Présentation de Corba

par Marc-Olivier Killijian

Date de publication :

Dernière mise à jour : 19/02/2006

Ce document est tiré d' "Etude des documents de l'OMG en vue de leur utilisation dans des applications multimedias distribuées" par Franck Guerrero.

I - Introduction
II - L'ORB
    A - Object Request Broker
    B - Interface Definition Language
    C - Dynamic Invocation Interface
    D - Interface Repository
    E - Basic Object Adapter
III - Common Object Service Specification 1
    A - Naming Service
    B - Event Service
    C - LifeCycle Service

I - Introduction

L'OMG (Object Management Group) est un consortium regroupant :

- Des fournisseurs de systèmes
- Des fournisseurs de logiciels
- Des utilisateurs finaux

Il a pour but le développement d'applications distribuées dont les composants collaborent avec :

- Efficacité
- Fiabilité
- Transparence
- *Scalability, i.e.,* une capacité d'évolution importante

L'OMG a défini un modèle de référence pour des applications distribuées utilisant des techniques orientées objet. Ce modèle comprend quatre points de standardisation :

- Object Model: c'est un modèle générique pour assurer la communication avec des systèmes orientés objet conformes au modèle de l'OMG
- Object Request Broker (ORB): c'est l'élément clé de communication, il assure la distribution des messages
- ObjectServices (ou encore CORBAServices): ces services fournissent les principales fonctions de base nécessaires à la gestion des objets (nommage, persistance,gestion d'évènements...)
- CommonFacilities (ou encore CORBAFacilities): ce sont des utilitaires destinés aux applications

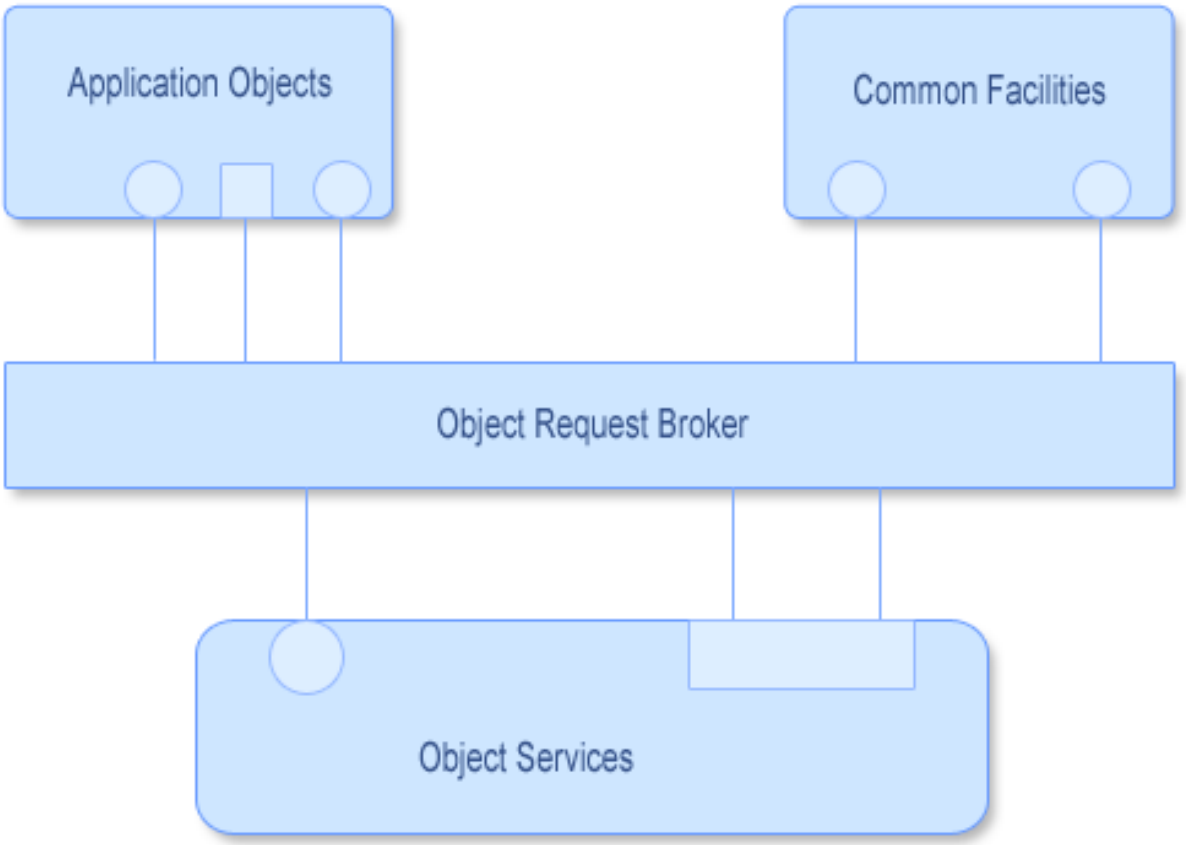

Fig 1

L'OMG a donc défini CORBA (Common Object Request Broker Architecture), une architecture respectant la standardisation ci-dessus. Les principes de CORBA sont :

- Une séparation stricte Interface/Implémentation
- La transparence de la localisation des objets
- La transparence de l'accès aux objets
- Le typage des Object References par les interfaces
- L'héritage multiple d'interfaces

II - L'ORB

A - Object Request Broker

L'ORB fournit les mécanismes par lesquels des objets font des requêtes et reçoivent des réponses, et ce de manière transparente. Il fournit également l'interopérabilité entre des applications sur différentes machines dans des environnements distribués hétérogènes et il interconnecte *sans coutures* de multiples systèmes objets. D'une façon simplifiée, on peut définir l'ORB comme une entité qui fournit des mécanismes d'interrogations permettant de récupérer des objets, des procédures qui constituent une application.

L'ORB est défini plutôt par ses interfaces que comme un unique composant. Il comprend les composants suivants :

- ORB Core/ORB interface
- Interface Definition Language (IDL)
- Dynamic Invocation Interface (DII)
- Interface Repository (IR)
- Basic Object Adapter (BOA)

L'ORB est responsable de tous les mécanismes nécessaires pour :

- Trouver l'implémentation de l'objet pour la requête
- Préparer cette implémentation à recevoir la requête
- Communiquer les données constituant la requête

L'interface que voit le client est complètement indépendante :

- De l'endroit où l'objet est situé
- Du langage dans lequel l'objet est implémenté

L'implémentation *objet, ie. le serveur,* fournit la sémantique de l'objet, définit les données pour une instance d'objet, définit le code pour les méthodes des objets. Elle peut utiliser d'autres objets ou peut manipuler d'autres systèmes softwares qui ne sont pas eux-mêmes des objets.

Le client exécute une requête en ayant accès à une référence objet (ObjRef) pour un objet et en ayant connaissance du type de l'objet et de l'opération destinée à être exécutée.

Le client initie la requête en appelant des routines *stubs* spécifiques à l'objet (invocation statique) ou bien par construction dynamique de la requête (via la DII).

Les invocations statiques et dynamiques satisfont la même sémantique et donc le récepteur du message ne sait pas comment la requête a été construite. L'ORB localise le code d'implémentation approprié, transmet le contrôle de paramètres et de transfert à l'implémentation objet via un *skeleton* IDL. Les *skeletons* sont spécifiques à l'interface et à l'OA.

En exécutant la requête, l'implémentation objet peut obtenir certains services de l'ORB via l'OA.

Quand la requête est complétée, le contrôle et les valeurs de sortie sont retournés au client.

L'ORB interface est une interface aux fonctions de l'ORB qui sont indépendantes de l'OA utilisé. Ces opérations sont les mêmes pour tous les ORB et toutes les implémentations objet, et peuvent être exécutées soit par un

client, soit par une implémentation (Par exemple la conversion d'ObjRef en String).

B - Interface Definition Language

Développer des applications distribuées flexibles sur des plateformes hétérogènes nécessite une séparation stricte interface/implémentation(s). IDL aide à accomplir cette séparation.

En effet, IDL est un langage de définition d'interface orienté objet. Il définit les types des objets en spécifiant leurs interfaces. Une interface consiste en un jeu d'opérations et de paramètres pour ces opérations.

IDL est le moyen par lequel une implémentation d'un objet indique à ses clients potentiels quelles opérations sont disponibles et comment elles doivent être invoquées.

IDL a été conçu pour assurer la correspondance avec des langages de programmation (C,C++,SmallTalk sont actuellement standardisés).

Un compilateur IDL génère des *stubs* client et des *skeletons* serveurs. Ceux-ci automatisent les actions suivantes (en conjonction avec l'ORB):

- Les *factories* client (un factory est un objet créant un autre)
- Le codage/décodage des paramètres
- La génération des implémentations des classes d'interfaces
- L'enregistrement et l'activation des objets
- La localisation et les liens des objets

C - Dynamic Invocation Interface

L'interface d'invocation dynamique d'un ORB autorise la création et l'invocation dynamiques de requêtes. Un client utilisant cette interface pour envoyer une requête à un objet obtient la même sémantique qu'un client utilisant l'opération *stub* générée à partir de la spécification de type IDL.

Pour invoquer une opération sur un objet, un client doit faire appel, et être lié statiquement, au *stub* correspondant. Puisque le développeur détermine, à l'écriture du code, les *stubs* qu'un client contient, l'invocation statique ne peut pas accéder à de nouveaux objets qui ont été ajoutés au système plus tard. La DII fournit cette capacité. Cette interface permet à un client, à l'exécution, de :

- Découvrir de nouveaux objets
- Découvrir leurs interfaces
- Retrouver les définitions d'interfaces
- Construire et distribuer des invocations
- Recevoir les réponses

Et ceci de la part d'objets dont les *stubs* du client ne sont pas liés dans son module.

La DII est donc une interface de l'ORB qui comprend des routines autorisant le client et l'ORB, travaillant ensemble, à construire et invoquer des opérations sur tout objet disponible à l'exécution.

En essence, la DII est un *stub* générique côté client capable de faire suivre toute requête vers tout objet, cela en interprétant, à l'exécution, les paramètres des requêtes et les identifiants des opérations. Cependant, la flexibilité à l'exécution fournie par la DII peut se révéler coûteuse. Par exemple, Une requête à distance faite à travers une paire *stub/skeleton* générée par un compilateur peut être accomplie en une seule RPC; mais le même appel via la

DII nécessite des appels à :

- Object::getinterface: pour obtenir l'objet InterfaceDef
- InterfaceDef::describe: pour obtenir l'information sur les opérations supportées par l'objet
- Object::createrequest: pour créer la requête
- Request::addarg: pour chaque argument de la requête
- Request::invoke: pour invoquer effectivement la requête

Pour une opération sans argument et un type de retour *void,* la DII requiert un minimum de deux appels de fonctions, dont au moins une sera une RPC. En fait, pour la plupart des applications, particulièrement celles écrites dans un langage compilé comme C++, il est beaucoup plus efficace de passer les requêtes à travers les *stubs* statiques IDL.

D - Interface Repository

L'IR est le composant de l'ORB qui fournit un stockage persistant des définitions d'interfaces, il gère et permet l'accès à une collection de définitions d'objets spécifiés en IDL.

L'ORB peut utiliser les définitions d'objets contenues dans l'IR pour interpréter/manipuler les valeurs fournies lors d'une requête :

- Pour permettre la vérification du type des signatures des requêtes
- Pour aider à fournir l'interopérabilité entre différentes implémentations d'ORB

Comme l'interface vers les définitions d'objets maintenue dans une IR est publique, l'information maintenue dans l'IR peut aussi être utilisée par des clients et des services. Par exemple, l'IR peut être utilisé :

- Pour gérer l'installation et la distribution des définitions d'interfaces
- Pour fournir des composants dans un environnement CASE (*browser* d'interface)
- Pour fournir l'information interface aux compilateurs
- Pour fournir des composants dans des environnements *utilisateurs finaux* (un constructeur de menus)

L'IR utilise des modules pour grouper des interfaces et pour naviguer à travers ces groupes au moyen de leurs noms. Un module peut contenir :

- Des constantes
- Des définitions de types
- Des exceptions
- Des définitions d'interfaces
- D'autres modules

La spécification de CORBA pour l'IR définit seulement des opérations pour retrouver de l'information dans l'IR. Il peut y avoir plusieurs manières d'insérer de l'information dans l'IR (compiler des définitions IDL, construire des objets *repository* a travers la DII, copier des objets d'une IR à une autre...)

E - Basic Object Adapter

Un Object Adapter est l'interface principale pour une implémentation objet pour accéder aux services fournis par un ORB. On s'attend à ce qu'il y ait peu d'OA disponibles, avec des interfaces appropriées à des types spécifiques d'objets.

L'éventail important des granularités, temps de vie, politiques et styles d'implémentations d'un objet rend difficile pour l'ORB Core la fourniture d'une seule interface pratique et efficace pour tous les objets. Ainsi, à travers les OA, il est possible pour l'ORB de cibler des groupes particuliers d'implémentations obiet qui ont des besoins similaires.

Le BOA est une interface visant à supporter un large éventail d'implémentations objet. Il fournit les fonctions suivantes :

- Génération et interprétation des ObjRef
- Authentification du principal effectuant l'appel
- Activation/désactivation des objets individuels
- Activation/désactivation de l'implémentation
- Invocation des méthodes à travers des skeletons

Le BOA supporte des implémentations objet construites d'un ou plusieurs programmes. Il communique avec ces programmes en utilisant les facilités du système d'exploitation. Il nécessite donc des informations qui sont, de façon inhérente, non-portables.

Bien qu'il ne définisse pas cette information, le BOA définit le concept d'une Implementation Repository qui peut détenir cette information, autorisant chaque système à installer et démarrer des implémentations de manière appropriée à chaque système.

Le BOA est donc l'interface qui permet aux implémentations objet de pouvoir communiquer avec l'ORB et d'accéder à ses services (cf. Fig 2: Structure et scénario d'un BOA)

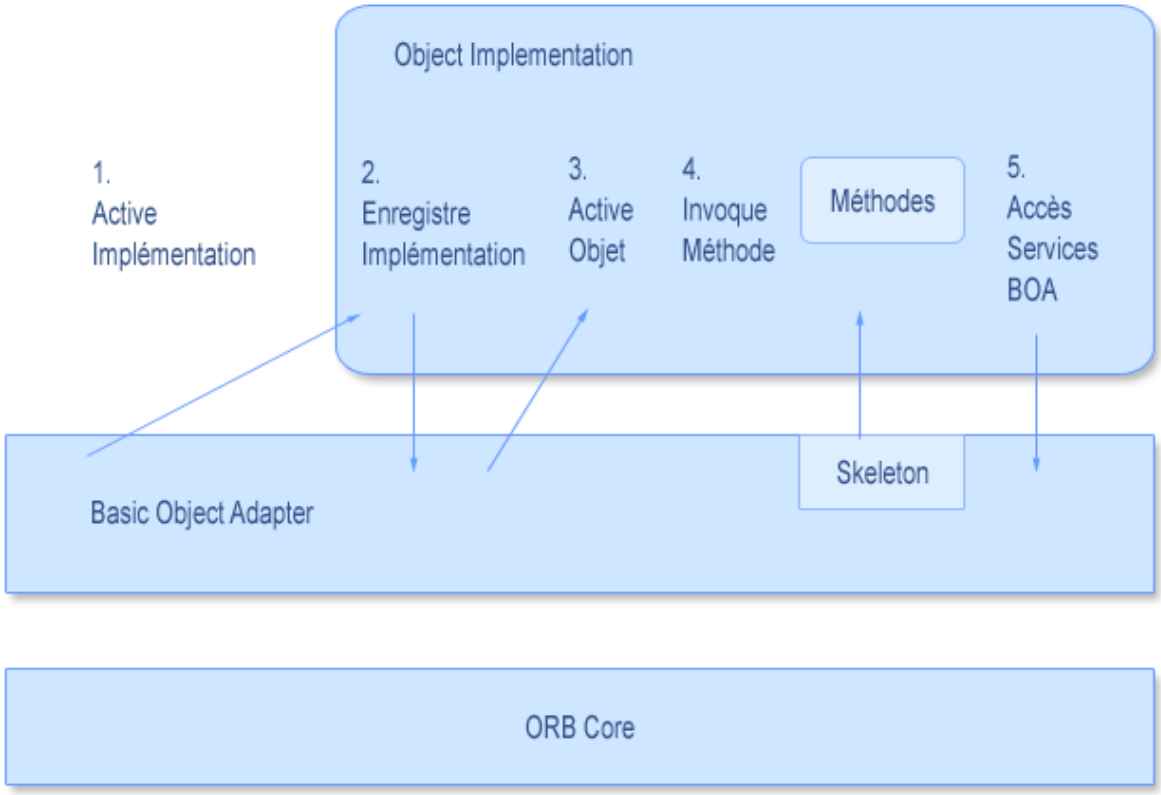

Fig 2

1 Le BOA démarre un programme pour fournir l'implémentation objet
2 L'implémentation objet notifie le BOA que son initialisation est terminée et qu'elle est prête à manipuler des requêtes.
3 Quand la première requête pour un objet particulier arrive, l'implémentation est avertie qu'elle doit activer l'objet.
4 Dans les requêtes suivantes, le BOA appelle la méthode appropriée en utilisant les *skeletons.*
5 A certains moments, l'implémentation objet peut accéder aux services du BOA tels que création d'un objet, désactivation...

III - Common Object Service Specification 1

Avant de détailler les services du COSS I à proprement dit, décrivons l'architecture générale d'un ObjectService. Un service est caractérisé par les interfaces qu'il fournit et par les objets qui fournissent ces interfaces. Un service peut impliquer un seul objet (par exemple un timer), plusieurs objets qui fournissent le même type d'interface (par exemple des objets *threads) ou* plusieurs objets qui fournissent des types d'interfaces distincts qui héritent d'un type d'interface de service (par exemple tout objet fournissant le service *Cycle de Vie).*

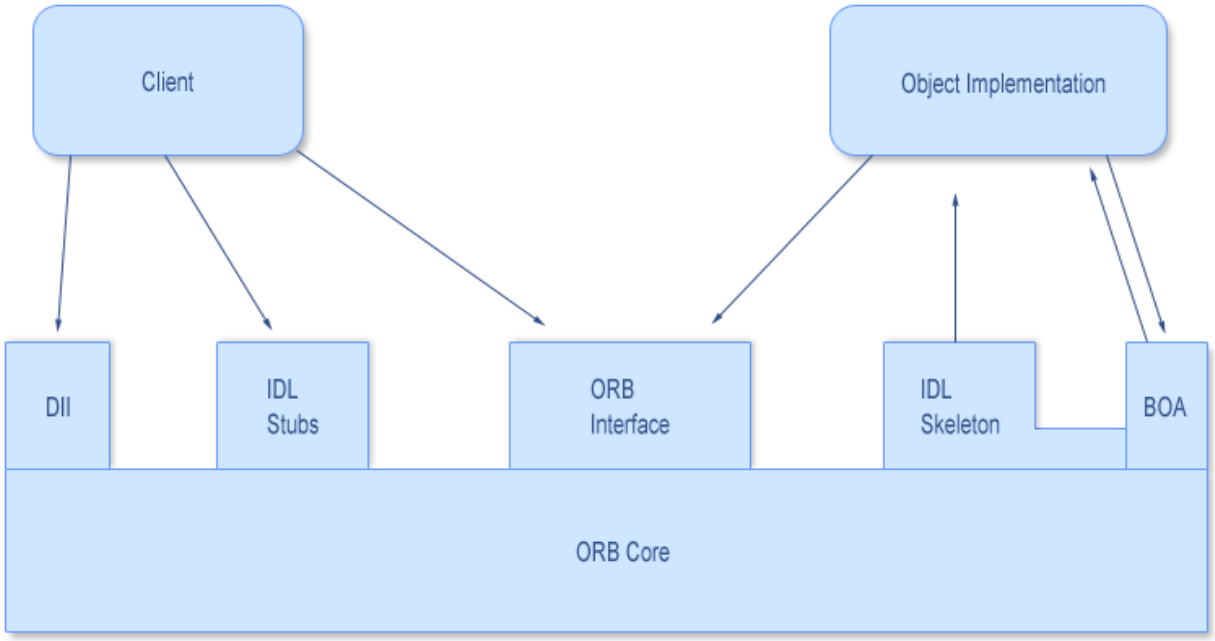

Fig 3

Chaque ObjectService fournit son service à un jeu d'utilisateurs qui sont typiquement des applications ou des *Common Facilities,* mais qui peuvent aussi être d'autres ObjectServices (cf. Fig 3).

A - Naming Service

Le service de nommage fournit le principal mécanisme à travers lequel la plupart des objets d'un système basé-ORB situent les objets qu'ils veulent utiliser. Pour cela, il utilise des noms. Un nom est une séquence ordonnée de composants. Chaque composant sauf le dernier nomme un contexte, le dernier dénote les objets liés par le nom. Un composant est composé de deux attributs :

- Attribut *identifier:* identifiant
- Attribut *kind:* type du composant

Un contexte est un jeu de liens noms-objets dans lequel chaque nom est unique.

Le service de nommage permet :

- De lier un nom à un objet, ceci dans un contexte de nommage
- De résoudre un nom, i.e., déterminer l'objet associé au nom dans un contexte donné

Les implémentations de ce service peuvent être spécifiques à une application ou bien basées sur une variété de systèmes de nommage actuellement disponibles, ceci grâce à la généralité du modèle utilisé et car les noms sont traités dans leur forme structurelle.

Puisque les valeurs des attributs des noms ne sont ni assignées ni interprétée par ce service, les logiciels de plus haut niveau n'ont pas de contraintes en terme de politique sur l'utilisation ou la gestion de ces valeurs.

Par l'utilisation de *bibliothèques de noms,* la manipulation des noms est simplifiée, et les noms peuvent être indépendants de la représentation, permettant ainsi à cette représentation d'évoluer sans requérir de changements du côté client.

La localisation d'applications est facilitée par l'indépendance des noms au niveau syntaxique et par la présence de l'attribut *kind.*

B - Event Service

Dans CORBA, l'invocation standard d'une méthode consiste en une exécution synchrone d'une opération fournie par un objet. Or, pour la plupart des applications, un modèle de communication plus découplé entre objets est nécessaire. L'OMG a donc défini un jeu d'interfaces *event service* qui permet la communication asynchrone entre objets. Le modèle de l'OMG est basé sur le paradigme *publish/subscribe.*

Le service évènement définit deux rôles pour les objets: producteur et consommateur. Les producteurs fournissent les données évènement, les consommateurs consomment ces données. La communication de ces données entre producteurs et consommateurs se fait par les requêtes CORBA standards. Les producteurs peuvent générer des évènements sans connaitre les identités des consommateurs. De même, les consommateurs peuvent recevoir des évènements sans connaitre les identités des producteurs.

Il existe deux approches pour initier la communication d'évènements :

- Le modèle *push:* il permet à un producteur d'initier le transfert des données évènements vers le consommateur. Le producteur a donc l'initiative.
- Le modèle *pull:* il permet à un consommateur de solliciter des données évènements d'un producteur. Le consommateur a donc l'initiative.

La communication elle-même peut être soit générique soit typée:

- Communication générique: toute communication se fait au moyen d'opérations *push* et *pull* génériques qui prennent un seul paramètre englobant toutes les données évènements.
- Communication typée: toute communication se fait via des opérations définies en IDL.

Ce service définit également des objets *event channel.* Un *event channel* est un objet qui permet à de multiples producteurs de communiquer avec de multiples consommateurs, et ce de manière asynchrone. Un *event channel* est à la fois un producteur et un consommateur d'évènements; c'est un objet CORBA standard et donc la communication avec un *event channel* se fait par les requêtes CORBA standards. De plus, l'interface *event channel* peut être sous-typée pour supporter des facultés étendues. Les interfaces producteur et consommateur étant symétriques, on peut chainer des *event channel* (pour supporter par exemple divers modèles de filtrage des évènements).

Ce service fournit des capacités basiques qui peuvent être configurées ensemble de façon flexible et puissante, il supporte :

- Evènements asynchrones (découplés en producteurs et consommateurs)
- Evènements fan-in
- Evènements fan-out
- Acheminement fiable d'évènements (par des implémentations appropriées *d'event channel.)*

Un serveur centralisé n'est pas nécessaire, ce service ne dépend d'aucun service global. De plus, les interfaces de ce service autorisent différentes qualités de service, pour différents niveaux de fiabilité et permettent des extensions futures pour des fonctionnalités additionnelles. Elles peuvent être implémentées et utilisées dans différents environnements (qu'ils supportent le *multithreading* ou pas). Les producteurs,consommateurs et *event channel* étant des objets, on peut tirer des avantages des optimisations de performance fournies par les implémentations des ORB pour les objets locaux ou éloignés.

C - LifeCycle Service

Le service cycle de vie définit les services et les conventions nécessaires à la création, la destruction, la copie et au déplacement des objets. Puisque les environnements basés sur CORBA supportent des objets distribués, ce service autorise les clients à exécuter ces opérations sur des objets situés dans différents endroits.

Le modèle client pour la création d'objets est défini en termes d'objets *factory.* Un *factory* est un objet qui crée un autre objet. Ce n'est pas un objet spécial, il a des interfaces IDL bien définies et des implémentations dans un langage de programmation. Il n'y a pas d'interface standard pour un *factory,* cependant ce service définit également une interface pour un *factory* générique, ceci permet la définition de services de création standard. Les opérations *remove, copy* et *move* sont définies dans une interface *LifeCycleObject.*

- *Creation:* La plupart des paramètres fournis à un opérateur *create* seront dépendants de l'implémentation, une signature IDL standardisée et universelle pour la création n'est donc pas possible. Les signatures IDL pour la création seront définies pour différentes sortes d'objets *factory* mais elles seront spécifiques au type, à l'implémentation et au mécanisme de stockage persistant de l'objet devant être crée.
- *Deletion:* Un opérateur *remove* est défini pour tout objet supportant l'interface *LifeCycleObject.* Ce modèle pour la suppression supporte tout paradigme pour garantir l'intégrité référentielle. Le service décrit un support pour les deux paradigmes les plus communs, basés sur les relations de référence et de contenu. Un seul type de suppression est autorisé, une opération différente devra être utilisée pour archiver un objet.