Common Object Request Broker Architecture (CORBA)

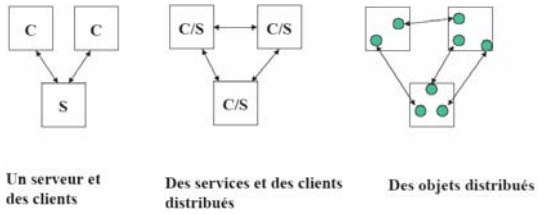
1

De nouveaux besoins

- Développement des réseaux hétérogènes tant sur les architectures que sur les systèmes
- Intégration de logiciels d'origines diverses
  - beaucoup d'approches OO incompatibles
- Accès aux logiciels à l'intérieur et à l'extérieur des sociétés via Internet :
  - gestion d'agences ou succursales pour une entreprise
  - échange clients et fournisseurs (e-commerce par exemple)

2

L'évolution des modèles d'architectures distribuées



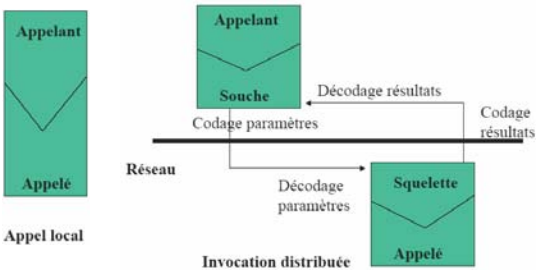
Un serveur et des clients

Des services et des clients distribués

Des objets distribués

3

Les mécanismes de l'invocation distribuée



4

Motivations pour CORBA

- Construire un environnement dont les spécifications sont standardisées
- s'affranchir des solutions purement propriétaires
- l'hétérogénéité et l'interopérabilité entre les différents langages et les environnements informatiques (machines, OS)

Motivations pour l'approche Objet

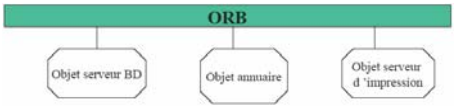
- l'application modélise directement des objets réels du domaine
- obtenir une architecture logicielle claire et simple
- avoir une conception modulaire (masquage des détails d'implantation)
- la réutilisation et portabilité des composants logiciels

➡ **UNE MEME NORME POUR DIVERSES TECHNOLOGIES**

5

Quelques Définitions

- **CORBA :** Common Object Request Broker Architecture
- **ORB** ou bus logiciel : fournit une infrastructure de communication pour des objets hétérogènes et distribués



Objets : entités identifiés qui fournissent des services

- **distribués** : pouvant être accédés à distance au travers d'un réseau
- **hétérogènes** : proviennent de langages et de systèmes différents

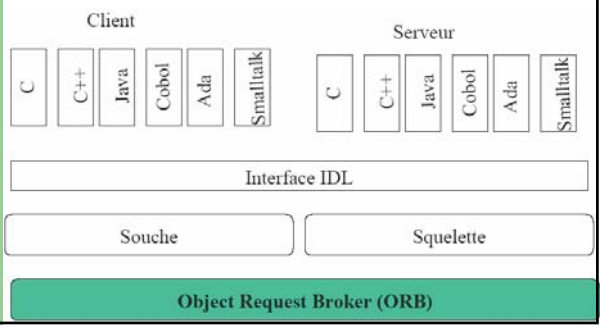
6

Consortium OMG (Object Management Group)

- OMG : organisation non commerciale fondée en 1989 (USA) produit des spécifications
- Aujourd'hui environ 1000 membres :
  - constructeurs (IBM, SUN, HP, INTEL, …)
  - éditeurs (Netscape, MicroSoft, …)
  - utilisateurs (Boeing, Alcatel, NASA)
  - laboratoires (INRIA, CERN, LIFL)
- OMA **:** Object Management Architecture
  - architecture générale pour la gestion d'objets distribués
- CORBA **:** est un des composants de l'OMA
  - permet aux objets distribués de communiquer

7

Vue générale de CORBA



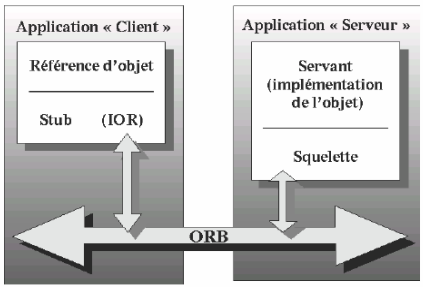
8

IDL/CORBA

- Nécessité d'un langage de description des interfaces contractuelles pour les services et requêtes
- IDL (Interface Description Language) : est le langage de description de toutes les technologies CORBA. Il permet notamment la description des interfaces des objets distribués.
- Projection des interfaces IDL dans le langage des applicatifs.
- Toutes les technologies CORBA sont décrites en IDL.

9

Architecture Corba



10

Invocation statique

- Quand un client invoque une méthode sur l'objet *proxy* (stub) local,
- l'ORB empaquette les paramètres de l'appel ;
- il les envoie au gestionnaire de services (*serveur*);
- le serveur les dépaquette ;
- le serveur invoque la méthode sur le *servant* (en anglais);
- les éventuels paramètres de sortie ou valeurs de retour suivent le chemin inverse.

11

Langage IDL

IDL (*Interface Definition Language*)
- Langage de définition des services proposés par un objet
- Une interface comprend les opérations et les attributs d'un objet
- Langage déclaratif (l'implantation ne se fait pas à ce niveau)

Mapping
- Correspondance entre IDL et un langage de programmation
- 6 mappings normalisés OMG (C, C++, Java, Ada, Cobol, Smalltalk)
- D'autres plus exotiques existent (Tcl, Perl, CLOS, Eiffel, Python, Modula).

IDL de CORBA
- Orienté objet
- Supporte l'héritage
- Dédié à la programmation distribuée

12

Langage IDL

- Structure d'un fichier IDL

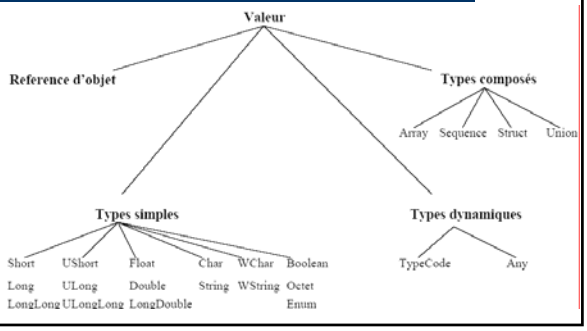
```
<types>
<constantes>
<exceptions>
<modules>
    <interfaces>
        <attributs>
        <operations>
```

```
typedef string Tadresse;
const Tadresse adresseMairie="…";
exception jourFerie{};
module TP {
    interface Etudiant {
        attribute long age;
        boolean present (in long jour)
        raises jourFerie;
        }
}
```

• les modules peuvent être emboîtés les uns dans les autres
• les types, constantes, exceptions peuvent être déclarés à différents niveaux (globalement, localement à un module, localement à une interface).

13

Types de données d'IDL



14

Les types de données dans IDL

Types de base :
• short, unsigned short (2 octets)
• long, unsigned long (4 octets)
• long long, uns. long long (8)
• float (4 octets)
• double (8), long double (16)
• boolean (TRUE, FALSE)
• octet (1 octet, transmis tel quel)
• char (1 octet, ISO Latin 1)
• wchar (2 octets, Unicode)
• void (type vide)

Types construits :
• struct (structures)
• enum (énumération)
• union (type discriminé)
• string, wstring (chaîne)
• sequence (tableau)

Exemple :

```
typedef long type_age;
struct personne {
    string nom;
    type_age age;
};
enum couleur{rouge, vert, bleu};
union carre switch(couleur) {
    case rouge : short num;
    default: char n;
};
string <16> chaine;
string nom;
sequence <long> vecteur;
sequence <float, 8> tableau;
```

15

Les constantes dans IDL

Des variables typées dont la valeur est fixe
const <type> <ident> = <expr>;

Types autorisés pour les constantes
• long, unsigned long
• short, unsigned short
• long long, unsigned long long
• float, double, long double
• boolean
• char, wchar
• string

Exemple

```
const long max=255;
const long taille=8;
const long segments=1024/taille;
const string usage="Hello";
```

Opérateurs autorisés
• unaires + - ~
• binaires + - * / % << >> & | ^

16

Les exceptions dans IDL

- Des structures de données qui signalent une situation exceptionnelle
  - définies par le programmeur
  - levées par le système

Exceptions spécifiées par l'utilisateur

```
exception <ident> { <données>* };
```

Exemple

```
exception overflow {
        float limit;
    };
long exp(in float x)  raises(overflow);
```

Quelques exceptions système
- OBJECT_NOT_EXIST
- BAD_PARAM
- COMM_FAILURE
- NO_MEMORY
- ...

17

Les modules IDL

- Des conteneurs de définitions
- Peuvent contenir : types, constantes, exceptions, interfaces, modules
- permettent de structurer les applications.

```
module <ident> { … }
```

- **Exemple**

```
module mesAnimaux {
    …
        module alaVille { …};
        module alaCampagne { …};
    …
};
```

- Toutes les définitions d'un module sont «visibles» grâce à l'opérateur de résolution de portée **::**
- Ex. mesAnimaux**::**alaVille**::**...

18

Les interfaces IDL

Ce sont des points d'accès aux objets CORBA
(identiques aux interfaces de Java ou aux classes abstraites de C++).

- Peuvent contenir : types, constantes, exceptions, attributs, opérations
- Peuvent hériter leur structure d'une ou de plusieurs autres interfaces
- Contiennent des opérations qui prennent des paramètres et retournent un résultat

  in : entrée **inout** : entrée/sortie **out** : sortie

  interface <ident> : <heritage> { …};

- Exemple

```
interface B {
    void f (in float T);
};
interface A : B { …}; // utilise B
```

19

Les interfaces IDL

- En plus des opérations, les interfaces contiennent aussi des **Attributs** :
  - les variables publiques des interfaces (accessibles par tous les clients)
  - Peuvent être lus ou écrits par les clients (sauf si déclarés readonly)
  - Chaque attribut est associé à deux méthodes (appelées setter/getter) permettant de le lire et de l'écrire (seulement setter pour readonly)
- **attribute** <type> <ident>;
- **readonly attribute** <type> <ident>;

20

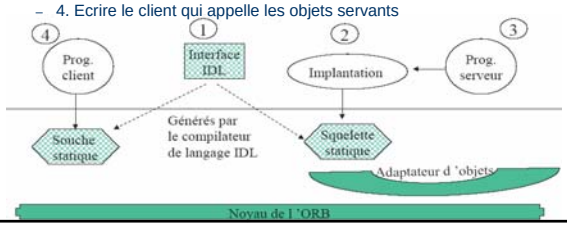
Exemple

```
module GestionBancaire {
    struct FicheClient { string nom; string adresse; } ;
    interface compte {
        attribute FicheClient client;
        readonly attribute float balance;
        void faireDepot (in float f);
        void faireRetrait (in float f);
    };
    interface livret : compte {
        readonly attribute float taux;
    };
    interface banque {
        / exception 'GestionBancaire::banque::refuser' en cas de problemes
        exception refuser {string raison;};
        compte creationCompte (in FicheClient client) raises (refuser);
        livret creationLivret (in FicheClient client) raises (refuser);
        compte retrouverCompte (in string nom) raises (refuser);
        void destructionCompte (in compte a);
    };
} ;
```

21

Développement Java

- **Développement d'une application client/serveur CORBA/Java**
  - 1. Ecrire en IDL les interfaces des objets servants
  - 2. Implanter en Java ces interfaces
  - 3. Ecrire le serveur qui crée les objets servants
  - 4. Ecrire le client qui appelle les objets servants



22

Projection des types

- L 'étape 2 nécessite de connaître la traduction (*mapping*) Java des éléments IDL

IDL	Java
[unsigned] short	short
[unsigned] long	int
[unsigned] long long	long
float	float
double	double
char, wchar	char
string, wstring	Java.lang.String
boolean	boolean
octet	byte
any	org.omg.CORBA.Any
void	void
long double	non supporté

types de base

IDL	Java
module	package
interface	interface
attribute	Méthodes *setter/getter*
opération	Méthode Java
struct, enum, union	Classes Java
sequence	Tableau Java
const	static final
exception	Sous-classe de java.lang.Exception

types construits

23

Projection des interfaces

- Exemple de correspondance IDL - Java

IDL

```
module M {
   interface I {
     long meth (in long arg)
       raises (e);
     attribute float a;
     readonly
        attribute double d;
   }
   exception e { … }
}
```

Java

```
package M;
interface I {
    int meth ( int arg)
      throws e;
    float a();
    void a ( float value);
    double d();
}
class e extends Java.lang.Exception
  { ... }
```

24

Projection des interfaces vers Java

- Pour chaque interface IDL, le traducteur IDL vers Java 1.4 et plus
  - une interface Java <interface>
  - une classe pour le squelette _<interface>Operations
  - une classe pour la souche _<interface>Stub
  - une classe dite Helper <interface>Helper
  - une classe dite Holder <interface>Holder
  - une classe dite Holder <interface>POA
- La classe <interface>Helper permet de :
  - lire et écrire des objets implantant cette interface dans un flux (read() et write())
  - convertir un objet CORBA en type Java (Cast) : narrow()
  - les classes *Helper* existent déjà également pour les types simples (IntHelper, …)
- La classe <interface>Holder gère les modes de passage inout et out (inexistant en Java natif)

25

Projection des interfaces vers Java

- **Depuis java 1.4 :**
  - **Introduction de la notion de POA et du POA Manager** (déjà utilisé dans la projection vers C++) pour une meilleure portabilité des ORBs car le POA est défini dans les spécifications de CORBA.
  - **Le bus ORB doit être lancé** (il intègre le service de noms et d'autres fonctionnalités) : c'est le démon orbd.
- **Remarque** :
  - orbd doit être lancé obligatoirement sur la même machine que le serveur

26

Le POA

- *Adaptateur d'Objets (OA), son rôle* :
  - Module de connexion du serveur sur l'ORB
  - crée ou active un objet suite à une invocation
  - désactive un objet
  - assure la réception des requêtes auprès des objets et informe l'ORB du bon cheminement
  - assure la sécurisation des échanges

27

Le POA

Adaptateur d'Objets (OA), sa fonction principale :

- Fournir une transition entre la notion abstraite d'objet CORBA et la réelle implantation du comportement de l'objet sous la fome d'un servant.
- Chaque servant actif sur l'ORB est rattaché à un POA qui lui fournit un espace de noms (*namespace*).
- Un POA peut gérer plusieurs servants.
- Un serveur doit toujours commencer par se lier à un POA racine, appelé RootPOA.
- Chaque POA est associé à un gestionnaire de POA, *POAManager*.

28

Créer et utiliser un POA

Obtenir la racine du POA

●Obtenir le POA appelé *rootPOA* qui est géré par l'ORB et fourni aux applications.

Exemple de code :

```
ORB orb = ORB.init( args, null );
POA rootPOA = POAHelper.narrow( orb.resolve_initial_references("RootPOA"));
```

● Chaque objet POA est associé à un POA Manager qui contrôle l'état des POAs qu'il gère. Le POA Manager peut avoir les états suivants :

● **État:**
- « ***Holding*** » : les POAs associés stocke les requêtes qui arrivent.
- « ***Active*** » : les POAs associés démarrent le traitement des requêtes.
- « ***Discarding*** » : les POAs associés rejettent les requêtes qui arrivent.
-« ***Inactive*** » : les POAs associés rejettent les requêtes qui n'ont pas commencé à s'exécuter.

● Si le POA Manager n'est pas activé, tous les appels vers le servant seront suspendus car le POA Manager est par défaut à l'état HOLD.

```
// Activer le POAManager
persistentPOA.the_POAManager().activate( );
```

29

Comportement du serveur

La classe du serveur possède une classe main qui doit :

- Créer et initialiser une instance de l'ORB ;
- Obtenir une référence au *rootPOA* et activer le POAManager ;
- Créer une instance pour chaque servant qu'il attache à l'ORB ;
- Obtenir la référence CORBA du servant créé ;
- Obtenir une référence vers le contexte de nommage (qui est un objet CORBA);
- Enregistrer les nouveaux objets dans le contexte de nommage ;
- Attendre les invocations du nouvel objet par les clients.

30

Comportement du client

- Le client doit :
  - Créer et initialiser l'ORB ;
  - Obtenir une référence au contexte de nommage ;
  - Demander la référence de l'objet au service de nommage : la référence obtenue est une référence CORBA ;
  - Invoquer les méthodes de l'objet recherché et
  - Fermer l'ORB.

31

Projection en Java : Exemple

- **Construction d'une interface IDL** qui sera partagée par le client et le serveur

```
// fichier specification.idl
module ReverseApp
   {     interface Reverse
                  {   string reverseString(in string chaineOrigine);
                  };
   };
```

- En utilisant JDK1.4 ou plus, on peut générer les squelettes et les souches avec la commande suivante :

  idlj –fall specification.idl

32

Exemple (suite)

Un sous-répertoire pour le package **ReverseApp** est généré : il correspond au module de même nom.
Les fichiers générés dans le répertoire ReverseApp par le compilateur idlj sont :
- **Reverse.java** : interface Java de l'interface IDL dérivant elle-même de l'interface ReverseOperations, de CORBA.Object et de IDLEntity.
- **ReverseHelper.java** : Classe qui regroupe des méthodes d'utilisation (lect/écrit, conversion) des objets distribués;
- **ReverseHolder.java** : Classe "enveloppe" pour le passage de paramètres
- **ReverseOperations.java** : contient toutes les opérations définies dans l'interface IDL. Dans cet exemple, elle contient la méthode reverseString().
-**ReversePOA.java** : squelette de l'objet Reverse utilisé pour implémenter le service, par dérivation.
- _**ReverseStub.java** : souche de l'objet Reverse, représentant local de l'objet côté client.

33

Exemple (suite)

Ecrire le serveur : ReverseServer.java
Ecrire le servant : ReverseServant.java
Écrire le client : ReverseClient.java

- **Compilation du serveur**

  idlj –fall specification.idl
  javac ReverseServer.java ReverseServant.java ReverseApp/*.java
- **Compilation du Client**

  idlj –fall specification.idl
  javac Reverseclient.java ReverseApp/*.java
- **Lancement du serveur orbd**

  orbd –ORBInitialPort 1500&

34

Exemple (suite)

- **Lancement de l'application serveur :**

  java ReverseServer –ORBInitialPort 1500&
- **Lancement de l'application cliente :**

java ReverseClient paris –ORBInitialHost localhost -ORBInitialPort 1500

Résultat après exécution:

« La chaine inversée de paris est : sirap »

35