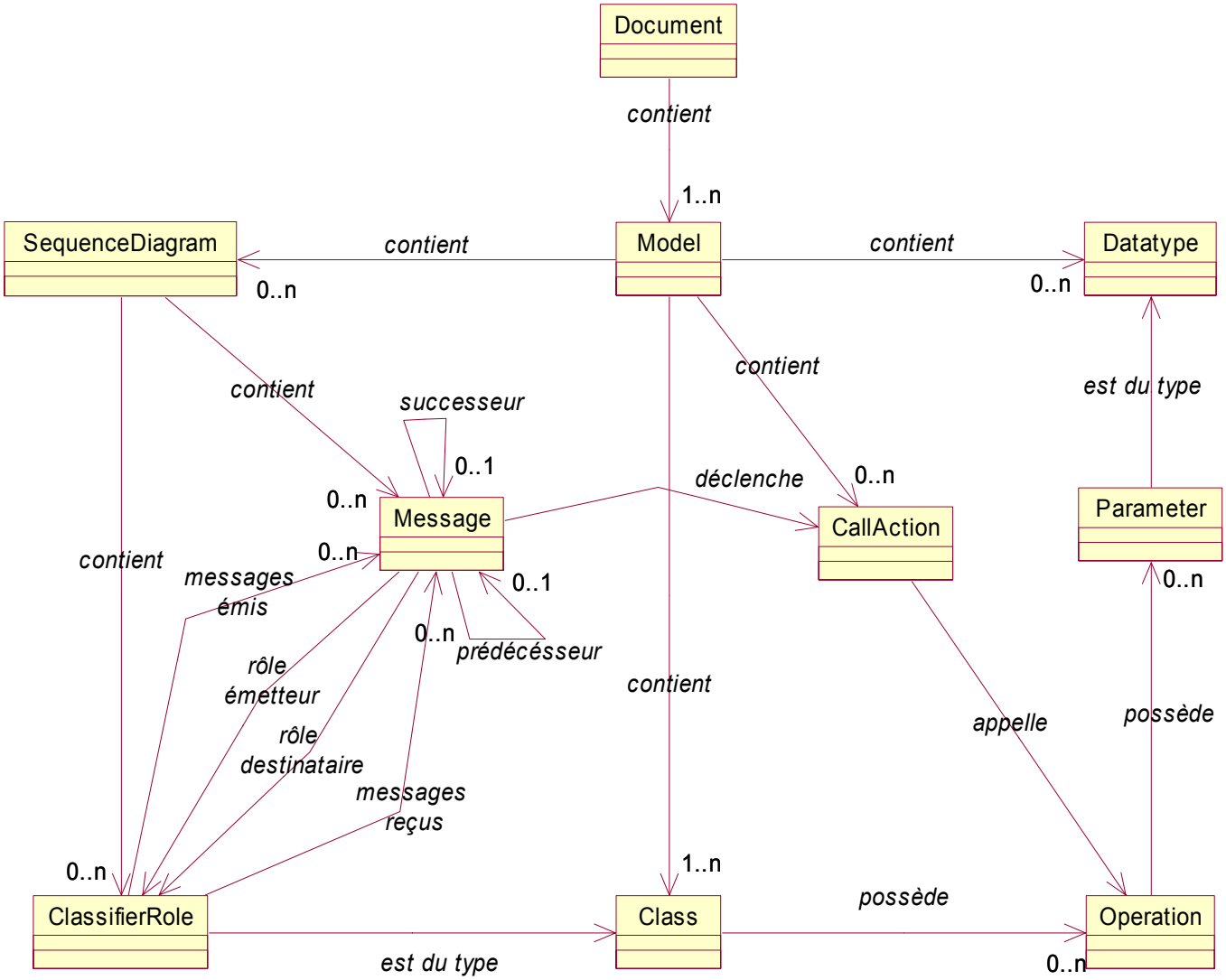
C'est donc tout naturellement que le format XMI s'est imposé pour cette partie de notre projet. XMI présente de nombreux avantages pour CorbaTrace :

- Le format de fichier est XML, tout comme les logs des interceptions.
- Ce n'est pas le même logiciel qui fait l'interprétation des messages Corba et la visualisation.
- CorbaTrace est indépendant de l'outil de visualisation des diagrammes de séquence.
- Nous n'avons pas à traiter l'aspect visualisation graphique à proprement parler, puisque les AGL UML le feront à notre place.
- Les sorties générées par CorbaTrace ne sont pas propriétaires, et elles sont portables.

5.1.3 Structure des documents XMI

XMI permet évidemment de stocker tous les types de diagrammes UML. Nous ne nous intéressons dans le cadre de CorbaTrace uniquement aux diagrammes de séquence. Pour pouvoir générer du XMI à partir du programme, il a d'abord fallu comprendre comment étaient structurés les documents XMI. Etant donné que les spécifications de l'OMG sont souvent difficiles à cerner, nous avons tout simplement créé un diagramme de séquence de test en UML puis sauvegardé ce diagramme au format XMI, afin de pouvoir le « disséquer ».

Voici la structure globale d'un document XMI en ce qui concerne les diagrammes de séquence :



Un document XMI est composé de différents objets bien distincts qui ont chacun des propriétés spécifiques (non représentées ici). On remarquera que par rapport à un diagramme de séquence UML classique on introduit ici le concept d'Action. Cette action est l'information portée par un message. Pour nous, toutes les actions seront des CallAction, c'est à dire des actions qui correspondent à des opérations déclenchées sur l'objet destinataire.

On remarque également que les messages existent à plusieurs endroits dans le document (rôles, message eux mêmes, diagramme de séquence) provoquant des redondances. Ceci a l'avantage de permettre de trouver une information directement dans l'objet sur lequel on travaille : par exemple dans un rôle, sans avoir à travailler sur l'objet diagramme pour trouver les messages émis par ce rôle.

Chaque objet XMI est identifié par un identifiant unique sous forme de chaîne de caractères (mais indépendant de son nom). Un même objet figure une seule et unique fois dans le document XMI. Lorsqu'un autre objet a besoin de le relier, il utilise une référence vers lui grâce à son identifiant.

5.1.4 Le problème des extensions

5.1.4.1 Informations graphiques pour les diagrammes

Il y a malgré tout un problème important dans la spécification d'XMI ou l'utilisation qui en est faite par les éditeurs de logiciels (au choix !). En effet, XMI ne permet pas de stocker les informations de rendus graphiques des diagrammes (disposition des classes, des messages, des rôles…).

Dans les logiciels actuels qui utilisent le format XMI, aucun ne sait extrapoler pour recréer ces informations, afin de générer une représentation graphique la plus logique possible. Au niveau de XMI, la seule balise qui n'a pas réellement quelque chose à voir avec UML est la balise <XMI.extensions>. Cette balise est justement destinée à offrir un mécanisme pour stocker des informations hors UML.

5.1.4.2 Format propriétaire des extensions

Les logiciels de modélisation UML ont donc utilisé ce mécanisme pour stocker toutes les informations purement graphiques (coordonnées et taille pour la disposition des éléments). Le gros problème est que ces informations ne sont pas normalisées dans les spécifications de XMI, ce qui est logique d'ailleurs, puisque d'une part ces informations ne sont pas des données UML, et que d'autre part les logiciels d'UML devraient être capables de dessiner des diagrammes sans autres informations. Et ce n'est malheureusement pas le cas.

En conséquence, chaque éditeur de logiciel a utilisé des balises et une structure pour les extensions XMI qui ne sont pas uniformisées, elles sont propriétaires (le mot qui tue !). Un effet de bord direct est qu'il faut également étendre la dtd de XMI pour décrire les balises des extensions. De plus, les fichiers XMI ne sont donc plus complètement standards, donc plus aussi facilement échangeables, puisque chaque logiciel gère les extensions de manière différente. Heureusement, il est possible de regrouper toutes les informations d'extensions tout à la fin du document XMI, plutôt que d'inclure ces extensions à l'intérieur de chaque objet, ce qui est tout aussi permis.

Conclusion : pour notre projet CorbaTrace, nous avons été obligés de choisir certains logiciels de modélisation UML, de décortiquer le format de leurs extensions, et de développer une méthode de génération des extensions à partir du document XMI (données UML) pour chacun d'eux.

5.1.5 Les outils de visualisation choisis

XMI était le format idéal à utiliser dans notre application. Cependant, même si son développement était certain, il nous fallait prospecter et tester pour trouver des logiciels qui permettaient déjà d'utiliser le standard XMI. Lors de nos recherches sur internet, nous avons trouvé plusieurs logiciels d'UML permettant de sauvegarder en XMI :

5.1.5.1 Poseidon for UML

La version « Community Edtion 1.2 » est utilisable librement. XMI est son format de stockage natif. Les fichiers sont sauvegardés dans un fichier « .xml.zip ». Ce fichier compresse les deux fichiers qui composent un document Poseidon : un fichier XMI, et un fichier SGML qui stocke les extensions. Cet outil est intéressant puisque totalement libre d'utilisation, cependant nous ne l'avons pas retenu dans notre liste finale car il ne stocke pas les extensions XMI de la même façon que les deux autres logiciels que nous avons testés.

5.1.5.2 Rational Rose

C'est le logiciel de modélisation UML le plus répandu dans les entreprises. Son prix est très élevé, mais il a de nombreuses fonctionnalités très intéressantes qui étendent les fonctionnalités UML de base. Il stocke les diagrammes dans un format propriétaire (fichier MDL), mais on peut installer un plugin pour pouvoir importer et exporter des fichiers au format XMI.

Ce plugin est développé par Unisys et est utilisable librement. La nouvelle version du plugin sortie pendant la durée de notre projet est censée exporter en métamodèle UML1.3, mais c'est manifestement faux puisque l'export se fait toujours au format 1.1, et que les balises XMI correspondant à UML 1.3 ne sont pas comprises par le moteur d'importation. Malgré tout, on arrive à importer des fichiers XMI avec la dtd de UML1.3, moyennant quelques « warnings ». Les extensions sont toutes regroupées à la fin du fichier XMI.

5.1.5.3 MagicDraw UML

Nous avons totalement découvert ce produit. Il a la particularité d'avoir basé toute son architecture sur le vrai standard UML (et non sur une interprétation comme Rational) et sur le format XMI. L'enregistrement des diagrammes se fait évidemment en XMI avec le métamodèle UML1.3. Dans ces fichiers, les extensions sont ajoutées dans le corps de chaque objet, ce qui complique leur compréhension. Cependant, il est aussi possible d'exporter au format Unisys (plugin Rational rose !). De toute façon, les extensions peuvent toujours être rassemblées et figurer à la fin du fichier.

D'une manière générale, ce logiciel est très avancé au niveau UML, suivant à la lettre les standards UML et XMI, et son interface est intuitive et agréable. Il est écrit en java, donc multi plateforme. C'est un logiciel payant, mais dont on peut librement utiliser une version d'évaluation qui n'est pas limitée en temps, la seule limite fixée étant le nombre de classes et d'acteurs au moment de la sauvegarde (20). Ceci ne nous gène pas puisque nous ne voulons que visualiser des fichiers déjà existants.

Nous avons choisi de développer des extensions XMI pour les logiciels MagicDraw UML et Rational Rose. Ceci dit, n'importe qui peut facilement développer du code pour générer des extensions pour d'autres outils.

5.2 Les APIs XMI

5.2.1 Standards utilisés

Les API XMI ont été développées en utilisant les standards SAX et DOM. DOM est une spécification permettant de construire des documents XML, et SAX une spécification permettant de parser des documents XML. Les packages standard utilisés sont *org.xml.sax* et *org.w3c.dom*.

Ces deux packages ne contiennent que des interfaces, et d'autres packages doivent être utilisés pour pouvoir traiter correctement du XML. Travaillant avec Java, nous avons choisi la bibliothèque JAXP de SUN. JAXP propose tous les outils XML dont on peut avoir besoin. Il est d'ailleurs inclus dans le J2SDK de SUN depuis la version 1.4.0. Pour les versions antérieures, JAXP est téléchargeable sur le site de SUN, puis il doit être ajouté au CLASSPATH. Les packages JAXP utilisés sont les suivants : *javax.xml.parsers*, *javax.xml.transform*, *javax.xml.transform.dom*, *javax.xml.transform.stream*.

5.2.2 Problèmes rencontrés

Les classes *javax.xml* sont nécessaires pour écrire le document DOM dans un fichier. Pour cela, on crée un transformer XSLT qui n'utilise pas de feuille de style, qui parse le document et l'écrit dans un stream de destination. Deux problèmes importants ressortent de ce procédé :

- le fichier XML ne peut pas être écrit de façon indentée, ce qui n'est pas très propre et rend la lisibilité difficile.
- Il n'y a de méthode permettant d'étendre la DTD du document XML en ajoutant des balises supplémentaires entre crochets.

Ce dernier point n'a pu être contourné qu'en réouvrant le fichier en lecture et écriture avec un *RandomFileAccess*, afin de pouvoir y insérer la déclaration des balises dont nous avions besoin en plus de la DTD d'UML. Ces balises supplémentaires sont celles des extensions Rational Rose, uniquement. En effet, MagicDraw UML ne parse que la partie contenu XMI, et reconnaît ses propres balises d'extension sans que celles-ci soient déclarées.

Il est à noter que si les extensions pour Rational Rose ont été assez faciles à comprendre et à coder, celles de MagicDraw sont très complexes, et il n'existe évidemment aucune documentation sur ce sujet. En effet, il faut calculer les coordonnées de taille et de position pour chaque élément du diagramme de séquence. Toutefois, le moteur graphique de MagicDraw possède l'avantage de recalculer automatiquement certaines coordonnées si celles qui sont fournies ne sont pas tout à fait exactes.

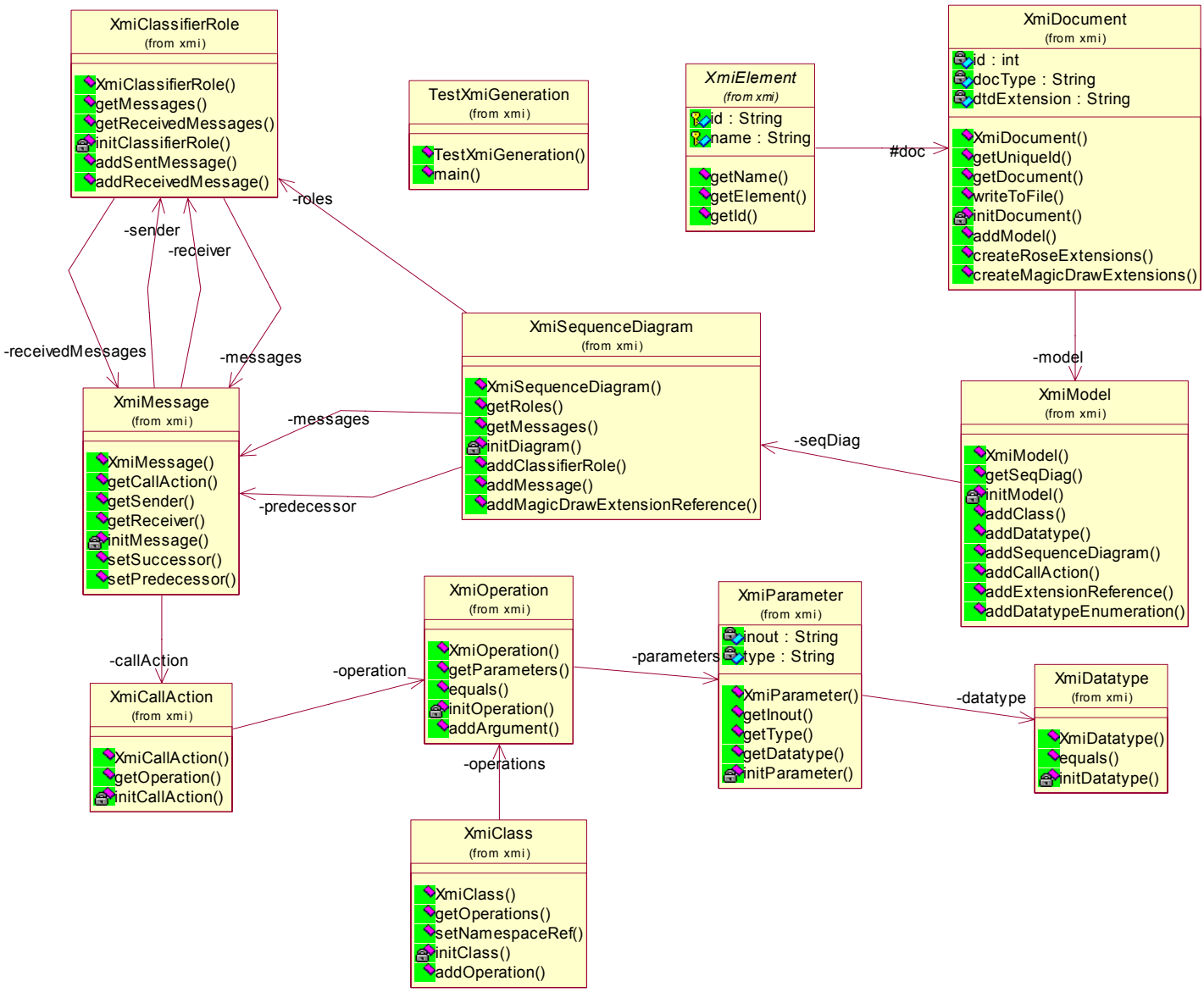
5.2.3 Structure des APIs

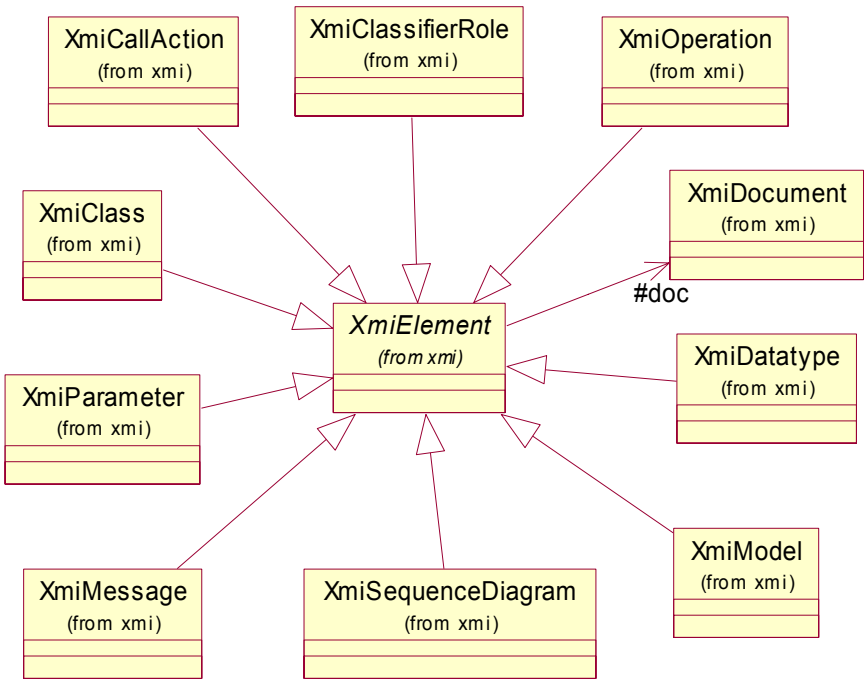
Les APIs développées pour la génération de XMI sont rassemblées dans le package *corbaTrace.xmi*. Elles sont totalement indépendantes des applications corbaTrace, ce qui permet de les réutiliser facilement dans d'autres développements. Les API suivent la structure XMI : une classe pour chaque objet XMI. La classe principale est la classe document. Toutes les autres classes étendent la classe abstraite XmiElement, qui stocke les attributs communs : le nom, l'identifiant, et l'Element DOM qui stocke les informations XMI sous forme XML.

Les relations entre les classes XMI ne correspondent pas à celles du modèle XMI, tout simplement parce que ces relations existent déjà dans l'Element DOM que possède chaque classe XMI. Les seules relations qui sont stockées le sont parce qu'on a besoin de les retrouver lors de la génération des extensions. Dans chaque classe, le constructeur initialise les attributs tels que les vecteurs, le nom qui est passé en paramètre, et appelle la méthode init. C'est cette méthode qui construit l'Element DOM (le XML).

Le document DOM lui-même (interface *Document*) est un attribut de la classe XmiDocument. L'instance de XmiDocument doit être passée en argument de tous les constructeurs des autres classes Xmi, car le document DOM sert de *factory* pour construire tous les noeuds à ajouter au document DOM.

5.2.4 Diagrammes de classes





5.2.5 Exemple de code

```
//création du document XMI et du modèle
XmiDocument doc=new XmiDocument();
XmiModel model=new XmiModel(doc, "FirstTest");
doc.addModel(model);

//création des classes et des types de données
XmiClass _class=new XmiClass(doc,"FirstClass");
model.addClass(_class);
XmiDatatype dt1=new XmiDatatype(doc,"string");
model.addDatatype(dt1);
XmiDatatype dt2=new XmiDatatype(doc,"boolean");
model.addDatatype(dt2);

//création des opérations avec leurs arguments, et ajout à la classe
XmiOperation op=new XmiOperation(doc, "say_hello");
_class.addOperation(op);
XmiParameter arg1=new XmiParameter(doc,"argument1","in",dt1);
op.addArgument(arg1);
XmiParameter arg2=new XmiParameter(doc,"argument2","inout",dt2);
op.addArgument(arg2);
XmiOperation op2=new XmiOperation(doc,"say_goodbye");
_class.addOperation(op2);

//création du diagramme de séquence et des rôles
XmiSequenceDiagram seqDiag=new XmiSequenceDiagram(doc, "FirstDiagram");
model.addSequenceDiagram(seqDiag);
XmiClassifierRole role1=new XmiClassifierRole(doc,"InstanceTest1",_class);
seqDiag.addClassifierRole(role1);
XmiClassifierRole role2=new XmiClassifierRole(doc,"InstanceTest2",_class);
```

```
    seqDiag.addClassifierRole(role2);
    XmiClassifierRole role3=new XmiClassifierRole(doc,"InstanceTest3",_class);
    seqDiag.addClassifierRole(role3);

    //création des actions et ajout au modèle
    XmiCallAction action=new XmiCallAction(doc,"action1",op);
    model.addCallAction(action);
    XmiCallAction action2=new XmiCallAction(doc,"action2",op2);
    model.addCallAction(action2);

    //création des messages et ajout au diagramme
    XmiMessage msg=new XmiMessage(doc,"message1",role1,role2,action,null);
    XmiMessage msg2=new XmiMessage(doc,"message2",role3,role1,action,null);
    XmiMessage msg3=new XmiMessage(doc,"message3",role2,role1,action2,null);
    seqDiag.addMessage(msg);
    seqDiag.addMessage(msg2);
    seqDiag.addMessage(msg3);

    //ajout des extensions XMI, au choix: MagicDraw OU Rational Rose
    //doc.createMagicDrawExtensions();
    doc.createRoseExtensions();

    //écriture du document XMI dans un fichier
    doc.writeToFile("./testXmi.xml");
```

5.3 Visualisation des diagrammes de séquence obtenus

5.3.1 Avec MagicDraw UML

5.3.1.1 Comment procéder

Il est nécessaire d'avoir le fichier *uml13.dtd* dans le même répertoire que le fichier XMI, parce que MagicDraw valide le document XML. Pour importer le fichier, il suffit de lancer MagicDraw, de cliquer sur « File->Open Project… », puis de sélectionner le fichier en question. Pendant le chargement, MagicDraw affiche un message d'erreur disant que le fichier n'utilise pas le bon méta modèle. Ceci est tout simplement dû au fait que pour avoir la compatibilité avec Rational Rose nous avons dû mettre la version du méta modèle à 1.1 au lieu de 1.3, même si la DTD utilisée est bien celle du format 1.3. Il est donc tout à fait possible de changer cette indication de version dans le fichier XMI.

5.3.1.2 Résultats obtenus

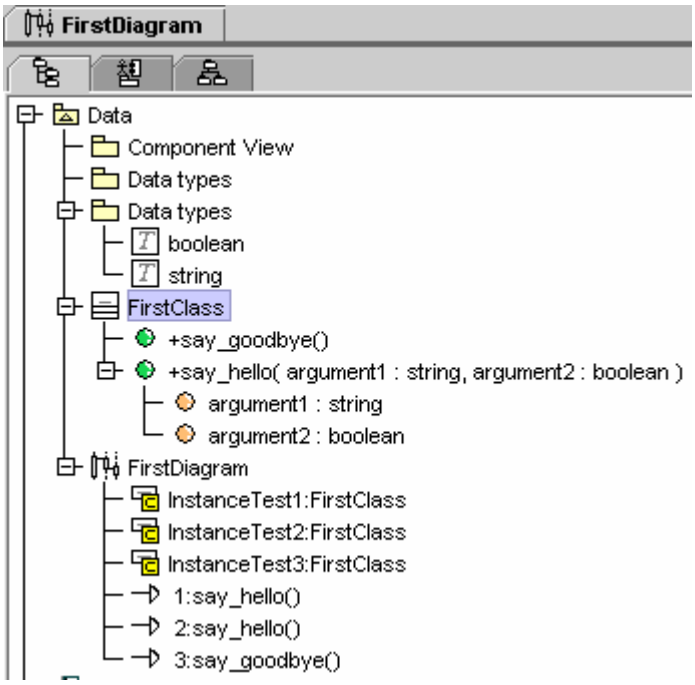


Figure 11 : L'arborescence

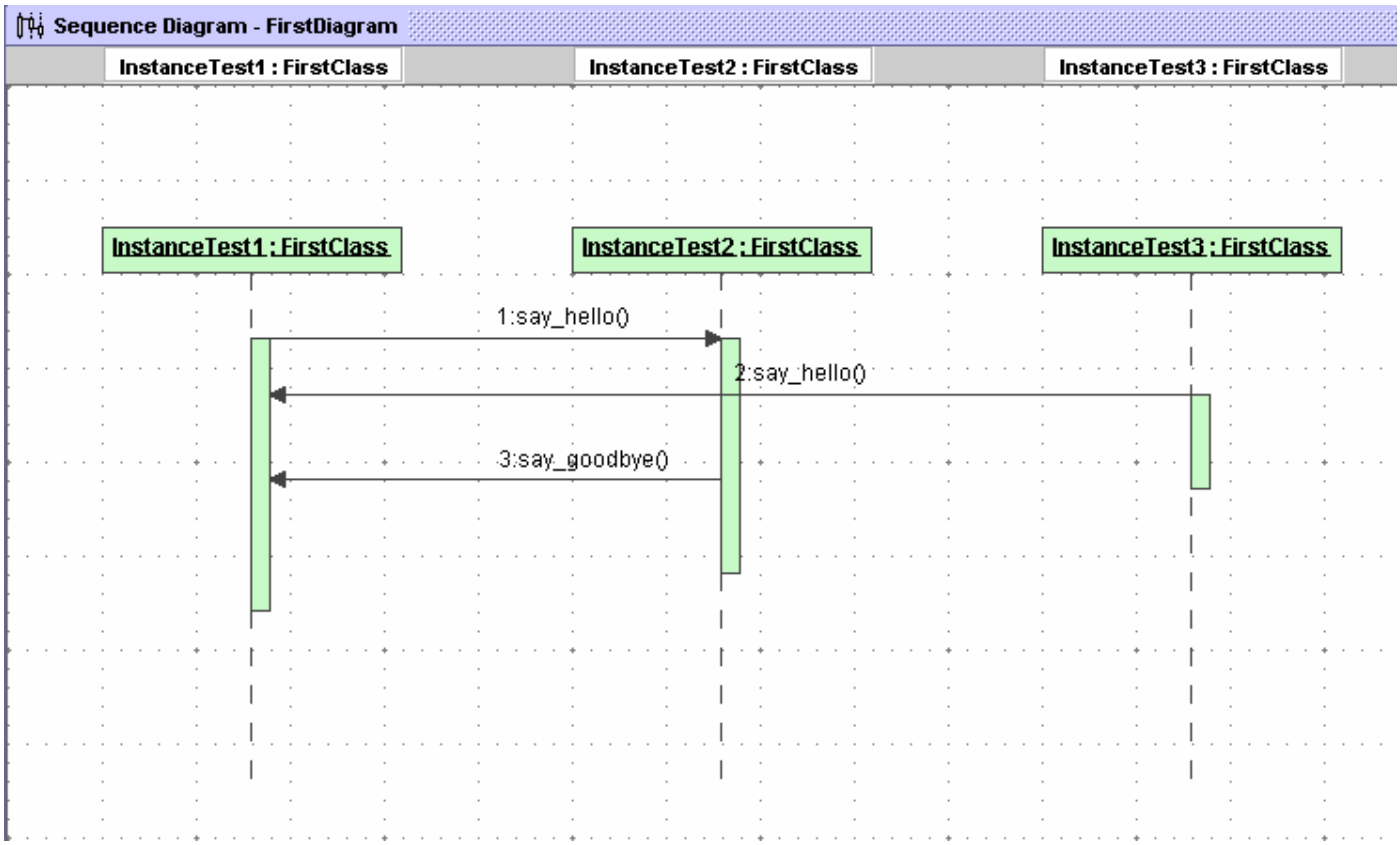


Figure 12 : Le diagramme de séquence

5.3.2 Avec Rational Rose

5.3.2.1 Comment procéder

Rational Rose ne supporte pas le XMI nativement (même pas dans la version 2002, une honte !). Pour pouvoir importer des fichiers XMI dans Rational Rose, il faut donc installer un plugin. Ce dernier est fourni par Unisys, et il est libre d'utilisation, mais n'existe que pour Windows.

Note : attention ! Il y a presque toujours un problème d'enregistrement des DLL du plugin. Il faut relancer le fichier « *reg.bat* » en étant positionné dans le répertoire *system32* de windows, ou mettre le chemin du répertoire *sysem32* dans le *PATH*.

Avant d'importer un fichier XMI, il faut soit le placer dans le répertoire « XMI Files » du répertoire d'installation du plugin, soit copier le fichier uml13.dtd dans le même répertoire que le fichier XMI. Ensuite, il suffit de cliquer sur « File->Import UML 1.1 XMI file ».

Si l'assistant de création de projet s'ouvre, il faut cliquer sur « cancel », pour continuer l'import. Pendant l'importation, plusieurs messages d'erreur apparaissent (normalement 2), il faut cliquer sur « No ». Ils apparaissent parce que la DTD que nous utilisons est celle de UML1.3, et que le plugin ne supporte que le méta modèle 1.1, alors certaines balises ne sont pas comprises par le parseur. A la fin, il y a un message critique s'affiche, disant que « le fichier MDL n'existe pas ». C'est une erreur de programmation du plugin, il suffit de cliquer sur « OK » et un nouveau message s'affiche indiquant le l'import est terminé. Ouf !

Le diagramme de séquence que l'on peut alors visualiser comporte des messages qui sont dans le bon ordre chronologique, mais mal disposés. On n'y peut malheureusement rien au niveau des extensions XMI. De plus, Rational Rose ne connaît pas le concept UML de *CallAction*, ce qui fait qu'on ne peut pas retrouver sur le diagramme les actions qui sont déclenchées par chaque message. Cette information n'est pas représentée, et de ce fait, le libellé affiché au-dessus des messages est leur nom, et non pas le nom de l'Opération appelée par l'Action.

Le package « Data types » qui contient normalement tous les types de données utilisés et définis dans le fichier XMI apparaît vide, car le moteur d'importation du plugin ne connaît pas ce concept. Il est clair qu'une fois que l'on a lu tout ça, on a envie d'essayer MagicDraw UML qui est quand même bien mieux !

5.3.2.2 Résultats obtenus

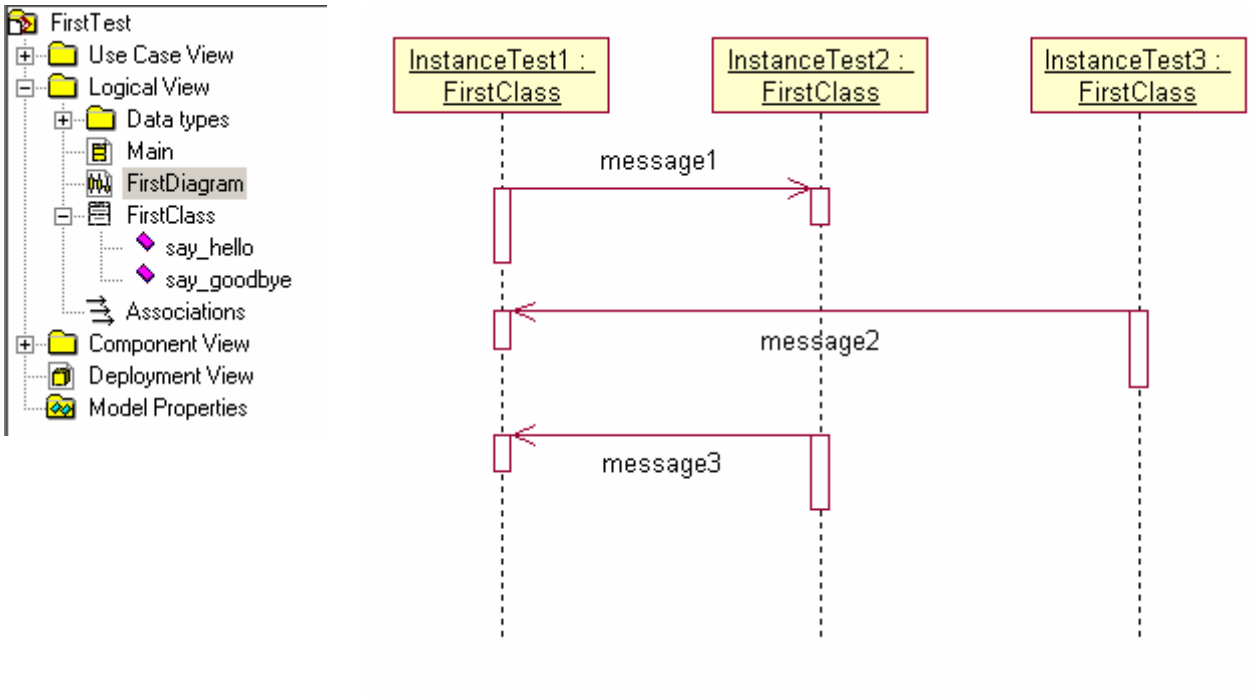


Figure 13 : L'arborescence et le diagramme de séquence

5.4 L'application Xmi2tex

5.4.1 Utilité et fondements

Comme cela avait été fait dans le projet de l'année précédente, nous avons souhaité proposer une visualisation des diagrammes de séquence en utilisant l'outil Latex. Ce dernier est disponible pour toutes les plateformes et c'est un outil largement utilisé et puissant. Pour créer des diagrammes de séquence, nous utilisons la librairie *msc.sty*, qui est téléchargeable gratuitement sur le site de ses créateurs.

Plutôt que d'écrire une deuxième version de l'application log2xmi ou une extension pour celle-ci, le plus simple était d'utiliser comme point de départ les fichiers XMI déjà générés. Ceci a donc entraîné le développement d'une seconde application, appelée *xmi2tex*, qui prend un fichier XMI (*.xml*) en entrée et livre un fichier *.tex* en sortie.

5.4.2 Fonctionnement

L'application *Xmi2tex* se trouve dans le package *corbaTrace.xmi2tex*. Elle est composée d'une seule classe qui contient un *main* et qui s'appelle *Xmi2tex*.

En fait, le plus simple pour passer d'un fichier XMI à un fichier TEX était d'utiliser un processeur XSLT. Celui-ci applique un fichier de style XSL au fichier XML donné en entrée. Le fichier XSL comporte des commandes qui spécifient les données à écrire en sortie pour certaines des balises rencontrées dans le fichier XML.

L'application xmi2tex parse le document XML pour le transformer en structure DOM, le valide avec sa DTD, puis crée un transformer XSLT qui combine la DOM avec le fichier XSL, et écrit le résultat dans un fichier *tex*. Tous les outils utilisés sont les outils standard du package JAXP de SUN, qui sont inclus dans le J2SDK depuis la version 1.4.

5.4.3 Utilisation

Pour des questions de fiabilité, xmi2tex valide le fichier XMI avec la DTD indiquée. Celle-ci doit se trouver dans le même répertoire que le fichier XMI. Le fichier XSL doit également se trouver dans le même répertoire.

Pour lancer le processus, il suffit ensuite de lancer la commande :

```
java corbaTrace.xmi2tex.Xmi2tex <fichier XMI> <fichier sortie tex>
```

Enfin, il faut compiler le fichier *.tex* obtenu avec *Latex*, en ayant préalablement configuré la librairie *msc.sty*. Ceci permet d'obtenir un fichier PS ou PDF facile à visionner.

5.4.4 Fichier XSL

```
<?xml version='1.0' encoding="UTF-8" ?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.1">
      <xsl:output method="text" version="1.0" encoding="UTF-8" indent="yes" />

      <xsl:template match="/">
\documentclass{article}
\usepackage{msc/msc}
\begin{document}

\begin{msc}{Sequence chart}
            <xsl:for-each
select="XMI/XMI.content/Model_Management.Model/Foundation.Core.Namespace.ownedEl
ement/Behavioral_Elements.Collaborations.Collaboration">
                  <xsl:apply-templates
select="Foundation.Core.Namespace.ownedElement/Behavioral_Elements.Collaboration
s.ClassifierRole" />
                  <xsl:apply-templates
select="Behavioral_Elements.Collaborations.Collaboration.interaction/Behavioral_
Elements.Collaborations.Interaction/Behavioral_Elements.Collaborations.Interacti
on.message/Behavioral_Elements.Collaborations.Message" />
            </xsl:for-each>
\end{msc}

\end{document}
      </xsl:template>

      <xsl:template
match="Foundation.Core.Namespace.ownedElement/Behavioral_Elements.Collaborations
.ClassifierRole">
\declinst{<xsl:value-of select="@xmi.id" />}{}{<xsl:value-of
select="Foundation.Core.ModelElement.name" />}
      </xsl:template>


      <xsl:template
match="Behavioral_Elements.Collaborations.Collaboration.interaction/Behavioral_E
lements.Collaborations.Interaction/Behavioral_Elements.Collaborations.Interactio
n.message/Behavioral_Elements.Collaborations.Message">
\mess{<xsl:value-of select="Foundation.Core.ModelElement.name" />}{<xsl:apply-
templates
select="Behavioral_Elements.Collaborations.Message.sender/Behavioral_Elements.Co
llaborations.ClassifierRole" />}{<xsl:apply-templates
select="Behavioral_Elements.Collaborations.Message.receiver/Behavioral_Elements.
Collaborations.ClassifierRole" />}[0]
\nextlevel[1]
      </xsl:template>

      <xsl:template
match="Behavioral_Elements.Collaborations.Message.sender/Behavioral_Elements.Col
laborations.ClassifierRole">
            <xsl:value-of select="@xmi.idref" />
      </xsl:template>
```

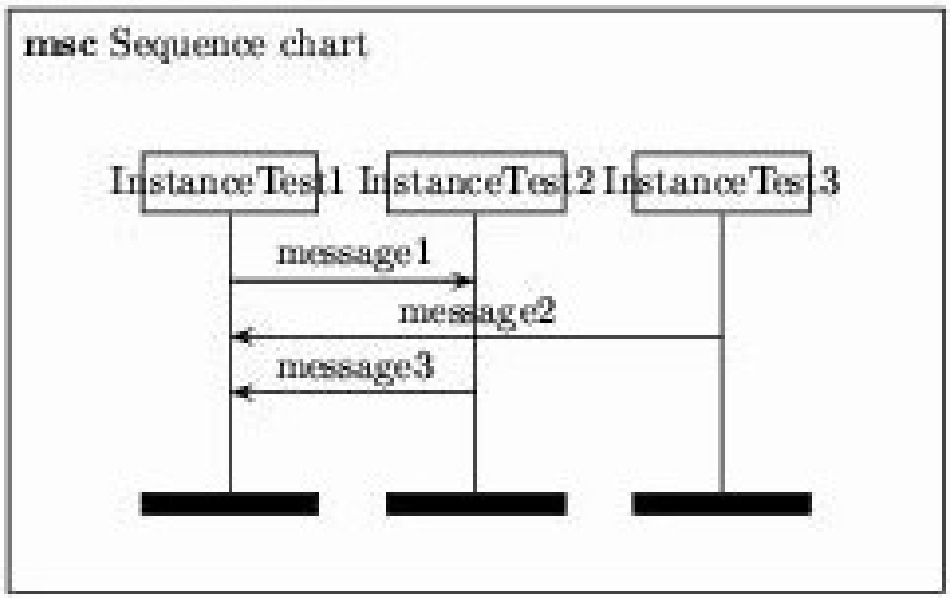
```
      <xsl:template
match="Behavioral_Elements.Collaborations.Message.receiver/Behavioral_Elements.C
ollaborations.ClassifierRole">
            <xsl:value-of select="@xmi.idref" />
      </xsl:template>
</xsl:stylesheet>
```

5.4.5 Résultat

5.4.5.1 Exemple de fichier TEX

```
\documentclass{article}
\usepackage{msc/msc}
\begin{document}
\begin{msc}{Sequence chart}
\declinst{ID12}{}{InstanceTest1}
\declinst{ID13}{}{InstanceTest2}
\declinst{ID14}{}{InstanceTest3}
\mess{message1}{ID12}{ID13}[0]
\nextlevel[1]
\mess{message2}{ID14}{ID12}[0]
\nextlevel[1]
\mess{message3}{ID13}{ID12}[0]
\nextlevel[1]
\end{msc}
\end{document}
```

5.4.5.2 Le diagramme de séquence correspondant

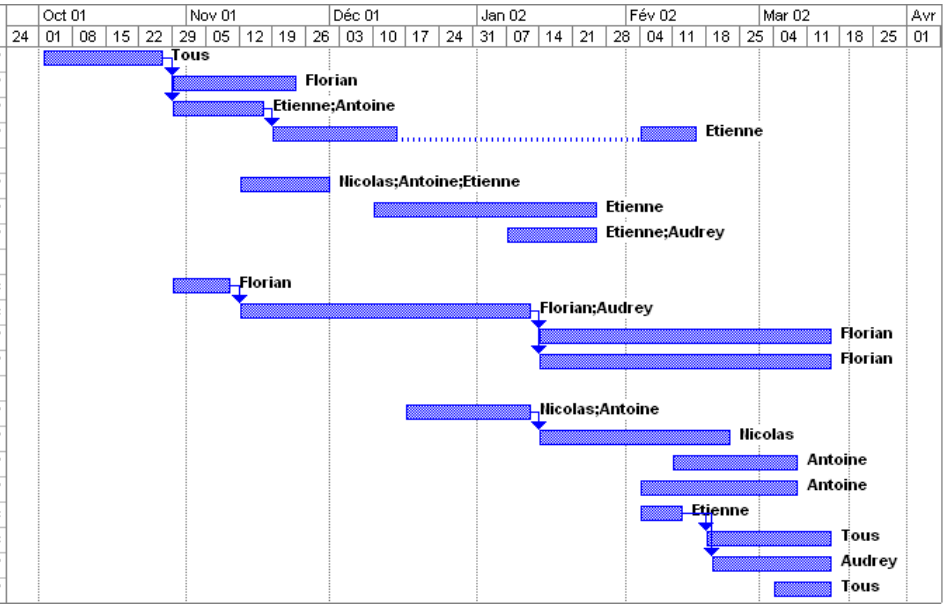


6 Gestion du projet

6.1 Travail d'équipe

6.1.1 Planning

N°	Nom de la tâche	Durée
1	Prise en main	19 jours?
3	Essais sur Naming Service	20 jours?
4	Portage ant + JDK 1.4 + OOB 4.1	15 jours?
5	Réarrangement des sources des PI	30 jours?
11		
12	DTD des logs	15 jours?
13	heures + IDs dans PI	35 jours?
14	Simplification de l'intégration dans l'hôte	15 jours?
15		
16	Génération Latex	10 jours
17	Génération XMI générale	45 jours
18	Génération XMI Rational Rose	45 jours?
19	Génération XMI Magic Draw	45 jours?
20		
21	Parsage des logs	20 jours?
22	Fusion des messages	30 jours?
23	Fichier de configuration des filtres	20 jours?
24	Synchronistation	25 jours?
25	Création du site web	7 jours
26	Création pages HTML anglaises	20 jours
27	traduction pages HTML en français	19 jours?
28	Rapport et exposé	10 jours?



6.1.2 CVS

CVS est un outil GNU de gestion des versions de sources. C'est un outil qui est indispensable après l'avoir connu. Il permet :

- D'archiver toutes les versions successives d'un même fichier, et donc de pouvoir revenir en arrière sans jamais rien perdre
- De travailler simultanément à plusieurs sur les mêmes fichiers
- De fusionner le code automatiquement ou manuellement en cas de conflit sur une même ligne de code
- De centraliser la sauvegarde des sources et ainsi d'avoir toujours la dernière version des sources disponibles
- De pouvoir naviguer avec un browser et CVSWeb les sources et voir des résumés des évolutions apportées aux fichiers

Grâce à cet outil, nous n'avons jamais eu les problèmes habituels de travail en équipes comme par exemple gérer une dizaine de versions différentes du même programme ou une disquette défectueuse qui fait perdre une semaine de travail !

Bien que son utilisation soit très simple, il est important de ne pas en faire n'importe quoi. Nous avons par exemple interdit à quiconque de « commiter » (valider) des fichiers qui ne compilent pas ou de changer l'architecture du CVS sans en parler aux autres. Le CVS doit toujours être stable et utilisable au moins en partie à n'importe quel moment.

Quand nous sommes passés sur TuxFamily (voir ci-dessous), nous avons dû gérer deux CVS. Le choix a été fait de garder deux CVS et de ne pas travailler directement sur celui-ci pour ne pas prendre de risque de mauvaise manipulation. Régulièrement, les fichiers du CVS local sont donc mis à jour sur TuxFamily, mais uniquement en cas de nouvelle fonctionnalité ou de bug majeur. Une fois le projet terminé, nous ne travaillerons bien évidemment que sur le CVS accessible par le web.

Sur TuxFamily, nous avons pu également mettre en place la fonctionnalité de récupérer le CVS anonymement. Ainsi, n'importe qui peut récupérer la dernière version de CorbaTrace et pourquoi pas modifier les sources et nous envoyer ses modifications. Dans ce cas, il nous est possible d'autoriser un utilisateur en écriture sur le CVS.

6.1.3 XML

Le choix du XML comme format intermédiaire entre nos modules nous a permis d'améliorer le travail d'équipe. Ainsi, la partie d'interception n'avait besoin que d'écrire dans un fichier du XML, indépendamment de son traitement. Et inversement, si la partie Log2XMI prenait de l'avance, il était tout à fait aisé d'écrire un fichier de log XML à la main, avec son propre jeu de test. Le résultat était le même avec la partie XMI qui a pu faire des tests de génération de tests de fichiers XMI sur différents AGL sans être dépendant de Log2XMI.

Chaque groupe a pu ainsi travailler à son rythme sans pénaliser les autres.

6.2 Site Web

Pour diffuser CorbaTrace, il est essentiel d'avoir un site web. La première étape a été de trouver un hébergeur.

6.2.1 Hébergeur

Le premier auquel nous avons pensé est bien évidemment SourceForge. C'est le plus gros et le plus complet de tous les hébergeurs libres du Web. Le problème de SourceForge vient d'un changement récent de sa licence d'exploitation. En effet, VA Software (anciennement VA Linux) a changé le contrat entre les développeurs et eux. Selon des développeurs expérimentés du libre et selon de nombreux articles de journaux, ce nouveau contrat laisse la possibilité à VA Linux de s'approprier les travaux hébergés sur SourceForge.

Nous nous sommes donc tournés vers un nouvel hébergeur libre nommé Savannah. Celui-ci reprend le moteur GPL de SourceForge (qui est très complet) mais il est géré directement par GNU. Nous avons soumis notre demande d'hébergement et une longue négociation a alors débuté. Le problème est la dépendance de CorbaTrace vis-à-vis de logiciels propriétaire. Nous avons donc transformé le code pour qu'il soit indépendant de tout ORB et de tout parseur XML propriétaire. Mais le problème subsiste à cause de notre dépendance vis-à-vis du JDK 1.3 et 1.4. En effet, le JDK de Sun est gratuit et Open Source mais il n'est pas libre. Or, le projet GNU se veut cohérent et ne veut pas s'appuyer sur du code appartenant à une entreprise.

Un projet GNU de machine virtuelle Java existe ainsi qu'un début de compilateur. Celui-ci n'est autre que GCC 3.0, mais il ne supporte que le JDK 1.1. Avec quelques efforts, nous pourrions rendre compatible CorbaTrace avec cette version mais nous n'en avons pas eu le temps et ce n'était pas prioritaire.

Enfin, la décision a été prise d'utilisé un hébergeur libre et français : TuxFamily. Nous avons ainsi eu à disposition un CVS, CVSWeb, un accès FTP, un site web, des statistiques, etc.
L'adresse du site est : http://corbatrace.tuxfamily.org

6.2.2 Conception

La première étape de la conception du site web a été de créer un logo. Celui que nous avons fait n'ai pas parfait et mériterait d'être amélioré pour avoir une meilleure identité visuelle, mais il a déjà l'avantage d'exister.

Figure 14 : Logo de CorbaTrace

Ensuite, le design général a été réalisé sous GIMP avec une maquette en XHTML. Le Javascript a été évité et le site avait comme impératif de marcher sur tous les navigateurs. Nous avons donc respecté au plus près la dernière norme du W3C en matière de présentation web.

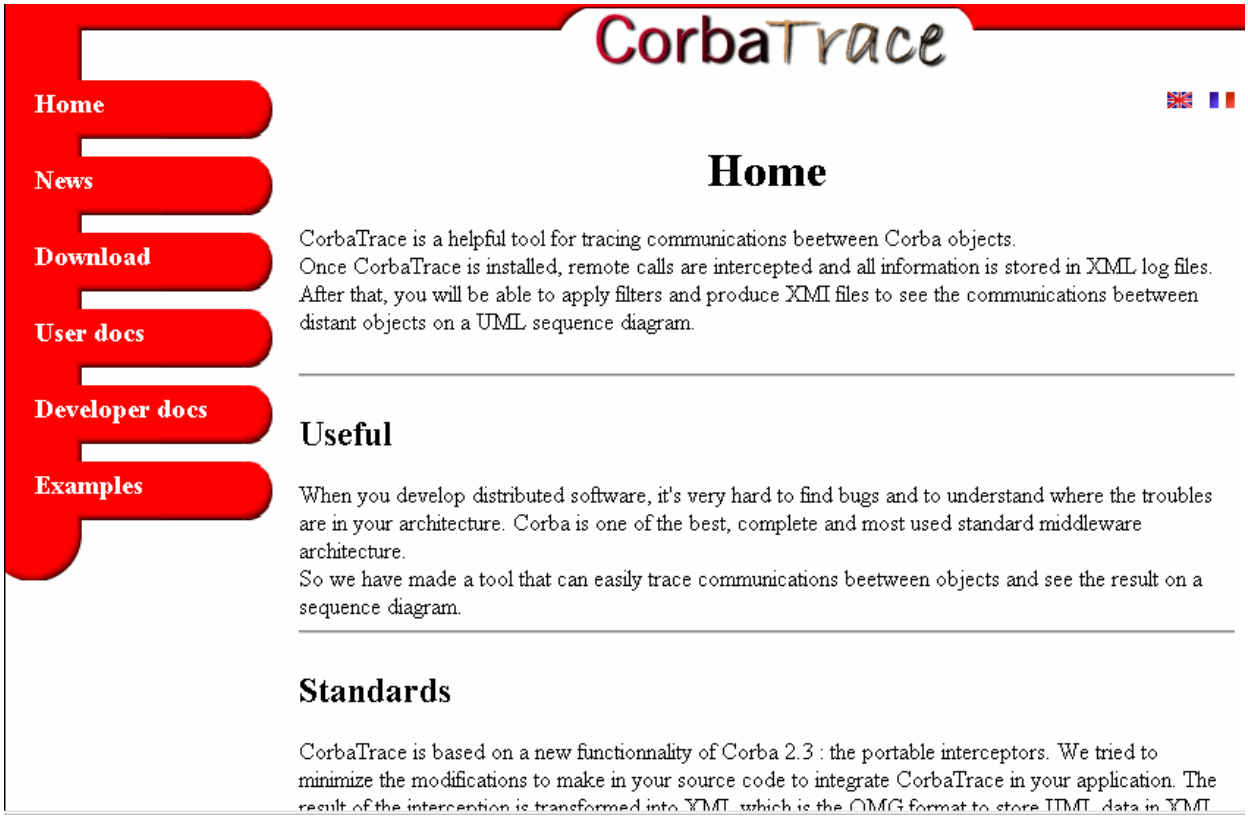


Figure 15 : page d'accueil anglaise du site CorbaTrace

Pour permettre une évolution facile du site, un moteur PHP a été utilisé pour générer le HTML. Une solution à base de XML et de XSLT a été commencée puis abandonner car le PHP suffisait.

Le PHP nous permet par exemple d'ajouter ou modifier un bouton d'une manière très simple. Le code suivant suffit à l'utilisateur pour générer la barre de boutons sur la gauche :

```
if(!strcmp($lang, "fr")) {
      button_titled("home", "Présentation", $lang);
      button_titled("news", "Nouveautés", $lang);
      button_titled("download", "Téléchargement", $lang);
      button_titled("docs", "Docs utilisateur", $lang);
      button_titled("install", "Docs développeur", $lang);
      button_titled("example", "Exemples", $lang);
}
```

Le PHP a également été mis à contribution pour ne pas utiliser les frames du HTML qui sont proscrites du XHTML. Ainsi, tous les liens du site passent par index.php avec un argument `$page` passé par l'URL qui permet de savoir la page à inclure.

Nous avons aussi voulu rendre le site multi-langages en écrivant des pages en français et d'autres en anglais. Un autre argument `$lang` est donc passé en argument des liens hypertextes. Si la page dans la langue n'est pas disponible, on l'affiche en anglais.

6.2.3 Statistiques

Pour permettre de faire connaître CorbaTrace, une petite campagne de publicité a dû être nécessaire. Nous avons donc enregistré notre site sur des moteurs de recherche, nous avons discuté avec des développeurs Corba sur des forums de discussion (comp.langage.corba, fr.comp.langage.corba, comp.langage.java, …), et nous l'avons inscrit sur des sites de recherches de programmes informatiques.

Ainsi, nous sommes aujourd'hui référencé sur quelques uns des plus sites informatiques du Web :

- www.jars.com : répertoire des programmes java (après java.sun.com et javaworld.com, le plus gros site du monde sur Java).
- www.freshmeat.net : répertoire de programmes libre
- www.jesuislibre.org : répertoire de programmes libre français
- www.linuxfr.org : site français de nouvelles sur le libre
- www.linuxjournal.org : journal en ligne sur le libre (un article sur le débuggage de programmes Corba faisait un encart complet sur CorbaTrace)

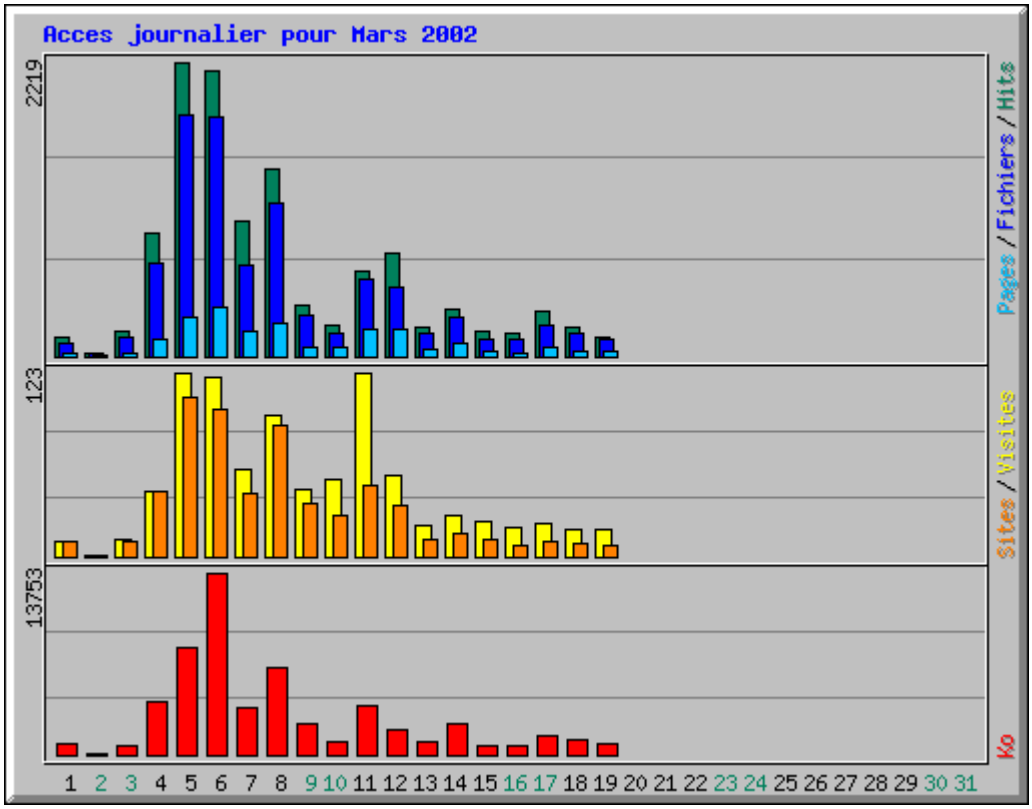


Figure 16 : Statistiques pour le mois de Mars

Grâce à TuxFamily, nous avons également des statistiques complètes sur la fréquentation du site. On a ainsi pu constater 10000 hits et 800 visites en deux semaines. 25 % venait d'Allemagne, 15 % du Japon et 10% de Russie. Le reste venait de France et des USA (les accès depuis Nantes représentent 20% du trafic). Nous avons donc bien fait de créer tout de suite le site en anglais et de commenter tout le code en anglais.

7 Conclusion

7.1 Ce que le projet nous a apporté

D'une manière générale, nous avons tous beaucoup appris pendant ce projet. D'abord, nous nous sommes tous familiarisés avec Corba et avons fait des tests. Puis nous nous sommes plongés dans les documentations, notamment les spécifications, pour décortiquer les envois de messages et les mécanismes d'interception. Ensuite, nous avons découvert la norme XMI, et également disséqué celle-ci afin de pouvoir l'exploiter correctement. Enfin, les transformations des logs vers le XMI ont nécessité un travail d'analyse et de réflexion approfondis au niveau des algorithmes à utiliser et leur mise en œuvre.

Ce projet nous a également permis de travailler en équipe. Cela a nécessité de planifier, de s'organiser, de se réunir, de se partager les tâches, de communiquer. Nous avons aussi suivi une relation maître d'œuvre / maître d'ouvrage avec les enseignants qui nous ont encadrés. Les réunions que nous avons faites avec eux ont été très constructives, et ont permis d'avancer dans nos idées sur le projet et d'apporter de nouvelles perspectives et solutions.

7.2 Un projet finalisé

CorbaTrace est un projet qui a été mené jusqu'à la fin. Ce projet fonctionne de bout en bout, depuis la mise en place facilitée de l'interception dans un programme jusqu'à la visualisation des diagrammes de séquence obtenus. Nous avons abouti à un ensemble de livrables qui sont même maintenant diffusés sous licence LGPL, avec l'accord de l'université. Un site web complet a été mis en place, permettant de publier tout CorbaTrace, avec Javadoc, CVS, versions packagées pour téléchargement, documentation utilisateur et documentation développeur en anglais et en français.

Notre satisfaction d'avoir développé un ensemble de programmes qui peuvent être utiles à toute personne développant en Corba est récompensée par le succès que rencontre le site au point de vue du nombre de téléchargements et du nombre d'articles ou liens sur CorbaTrace sur d'autres sites d'informatique.

8 Bibliographie

8.1 Références Corba

Spécification officielles Corba 2.3 : http://www.omg.org
Entre-aides entre développeurs Corba francophones : http://www.developper.com
Newsgroups sur Corba : comp.lang.java.corba, comp.orbject.corba, borland.public.corba
Orbacus : http://www.iona.com
Très bon site sur les intercepteurs Corba : http://corbatrace.tuxfamily.org

8.2 Références Java

Site officiel de java : http://java.sun.com
News sur Java : http://www.javaworld.com
Recherche de programmes Java : http://www.jars.com
Parsers XML : http://xml.apache.org
Ant : http://jakarta.apache.org/ant
JVM GNU : http://www.gnu.org/classpath

8.3 Références XMI

Object Management Group: http://www.omg.org
Spécifications XMI 1.2: http://www.omg.org/cgi-bin/doc?formal/2002-01-01
DTD de XMI pour UML 1.3 : http://www.omg.org/cgi-bin/doc?formal/01-04-03
Plugin XMI de Unisys pour Rational Rose :
http://www.rational.com/support/downloadcenter/addins/rose/
Rational Rose : http://www.rational.com/products/rose/
MagicDraw UML : http://www.magicdraw.com
Poseidon for UML : http://www.gentleware.com
Package msc pour Latex : http://www.win.tue.nl/cs/fm/sjouke/mscpackage.html

9 Annexes

9.1 Mode d'emploi utilisateur

Nous essayons de simplifier les modifications requises dans votre projet pour activer CorbaTrace. Nous utilisons uniquement les méthodes de l'OMG pour l'installer. Ce fichier peut ne pas être parfaitement synchronisé avec les derniers fichiers sources. Voir dans le CVS le fichier INSTALL et dans l'API JavaDoc pour les nouvelles instructions.

9.1.1 Installation dans votre projet

Vous avez un exemple du serveur et un du client dans ./src/java/corbatrace/hello.

Inclure uniquement CorbaTrace.jar dans votre CLASSPATH.

```
set CLASSPATH=$CLASSPATH:mypath/CorbaTrace.jar
```

Ajouter au début de vos fichiers sources :

```
import corbaTrace.Interceptor;
```

9.1.1.1 Client

Dans vos sources, avant la création de l'ORB, initialisez l'intercepteur en créant une instance d'`InterceptorClient`.

Après l'initialisation de l'ORB, et avant les appels distants, ajoutez :

```
obj = interceptorClient.active_interception(obj, orb);
```

où obj est créé de la façon suivante :

```
org.omg.CORBA.Object obj = orb.string_to_object(myIOR);
```

Vous pouvez obtenir votre propre objet en utilisant la fonction standard suivante :

```
Hello hello = HelloHelper.narrow(obj);
```

Une fois tous les objets créés, ajoutez :

```
interceptorClient.activate_log(orb, "My Name");
```

"My Name" est utilisé pour identifier votre composant. Tous les objets client sur le même orb font partis du même composant.

Si vous voulez changer la stratégie d'interception, appelez la methode suivante :

```
interceptorClient.change_level_interception(obj, orb, int);
```

(Se référer au site de l'OMG pour plus de détails).

Voici un exemple :

```
import corbatrace.InterceptorClient;
...
class MyClass {
      ...
      interceptorClient = new InterceptorClient();
      ...
      ORB orb = ORB.init(args, props);
      ...
      obj = orb.string_to_object(ref);
      obj = interceptorClient.active_interception(obj, orb);
      interceptorClient.activate_log(orb, "My Component");
      ...
      Hello hello = HelloHelper.narrow(obj);
      ...
}
```

9.1.1.2 Server

Dans vos sources, avant la création de l'ORB, initialisez l'intercepteur en créant une instance d'`InterceptorServer`.

Après l'initialisation de l'ORB, appelez la méthode ci-dessous pour obtenir les références sur le RootPOA :

```
obj = orb.resolve_initial_references("RootPOA");
```

où obj est un org.omg.CORBA.Object.

Obtenez votre propre rootPOA en utilisant la méthode standard suivante :

```
POA rootPOA = org.omg.PortableServer.POAHelper.narrow(obj);
```

Puis créez le POA en utilisant la méthode "create_poa" de la classe `InterceptorServer` :

```
poa_interceptor = interceptorServer.create_poa(orb, rootPOA, "myHelloPOA");
```

où "myHelloPOA" est le nom du nouveau POA créé dynamiquement, et rootPOA votre propre POA.

Activez l'objet sur ce POA :

```
obj = interceptorServer.activate_object(poa, helloImpl, "My Component");
```

Cette méthode enregistre et active l'objet sur le POA qui vient d'être créé. Ce POA permettra aux applications clientes d'y accéder.
Cette méthode active aussi le POAManager qui contrôle la manière avec laquelle le POA gère les requêtes des différents clients.

Obtenez votre propre objet en utilisant la méthode standard suivante :

```
Hello hello = HelloHelper.narrow(obj);
```

Enregistrez votre objet "hello" dans un fichier (ou dans le CosNaming) :

```
writeObjectToFile(orb, hello, filename);
```

Mise en attente du serveur : celui-ci écoute les demandes des différents clients et y répond en appelant les méthodes correspondantes sur l'objet serveur :

```
orb.run();
```

Destruction du serveur avant de quitter le programme :

```
orb.destroy();
```

Voici un exemple :

```
import corbaTrace.InterceptorServer;
...
class MyClass {
      ...
      interceptorServer= new InterceptorServer();
      ...
      ORB orb = org.omg.CORBA.ORB.init(args,props);
      obj = orb.resolve_initial_references("RootPOA");
      POA rootPOA = org.omg.PortableServer.POAHelper.narrow(obj);
      poa_interceptor = interceptorServer.create_poa(orb, rootPOA,
"myHelloPOA");
      ...
      obj = interceptorServer.activate_object(poa, helloImpl,"My
Component");
        Hello hello = HelloHelper.narrow(obj);
          writeObjectToFile(orb, hello, "My File");
      ...
      orb.run();
      orb.destroy();
}
```

9.1.2 Résultat

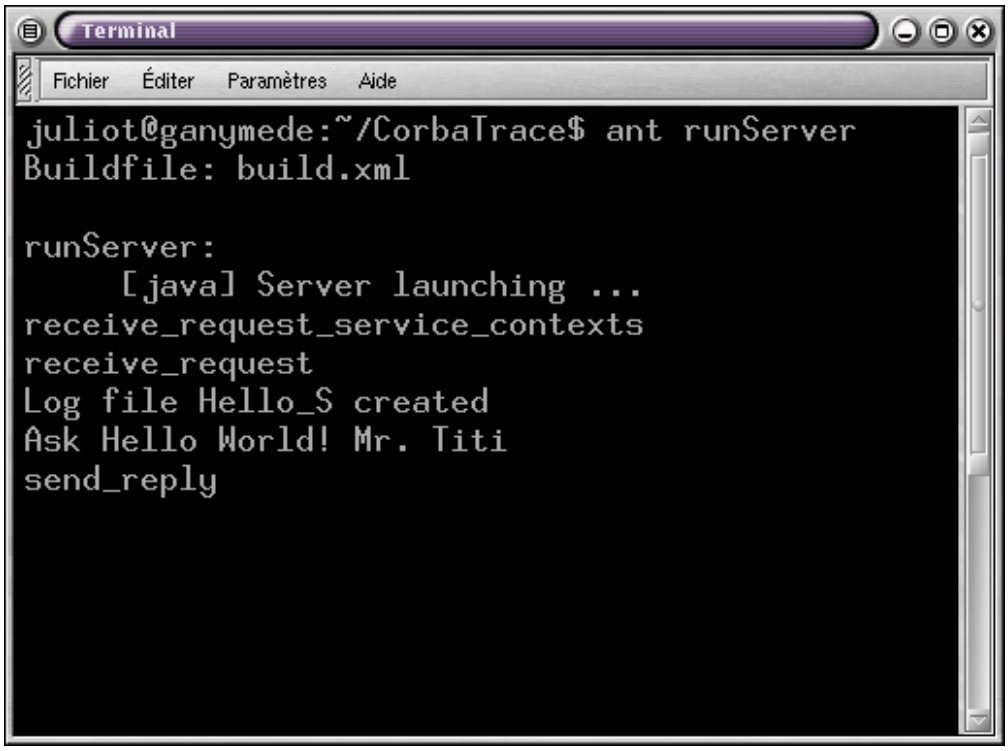
Compilez votre projet et exécutez-le. Des fichiers log sont générés (*_Ctrace.xml pour les objets client et *_Strace.xml pour les objets serveur). Leur compréhension est facile.

Vous pouvez les convertir en XMI et utilisez d'autres filtres (pas encore implémenté). Vous pouvez voir XMI dans tous les outils UML qui lisent les fichiers XMI standards. Vous pouvez aussi utiliser notre outil appelé : xmi2latex.

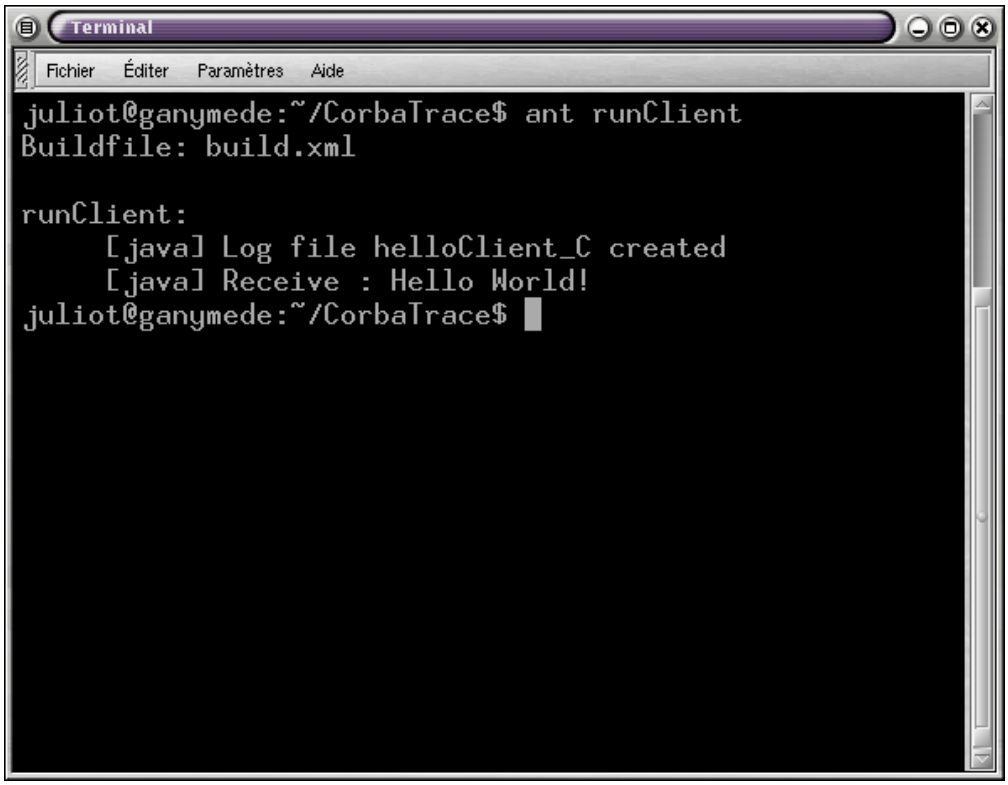
9.1.3 Captures d'écran

Lors de l'exécution de notre exemple vous obtenez les deux captures d'écran ci-dessous :

- **Server**



- **Client**



Après l'interception, deux logs sont générés (fichiers joints).

9.1.3.1 Le log du server

```
<message IDmessage="0 0" type="receive_request"/>
        <local_object IDobject="Hello" date="Wed Feb 27 14:47:39 CET 2002"/>
        <distant_object IDobject="helloClient" date="Wed Feb 27 14:47:39 CET 2002"/>

        <operation name="say_hello">
                <argument inout="in" value="Titi" type=""/>
        </operation>

        <options>
                <exceptions/>
                <responseExpected value="true">
        </options>
</message>


<message IDmessage="0 0" type="send_reply"/>
        <local_object IDobject="Hello" date="Wed Feb 27 14:47:39 CET 2002"/>
        <distant_object IDobject="helloClient" date="Wed Feb 27 14:47:39 CET 2002"/>

        <operation name="say_hello">
                <argument inout="in" value="Titi" type=""/>
        </operation>
        <result value="Hello World!"/>

        <options>
                <exceptions/>
                <responseExpected value="true">
                <reply status="successful"/>
        </options>
</message>
```

9.1.3.2 Le log du client

```
<message IDmessage="0 0" type="send_request"/>
      <local_object IDobject="helloClient" date="Wed Feb 27 14:47:39 CET 2002"/>
      <distant_object IDobject="ServerId" date="Thu Jan 01 01:00:00 CET 1970"/>

      <operation name="say_hello">
            <argument inout="in" value="Titi" type=""/>
      </operation>

      <options>
            <exceptions/>
            <responseExpected value="true">
      </options>
</message>


<message IDmessage="0 0" type="receive_reply"/>
      <local_object IDobject="helloClient" date="Wed Feb 27 14:47:39 CET 2002"/>
```

```
      <distant_object IDobject="ServerId" date="Thu Jan 01 01:00:00 CET 1970"/>

      <operation name="say_hello">
            <argument inout="in" value="Titi" type=""/>
      </operation>
      <result value="Hello World!"/>

      <options>
            <exceptions/>
            <responseExpected value="true">
            <reply status="successful"/>
      </options>
</message>
```

9.2 Generer un Diagramme de Sequence UML

Après la création des logs des messages interceptés, vous pouvez les utiliser avec notre outil (Log2xmi) pour générer un diagramme de séquence UML au format XMI.

Cet outil a quatre objectifs :

- parser vos fichiers de log, et fusionner les messages partiels pour obtenir des messages complets.
- synchroniser toutes les horloges locales des objets à une horloge commune (puisque souvent distribués sur différentes machines à travers le monde).
- appliquer quelques filtres personnels pour obtenir des informations plus pertinentes.
- générer un fichier XMI que vous pouvez directement visualiser sur des outils courants comme MagicDraw et Rational Rose.

La ligne de commande pour Log2xmi est la suivante:

```
java corbaTrace.log2xmi.Main [options] <XML logs>
```

Les logs XML sont les logs que vous avez généré à l'étape précédente. Ils peuvent être donnés sous la forme d'un chemin vers un fichier ou d'une URL. N'oubliez pas de placer le fichier log.dtd dans le même répertoire que vos logs (de même pour les filtres).

Voici un résumé des options :

-o est le fichier xmi dans lequel log2xmi sauve tous les messages lus. Par défaut, log2xmi utilise le nom de fichier "out.xml".

-f donne un fichier de filtre pour restreindre les messages générés (voir plus bas).

-x définit pour quelle application vous souhaitez générer le fichier XMI. -xr signifie pour Rational Rose, -xm pour Magic Draw, et -xrm (ou -xmr) pour les deux.

-v indique de ne pas valider les logs XML avec leur DTD (lors du parsage).
Cela ne devrait rien changer comme les logs générés sont validés, mais vous pouvez l'utiliser si vous voulez toujours générer le fichier xmi même si les logs sont un peu corrompus.
note : cette option s'applique aussi au fichier de filtres (car celui-ci est aussi basé sur log.dtd).
Nous vous recommandons d'utiliser cette option aussi rarement que possible.

-s indique de passer l'étape de synchronisation.
Cela peut être utilisé seulement si vous savez que tous vos objets fonctionnent déjà avec la même horloge (dans le même fuseau horaire), et ainsi l'étape de synchronisation n'est pas nécessaire.
Cela peur accélérer un peu l'ensemble du traitement des logs.

-d affiche plus d'informations à chaque étape.
Elle est inutile dans la plupart des cas et ne devrait être considérée qu'à des fins de débuggage.

9.3 Filtres

Les filtres sont écrits dans un fichier XML, comme défini dans la DTD (log.dtd)

Il fonctionne à deux niveaux :

- de manière globale
- au niveau d'un objet

Au niveau global, vous pouvez restreindre les messages à :

- vos types de messages donnés :
  les types sont : REQUEST, REPLY, EXCEPTION, BROKEN_REQUEST, BROKEN_REPLY, et BROKEN_EXCEPTION.

- vos dates données (ou intervalles) :
  Il y a trois types de filtres de date : après une date, avant une date, entre eux dates.
  Les dates utilisent le même format que les logs : annee-mois-jourTheures-minutes-secondes-millisecondes (par ex: 2002-03-09T17:00:00.000)

- vos opérations données, independamment des objets.
  pour chaque operation vous donnez son nom, et pouvez la restreindre plus précisemment avec les valeurs des arguments de l'opération.

- Les arguments définis sont de deux types
  - un type de données et une valeur
  - une position d'argument dans l'opération (de 1 à n) et une valeur
  - vos objets donnés avec leur ID (nom de l'objet)

- Pour des objets donnés, vous pouvez aussi restreindre vos messages plus précisemment à des types de messages donnés, des dates, et des opérations, avec les même principes qu'au niveau global.

Chaque filtre différent (date - type - objet - methode) fonctionne comme un "ET" avec les autres.

Chaque composant de filtre (les dates pour un filtre de date, les arguments pour un filtre de méthode, etc.) fonctionne comme un "OU" avec les autres.

Par exemple :

```
<filter>
      <message_types>
            <type value="BROKEN_REQUEST"/>
            <type value="BROKEN_REPLY"/>
            <type value="BROKEN_EXCEPTION"/>
      </message_types>
      <dates>
            <after date="2002-03-09T17:00:00.000"/>
            <before date="2002-03-09T15:00:00.000"/>
            <between from="2002-03-09T15:30:00.000" to="2002-03-
09T16:30:00.000"/>
      </dates>
      <methods>
            <method name="anOperation1">
                  <argumentAt position="3" value="ddd"/>
                  <typedArgument type="string" value="ccc"/>
            </method>
            <method name="anOperation2">
            </method>
      </methods>
      <objects>
            <object id="anObjectID1">
                  <message_types>
                        <type value="BROKEN_REQUEST"/>
                        <type value="BROKEN_REPLY"/>
                        <type value="BROKEN_EXCEPTION"/>
                  </message_types>
                  <methods>
                        <method name="anOperation">
                              <argumentAt position="1" value="aaa"/>
                              <argumentAt position="2" value="bbb"/>
                        </method>
                  </methods>
            </object>
            <object id="anObjectID2">
            </object>
      </objects>
</filter>
```

Ces filtres de messages conservent les messages qui :

```
            sont de type incomplets (n'importe lequel)
        ET sont après 17H (le 9/3/2002)
                OU avant 15H
                OU entre 15H30 et 16H30
        ET sont des operations nommées "anOperation1"
                   avec (un argument à la position 3 égal à "ddd"
                         OU un argument de type string égal à "ccc")
                OU une operation nommée "anOperation2"
        ET concerne un objet "anObject1"
                     avec un message de type incomplet
                          ET concernant l'operation "anOperation"
                                        avec un argument à la position 1 égal à
      "aaa"
                                             OU un argument à la position 2
      égal à "bbb"
           OU concerne un object "anObjectID2"
```

Bien sûr, ce n'est qu'un exemple. Ici, donner les types de messages comme "incomplets" pour l'objet "anObject1" n'est pas utile comme il est déjà défini de manière globale pour le même type de messages.

9.4 Mode d'emploi développeur

Si vous voulez compiler ou modifier les sources de CorbaTrace, suivez les instructions ci-dessous :

9.4.1 Conditions requises

Pour lancer CorbaTrace, vous devez avoir :

- un système d'exploitation compatible Java
- Java Runtime Environnement
- un ORB (Corba 2.3 ou plus)

Pour compiler CorbaTrace, vous devez également avoir :

- Ant 1.4

Ceci est un outil de compilation (comme "Make") écrit en Java et avec un fichier de configuration en XML. Il est gratuit et développé par Apache. Vous pouvez le télécharger sur le site : http://jakarta.apache.org/ant

9.5 Compilation

C'est très simple !

Tapez juste..... :

-> `ant`

(quand vous êtes positionné dans le repertoire CorbaTrace où se situe le fichier build.xml)

C'est tout !

Maintenant, vous avez le fichier jar dans ./dist/corbatrace.jar

Ci-dessous se trouve l'architecture des répertoires des sources :

./
: La racine du projet (with the build.xml of Ant)

./src

Les sources
./src/java
Les sources java
./src/idl
Les sources idl
./src/cpp
Les sources C++
./etc
The config files and XML-DTD example files
./build
The generated files
./build/classes
The generated class files
./build/idlgen
The java files generated by idl
./dist
The jar file
./javadoc
The JavaDoc generated documentation

9.6 Commandes

Pour compiler
```
ant
```
Pour nettoyer le projet
```
ant clean
```
Pour générer la javadoc
```
ant javadoc
```
Pour lancer l'exemple du Server
```
ant runServer
```
Pour lancer l'exemple du Client
```
ant runClient
```

9.7 Licence

Nous avons choisi une licence libre pour CorbaTrace. Cette licence est la LGPL : GNU Lesser General Public License. Notre projet est ainsi open source, modifiable, gratuit, et génial.

Elle est consultable en ligne sur le site de GNU ou dans l'archive de notre projet.