CORBA : principes et mécanismes

Développement d'applications distribuées

(par Z. Mammeri – UPS)

1. Objectifs et principes de base de CORBA

1.1. CORBA : bus pour l'interopérabilité

CORBA (Common Object Request Broker Architecture) est une architecture proposée et maintenue par l'OMG[1] (Object Management Group) pour permettre l'intégration d'une grande variété de systèmes **hétérogènes** distribués orientés objet. CORBA est la spécification d'architecture la plus répandue pour développer des systèmes distribués.

La première version de CORBA (version 1.1) date de 1992. CORBA a toujours été en pleine évolution pour prendre en compte de plus en plus de besoins d'utilisateurs. La version actuelle, CORBA 3.0, a été adoptée fin 2002 et éditée en 2004.

CORBA est conçue pour supporter des applications distribuées orientées **objet** et développées selon le modèle **client-serveur**. On dit aussi que CORBA est un bus **réparti** (à ne pas confondre évidemment avec la notion de bus physique utilisée dans les réseaux de transmission).

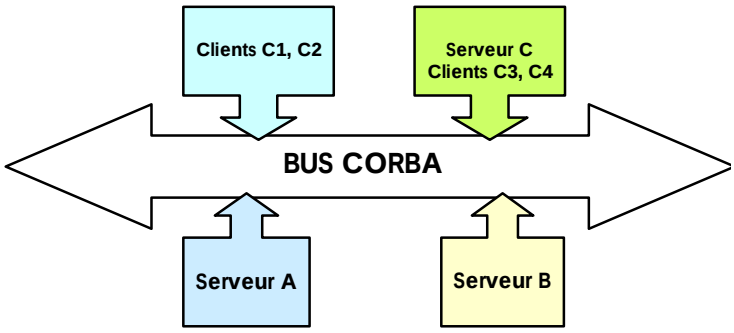

Figure 1. *Bus CORBA*

CORBA a été conçue pour développer des applications de la façon la plus simple possible, même si développer des applications distribuées n'est pas toujours facile. **Transparence**, **portabilité** et **interopérabilité** sont les principes de base de la spécification de CORBA. Quand le client fait une requête, il ne voit qu'un appel local, tous les mécanismes impliqués dans l'appel sont transparents au client. Toute machine, abritant un ORB (Object Request Broker), doit pouvoir exécuter des applications CORBA. Les applications doivent être portables sur des machines avec différentes architectures. Une application CORBA pour un ORB A doit pouvoir s'exécuter aussi sur un ORB B sans trop ou aucune modification. Les différents ORBs doivent donc inter-opérer sans nécessité de modifier les applications.

[1] L'OMG est un consortium créé en 1989. Il compte actuellement autour de 1000 membres (essentiellement des entreprises et laboratoires) qui travaillent sur les technologies orientées objet et qui interviennent dans la spécification et les extensions de CORBA, UML, CWM ('Common Warehouse Metamodel') et MDA ('Model Driven Architecture').

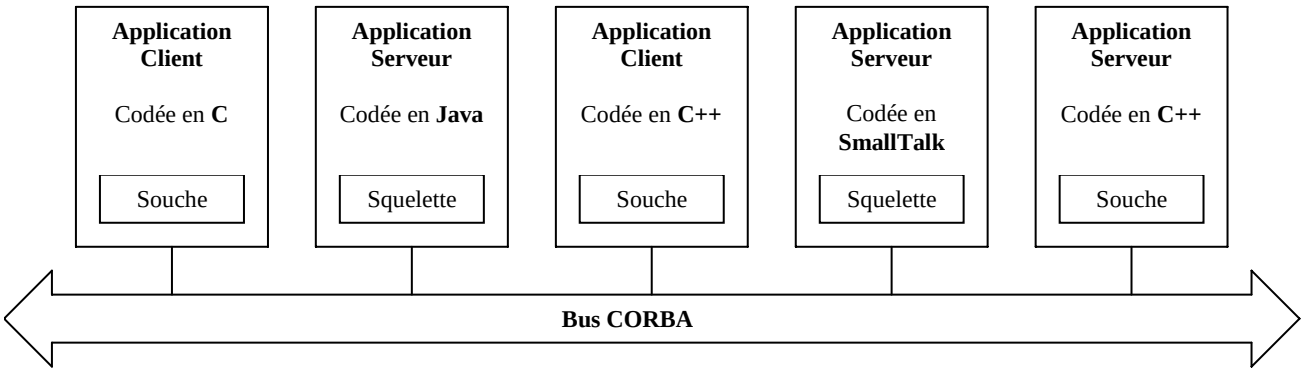

Figure 2. *Hétérogénéité avec CORBA*

Un des avantages les plus importants de la spécification CORBA est de faciliter le développement des applications distribuées en évitant au développeur des difficultés concernant les mécanismes de communication, la location des applications, les clients et serveurs, le protocole de communication à utiliser ou le format de message à utiliser.

Dans CORBA, le **client** est l'entité qui veut exécuter une opération sur l'objet. L'**implémentation d'objet** est le code qui implémente l'objet. L'ORB gère tous les mécanismes nécessaires pour trouver l'implémentation de l'objet demandé. L'ORB a pour fonction de préparer l'implémentation d'objet afin de recevoir la requête et de communiquer les données qui la constituent.

1.2. Déroulement de requête sous CORBA

Le **client n'a pas besoin de connaître :**
- la localisation de l'objet invoqué,
- le langage avec lequel a été programmé l'objet,
- le système d'exploitation sur lequel est implanté l'objet,
- la façon dont l'objet est implanté par le serveur.

Le client doit connaître seulement l'**interface** d'accès à l'objet et la **référence** de l'objet.

Les interactions entre les applications sont matérialisées par des **invocations**, à distance ou en local, des **méthodes** (ou **opérations**) des objets.

La notion de client/serveur intervient uniquement lors de l'utilisation d'un objet. Une même application peut être tantôt client tantôt serveur.

Pour pouvoir faire une requête, le client peut utiliser l'interface d'invocation dynamique ou un *stub* (**souche**) produit pendant la compilation d'un script IDL. L'implémentation de l'objet et le client peuvent dialoguer avec l'ORB à travers les interfaces de l'ORB.

L'implémentation de l'objet peut recevoir les requêtes, soit par les squelettes générés par IDL, soit par les squelettes dynamiques, mais elle peut aussi dialoguer avec l'ORB à travers l'adaptateur d'objets.

Les interfaces pour les objets peuvent être définies de deux façons :

- **Statique** : en utilisant le langage de définition d'interfaces (IDL). Le langage IDL permet de définir les opérations sur les objets et les paramètres des opérations.
- **Dynamique** : où les interfaces peuvent être stockées dans un dépôt d'interfaces. Les composantes des interfaces sont représentées comme objets afin de permettre leur accès pendant l'exécution.

Pour pouvoir faire une requête, le client doit connaître la référence de l'objet, le type et l'opération à exécuter. Une fois cette information obtenue, le client peut initialiser la requête. La requête se fait en appelant les routines représentées par les stubs ou en les construisant de manière dynamique. Un stub est spécifique à un objet.

Quand la requête a quitté le client, l'ORB est le responsable de la localisation de l'implémentation de l'objet, de la transmission des paramètres et du contrôle de l'implémentation d'objet. Le transfert d'information se réalise au travers des squelettes IDL ou des squelettes dynamiques. Les squelettes sont spécifiques à l'interface et à l'adaptateur d'objets. Si au moment d'exécuter la requête, l'implémentation de l'objet a besoin des services de l'ORB, elle peut communiquer avec l'ORB à travers l'Adaptateur d'Objets. Une fois la requête satisfaite le contrôle et les résultats sont retournés au client.

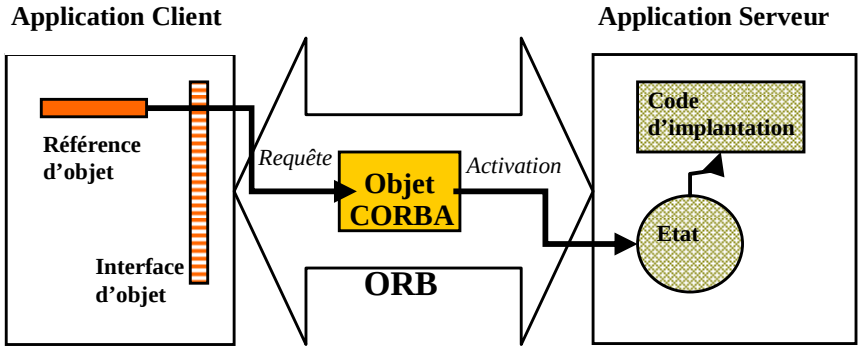

Figure 3. *Principe simplifié requête via CORBA*

Ce que nous résumons ici est une présentation générale de CORBA, la spécification complète de CORBA comprend à peu près 1200 pages et la spécification des services d'objets comprend plusieurs centaines de pages, ce chapitre ne tente de présenter que les principales composantes de CORBA.

Définitions à retenir

- ***Application client*** : programme qui invoque des objets via le bus CORBA.
- ***Référence d'objet*** : une structure de données désignant l'objet CORBA et contenant l'information nécessaire pour le localiser sur le bus.
- ***Interface de l'objet*** : type de l'objet CORBA définissant ses opérations et attributs.
- ***Requête*** : mécanisme d'appel d'une opération (ou méthode) ou d'accès à un attribut d'un objet.
- ***Bus CORBA*** : il achemine les requêtes de l'application client vers l'objet en masquant tous les problèmes d'implantation.
- ***Objet CORBA*** : c'est le composant cible.
- ***Implantation de l'objet*** : entité codant l'objet CORBA à un instant donné et gérant un état de l'objet. Au cours du temps, un même objet peut avoir plusieurs implantations.
- ***Application serveur*** : c'est la structure d'accueil des objets d'implantation et des exécutions des opérations. En général, l'application serveur est un processus cyclique.

1.3. Composants de l'architecture CORBA

Les principaux composants (obligatoires ou optionnels) de CORBA sont résumés par la figure 4.

- *Souches (stubs) :* permettent au client de faire des requêtes (mais ces requêtes doivent être connues avant la compilation). Lors d'un appel de méthode, le stub associé permet de préparer les paramètres d'entrée et de décoder les paramètres de sortie et résultat.
- *Interface d'invocation statique (SII : static invocation interface)* : interface pour les invocations statiques (c'est-à-dire des requêtes connues à la compilation).
- *Interface dynamique d'invocation* (DII : *dynamic invocation interface)* : interface pour les invocations dynamiques (c'est-à-dire des requêtes construites seulement à l'exécution).
- *Interface de l'ORB* : permet l'accès aux services de l'ORB par les clients et serveurs.
- *Squelettes statiques* : permettent aux objets de recevoir les requêtes définies avant la compilation. Lors d'un appel de méthode, le squelette associé décode les paramètres d'entrée et prépare les paramètres de sortie et résultat.
- *Interface de squelette dynamique (DSI : dynamic Skeleton Interface)* : permet de gérer dynamiquement les requêtes pour lesquelles des interfaces statiques n'ont pas été spécifiées.
- *POA (Portable Object Adaptor)* : c'est le composant qui s'occupe de créer les objets, de maintenir les associations entre objets et implantations et de réaliser l'activation des objets.
- *ORB* : c'est le noyau de CORBA pour le transport des requêtes et des réponses. Il Intègre, au minimum, les protocoles d'interopérabilité GIOP (General Inter-ORB Protocol) et IIOP (*Internet Inter-ORB Protocol*).

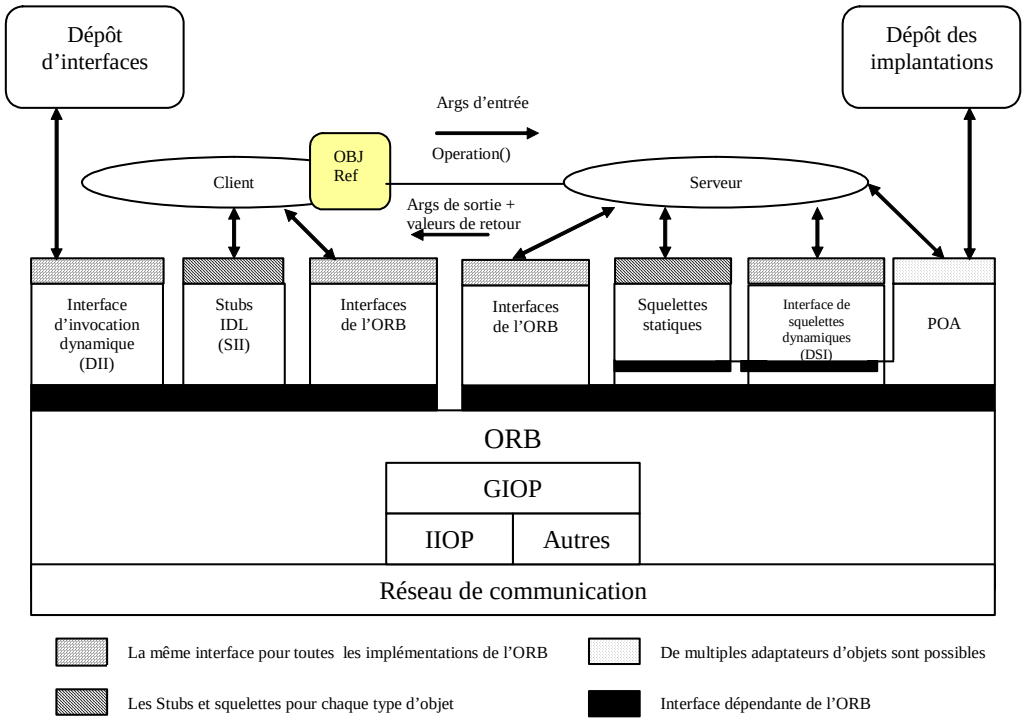

Figure 4. *Structure de CORBA*

- *Dépôt d'interfaces ("Interface repository")* : contient la description des interfaces IDL accessibles aux applications à l'exécution.
- *Dépôt d'implantations ("implementation repository")* : contient la description les implantations d'objet.
- *Référence d'objet* : à chaque objet est associé une référence (chaîne de bits) permettant de faire le lien entre l'objet et son nom. Un même objet peut être associé à plusieurs références dans le temps. A tout instant, une référence ne correspond qu'à un seul objet.

Comme le montre la figure 5, les services offerts par le bus CORBA sont généralement regroupés en plusieurs classes :

1) Services communs (appelés **`CORBAservices`**) : ils fournissent les fonctions nécessaires à la plupart des applications. Les principaux services communs définis actuellement sont :
   - annuaires (pour le nommage),
   - cycle de vie des objets,
   - relations entre objets,
   - événements,
   - transactions,
   - sécurité,
   - persistance.

2) Utilitaires communs (appelés **`CORBAfacilities`**) : ce sont des canevas d'objets qui répondent à des besoins des utilisateurs. Ils incluent les outils d'administration, IHM…

3) Les interfaces de domaine (appelés **`CORBA Interfaces`**) : elles définissent des objets concernant des métiers spécifiques à des domaines d'activités comme la finance, santé, télécommunications…

4) Les objets applicatifs : ils sont spécifiques à chaque application et ne peuvent pas être standardisés a priori.

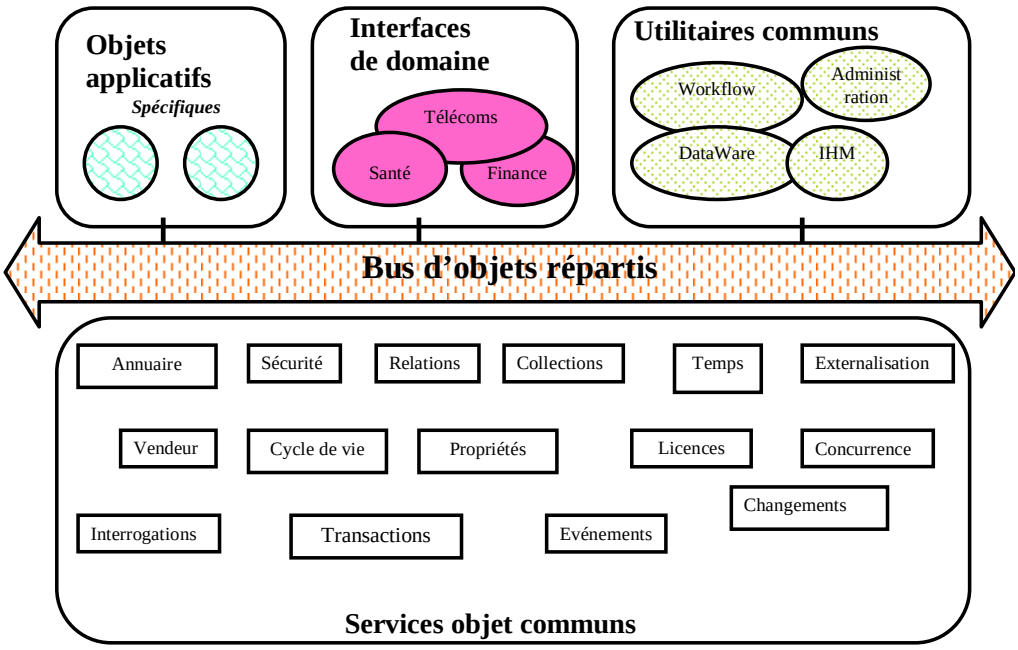

Figure 5. *Aperçu général des services de CORBA*

1.4. Service de recherche d'objets

Exporter/importer des références d'objet entre programmes est fastidieux. Saisir des références à partir de clavier est fastidieux aussi. Pour faciliter, la création et recherche de référence, on fait appel au service de nommage (qui constitue une sorte de DNS). En effet, parmi les services mentionnés dans le paragraphe précédent, le service **Nommage** (dit aussi **annuaire**) joue un rôle important pour les applications CORBA. Ce service permet de gérer de manière dynamique les associations entre les noms d'objet et les références d'objet. Le fonctionnement est en trois phases :

1) Les applications qui fournissent des objets enregistrent les références de leurs objets auprès du service de nommage en leur associant, en général, des noms symboliques.

2) Les clients interrogent le service pour obtenir les références des objets qu'ils veulent invoquer.

3) Le service de nommage recherche dans l'ensemble des références enregistrées celles qui répondent aux critères d'interrogations fournis par les applications clientes.

Le service de Nommage est défini dans le module **`CosNaming`**. L'espace de désignation symbolique des objets est structuré par un graphe (arbre) de **contextes** de nommage. L'annuaire a une racine et des répertoires, appelés contextes de nommage. Chaque contexte maintient une liste d'associations des noms symboliques et des références d'objet. Les feuilles de l'arbre de nommage contiennent les références d'objet.

Les opérations suivantes sont définies par le service de Nommage :

- `bind` : rajouter une association entre un nom et une référence.
- `rebind` : mettre à jour une association.
- `unbind` : détruire une association.
- `resolve` : rechercher une référence désignée par un chemin (un chemin est une concaténation de noms).
- `new_context` : création d'un nouveau contexte.
- `destroy` : détruire un contexte.

Obtention du service de Nommage

En Java, pour obtenir la référence du service de Nommage, on utilise le module `org.omg.CORBA` et l'opération **`resolve_initial_references`** comme suit :

```
org.omg.CORBA.Object objRef =
       orb.resolve_initial_references("NameService");
```

Ensuite, pour obtenir le nom du contexte associé à la référence d'objet que l'on vient d'obtenir, on utilise l'opération **`narrow`** :

```
org.omg.CosNaming.NamingContext nsRef =
       org.omg.CosNaming.CosNamingContextHelper.narrow(objRef);
```

Enregistrement d'une référence d'objet

Toute application Serveur doit diffuser aux applications Clients les références de ses objets pour qu'ils puissent les invoquer. Pour cela, il faut :

1) obtenir une référence du service de Nommage,
2) créer un chemin valide
3) associer à ce chemin la référence d'objet via l'opération **`bind`**.

En Java, l'enregistrement d'une référence d'objet se fait comme suit :

```
// 1) obtenir une référence du service de Nommage
org.omg.CORBA.Object objRef =
       orb.resolve_initial_references("NameService");
org.omg.CosNaming.NamingContext nsRef =
       org.omg.CosNaming.CosNamingContextHelper.narrow(objRef);

 // 2) créer un chemin valide :
 // Un nom de composant est un doublon (identificateur, qualificatif).
 org.omg.CosNaming.NameComponent[] nsNom =
      new org.omg.CosNaming.NameComponent[1];
 nsNom[0] = new org.omg.CosNaming.NameComponent["Repertoire", ""];

 // 3) création de la liaison
 nsRef.bind(nsNom, repertoire);
```

Recherche d'une référence d'objet

L'application Client doit obtenir la référence de tout objet avant d'invoquer des opérations sur cet objet. Il y a deux manières d'y parvenir : soit utiliser l'opération **`string_to_object`**, soit utiliser le service de Nommage. Dans le cas où le service de Nommage est utilisé, la recherche de référence s'effectue comme suit :

1) obtenir une référence du service de Nommage,
2) créer un chemin valide,
3) rechercher la référence associée à ce chemin l'opération **`resolve`**.

En Java, la recherche d'une référence d'objet se fait comme suit :

```
// 1) obtenir une référence du service de Nommage
org.omg.CORBA.Object objRef =
       orb.resolve_initial_references("NameService");
org.omg.CosNaming.NamingContext nsRef =
       org.omg.CosNaming.CosNamingContextHelper.narrow(objRef);

 // 2) créer un chemin valide
 org.omg.CosNaming.NameComponent[] nsNom =
       new org.omg.CosNaming.NameComponent[1];
 nsNom[0] = new org.omg.CosNaming.NameComponent["Repertoire", ""];

 // 3) obtenir la référence associée au chemin
 org.omg.CORBA.Object objRef = nsRef.resolve(nsNom);
```

2. Langage de Définition d'Interfaces IDL

2.1. IDL dans le processus de développement d'application

IDL (Interface Definition Language) est le langage utilisé pour décrire les interfaces que les clients vont appeler et que les implémentations d'objets vont fournir. Les interfaces définies en IDL fournissent toute l'information nécessaire pour pouvoir implémenter les clients. Les clients et les objets sont implémentés avec un langage de programmation, comme C++ et non avec IDL. IDL est uniquement un langage descriptif. Les scripts en IDL doivent être projetés sur un langage de programmation (tels que C++, Java…) pour lequel la projection d'IDL a été définie, voir la figure 6. Les fichiers IDL doivent avoir l'extension '`.idl`'.

Le langage IDL permet d'exprimer, sous forme de **contrats**, la coopération entre les clients et serveurs de services, en séparant les interfaces des implantations et en masquant les divers problèmes liés à l'interopérabilité, hétérogénéité et localisation de ceux-ci.

Un contrat IDL spécifie les types manipulés par un ensemble de processus répartis. Les contrats IDL sont projetés en **souches** IDL dans l'environnement de programmation du client et en **squelette**s dans l'environnement de programmation du serveur.

Le client invoque localement les souches pour accéder aux objets. Les souches construisent des requêtes, qui sont transportées par le bus CORBA, puis délivrées aux squelettes qui les délègueront aux objets concernés.

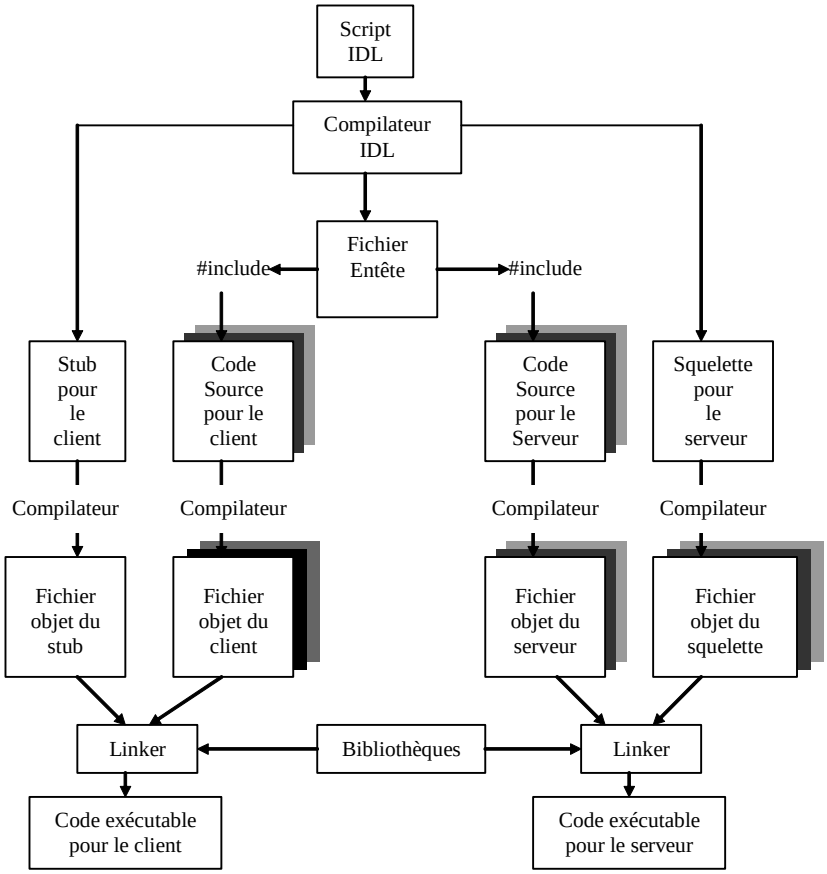

Figure 6. *Processus de compilation pour l'exécution d'application avec CORBA*

2.2. Langage IDL

IDL a été conçu de façon à être simple. Un script IDL est un ensemble de définitions d'un ou plusieurs types de données, de constantes, d'exceptions et ou de modules.

De façon générale, les définitions IDL ont la structure suivante (en notation BNF) :

```
<specification> ::= <importation>* <définition>+
<définition> ::=     <module> ";"
                   | <interface> ";"
                   | <declaration_constante> ";"
                   | <declaration_type> ";"
                   | <declaration_exception> ";"
                   | <autres_déclarations> ";"
```

Nous n'allons pas ici décrire toute la syntaxe de IDL, mais seulement les éléments les plus utilisés.

a) Module

Un module IDL permet de regrouper un ensemble de définitions afin d'inclure ces règles dans différentes spécifications sans avoir à les réécrire à chaque fois.

```
<module> ::= "module" <nom_module> "{"<définition>+ "}"
```

Tous les noms définis par la spécification CORBA sont contenus dans un module appelé `CORBA` stocké dans le fichier `orb.idl`. Ce fichier doit être inclus dans toute spécification en IDL qui utilise des noms CORBA.

Exemple :

```
#include <orb.idl>
module M1 {
      typedef ...; // définition de type dans le module
}
```

b) Interface

La notion d'interface est fondamentale en IDL : elle permet de spécifier notamment les signatures (paramètres et leurs types) des opérations (méthodes) appelées.

Une interface est définie par un identificateur (c'est-à-dire un nom) et un corps. Une interface peut hériter des éléments d'autres interfaces.

Le corps de l'interface décrit notamment les définitions de types, de constantes, les attributs et les opérations que l'interface exporte.

Une interface se déclare avec la syntaxe suivante :

```
<interface> ::= <interface_dcl> | <forward_dcl>

<interface_dcl> ::= <entete_interface> "{" <corps_interface> "}"

<entete_interface> ::= ["abstract"|"local"] "interface" <identificateur>
            [ <héritage_interface> ]

<corps_interface> ::= <export>*

<export> ::=  <declaration_type> ";"
            | <declaration_constante> ";"
            | <declaration_exception> ";"
            | <declaration_attribut> ";"
            | <declaration_operation> ";"
            | <autres_declarations> ";"

<heritage_interface> ::= ":" <nom_interface> { "," <nom_interface> }*

<nom_interface> ::=  <identificateur> | "::" <identificateur>
                   | <identificateur> "::" <identificateur>
```

Héritage : une interface peut être dérivée d'une autre interface. Une interface peut donc définir ses proposes éléments et hériter des éléments d'autres interfaces. Dans IDL, on peut faire de l'héritage simple ou multiple.

Dans l'exemple suivant, l'interface B hérite le type L1 de l'interface A et l'interface C hérite les type L1 et L2 et les opérations opA et opB des interfaces A et B :

```
Interface A {typedef long L1;
             short opA(in L1 i1);} ;
Interface B: A {typedef short L2;
                   L1 opB (in ...} ;
Interface C : A, B {
                   L1 var1 ;
                  L2 var2 ;} ;
```

Comme le montre l'exemple suivant, en cas d'héritage multiple, il faut indiquer le nom de l'interface (suivi de '**::**') dont on hérite un élément si ce même élément est défini aussi dans d'autres interfaces mentionnées dans l'héritage. Dans l'exemple suivant, le type string_t est défini dans les deux interfaces A et B mais de manière différente.

```
interface A {
       typedef string<128> string_t;
       };
interface B {
       typedef string<256> string_t;
      };

interface C: A, B {
       attribute A::string_t Titre;
       attribute A::string_t Nom;
       attribute B::string_t Ville;
};
```

Avec le mot clé **local**, on définit des interfaces locales pour lesquelles il y a des restrictions d'utilisation.

c) Types de base

Les types suivants sont prédéfinis en IDL :

```
<type_basique> ::=   <type_entier>   | <type_reel>     | <type_char>
                   | <type_boolean>  | <type_any>      | <type_string>

     <type_reel> ::= "float" | "double" | "long" "double"
     <type_entier> ::= ["unsigned"] <entier_non_signé>
     <entier_non_signé> ::= "short" | "long" | "long" "long"
     <type_char> ::= "char"
     <type_boolean> ::= "boolean"
     <type_octet> ::= "octet"
     <type_string> ::= "string" ["<" <nombreMaxi_caractères>">"]
     <type_any> ::= "any"
```

Les valeurs des entiers sont définies comme suit :

short	$-2^{15} .. 2^{15} - 1$
long	$-2^{31} .. 2^{31} - 1$
long long	$-2^{63} .. 2^{63} - 1$
unsigned short	$0 .. 2^{16} - 1$
unsigned long	$0 .. 2^{32} - 1$
unsigned long long	$0 .. 2^{64} - 1$

d) Types construits

Par types construits, on entend : les structures, unions et énumérations.

Le type **union** d'IDL est une sorte de croisement de **union** et **switch** du langage C. L'entête d'un type **union** doit contenir un type de sélecteur qui permet de désigner le type d'élément à utiliser dans chaque instance du type.

```
<type_construit> ::= <type_struct> | <type_union> | <type_enum>

<type_struct> ::= "struct" <identificateur> "{" <liste_element>"}"

<type_union> ::= "union" <identificateur> "switch" "(" <type_selecteur>")"
            "{" <corps_selection> "}"
<corps_selection> ::= <cas>*
<cas> ::= { "case" <expression_constante> | "default" } " :"
            <type> <identificateur>

<type_enum> ::= "enum" <identificateur> "{" <liste_litteraux>"}"
```

Exemple :

```
struct enfant {
      string<40> nom ;
      string<40> prenom ;
      unsigned short age ;};

enum situation_fam { AvecEnfants, SansEnfants };

union Personne switch(situation_fam) {
      case AvecEnfants:
            string<40> nom ;
            string<40> prenom ;
            unsigned short nombre_enfants;
            sequence<enfant> LesEnfants;
      case SansEnfants:
            string<40> nom ;
            string<40> prenom ; };
```

Le type **TypeCode** est un méta-type qui peut correspondre à n'importe quel type. A chaque type IDL est associé un **TypeCode** qui le décrit. Le type **any** permet de stocker n'importe quelle valeur IDL.

e) Déclaration de constante

```
<const_dcl> ::= "const" <type_const> <identificateur> "=" <expr_const>

<type_const> ::= <integer_type>
       | <type_char> | <type_ boolean> | <type_reel>
       | <type_string> | <type_octet>
```

Voici quelques exemples de déclarations de constantes :

```
const short s = 655592;
const octet o = -54;
const long L1 = 3;
const long L2 = 5 + L1;
```

```
enum Color { red, green, blue };
const Color FAVORITE_COLOR = red;
module M1 {
        enum Size { small, medium, large };
      };
const M1::Size MYSIZE = M1::medium;
```

f) Déclaration de types

En plus des types prédéfinis, IDL permet la définition de types appropriés à une application particulière ou en vue d'être utilisés par plusieurs applications qui partagent certains aspects communs.

```
<declaration_type> ::= "typedef" <type_spec> <identificateur>
             | <type_struct> | <type_union>
             | <type_enum> | <autres_déclaration_type>
<type_spec> ::= <type_basique> | <type_construit>
```

On peut aussi déclarer des séquences et des tableaux multi dimensionnels.

Une séquence est un tableau à une dimension avec deux caractéristiques : une taille maximale (définie au moment de la compilation) et une taille courante définie au cours de l'exécution.

Un tableau peut avoir éventuellement plusieurs dimensions. La taille de chaque dimension doit être définie au moment de la compilation.

```
<declaration_sequence> ::= "sequence" "<" <type_element>
      [ "," <constante> ] ">"

<declaration_tableau> ::= <identificateur> {"["<dimension_tab> "]"}+
```

Exemple

```
typedef sequence<enfant> S1_enfants ; // déclare un type séquence de taille
                          // infinie d'enfants
typedef sequence<enfant, 10> S2_enfants ; // déclare un type séquence de 10
                          // enfants maximum.
```

g) Déclaration des opérations

Une **opération** se définit par une signature qui comprend le nom de l'opération, le type du résultat, la liste des paramètres et la liste des exceptions éventuellement déclenchées lors de l'invocation. Un paramètre se caractérise par un mode de passage, un type et un nom formel. Les modes de passages autorisés sont **`in`**, **`out`** et **`inout`**. Le résultat et les paramètres peuvent être de n'importe quel type exprimable en IDL. Par défaut, l'invocation d'une opération est **synchrone** (c'est-à-dire bloquant). Cependant, il est possible de spécifier qu'une opération est asynchrone (**`oneway`**), c'est-à-dire que le résultat est de type **`void`**, que tous les paramètres sont en mode `in` et qu'aucune exception ne peut être déclenchée. Malheureusement, CORBA ne spécifie pas la sémantique d'exécution d'une opération **`oneway`** : l'invocation peut échouer, être exécutée plusieurs fois sans que l'appelant ou l'appelé puissent en être avertis. Toutefois, dans la majorité des implantations CORBA, l'invocation d'une telle opération équivaut à un envoi fiabilisé de messages. De plus, l'OMG a proposé une spécification de service pour les communications asynchrones (il s'agit du service *CORBA Messaging*).

```
<declaration_operation> ::= ["oneway"] { <type_resultat> | "void" }
                  <identificateur>
                  { "(" ")" | "(" <parametre_spec>+ ")" }
                  [raises <nom>+ ]
```

```
<parametre_spec> ::= { “in” | “out” | “inout” } <type_param> <nom_param>
```

Exemple

```
interface I1 {
 unsigned short obtenir_nombre_articles ();
 float obtenir_prix_article (in long num_article) ;
 void operX (in T1 a, out T2 b, in/out T3 c) ;
}

interface AnnuairePersonnel {
 exception existeDeja {T_Nom nom);} ;
 exceptiion inconnu {T_Nom nom);} ;
 AjouterPersonne (in T_Personne personne) raises (existeDeja) ;
 T_Personne ObtenirInfoPersonne (in T_Nom nom) raises (inconnu) ;
 Void ModifierPersonne (in T_Nom nom, in T_Personne personne)
                        raises (inconnu) ;
}
```

h) Déclaration d’attributs

En plus des opérations (que l’on peut invoquer), une interface peut définir des attributs (des variables) accessibles (c’est-à-dire qui peuvent être lus ou écrits) par un appelant. Le mot clé **`readonly`** signifie que l’attribut est accessible en lecture seulement. Il faut noter que l’on peut se passer des attributs en définissant des variables internes auxquelles on associe des fonctions de lecture et écriture (voir exemple ci-dessous).

```
<declaration_attribut> ::= [“readonly”] “attribute” <type> <identificateur>
```

Exemple

```
interface Interf1 {
 enum T_materiau { ruban, glace };
       struct T_position {float x, y; };
 attribute float radius;
 attribute T_materiau materiau ;
 readonly attribute T_position position ;
};
```

La définition de `Interf1` est équivalente à celle de `Interf2` ci-dessous.

```
interface Interf2 {
 enum T_materiau { ruban, glace };
       struct T_position {float x, y; };
 float radius;
 T_materiau materiau ;
 T_position position ;
 float _get_radius();
 void _set_radius(in float r);
 T_materiau _get_materiau() ;
 void _set_materiau(in T_materiau m) ;
 T_position _get_position() ;
};
```

i) Remarques sur la syntaxe

1. Attention : IDL n’est pas sensible à la case (par exemple, les trois écritures d’identificateurs suivantes sont considérées comme indiquant le même identificateur : `AABB`, `aabb` et `AaBb`).

2. Les mots suivants sont des mots clés de DL et ne peuvent pas être utilisés comme identificateurs :

```
Abstract    exception   inout       provides    truncatable
Any         emits       interface   public      typedef
attribute   enum        local       publishes   typeid
boolean     eventtype   long        raises      typeprefix
case        factory     module      readonly    unsigned
char        FALSE       multiple    setraises   union
component   finder      native      sequence    uses
const       fixed       object      short       ValueBase
consumes    float       octet       string      valuetype
context     getraises   oneway      struct      void
custom      home        out         supports    wchar
default     import      primarykey  switch      wstring
double      in          private     TRUE
```

3. Une ligne commençant par # (par exemple `#include`) est une directive et nécessite un prétraitement.

4. Une ligne commençant par `//` indique un commentaire. Plusieurs lignes commençant par `/*` et se terminant par `*/` indiquent un texte de commentaire.

2.3. Projection de IDL vers un langage de programmation

La figure 7 résume les types IDL que l'on peut utiliser pour définir des données, des interfaces, des constantes ou des types.

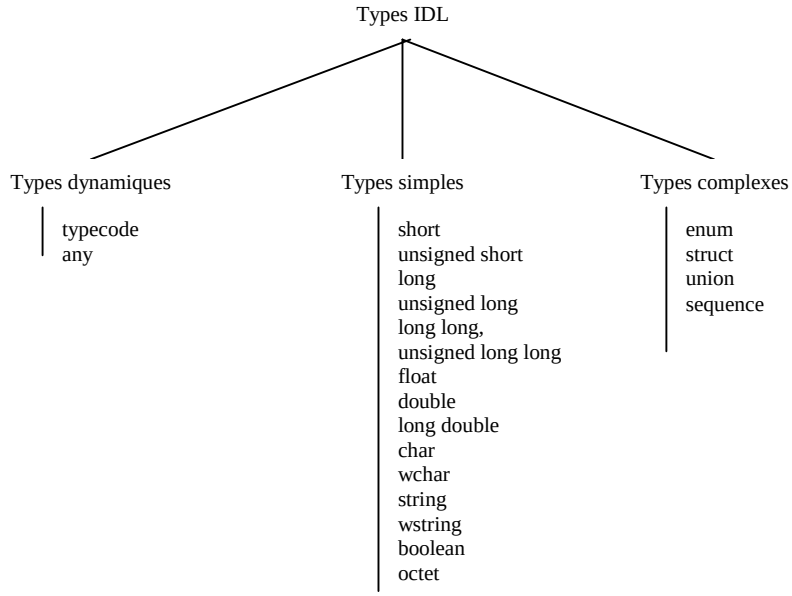

Figure 7. *Types de données IDL*

Les types IDL doivent être projetés sur les types des langages de programmation utilisés pour coder les programmes du client et serveur. Pour permettre la portabilité des applications, les règles de projection d'IDL vers les langages de programmation sont normalisées et fixent précisément la traduction de chaque type IDL en des constructions du langage cible. Actuellement les règles de traduction définies par l'OMG concernent les langages : C, C++, Java, Ada, SmallTalk et Cobol.

Comme le montre la figure 8, la projection est réalisée par un pré-compilateur IDL dépendant du langage de programmation cible et de l'implantation de l'ORB. Ainsi, chaque produit CORBA fournit son pré-compilateur IDL pour chacun des langages qu'il supporte. Le code des applications est alors portable d'un bus à un autre car les souches et squelettes générés s'utilisent toujours de la même manière quel que soit le

produit CORBA. Par contre, le code des souches et squelettes n'est pas toujours portable car il dépend de l'implantation de CORBA utilisée.

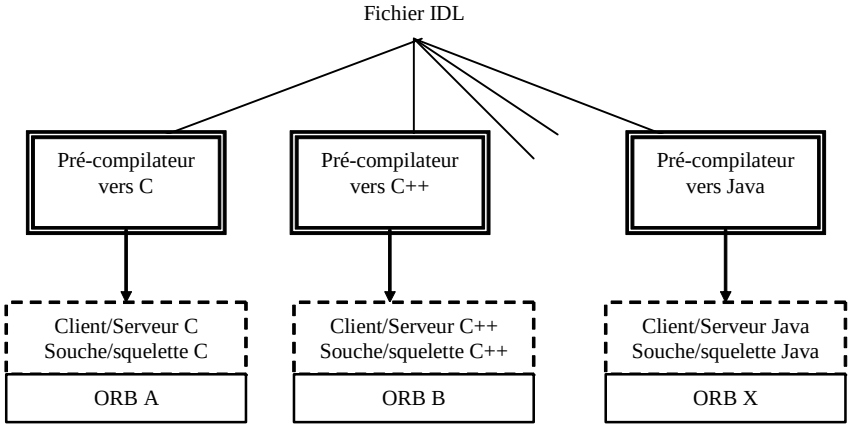

Figure 8. *Traduction de IDL vers les langages de programmation*

2.4. Mise en œuvre d'une application supportée par CORBA

Un schéma type de mise en place d'une application distribuée au dessus de CORBA peut être décrit comme suit :

1. ***Ecriture du contrat IDL***
   Dans une approche orientée objet (en UML par exemple), on définit dans un premier temps les objets de l'application distribuée, puis dans un second temps on modélise les objets sous forme de contrats IDL. Ces contrats en IDL décrivent les interfaces des objets et des types de données. On les enregistre dans un fichier texte (appelons-le `interface.idl`) .

2. ***Compilation des sources IDL et projection vers les langages de programmation***
   Le compilateur choisi opère un contrôle syntaxique et sémantique des définitions IDL contenues dans le fichier `interface.idl`. Le compilateur génère le code :
   - des stubs, utilisés par les applications clientes
   - des squelettes, pour les programmes serveurs implantant ces objets.

   La projection est spécifique à chaque langage de programmation.

3. ***Implantation des objets***
   L'implantation d'un objet définit le comportement pour les opérations et attributs de son interface. L'implantation complète ou réutilise le code généré par les squelettes et stubs. Le développeur implante les objets dans le langage de son choix en respectant les règles de projection vers le langage choisi.

4. ***Implantation des applications serveurs***
   Le développeur code les programmes serveurs qui incluent implantation des objets et squelettes pré-générés. Ces programmes contiennent le code pour :
   - se connecter à un ORB,
   - instancier les objets sur le serveur,
   - rendre publiques les références sur ces objets à l'aide du service de nommage,
   - se mettre en attente des requêtes des clients pour ces objets.

5. ***Ecriture des applications clients***

C'est dans ces programmes que l'on invoque les opérations sur les objets. Ces programmes incluent :

- le code des souches (méthodes de liaison avec les serveurs, ...),
- les références aux objets utilisés,
- le code spécifique à l'application du client.

6. ***Installation et configuration des serveurs***
   On enregistre les serveurs d'objets auprès du dépôt des implantations. Ceci permet d'automatiser leur activation lorsque les requêtes vont arriver.

7. ***Diffusion et configuration des clients***
   Après avoir écrit les programmes clients, il est nécessaire de diffuser les exécutables sur les sites clients.

8. ***Lancement de l'application distribuée***
   Le bus CORBA : l'ORB assure alors les communications entre les clients et les objets disponibles sur tels serveurs via le protocole IIOP.

2.5. Exemple d'application : *lecture d'horloge à distance*

1) Définition d'une interface (définition de contrat)

On décrit une interface simple, appelée `Horloge`, ne contenant qu'une opération qui renvoie une chaîne de caractères indiquant l'heure lue :

```
Interface Horloge {
      String Lire_horloge() ;
                  }
```

L'interface `Horloge` est placée dans un fichier `Horloge.idl`.

2) Pré-compilation du fichier `Horloge.idl`

On considère ici que le langage de programmation cible est Java. On fait une pré-compilation avec l'outil **`idl2java`**[3] utilisé avec l'ORB VisiBroker (attention, ici on n'insiste pas sur les options de la commande `Idl2java`) :

```
>Idl2java Horloge.idl
```

On obtient les fichiers et classe Java suivants :

- Un fichier souche (pour le client) ayant le nom `_HorlogeStub.java` qui est utilisé lors des appels de l'objet par le client.
- Un fichier squelette (pour le serveur) ayant le nom `_HorlogeImplBas.java` qui se charge des requêtes envoyées par le client.
- Une classe `Horloge.java` qui définit les services de l'objet `Horloge`.
- Une classe `HorlogeHelper.java` qui contient notamment la fonction **`narrow`** qui permet de convertir un objet en référence.
- Une classe `HorlogeHolder.java` utilisée lors de passage de paramètres d'appel de l'objet `Horloge`.

3) Implantation de l'objet

[3] Il y a d'autres outils de pré-compilation comme `idlj` qui permettent de passer de IDL à Java.

Comme on programme en Java, l'implantation de l'objet est une classe (que nous appelons `HorlogeImpl`) qui implante la classe (que nous avons appelée `_HorlogeImplBase`) correspondant au squelette obtenue à l'étape précédente. Ensuite, on implante l'opération définie dans l'interface :

```
// L'implantation de l'objet est placée dans le fichier HorlogeImpl.java
/** Implantation de l'interface pour la lecture d'horloge **/
Public class HorlogeImpl extends _HorlogeImplBase
{  public string lire_horloge()
      {
        java.util.Calendar calendar = new java.util.GregorianCalendar();
        string str = "[" + calendar.get(java.util.Calendar.HOUR_OF_DAY) +
        ":" + calendar.get(java.util.Calendar.MINUTE) + "]";
        return str;
      }
}
```

4) Implantation du serveur

Dans une communication entre un client et un serveur via un bus CORBA, l'ORB joue le rôle d'intermédiaire qui met les deux correspondants en relation. Client et serveur n'ont pas besoin de se connaître de manière directe. L'ORB a donc besoin de connaître la localisation exacte des objets.
L'implantation du client et serveur dépend de l'ORB utilisé. Dans ce qui suit, nous supposons que l'ORB utilisé est *Visibroker*.
Quelques remarques sur le code qui suit :

- L'importation de `org.omg.CORBA` permet d'avoir accès à tous les identificateurs (de types, d'opérations…) définis par CORBA.
- Le BOA (Basic Object Adaptor) est un adaptateur d'objet (POA) fonctionnant avec l'ORB VisiBroker.
- La fonction **`boa.obj_is_ready`** permet de signaler au BOA que l'objet donné en argument est prêt à recevoir des requêtes.
- La fonction **`boa.impl_is_ready`** est appelée une fois que tous les objets sont devenus prêts (signalés via **`boa.obj_is_ready`**) pour que le BOA rentre dans une boucle infinie d'attente des requêtes.

```
/** Code du serveur placé dans le fichier ServeurHorloge.java **/

import java.util.* ;
import org.omg.CORBA.* ;
import com.inprise.vbroker.orb.*;

public class Serveur
{  public static void main(string[] args)
      {
        // 1. Initialisation du bus CORBA (c'est-à-dire de l'ORB)
        org.omg.CORBA.ORB orb = org.omg.CORBA.ORB.init(args,null);
        // 2. Initialisation de l'adaptateur d'objet (BOA)
        com.inprise.vbroker.CORBA.BOA boa =
            ((com.inprise.vbroker.CORBA.ORB)orb).BOA_init();
        // 3. Création de l'implantation de l'objet Horloge et ajout de
        // l'implantation au BOA
        HorlogeImpl H1 = new  HorlogeImpl("HorlogeTOTO") ;
        boa.obj_is_ready(H1);
        System.out.println(H1+ " est prete");
        // 4. On lance le serveur qui va boucler en attente de requête
        boa.impl_is_ready() ;
      }
}
```

5) Compilation du serveur

Le code du serveur est compilé et édité (linker) avec l'ORB pour obtenir le code exécutable de l'application-serveur.

6) Implantation du client

On implante ici un client simple qui lit une seule fois l'horloge du serveur et l'affiche.
Quelques remarques concernant le code ci-dessous :

- `HorlogeHelper` : la classe `HorlogeHelper` est créée lors de la pré-compilation du contrat `Horloge.idl`.
- `bind` : cette fonction permet de lier un nom d'objet à une référence d'objet. L'opération **`bind`** qui est propriétaire des ORB et non une opération normalisée par CORBA. On peut aussi obtenir la référence d'un objet en utilisant l'opération CORBA `string_to_object()`.

```
import java.util.* ;
import org.omg.CORBA.* ;
/** Code du client placé dans le fichier ClientHorloge.java  **/
public class Client
{  public static void main(string[] args)
      {
        // 1. Initialisation du bus CORBA (c'est-à-dire de l'ORB)
        org.omg.CORBA.ORB orb = org.omg.CORBA.ORB.init(args,null);
        // 2. Recherche de l'horloge appelée Horloge ; obtention de la
        // référence d'objet. Ici on utilise l'opération bind propriétaire
        // de l'ORB VisiBroker.
        Horloge proxyHorloge = HorlogeHelper.bind(orb, "HorlogeTOTO");
        // Appel de l'opération lireHorloge
        string heure = proxyHorloge.lire_horloge() ;
        // Affichage du résultat
        System.out.println(("Il est : " + heure) ;
      }
}
```

7) Compilation du serveur

Le code du client est compilé et édité (linker) avec l'ORB pour obtenir le code exécutable de l'application-client.

Récapitulatif

Etapes pour développer le serveur :

- initialiser l'ORB (création d'une instance de l'objet ORB)
  - par : `org.omg.CORBA.ORB orb = org.omg.CORBA.ORB.init(args,null) ;`
- initialiser le POA (si le BOA est utilisé, voir dans els exemple précédents comment il est initialisé)
  - par : `POA rootpoa = POAHelper.narrow(orb.resolve_initial_refrecences("RootPOA");`
- créer et activer l'objet CORBA
  - par : `HorlogeImpl H1 new HorlogeImpl() ; // création d'objet rootPOA.activate_object(h1) ; // activation d'objet`
- éventuellement, enregistrer l'objet
  - par : `org.omg.CORBA.Object ref = rootpoa.servant_to_reference (H1) ;`
- activation du POA
  - par : `rootpoa.the_POAManager().activate();`
- attendre les invocations des méthodes de l'objet.

Etapes pour développer le client :

- initialiser l'ORB (création d'une instance de l'objet ORB)

    - par : `org.omg.CORBA.ORB orb = org.omg.CORBA.ORB.init(args,null) ;`
- récupérer la référence, `ref`, de l'objet à utiliser par exemple en la lisant à partir du clavier
- convertir la référence vers le type de l'objet à utiliser
    - par :
```
org.omg.CORBA.Object obj = orb.string_to_object(ref) ;
Horloge H1 = HorlogeHelper.narrow(obj);
```
- utiliser l'objet

Récapitulatif des fichiers et classes générés

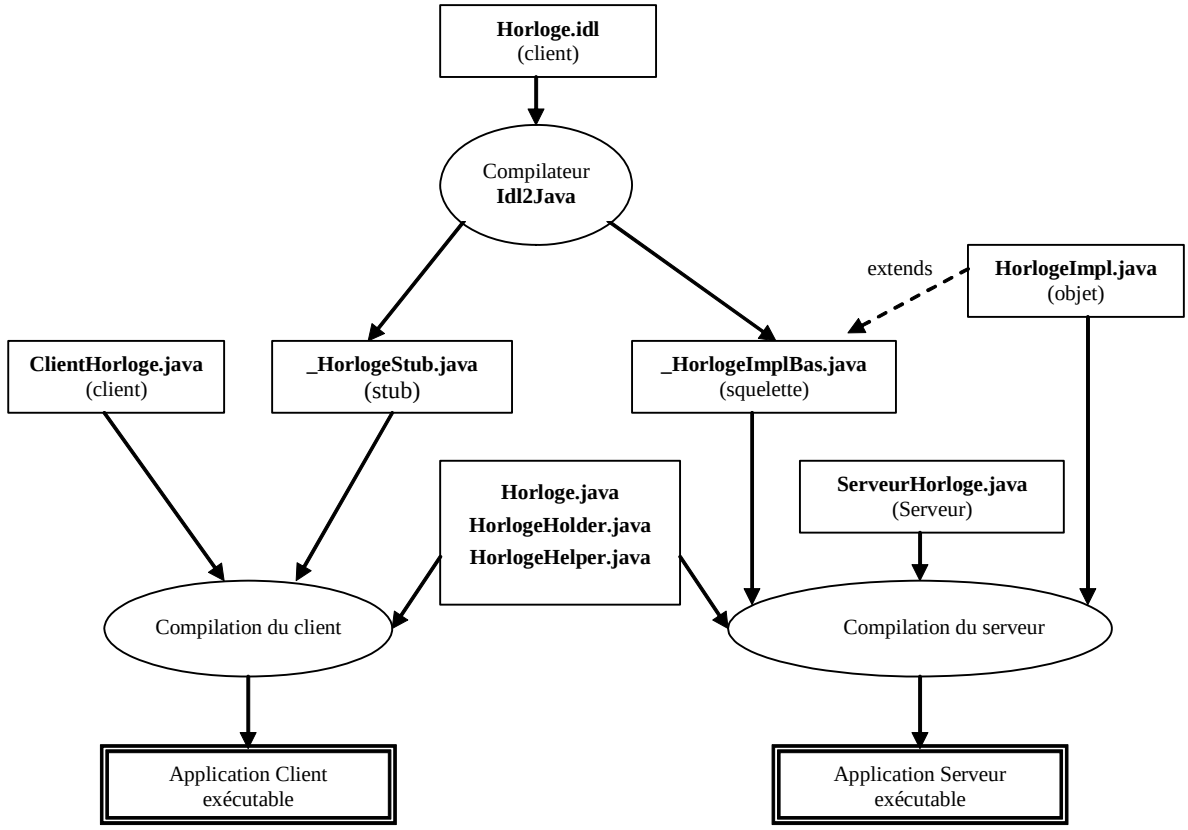

Figure 9. *Fichiers générés pour l'application Horloge*

3. Interface de l'ORB

L'interface de l'ORB définit les opérations qui sont implémentées dans le noyau de l'ORB. Cette interface définit toutes les opérations communes à tous les ORBs, les opérations qui ne sont pas dépendantes de l'adaptateur d'objets et les opérations de l'ORB qui peuvent être appelées par le client ou le serveur.

Toutes les opérations, pour identifier l'ORB, sont définies dans cette interface, par exemple, l'opération `ORBid()` qui retourne l'identité de l'ORB.

Quand le client veut avoir accès à un objet, il doit d'abord obtenir la référence de l'objet. Etant donné que les références d'objet peuvent varier d'un ORB à un autre, il est donc nécessaire de pouvoir convertir la référence d'objet dans un format que le client peut stocker. Il est nécessaire aussi de s'assurer que la valeur peut être transformée en une référence d'objet appropriée. Les opérations pour transformer des chaînes de caractères en références d'objet et les références d'objet en chaînes de caractères ont été définies.

Dans l'interface de l'ORB, des opérations sont définies pour identifier les services et les facilités qui sont supportées pour l'implémentation de l'ORB.

Nous donnons ci-dessous les principaux services (opérations) fournis par l'ORB :

- `ORBid`: renvoie l'identité de l'ORB utilisé.
- `object_to_string` *:* transforme une référence d'objet en une chaîne de caractères pour permettre l'interopérabilité (en cas de changement d'ORB).
- `string_to_object` *:* transforme une chaîne de caractères en une référence d'objet pour permettre l'interopérabilité (en cas de changement d'ORB).
- `resolve_initial_references` : renvoie la référence associée à un objet.
- `bind` : association d'un nom d'objet à une référence d'objet.
- `get_service_information` : renvoie la liste des services offerts par l'ORB.
- `run` : exécution d'une fonction interne à l'ORB.
- `shutdown` : arrêt de l'ORB.
- `destroy` : destruction de l'ORB pour libérer les ressources.
- `get_interface` : renvoie un objet, à partir du dépôt d'interfaces, qui correspond à une référence.
- `Duplicate` : duplique un objet.
- `release` : libérer une référence d'objet.
- `destroy` : destruction d'un objet.
- `create_policy` : création d'information de politique (de sécurité notamment).
- `get_policy` : renvoie la politique liée à un objet.
- `get_client_policy` : applique la politique du client à l'objet.
- `validate_connection` : valider la liaison entre objet et référence.
- `get_domain_managers` : renvoie les administrateurs de domaines.
- `get_orb` : renvoie l'ORB local qui traite un objet spécifié en argument.

Attention : les services (opérations) de l'ORB dépendent de l'ORB et peuvent avoir des noms qui diffèrent d'une implantation d'ORB à l'autre.

4. Interfaces d'invocation statique et dynamique

4.1. Interfaces d'invocation statique

Les stubs et les squelettes sont des interfaces d'invocation statique ; elles sont définies au moment de la compilation du script IDL. Les stubs, du côté du client, représentent le point d'accès aux opérations des objets définies par l'IDL. Ils ont pour fonction de faire des appels aux opérations qui se trouvent dans le serveur. Du point de vue du client, c'est comme si le client est en train de faire un appel à une opération locale.

Du côté du serveur, une fois la compilation du script IDL faite, les méthodes spécifiées dans les interfaces IDL doivent être implémentées. L'implémentation d'objet va créer les routines, qui en accord avec l'interface ORB, seront appelées à travers les squelettes. Il peut exister des squelettes sans qu'il existe de stub correspondant.

4.2. Interfaces d'invocation dynamique

Les interfaces d'invocation dynamique permettent de construire une requête dont la signature n'est pas connue jusqu'au moment d'exécution. Elles permettent de créer et d'invoquer les requêtes de façon dynamique, au lieu d'appeler un stub statique. Le client peut spécifier l'objet qui sera invoqué, l'opération qui sera exécutée, et l'ensemble des paramètres pour l'opération, au travers d'un appel ou d'une séquence d'appels. Le client doit fournir toute l'information concernant l'opération et les types de paramètres. Des

opérations pour créer des requêtes, pour manipuler des paramètres ou pour obtenir des résultats, ont été définies, par exemple, `create_request`, `add_arg` et `get_reponse`.

Du côté serveur, le squelette dynamique permet d'accepter des invocations. Au lieu d'utiliser un squelette statique, l'implémentation de l'objet est atteinte par une interface qui fournit l'accès au nom de l'opération et les paramètres de façon dynamique. Cette forme d'invocation est analogue à l'invocation dynamique du côté client. L'implémentation de l'objet doit fournir à l'ORB la description de tous les paramètres, et l'ORB doit fournir les valeurs de tous les paramètres d'entrée à utiliser pendant l'exécution.

5. Adaptateur Portable d'Objets (POA)

Le POA est une des principales composantes de l'architecture de CORBA, il a été conçu pour accomplir plusieurs tâches, notamment :

- permettre d'implanter des objets qui soient portables entre différents ORBs,
- permettre aux programmeurs d'implémenter des objets capables de conserver une référence d'objet même si le serveur a fini son exécution,
- permettre l'activation *transparente* des objets,
- permettre aux servants de gérer de multiples identités d'objets de façon simultanée,
- permettre plusieurs instances du POA dans le même serveur,
- associer des politiques aux objets,
- maintenir l'état persistant des objets et leurs identités.

Le POA est uniquement visible pour le serveur, toutes les implémentations fournies par le développeur doivent y être enregistrées. Le POA, l'ORB et les implémentations doivent travailler ensemble pour déterminer quel *servant* doit être invoqué. Le servant est l'entité qui implémente la requête sur les objets, c'est-à-dire que les requêtes faites sur les références d'objets sont transformées par l'ORB en invocations sur les servants. La figure 10 montre le modèle du POA.

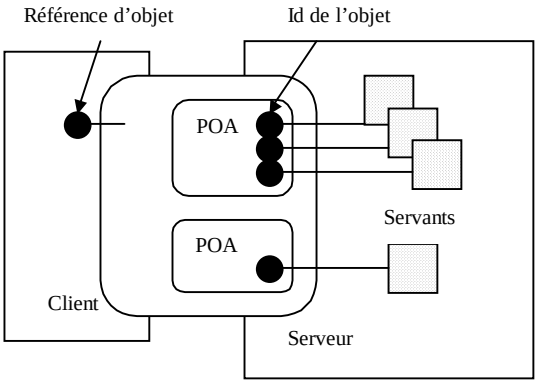

Figure 10. *Le modèle du POA*

Le POA négocie avec l'*ObjectId* et les servants *actifs*. Un servant actif est un objet qui se trouve en mémoire et qui a été enregistré dans le POA à travers les identités des objets.

Le terme *actif* est appliqué aux servants, ObjectIds et objets. Un objet est actif s'il a une entrée dans le répertoire d'objets actifs du POA associant un ObjectId avec un servant.

La requête doit être capable de transmettre l'ObjectId de l'objet et l'identité du POA qui a créé la référence de l'objet. Quand le client fait une requête, l'ORB doit, en première place, localiser le serveur (ou l'initialiser si c'est nécessaire) et ensuite localiser le POA adéquat dans le serveur. Une fois que l'ORB a localisé le POA, il lui transmet la requête. Le processus à suivre dépend des politiques associées au POA et de l'état courant d'activation des objets.

L'interface du POA contient essentiellement les opérations suivantes :

- `create_POA` : créer d'une nouvelle instance de POA.
- `get_POA` : renvoyer une référence sur le POA correspondant à un objet fourni en argument.
- `impl_is_ready` : signaler que l'implantation d'objet est prête à recevoir et traiter des requêtes.
- `destroy` : détruire une instance de POA.
- `get_servant_manager` : obtenir le nom du gestionnaire d'un objet.
- `set_servant_manager` : changer le nom du gestionnaire d'un objet.
- `activate_object` : activer une instance d'objet.
- `desactivate_object` : désactiver une instance d'objet.
- `create_reference` : créer de référence d'objet.

6. Dépôt d'interfaces

Le dépôt (ou répertoire) d'interfaces est la composante de l'ORB qui fournit le stockage permanent, la distribution et la gestion de l'ensemble d'interfaces. Les interfaces pour l'ORB et les définitions d'objets spécifiés par l'OMG sont gérées aussi par le dépôt.

L'ORB peut utiliser les définitions stockées dans le dépôt d'interfaces pour interpréter et manipuler les valeurs fournies dans la requête. De cette façon, l'ORB peut vérifier le type de données des paramètres dans la signature, corroborer l'exactitude de l'héritage des interfaces, et fournir l'interopérabilité entre les différentes implémentations d'ORBs. Etant donné que les interfaces stockées dans le dépôt sont publiques, l'information est disponible pour les clients et serveurs.

Pour faciliter la navigation dans le contenu du dépôt, les interfaces sont groupées en modules, ces modules peuvent contenir : constantes, types, exceptions, définitions d'interfaces et autres modules. Toute cette information est structurée comme un ensemble d'objets, des opérations sont définies pour gérer et pour utiliser l'information.

C'est au moment de connecter plusieurs ORBs que l'on trouve une des utilisations les plus importantes du dépôt d'interfaces. Quand un objet change d'ORB au cours d'une requête, il peut être nécessaire de créer un nouvel objet pour le représenter dans l'ORB qui reçoit la requête. Il peut être nécessaire de localiser l'information concernant l'interface dans le dépôt de l'ORB qui reçoit la requête. S'il est possible d'avoir l'Id du dépôt de l'ORB émetteur, il est possible alors de le trouver dans le dépôt de l'ORB récepteur.

L'information dans le dépôt d'interfaces permet aux programmes de déterminer et d'évaluer le type d'information au moment de l'exécution. Le client ou le serveur peut avoir accès à l'information en utilisant l'opération `get_interface` pour la référence de l'objet.

7. Interopérabilité dans CORBA

L'interopérabilité dans CORBA fournit la possibilité de supporter des réseaux d'objets distribués, qui sont (probablement) gérés par différents ORBs.

7.1. Architecture de l'interopérabilité de CORBA

L'interopérabilité de CORBA est basée sur les trois éléments suivants :

- une architecture pour l'interopérabilité,
- le support pour les ponts Inter-ORB,
- le Protocole Général et l'Internet Inter-ORB (GIOP et IIOP).

Il est possible, en plus, de spécifier des protocoles pour des environnements spécifiques, connus comme *inter-ORB protocoles d'environnement spécifique* (ESIOP).

En utilisant les ponts, les ORBs peuvent interopérer sans nécessité de connaître les détails d'implémentation de l'ORB, par exemple, le type de protocole utilisé.

L'architecture d'interopérabilité identifie les rôles des différents types de domaines : domaines des références d'objets, domaines de type, domaines de sécurité, domaines de transaction, etc.

Quand deux ORBs partagent le même domaine, ils peuvent communiquer directement, mais les organisations ont besoin d'établir des domaines locaux de contrôle. Si l'information doit quitter son domaine, l'invocation doit passer par un pont. Le rôle du pont est d'assurer que le contenu et la sémantique de l'ORB émetteur seront projetés sur la forme appropriée pour l'ORB récepteur. Le support des ponts inter-ORB peut être utilisé pour fournir interopérabilité avec des systèmes non-CORBA, par exemple le Modèle de Composant d'Objet (COM) de Microsoft.

La spécification de l'architecture d'interopérabilité de CORBA a été conçue pour permettre aux différentes implémentations d'ORBs d'interopérer à l'aide de ponts et d'éléments des protocoles communs.

Les objectifs suivis en spécifiant l'architecture d'interopérabilité sont les suivantes :

- Permettre une haute performance et des solutions légères d'interopérabilité.
- La conception ne doit pas être difficile à implémenter et ne doit pas non plus être trop restrictive en termes d'implémentation.
- Les solutions d'interopérabilité doivent être capables de fonctionner avec les implémentations des ORBs déjà existants.
- Toutes les opérations incluses dans le modèle d'objet de CORBA ainsi que la gestion de types de données doivent être supportées.

7.2. Domaines

Dans la spécification d'interopérabilité de CORBA, un domaine est une portée dans laquelle les membres du domaine sont associés selon certaines caractéristiques communes. Pour ces objets, la transparence de distribution doit être préservée. Les domaines peuvent être administratifs ou techniques et, normalement, ils ne peuvent pas correspondre aux limites d'installation de l'ORB.

Les domaines administratifs comprennent : les domaines des noms, les groupes de confiance, les domaines de gestion de ressources en général, les caractéristiques de temps d'exécution. Les domaines techniques comprennent les caractéristiques d'implémentation.

Conceptuellement, les mécanismes pour établir des ponts entre les domaines résident dans les frontières de chaque domaine, afin de transformer les requêtes exprimées en termes du domaine émetteur, en termes du domaine de destination.

Un des objectifs de la spécification de CORBA est la transparence afin de s'assurer que les objets client et serveur ont une vue uniforme de l'environnement du système distribué, mais du point de vue des développeurs le système n'est pas uniforme. La figure 11 présente un exemple de la cohabitation de différents types de domaines.

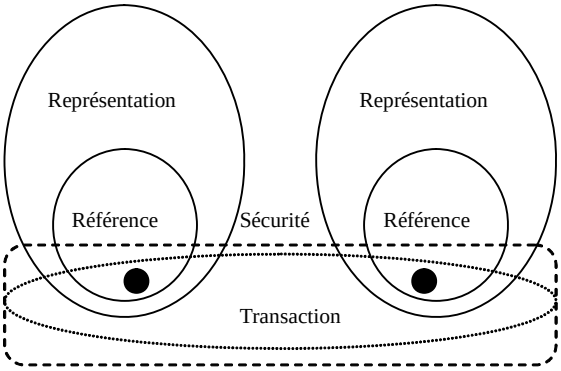

Figure 11. *Différents domaines peuvent exister dans le même réseau*

Pour obtenir une interopérabilité totale, il est indispensable que tous les concepts d'un domaine soient transformés en concepts de l'autre domaine avec lequel l'interopérabilité est nécessaire.

7.3. Ponts immédiats et médiats

Quand deux domaines doivent communiquer, les ponts transforment les donnés au niveau des frontières de chaque domaine. Il y a deux approches pour implémenter les ponts : les ponts médiats et les ponts immédiats ; ces concepts sont présentés dans la figure 12.

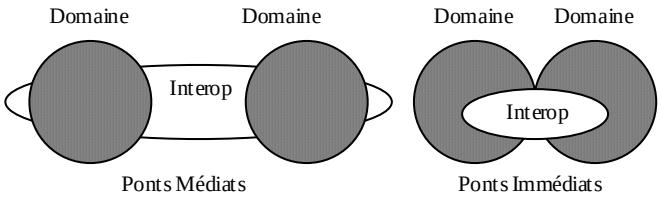

Figure 12. *Les deux types de ponts spécifiés en CORBA*

Avec les ponts médiats, les éléments essentiels d'interaction entre les domaines sont transformés aux frontières de chaque domaine, du format interne du premier domaine dans un format agréé pour les deux domaines. Il y a quelques observations à prendre en considération en utilisant les ponts médiats :

- Le niveau d'accord entre les domaines peut varier, d'un accord privé entre les ORBs à un format universel.
- Il peut y avoir plusieurs formats, chacun destiné à optimiser un critère spécifique.
- Si plusieurs formats sont présents, la sélection du format peut être statique ou dynamique.

Avec les ponts immédiats, les éléments essentiels d'interaction entre les domaines sont transformés directement, aux frontières de chaque domaine, du format interne du premier domaine dans le format interne du deuxième domaine. Il y a quelques observations à prendre en considération en utilisant les ponts immédiats :

- Cette approche est la plus optimale, mais la flexibilité et la généralité sont sacrifiées.
- Cette approche est applicable quand les domaines sont des domaines administratifs.

Le but de l'interopérabilité est de cacher les limites des domaines, afin de garantir la transparence. Les domaines administratifs existent, parce qu'il existe entre les domaines des différences en politique d'organisation ou de buts. Les ponts, pour de telles situations, nécessitent de la *médiation des politiques*, c'est-à-dire qu'il est nécessaire de contrôler, surveiller ou limiter le trafic. Des politiques pour la gestion de ressources peuvent être appliquées, pour limiter certains types de trafic durant une certaine période de temps.

7.4. Références d'objets

Dans le modèle d'objet de l'OMG, la référence d'objet est définie comme le nom de l'objet, qui de façon fiable dénote un objet particulier. La référence d'objet identifie le même objet à chaque fois que la référence est utilisée dans la requête, mais un objet peut être référencé par plusieurs références.

Le principe fondamental de l'interopérabilité de l'ORB c'est de permettre aux clients d'utiliser les noms des objets pour invoquer les opérations. Les objets peuvent se trouver dans d'autres ORBs. Les clients n'ont pas besoin de savoir si l'objet invoqué est un objet local ou distant.

Quand le pont doit manipuler une référence d'objet pour l'ORB, il doit le faire d'une façon compréhensible pour l'ORB, c'est-à-dire que la référence d'objet doit être dans le format natif du récepteur. Dans ce cas, le pont doit associer la nouvelle référence avec la référence originale. Plusieurs schémas ont été proposés pour implémenter cette correspondance entre références :

- Translation de la référence d'objet sur une référence incorporée : le pont garde la référence d'objet originale et passe une référence *proxy* différente vers le nouveau domaine. Le pont doit gérer l'état de chaque référence d'objet passée et transformer les références de format du premier ORB dans le format du deuxième et vice versa.
- Encapsulement des références : l'identificateur du domaine et le nom de l'objet sont enchaînés. Par exemple, si la référence $D_0.R$, créée dans le domaine D_0, traverse par les domaines $D_1 \ldots D_4$, dans le domaine D4, elle sera identifiée avec un proxy $d_3.d_2.d_1.d_0.R$, d_n étant l'adresse du domaine D_n. Voir figure 13.

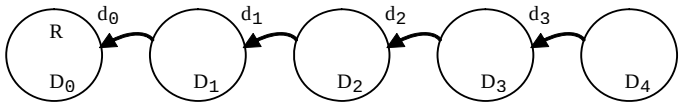

Figure 13. *L'encapsulement de référence ajoute l'information du domaine dans le pont*

- Translation des références de domaines : l'état des références d'objet est partagé, en ajoutant au proxy un identificateur de route basé sur les domaines. L'information de routage dans le domaine est encodée à chaque fois qu'une frontière de domaines est traversée. Par exemple, si la référence R, créée dans le domaine D_0, traverse les domaines $D_1 \ldots D_4$, elle peut être identifiée dans le domaine D_4 comme $(d_3.x_3)$. R et dans le domaine D_3 comme $(d_2.x_2).R$, et ainsi de suite, dn étant l'adresse du domaine D_n. Voir figure 14.
- La forme canonique de la référence : le proxy utilise un identificateur bien connu dans le domaine au lieu d'une route encodée. Par exemple, la référence R, créée dans le domaine D_0, sera connue comme $D_0.R$ dans d'autres domaines.

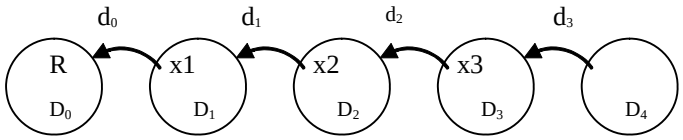

Figure 14. *Translation des références des domaines, les références des domaines substituts sont mises dans le pont*

L'information concernant les références, nécessaires pour le fonctionnement des ponts, peut inclure :

- *Référence nulle.* Les références nulles peuvent être transmises, mais ne produisent pas d'invocations des opérations.
- *Quel est le type ?* L'ORB peut avoir besoin de connaître le type de l'objet afin de préserver l'intégrité de types de données dans le système.
- *Quels protocoles sont supportés ?* Pour permettre aux clients de choisir le moyen de communication le plus adapté à ses besoins.
- *Quels sont les services de l'ORB qui sont disponibles ?* Plusieurs services peuvent être impliqués dans l'invocation, l'information concernant les services peut réduire la négociation entre les ORBs pour les sélectionner.

7.5. Protocole Général Inter-ORB (GIOP)

Le GIOP permet l'interopérabilité des ORBs et peut être projeté sur tout protocole de transport orienté connexion qui satisfait certaines conditions. La projection de GIOP sur TCP/IP est appelée Protocole Internet Inter-ORB (IIOP). Il y a trois versions de GIOP (1.0, 1.1, 1.2) dans la version 1.1 la fragmentation de messages est définie, dans la version 1.2 le support bidirectionnel de communication est ajouté.

Le protocole GIOP a été conçu pour supporter, de façon générale et à faible coût, l'interopérabilité des ORB. Les objectifs poursuivis sont :

- Disponibilité : GIOP et IIOP sont basés sur (TCP/IP) et définissent les couches minimum pour transmettre des requêtes CORBA entre les ORBs. La disponibilité est obtenue parce que TCP/IP est le mécanisme de transport le plus répandu.
- Simplicité : GIOP a été conçu pour être le plus simple possible.
- "Scalability" : GIOP et IIOP doivent supporter des ORBs au niveau d'Internet.
- Faible coût : l'addition de GIOP/IIOP dans un ORB doit requérir peu d'efforts, le coût en temps d'exécution pour supporter IIOP dans un ORB doit être aussi minimal.
- Généralité : les formats de messages de GIOP ont été conçus pour être utilisés dans toute couche de transport qui supporte certaines conditions basiques, spécifiquement les protocoles de transport orientés connexion.
- Neutralité d'architecture : pour le GIOP, les ORB sont des entités opaques avec une architecture inconnue.

7.5.1. Composantes de base de GIOP

La spécification de GIOP est composée de trois éléments basiques :

- la Représentation Commune de Données (CDR),
- le format de messages GIOP,
- les besoins au niveau du transport de GIOP.

CDR est la projection de la syntaxe de transfert des types de données IDL dans une représentation de bas niveau pour le transfert, au niveau du thread, des donnés entre ORBs et ponts inter-ORB (appelés Agents). Les messages GIOP sont échangés entre les agents pour faciliter les requêtes d'objets, la localisation des implémentations et la gestion des canaux de communication. Les besoins généraux, concernant la couche de transport pour le transfert des messages GIOP, sont aussi spécifiées. La spécification décrit la façon de traiter les connexions et les contraintes d'ordre des messages GIOP.

En plus des éléments de la spécification de GIOP, la spécification d'IIOP ajoute la description de la façon selon laquelle les connexions TCP/IP sont utilisées pour transférer les messages GIOP.

7.5.2. Besoins au niveau transport de messages

GIOP fait les hypothèses suivantes, concernant le transport de messages :

- Le transport est orienté connexion. Ceci pour permettre à l'émetteur d'établir une connexion en spécifiant l'adresse du récepteur. Une fois la connexion établie le transport rend une "poignée de main" à l'émetteur pour identifier la connexion. L'émetteur envoie le message par la connexion sans nécessité de spécifier l'adresse dans chaque message, l'adresse est implicite dans la poignée utilisée pour envoyer les messages.
- La connexion est bidirectionnelle en simultané (full-duplex). Le récepteur utilise la poignée pour recevoir les messages et aussi pour envoyer la réponse, c'est-à-dire que le récepteur peut répondre à l'émetteur en utilisant la même connexion, il n'a donc pas besoin de connaître l'adresse de l'émetteur.
- La connexion est symétrique. L'émetteur ou le récepteur peut fermer la connexion.
- Le transport est fiable. Le transport doit garantir que les messages sont livrés qu'une seule fois et dans le même ordre dans lequel ils ont été envoyés par l'émetteur. Si un message n'est pas livré le transport doit retourner un message d'erreur à l'émetteur.
- Le transport doit fournir une abstraction de flux constant d'octets. Le transport ne doit imposer aucune contrainte concernant la taille des messages, le récepteur voit la connexion comme un flux continu d'octets. L'émetteur et le récepteur ne sont pas concernés par la fragmentation, la duplication, la retransmission ou l'alignement des messages.
- Si le réseau faillit, les deux côtés de la connexion doivent recevoir une indication d'erreur.

Cette liste de besoins au niveau transport correspond aux garanties fournies par TCP/IP.

7.5.3. Représentation commune de données

CDR est la syntaxe de la projection des types de données IDL dans un format de bas niveau pour le transfert entre les agents. CDR a les caractéristiques suivantes :

- Support des formats d'ordre d'octets *big-endian*[4] et *little-endian*. Si l'émetteur et le récepteur ont des représentations différentes dans l'ordre d'octets, le récepteur est le responsable de la conversion d'ordre.
- Chaque type de donnée élémentaire est aligné conformément à sa limite naturelle. Par exemple, les valeurs *short* sont alignées sur 2 octets, les valeurs *long* sont alignées sur 4 octets et ainsi de suite. Le codage d'octets en CDR consiste à remplir les octets.
- Les donnés encodées avec CDR ne s'identifient pas elles-mêmes. Si par exemple un type de données *long* (4 octets) est suivi par un type de données *double* (8 octets), les 4 premiers octets vont contenir la valeur du type de données *long*, suivis par 4 octets de remplissage. Les 8 octets qui suivent vont contenir la valeur du type de donnée *double*, ce qui fait un total de 16 octets. Le récepteur ne voit

[4] En représentation *big-endian* on commence par les bits de poids forts (MSB suivi de LSB) – le premier bit est le bit de poids le plus fort. En représentation *little-endian* on commence par les bits de poids faibles (LSB suivi de MSB).

que 16 octets. Un accord préalable (sur les types de données à échanger) entre le récepteur et l'émetteur est nécessaire. Cet accord est établi par la définition des interfaces avec IDL.

On notera que CDR joue le rôle de ASN.1.

7.5.4. Formats des messages GIOP

GIOP définit huit types de messages (voir tableau 1). Les messages `Request` et `Reply` sont les plus importants parce qu'ils permettent d'établir le mécanisme basique de RPC (Appel de procédure distante). Afin de pouvoir envoyer un message GIOP, l'émetteur doit envoyer un *entête de message* suivi par le *corps du message*. Le contenu du message est dépendant du type de message indiqué dans l'entête. La figure 15 montre la structure de l'entête de message.

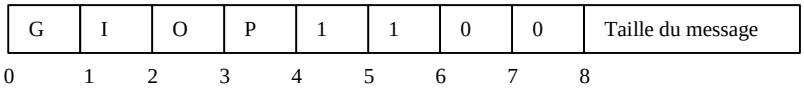

Figure 15. *Structure de l'entête de message Request pour la version 1.1, codé big-endian et sans fragments*

- Les 4 premiers octets contiennent toujours les caractères GIOP qui indiquent qu'il s'agit d'un message GIOP et aussi pour indiquer les limites du message.
- Les octets 4 et 5 indiquent la version majeure et mineure du GIOP.
- L'octet 6 est un drapeau, le bit le moins significatif est utilisé pour indiquer si le message est en format big-endian ou little-endian; zéro indique un codage big-endian. Le deuxième bit moins significatif indique si le message est un message fragmenté; un zéro indique, soit que c'est un message complet ou que c'est le dernier fragment.
- L'octet 7 indique le type de message; un zéro indique un type de message `Request` (les valeurs pour chaque type de message sont énumérées dans le tableau 1).
- Les octets 8-11 indiquent la taille du message (sans compter l'entête à 12 octets), le message est codé dans le format indiqué par le bit le moins significatif de l'octet 6.

7.5.5. Types de messages

- Le message `Request` est le message pour faire une requête, il est toujours envoyé par le client et il est utilisé pour invoquer une opération, pour écrire ou pour lire un attribut. Dans un message de requête, il est possible de spécifier uniquement des paramètres `in` et `inout`.
- Le message `Reply` est le message utilisé pour répondre à une requête, il est toujours envoyé par le serveur. Il contient les résultats de l'invocation de l'opération. Dans un message de réponse, il est possible de spécifier uniquement des paramètres `out` et `inout`. Si l'opération produit une exception, le message contient l'exception produite.
- Le message `CancelRequest` est utilisé pour indiquer que le client n'a plus besoin des résultats de l'opération.
- Le message `LocateRequest` est utilisé pour le client afin d'obtenir l'information relative à l'adresse de l'objet.
- Le message `LocateReply` est utilisé pour le serveur en réponse à un message `LocateRequest`.
- Le message `CloseConnection` est utilisé pour le serveur et le client pour signaler la fermeture de la connexion. Ce message est nécessaire afin d'indiquer que la fermeture est une déconnexion intentionnée et non pas une déconnexion en désordre.

- Le message `MessageError` est utilisé pour indiquer qu'il y a eu une erreur dans la production d'un message GIOP.
- Le message `Fragment` est utilisé pour envoyer un message dans plusieurs fragments. Le premier message est un message `Request` ou `Reply`, avec le bit de message fragmenté mis à 1. Chaque fragment qui suit contient le bit de message fragmenté mis à 1. Le dernier message de type Fragment met le drapeau à 0.

Type de message	Emetteur	Valeur
`Request`	Client	0
`Reply`	Serveur	1
`CancelRequest`	Client	2
`LocateRequest`	Client	3
`LocateReply`	Serveur	4
`CloseConnection`	Serveur/Client	5
`MessageError`	Serveur/Client	6
`Fragment`	Serveur/Client	7

Tableau 1 *Types de messages de GIOP*

7.6. Protocole Internet Inter-ORB

GIOP spécifie les détails pour la communication entre le client et le serveur. GIOP est un protocole abstrait, dans le sens qu'il est indépendant du transport, alors que IIOP est une implémentation concrète de GIOP au-dessus de TCP/IP.

Pour pouvoir implémenter GIOP au-dessus de TCP/IP, il est nécessaire de spécifier comment coder les Références Interopérables d'Objets (IOR). L'agent qui veut traiter des requêtes d'objets doit publier son adresse TPC/IP dans un IOR. L'adresse TCP/IP est composée de l'adresse IP de l'hôte (le nom de l'hôte) et le numéro de port.

Quand le client a besoin d'un service offert par un objet, il doit établir la communication avec l'adresse spécifiée dans l'IOR. Le serveur écoutant sur l'adresse spécifiée peut accepter ou refuser la connexion. Il peut avoir des politiques pour l'acceptation des connexions. Une fois que la connexion a été acceptée le client peut envoyer les messages `Request`, `CancelRequest`, `LocateRequest`, `MessageError` ou `Fragment` en écrivant sur la *socket*. Le serveur peut envoyer les messages `Reply`, `LocateReply`, `CloseConnection` et `Fragment`.

Apres avoir envoyé ou reçu un message du type `CloseConnection`, le client ou le serveur doit fermer la connexion TCP/IP.

7.7. Références Interopérables d'Objets (IOR) et profils IOR

Les IORs sont créés (au niveau de la frontière de domaines) à partir de référence d'objet lorsqu'il y a communication entre différents domaines. Les IORs sont le moyen général pour identifier un objet dans un environnement réparti multi-domaine. Les références d'objets sont opaques aux clients et spécifient toute l'information nécessaire pour envoyer les requêtes, y compris le transport et les protocoles à utiliser.

Les profils IIOP servent pour identifier des objets individuels qui peuvent être accédés à travers d'IIOP. Les profils se distinguent par les protocoles qu'ils supportent. En général, un profil IOR inclut au moins les informations suivantes : la version de IIOP (1.0, 1.1 ou 1.2), le nom de l'hôte récepteur du message GIOP, le numéro de port TCP/IP et le nom de l'objet vers lequel la requête est dirigée. Les adresses d'hôte sont limitées aux classes A, B ou C d'adresses d'Internet. Dans le cas de IIOP (c'est-à-dire quand CORBA est

supporté par TCP/IP), les profils de IOR incluent : le mécanisme de sécurité (politique de sécurité dans IP, environnement utilisé (SSL, Kerberos…), des infos sur le pare-feu…)

Un IOR doit spécifier l'ID du dépôt d'interfaces et il peut contenir plusieurs profils, pour différents protocoles ou pour le même, mais en indiquant différents hôtes et numéros de port. L'ID du dépôt fournit un identificateur unique pour chaque type qui est spécifié dans le script IDL.

8. Services d'objets

Quand le client envoie une requête, cette dernière peut contenir des attributs qui affectent la façon d'envoyer la requête au serveur. Ces attributs peuvent inclure des caractéristiques concernant la sécurité, la transaction, la récupération, la réplication, etc. Ces caractéristiques sont fournies par les services de l'ORB, elles peuvent résider dans le noyau de l'ORB ou dans une couche au-dessus de l'ORB. La spécification de CORBA et des services d'objets est assez ouverte pour permettre d'ajouter de nouveaux services.

Les services sont invoqués de façon implicite au moment des interactions avec les applications. Les services de l'ORB peuvent comprendre des mécanismes fondamentaux, par exemple la résolution des références, jusqu'aux caractéristiques plus avancées, par exemple sécurité, ordonnancement, transaction ou réplication. L'utilisation des services de l'ORB doit être transparente pour les agents.

La sélection des services de l'ORB à utiliser est déterminée par :

- les caractéristiques statiques du serveur et du client,
- les attributs dynamiques déterminés par le contexte dans lequel la requête est réalisée,
- les politiques administratives impliquées.

Quand il y a interaction entre plusieurs ORBs, il est nécessaire d'établir un accord concernant les services qui seront utilisés et la façon selon laquelle chacun d'entre eux va traiter la requête.

Les Services d'Objets de CORBA ont été définis comme un ensemble d'interfaces et de séquences de sémantiques qui sont utilisés pour implémenter des applications bien formées dans un environnement conforme à la spécification de CORBA. Les services d'objets fournissent des interfaces nécessaires pour construire des applications distribuées. Les services d'objets de l'OMG présentent les caractéristiques suivantes :

- Les opérations fournies par les services d'objets doivent servir comme des blocs de construction des facilités communes ou des applications.
- Les services d'objets sont modulaires et les objets utilisent les services dont ils ont besoin.
- Toutes les opérations fournies par les services d'objets sont spécifiées en IDL.

Les services sont conçus pour être les plus simples possibles et pour ne faire qu'une seule chose, mais bien faite. Les services sont conçus aussi pour être génériques, dans le sens où ils ne dépendent pas du type du client ni du type de données passées dans la requête. Les services sont structurés comme objets CORBA avec des interfaces IDL, qui peuvent être atteints localement ou à distance. Les services sont composés de plusieurs interfaces qui fournissent différentes vues pour différents types de clients. Certains des services spécifiés par l'OMG comprennent :

1. **Le service de noms** qui fournit la possibilité de relier le nom avec l'objet dans un contexte de noms. Le contexte de noms est un objet qui contient un ensemble de liaisons dans lequel chaque nom est unique. Le fait de résoudre un nom détermine l'objet associé à ce nom dans le contexte de noms. Différents noms peuvent être liés au même objet.

2. **Le service de cycle de vie** qui définit les conventions pour créer, effacer, copier et déplacer des objets. Le service de cycle de vie permet aux clients d'exécuter des opérations du cycle de vie des objets qui se trouvent dans des endroits différents.

3. **Le service de persistance d'objets** qui fournit un ensemble d'interfaces pour conserver et gérer l'état persistant des objets. L'objet a la responsabilité de la gestion de son état, cependant un délégué peut être utilisé pour réaliser cette fonction. Le service de persistance d'objets est optionnel mais fournit des capacités utiles, par exemple, stockage ou mécanismes pour des ordinateurs mobiles.

4. **Le service de transactions** qui fournit le support pour assurer l'exécution d'une tâche composée de plusieurs opérations dans plusieurs objets. Il fournit les propriétés d'*atomicité, de consistance, d'isolement* et *de durabilité* (ACID). Ce service fournit deux types de transactions : *plat* et *imbriqué*, dans le premier type les transactions regroupent toutes les opérations dans une seule entité transactionnelle, dans le deuxième type la transaction est imbriquée avec autres transactions. Deux modèles de propagation de transactions sont supportés, implicite (la transaction est gérée par le système) et explicite (la transaction est gérée par l'application).

5. **Le service de contrôle de concurrence** qui permet aux clients de coordonner les accès aux ressources partagées afin de conserver la cohérence de l'état des ressources. Ce service assure que les clients transactionnels et non-transactionnels sont coordonnés les uns par rapport aux autres. Le service de contrôle de concurrence travaille ensemble avec le service de transactions.

6. **Le service des relations** qui permet aux entités et relations d'être représentées de façon explicite. Les entités sont des objets CORBA. Deux types d'objets sont définis, les relations et les rôles. Un rôle représente un objet CORBA dans une relation. Une relation est créée au moment de passer un ensemble de rôles dans un ensemble de relations. Trois niveaux de service sont disponibles :
   - Le niveau basique qui définit des relations et rôles.
   - Le niveau de service arborescent est utilisé quand les objets sont impliqués dans une relation, en obtenant un arbre d'objets connectés.
   - Le niveau de service des relations définit des relations spécifiques.

7. **Le service de sécurité** qui comprend les fonctions de :
   - *Identification* et *authenticité* d'utilisateurs et d'objets. L'identification d'objets et d'utilisateurs peut être réalisée par attributs et contrôles de sécurité.
   - *Autorisation* et *contrôle d'accès* pour décider si l'accès aux objets peut être autorisé.
   - *Sécurité* et *audit*, les utilisateurs sont responsables des actions concernant la sécurité.
   - *Sécurité de communication* entre objets, un niveau de confiance doit être établi entre le client et l'objet, pour cela, l'authentification des clients dans les objets, l'authentification des objets chez les clients, la protection d'intégrité et de confidentialité peuvent être nécessaires.
   - *Administration* de l'information de sécurité qui peut être aussi nécessaire.

8. **Le service de temps** qui fournit les mécanismes pour synchroniser les horloges dans les systèmes distribués. Le temps est obtenu avec un certain taux d'erreur estimée. Ce service est utilisé pour établir l'ordre dans lequel les événements sont apparus, générer des événements basés sur des alarmes et temporisateurs, calculer le temps entre deux événements. L'objectif du service de temps est de fournir un mécanisme pour la synchronisation périodique des horloges dans tous les composants compris dans l'environnement du système distribué. Deux spécifications du service de temps ont été définies.

Il y a d'autres services, tels que le service d'événements, de notification, de message ou le service d'ordonnancement. Ces services constituent les extensions de CORBA pour la Qualité de Service (QoS) et les applications temps réel.

9. Conclusions

La spécification de CORBA est flexible parce qu'elle permet de développer des applications avec plusieurs langages de programmation. Il est uniquement est nécessaire que la projection du langage IDL sur le langage de programmation soit spécifiée. CORBA est aussi facile à manipuler car après la compilation du script IDL, les classes (les entêtes des opérations et les paramètres) pour développer les applications sont définies, il est donc uniquement nécessaire d'implémenter les opérations dans les classes.

Avec CORBA, le client et le serveur peuvent être implémentés avec différents langages de programmation et systèmes d'exploitation. Cette liberté d'implémentation est obtenue grâce aux squelettes et aux stubs qui sont les responsables de la transformation des requêtes ou des réponses dans un format reconnu par l'ORB.

Pour développer une application, en plus de définir et compiler un script IDL, il est nécessaire d'ajouter les services dont l'application aura besoin pendant son exécution. Les services ont été conçus de façon modulaire, c'est-à-dire que les services peuvent être enlevés ou ajoutés selon les caractéristiques recherchées par le développeur.

Les concepts de base pour l'interopérabilité sont le domaine, les ponts et le protocole GIOP. Toute application développée selon la spécification de CORBA est implantée dans un domaine. La requête, le long de son chemin vers le serveur, peut passer par plusieurs domaines. Des mécanismes pour projeter les références d'objets sur les différents domaines sont pris en compte dans la spécification de CORBA.

Les ponts sont les responsables d'établir la communication entre les domaines. Les ponts font la transformation des données afin de permettre la communication entre les domaines. Le protocole GIOP spécifie la représentation des données, les types de messages et les besoins au niveau transport. GIOP a été conçu comme un protocole abstrait, dont une des implémentations est le protocole IIOP, qui est la projection de GIOP sur TCP/IP.

Bien que CORBA soit la spécification la plus répandue, elle est toujours en évolution, des services continuent à être ajoutés ou modifiés, cela est très important pour tenir compte des besoins qui ne cessent de changer de forme à cause des exigences des nouvelles applications.

Quelques implémentations de CORBA

Pour finir, intéressons-nous quelques instants aux implémentations de CORBA. Les implémentations de la spécification de CORBA sont très nombreuses et majoritairement commerciales. Cependant, les produits gratuits sont de plus en plus performants et MICO est un bon exemple de produit gratuit et puissant. Voici une liste des principales implémentations existantes (ces implémentations prennent en considération quelques aspects de CORBA et pas tous en même temps) :

> AdaORB, BlueORB (Berry Software), Chorus ORB (Sun),Component Borker (IBM), CORBAplus (ExperSoft), CORBASript, DIMMA (ANSA), FnORB (DSTC), GemORB (GemStone), Broker, Internet Servcie Broker, JacORB (Gerald Brose), JavaIDL (JavaSoft), JavaORB (Distributed Objects Group), Jonathan (France-Télécom/CNET), Jorba (Roland Turner), LiveContent Broker (Peerlogic), MICO, Netscape Internet Service Broker (Netscape), OmniORB (AT&T Laboratories), OpenORB, OpenCORBA, OrbAda (Top Graph'X), ORBAcus (Object-Oriented Concepts), ORBexpress (Objective Interface Systems), ORBit-Eiffel, ORBbit-C++, ORBit-Perl, Orber, ORBit (RHAD LabsGNOME Project), Orbix (Iona Technologies), TAO (Distributed Object Computing Group at Washington University), Tatanka (Bionic Buffalo), TIB/ObjectBus (TIBCO), TORB, TPBROKER, UNAS, WebOTX, VisiBroker (Borland), Voyager ORB (ObjectSpace)