

# Introduction aux systèmes et applications répartis

---

**Sacha Krakowiak**  
Université Joseph Fourier  
Projet Sardes (INRIA et IMAG-LSR)

<http://sardes.inrialpes.fr/~krakowia>

Voir aussi Bureau Virtuel et placard

# Pourquoi des applications réparties ?

---

La **répartition** est un état de fait pour un nombre important d'applications

## ■ Besoins propres des applications

- ◆ Intégration d'applications existantes initialement séparées
- ◆ Intégration massive de ressources
  - ❖ Grilles de calcul, gestion de données
- ◆ Pénétration de l'informatique dans des domaines nouveaux d'application
  - ❖ Intégration d'objets du monde réel (informatique omniprésente (*ubiquitous computing*))
    - ▲ Surveillance et commande d'installations

## ■ Possibilités techniques

- ◆ Coût et performances des machines et des communications
- ◆ Interconnexion généralisée
  - ❖ Exemple 1 : interpénétration informatique-télécom-télévision
  - ❖ Exemple 2 : Réseaux de capteurs

# Caractéristiques des systèmes répartis (1)

---

---

## ■ Définition d'un système réparti

- ◆ Ensemble composé d'**éléments** reliés par un **système de communication** ; les éléments ont des fonctions de traitement (processeurs), de stockage (mémoire), de relation avec le monde extérieur (capteurs, actionneurs)
- ◆ Les différents éléments du système ne fonctionnent pas indépendamment mais collaborent à une ou plusieurs tâches communes. Conséquence : **une partie au moins de l'état global du système est partagée** entre plusieurs éléments (sinon, on aurait un fonctionnement indépendant)

De manière plus précise : toute expression de la spécification du système fait intervenir plusieurs éléments (exemple : préserver un invariant global, mettre des interfaces en correspondance, etc.)

# Caractéristiques des systèmes répartis (2)

---

---

## ■ Propriétés souhaitées

- ◆ Le système doit pouvoir fonctionner (au moins de façon dégradée) même en cas de défaillance de certains de ses éléments
- ◆ Le système doit pouvoir résister à des perturbations du système de communication (perte de messages, déconnexion temporaire, performances dégradées)
- ◆ Le système doit pouvoir résister à des attaques contre sa sécurité (violation de la confidentialité, de l'intégrité, usage indu de ressources, déni de service)

## ■ Conséquences

- ◆ Des décisions doivent pouvoir être prises localement, et dans une situation d'incertitude, ( sans connaissance d'un état global, d'ailleurs difficile à définir)

# Caractéristiques des systèmes répartis (3)

---

## ■ Difficultés

- ◆ Propriété d'asynchronisme du système de communication (pas de borne supérieure stricte pour le temps de transmission d'un message)
  - ❖ Conséquence : difficulté pour détecter les défaillances
- ◆ Dynamisme (la composition du système change en permanence)
  - ❖ Conséquences : difficulté pour définir un état global
  - ❖ Difficulté pour administrer le système
- ◆ Grande taille (nombre de composants, d'utilisateurs, dispersion géographique)
  - ❖ Conséquence : la capacité de croissance (*scalability*) est une propriété importante, mais difficile à réaliser

Malgré ces difficultés, des grands systèmes répartis existent et sont largement utilisés

le DNS (*Domain Name System*)

le World Wide Web

# Systemes et applications repartis

---

---

## ■ Distinction entre “systeme” et “application”

- ◆ **Systeme** : gestion des ressources communes et de l'infrastructure, lie de maniere etroite au materiel sous-jacent
  - ❖ **Systeme d'exploitation** : gestion de chaque element
  - ❖ **Systeme de communication** : echange d'information entre les elements
  - ❖ **Caracteristiques communes** : cachent la complexite du materiel et des communications, fournissent des services communs de plus haut niveau d'abstraction
- ◆ **Application** : reponse a un probleme specifique, fourniture de **services** a ses utilisateurs (qui peuvent etre d'autres applications). Utilise les services generaux fournis par le systeme
- ◆ La distinction n'est pas toujours evidente, car certaines applications peuvent directement travailler a bas niveau (au contact du materiel). Exemple : systemes embarques, reseaux de capteurs

# Services et interfaces

---

---

## ■ Définition

- ◆ Un système est un ensemble de composants (au sens non technique du terme) qui interagissent
- ◆ Un service est “un comportement défini par contrat, qui peut être implémenté et fourni par un composant pour être utilisé par un autre composant, sur la base exclusive du contrat” (\*)

## ■ Mise en œuvre

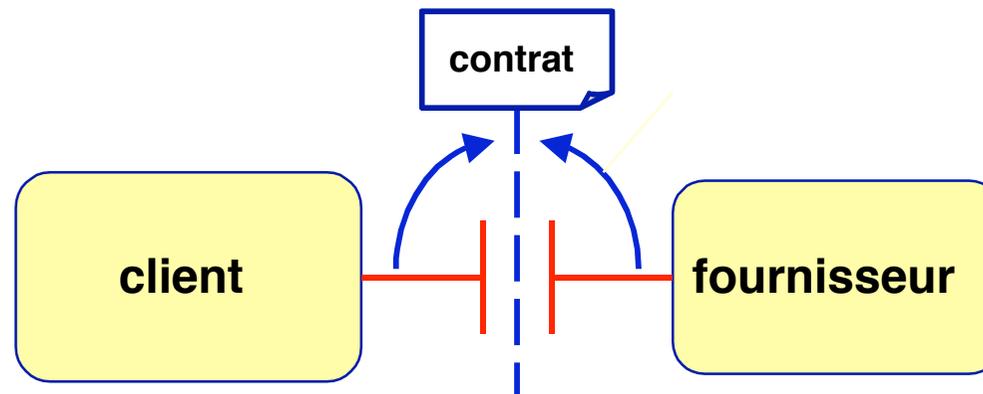
- ◆ Un service est accessible via une ou plusieurs interfaces
- ◆ Une interface décrit l'interaction entre client et fournisseur du service
  - ❖ Point de vue opérationnel : définition des opérations et structures de données qui concourent à la réalisation du service
  - ❖ Point de vue contractuel : définition du contrat entre client et fournisseur

(\*) Bieber and Carpenter, *Introduction to Service-Oriented Programming*, <http://www.openwings.org>

# Définitions d'interfaces (1)

---

---



- ◆ La fourniture d'un service met en jeu **deux** interfaces
  - ❖ Interface requise (côté client)
  - ❖ Interface fournie (côté fournisseur )
- ◆ Le contrat spécifie la compatibilité (conformité) entre ces interfaces
  - ❖ Au delà de l'interface, chaque partie est une "boîte noire" pour l'autre (principe d'encapsulation)
  - ❖ Conséquence : client ou fournisseur peuvent être remplacés du moment que le composant remplaçant respecte le contrat (est conforme)

# Définitions d'interfaces (2)

---

---

## ■ Partie “opérationnelle”

### ◆ Interface Definition Language (IDL)

#### ❖ Pas de standard, mais s'appuie sur un langage existant

- ▲ IDL CORBA sur C++
- ▲ Java et C# définissent leur propre IDL

## ■ Partie “contractuelle”

### ◆ Plusieurs niveaux de contrats

- ❖ Sur la forme : spécification de types -> conformité syntaxique
- ❖ Sur le comportement (1 méthode) : assertions -> conformité sémantique
- ❖ Sur les interactions entre méthodes : synchronisation
- ❖ Sur les aspects non fonctionnels (performances, etc.) : contrats de QoS

# Quelques classes d'applications réparties

---

---

## ■ Coordination d'activités

- ◆ Systèmes à flots de données (“workflow”)
- ◆ Systèmes à “agents”

## ■ Communication et partage d'information

- ◆ Bibliothèques virtuelles

## ■ Collecticiels

- ◆ Édition coopérative
- ◆ Téléconférence
- ◆ Ingénierie concourante

## ■ Applications “temps réel”

- ◆ Contrôle et surveillance de procédés et d'installations
- ◆ Avionique, etc.
- ◆ Localisation de mobiles
- ◆ Réseaux de capteurs

## ■ Nouveaux services grand public

- ◆ Presse électronique
- ◆ Télévision interactive
- ◆ Commerce électronique

# Organisation des applications réparties

---

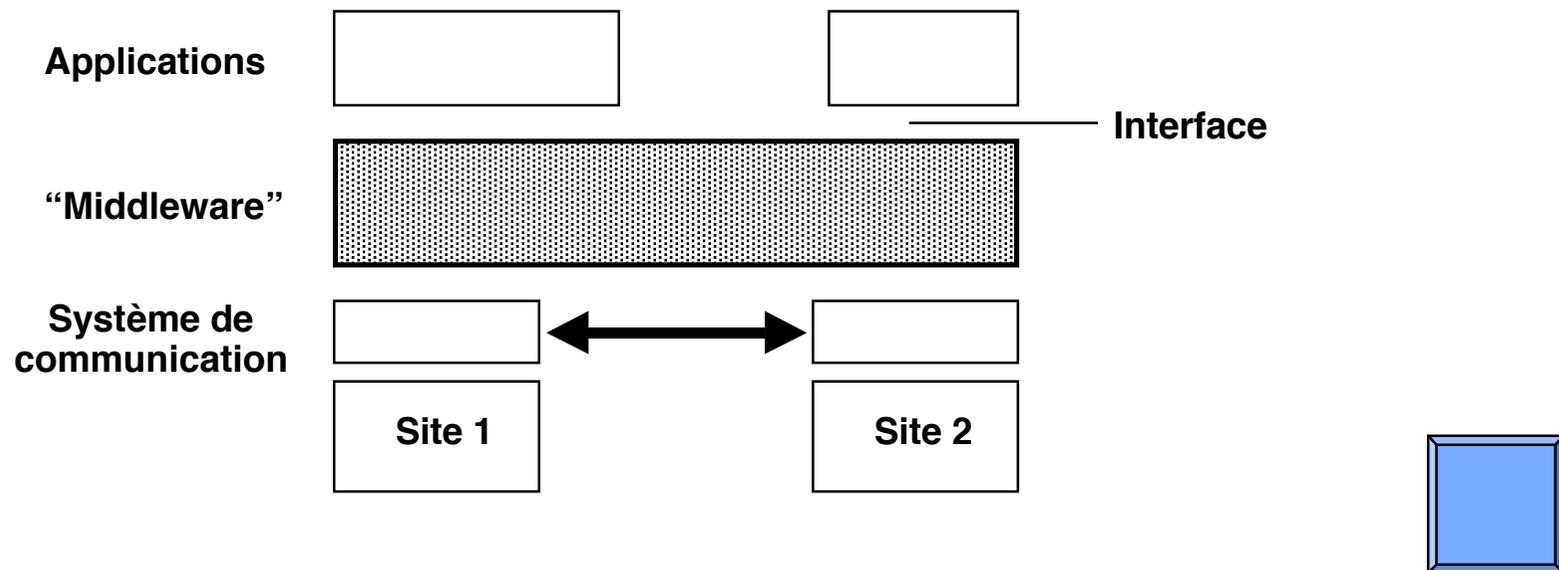
---

- **Client-serveur**
  - ◆ Exécution “synchrone” requête-réponse ; base : RPC
  - ◆ Extensions : serveurs coopérants ; service discontinu ; ...
- **Objets partagés** (organisations diverses)
- **Flots de communication**
  - ◆ Discrets (messages) ou continus (multimédia)
- **Code mobile**
  - ◆ Machine virtuelle (masque l’hétérogénéité)
  - ◆ Problèmes de sécurité
- **“Agents”**
  - ◆ Programme agissant pour le compte d’une entité cliente
  - ◆ Agents fixes ou mobiles, statiques ou adaptatifs
  - ◆ Coopération entre agents

# Organisation des applications réparties

## Un schéma commun : le “middleware” (intergiciel)

- **“Middleware” : couche de logiciel (réparti) destinée à**
  - ◆ masquer l’**hétérogénéité** des machines et systèmes
  - ◆ masquer la **répartition** des traitements et données
  - ◆ fournir une **interface** commode aux applications (modèle de programmation + API)



# Organisation des applications réparties

## Importance de la normalisation

---

---

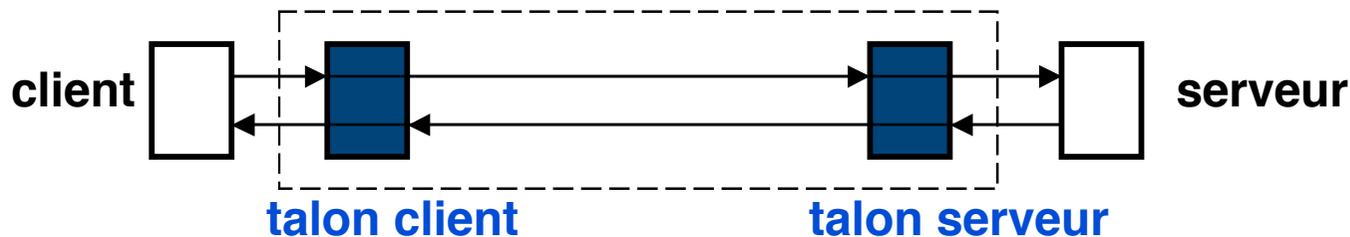
- **Le développement du “middleware” impose une normalisation des interfaces**
  - ◆ Logiciel de base
  - ◆ Domaines spécifiques d'applications
  
- **Nombreux consortiums et standards**
  - ◆ Open Group (ex-OSF) : systèmes, outils de base
  - ◆ Web Consortium (W3C) : Web et outils associés
  - ◆ OMG : objets répartis (CORBA, IIOP, etc.)
  - ◆ ODMG : bases de données à objets
  - ◆ ODP : organisation “ouverte” des applications
  - ◆ Workflow Management Coalition : applications à flots de données
  - ◆ ...

# Construction d'applications réparties

## Un outil de structuration : le mandataire (*proxy*)

---

- ◆ **Initialement** : réalisation de l'appel de procédure à distance

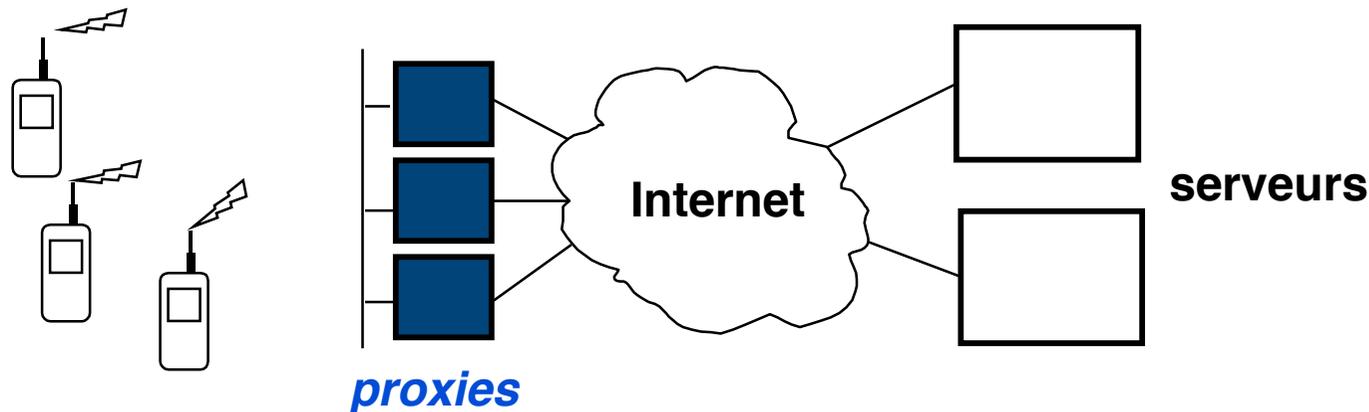


- ◆ **Fonction** : réalisation de la communication
  - synchronisation
  - adaptation de protocole
  - conversion de données
- ◆ **Évolution ultérieure**
  - fonctions propres à l'application
  - gestion d'objets
  - sécurité
  - adaptation à des conditions variables de fonctionnement

# Utilisation de *proxy* pour l'adaptation : applications pour clients légers mobiles

**Objectif** : porter une application répartie complexe sur des clients "légers" (PDAs)

**Méthode** : reporter les fonctions coûteuses dans un ensemble de *proxies*



**Fonctions des *proxies* :**

filtrage et compression (avec perte) des images

filtrage de texte (HTML)

agrégation de réponses aux requêtes

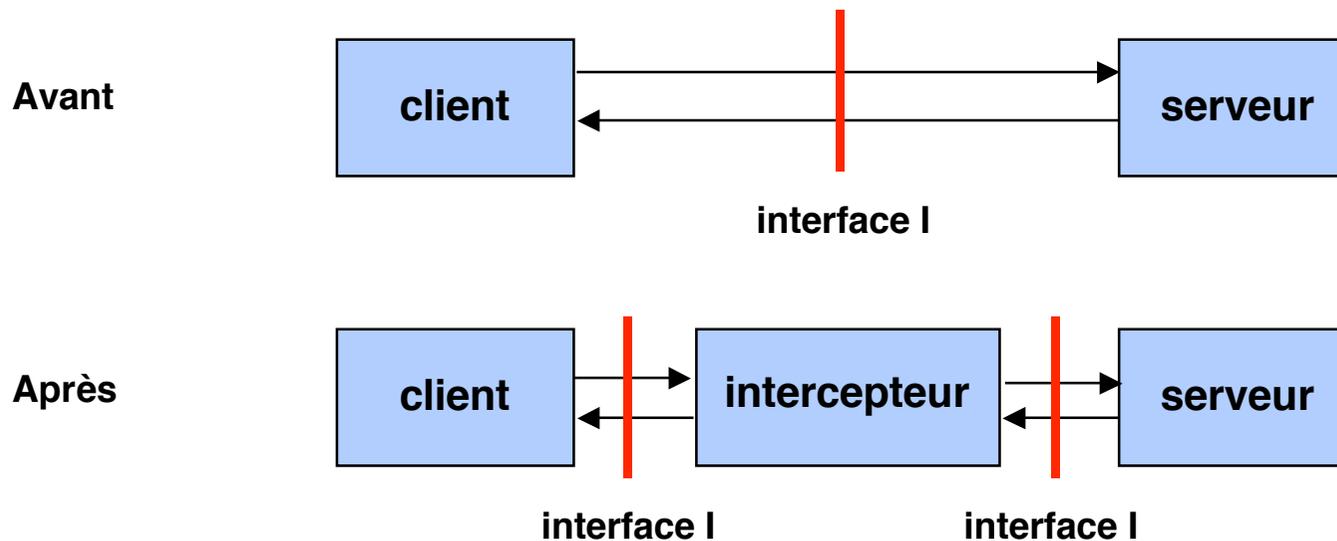
gestion de caches

**Programmation indépendante du serveur, adaptée aux caractéristiques des clients**

# Une autre construction de base : l'intercepteur

## ■ Fonction

- ◆ Interposition d'un traitement entre client et serveur
- ◆ Propriété : doit être "transparent (ne pas modifier l'interface)"



**Effets possibles : modifier requête, modifier réponse, ajouter traitement, rediriger requête, prélever mesures, etc.**

# Plan du cours

---

---

## Première partie : Outils de construction d'applications réparties

- Rappels (client-serveur, objets répartis, RMI)
- CORBA, principes, utilisation, fonctionnement, services
- Programmation par événements, bus logiciels, MOM
- Composants : J2EE, EJB, composants CORBA, ...
- Coordination de services : *Web services*

## Deuxième partie : Services système

- Tolérance aux fautes : client-serveur fiable, techniques de groupe
- Sécurité : confidentialité, authentification, pare-feux, code mobile
- Gestion répartie de données : principes, exemples (SGF répartis, P2P)

## Pratique :

TD (4 séances)

Expérimentation avec CORBA (2 séances)

Projet ECOM (projet d'intégration)

# Travaux Dirigés et Travaux Pratiques

---

---

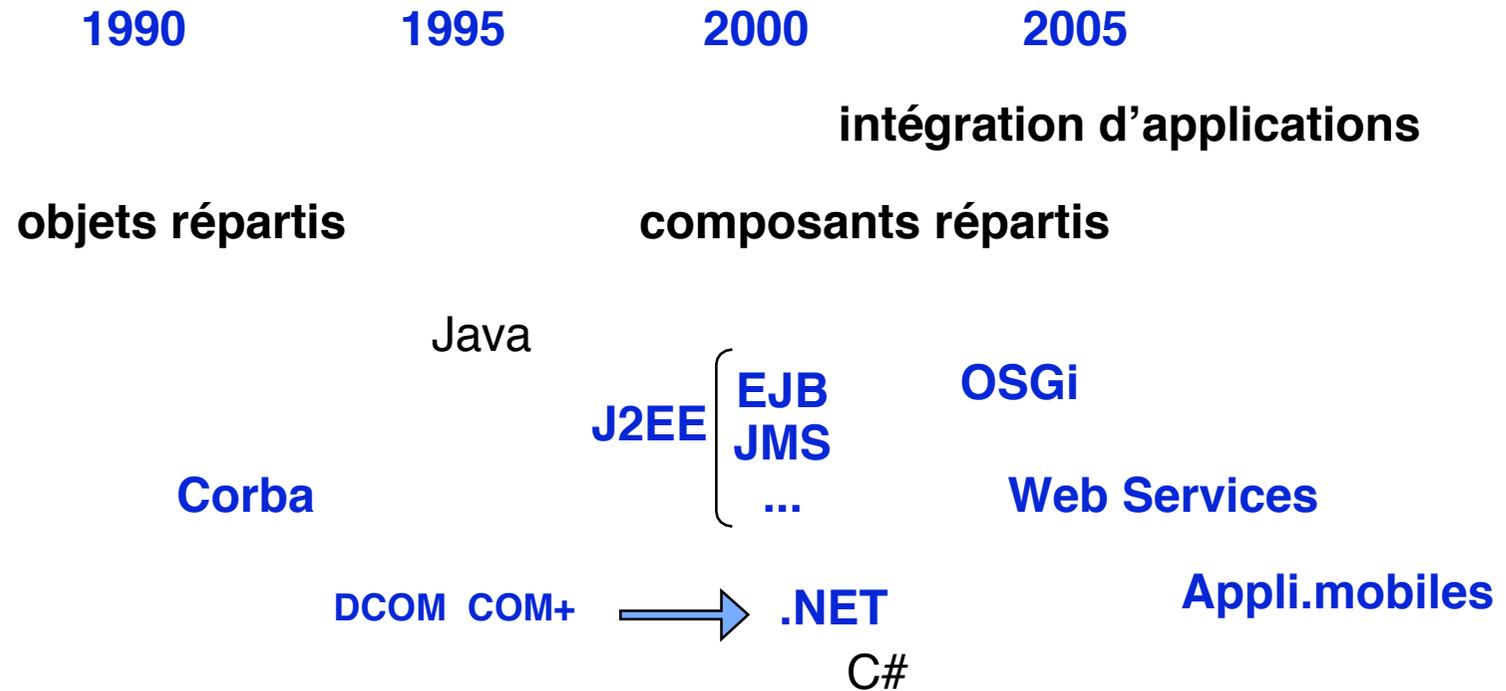


- **TD-1 : Communication asynchrone, messages**
  - ◆ Applications : messagerie, intégration d'applications
- **TD-2 : Tolérance aux fautes**
  - ◆ Applications : estimations de disponibilité, protocoles de groupe et duplication
- **TD-3 : Sécurité**
  - ◆ Application : vote sur l'Internet (spécifications, mise en œuvre)
- **TD-4 : Données réparties**
  - ◆ Applications : systèmes P2P, recherche de services
- **TP 1 et 2**
  - ◆ Développement d'une application en CORBA

# Sur le choix des outils

---

---

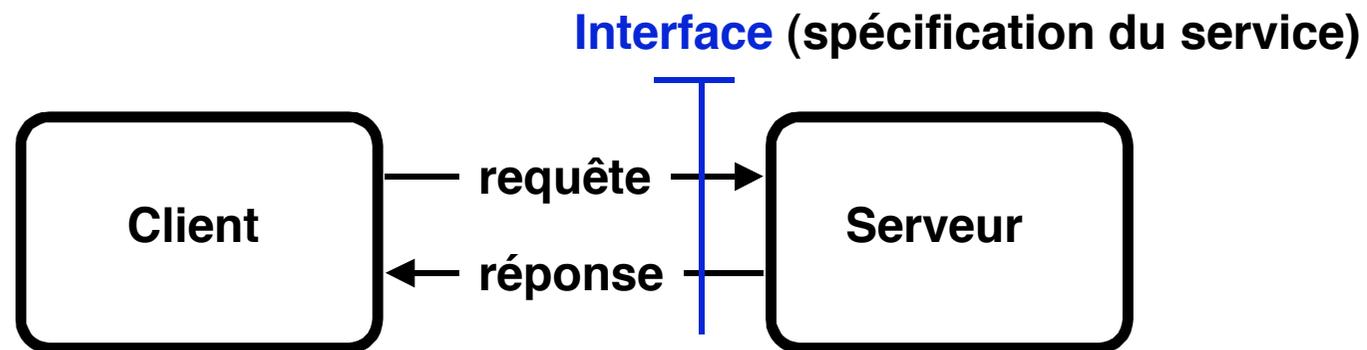


# Rappels sur le modèle Client-Serveur

---

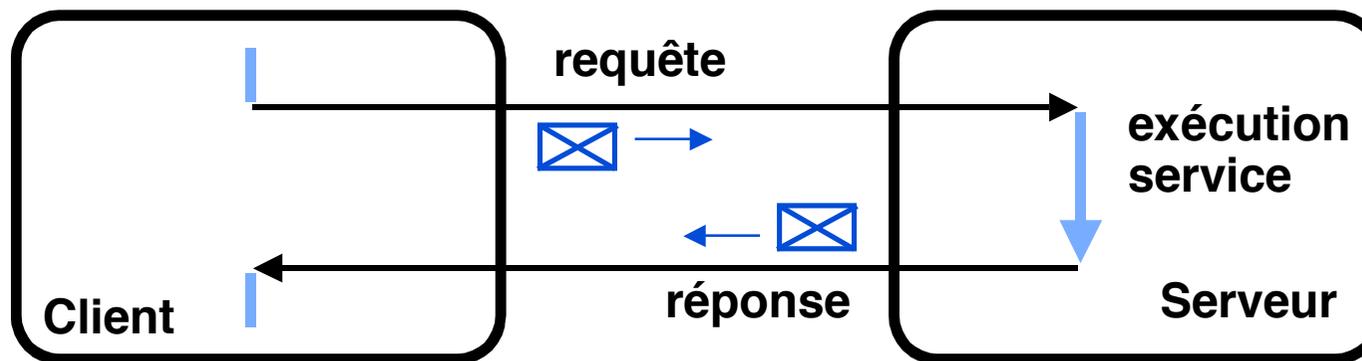
---

- Le *client* demande l'exécution d'un service
- Le *serveur* réalise le service
- Client et serveur sont (en général, pas nécessairement) localisés sur deux machines distinctes
- Indépendance interface-réalisation



# Modèle Client-Serveur : fonctionnement

- **Communication par messages** (plutôt que par partage de données, mémoire ou fichiers)
  - ◆ Requête : paramètres d'appel, spécification du service requis
  - ◆ Réponse : résultats, indicateur éventuel d'exécution ou d'erreur
  - ◆ Communication *synchrone* (dans le modèle de base) : le client est bloqué en attente de la réponse



# Modèle Client-Serveur : Intérêt

---

---

## ■ Structuration

- ◆ fonctions bien identifiées
- ◆ séparation interface du service - réalisation
- ◆ client et serveur peuvent être modifiés (remplacés) indépendamment

## ■ Protection

- ◆ client et serveur s'exécutent dans des domaines de protection différents

## ■ Gestion des ressources

- ◆ le serveur peut être partagé entre de nombreux clients
- ◆ En contrepartie, il doit assurer la gestion des ressources partagées

**ces considérations sont indépendantes de la répartition**

# Modèle Client-Serveur

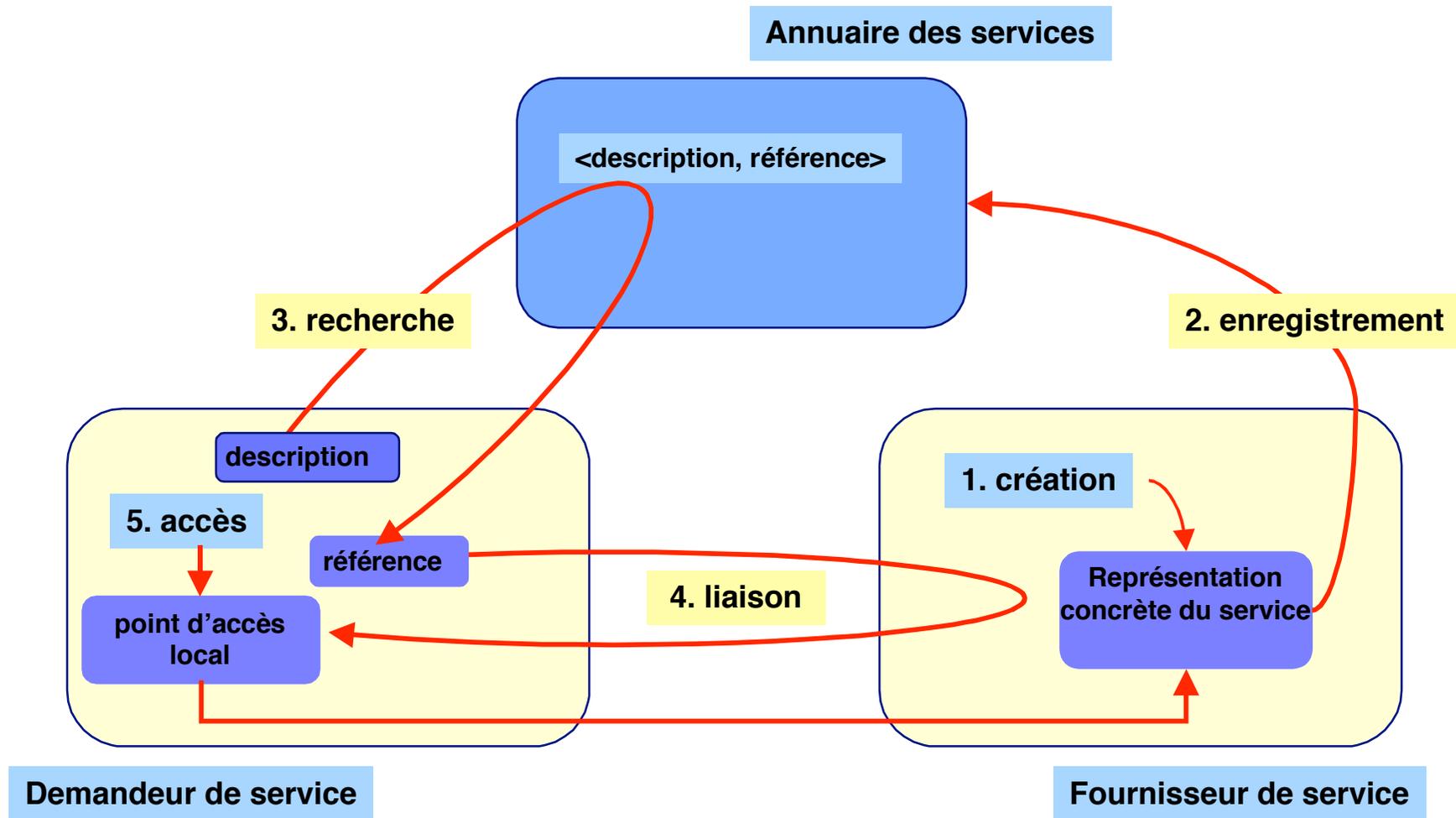
## Exemples

---

---

- **Serveur de fichiers (AFS, NFS)**
- **Serveur d'impression (lpd)**
- **Serveur de calcul**
- **Serveur d'application (spécifique à l'application)**
- **Serveur de bases de données**
- **Serveur de temps**
- **Serveur de noms (annuaire des services)**

# Accès à un service



# Mise en œuvre du schéma client-serveur

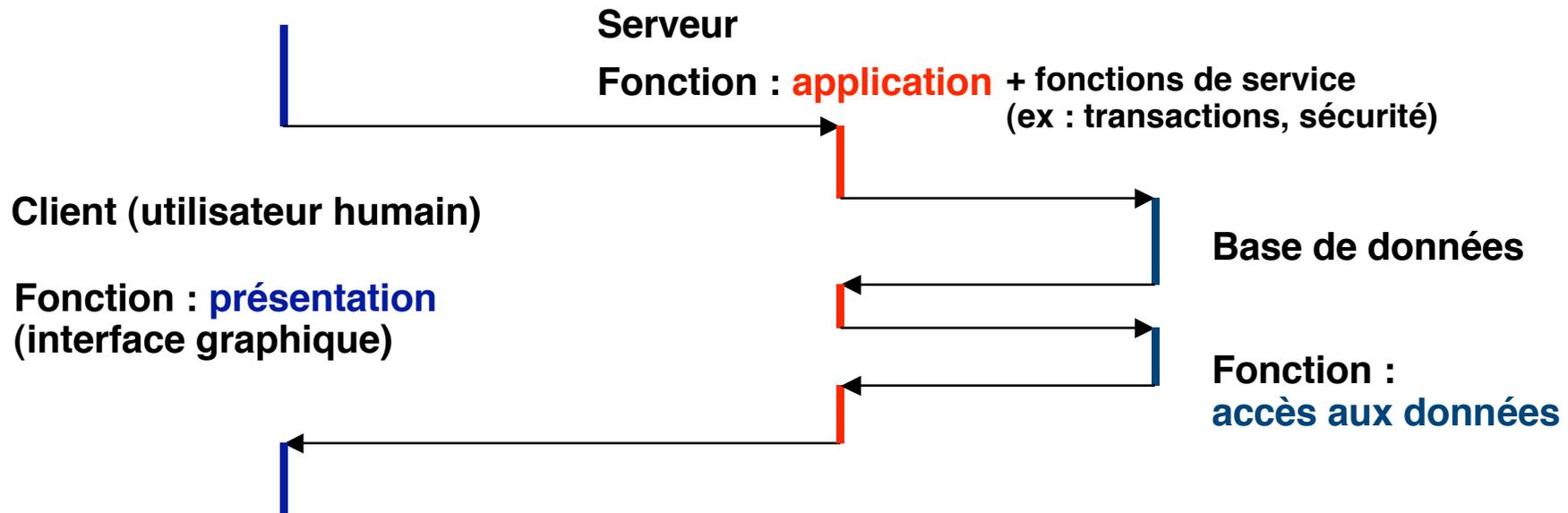
---

- **Par des opérations de “bas niveau”**
  - ◆ Utilisation de primitives du système de communication
  - ◆ Exemple : *sockets*
    - ❖ Mode non connecté (UDP)
    - ❖ Mode connecté (TCP)
  
- **Par des opérations de “haut niveau”**
  - ◆ Utilisation d'un *middleware* spécialisé
  - ◆ Contexte : langage de programmation
    - ❖ Appel de procédure à distance
  - ◆ Contexte : objets répartis
    - ❖ Appel de méthodes, création d'objets à distance

# Généralisations du schéma client-serveur (1)

## ■ Les notions de client et de serveur sont relatives

- ◆ Un serveur peut faire appel à d'autres serveurs dont il est client
- ◆ Exemple usuel : traitement utilisant une base de données



Ce cas sera traité en détail plus tard  
Architecture à 3 niveaux (3-tier architecture)

# Généralisation du schéma client-serveur (2)

---

---

## ■ Clients et serveurs jouent un rôle symétrique

- ◆ Tout site joue le rôle de serveur pour les autres
- ◆ Certaines fonctions (service de noms) peuvent éventuellement être centralisées

## ■ Systèmes pair à pair (*Peer to Peer*, P2P)

- ◆ Utilistion : partage de données à grande échelle
- ◆ Initialement : partage (illégal) de fichiers (Napster, etc.)

Ce cas sera traité en détail plus tard

# Construction d'applications réparties à base d'objets



**S. Krakowiak**

**Université Joseph Fourier**

**Projet Sardes (INRIA et IMAG-LSR)**

**<http://sardes.inrialpes.fr/people/krakowia>**

# Plan

---

---

- **Introduction aux objets répartis**
  - ◆ Rappel rapide sur Java RMI
- **Présentation générale de CORBA**
  - ◆ Motivations et historique
  - ◆ Architecture générale de CORBA
    - ❖ Composants de l'architecture
    - ❖ Désignation des objets et mécanismes d'appel
    - ❖ Langage de description d'interfaces
- **Exemple détaillé de construction d'une application avec CORBA**
- **Aspects techniques**
  - ◆ Appel dynamique et référentiel des interfaces
  - ◆ Références d'objets
  - ◆ Adaptateur d'objets et référentiel des implémentations
- **Services**

# Rappel des conclusions sur l'appel de procédure à distance

---

## ■ Avantages

- ◆ Abstraction (les détails de la communication sont cachés)
- ◆ Intégration dans un langage : facilite portabilité, mise au point
- ◆ Outils de génération, facilitent la mise en œuvre

## ■ Limitations

- ◆ La structure de l'application est statique : pas de création dynamique de serveur, pas de possibilité de redéploiement entre sites
- ◆ Pas de passage des paramètres par référence
- ◆ La communication est réduite à un schéma synchrone
- ◆ La persistance des données n'est pas assurée (il faut la réaliser explicitement par sauvegarde des données dans des fichiers)
- ◆ Des mécanismes plus évolués visent à remédier à ces limitations
  - ❖ Objets répartis (ex : Java RMI, CORBA) pour les 2 premières
  - ❖ Bus à messages (*Message Oriented Middleware*)
  - ❖ Composants répartis (ex : EJB, Corba CCM, .Net)

# Intérêt des objets pour la construction d'applications réparties

---

---

## ■ Encapsulation

- ◆ L'interface (méthodes + attributs) est la seule voie d'accès à l'état interne, non directement accessible

## ■ Classes et instances

- ◆ Mécanismes de génération d'exemplaires conformes à un même modèle

## ■ Héritage

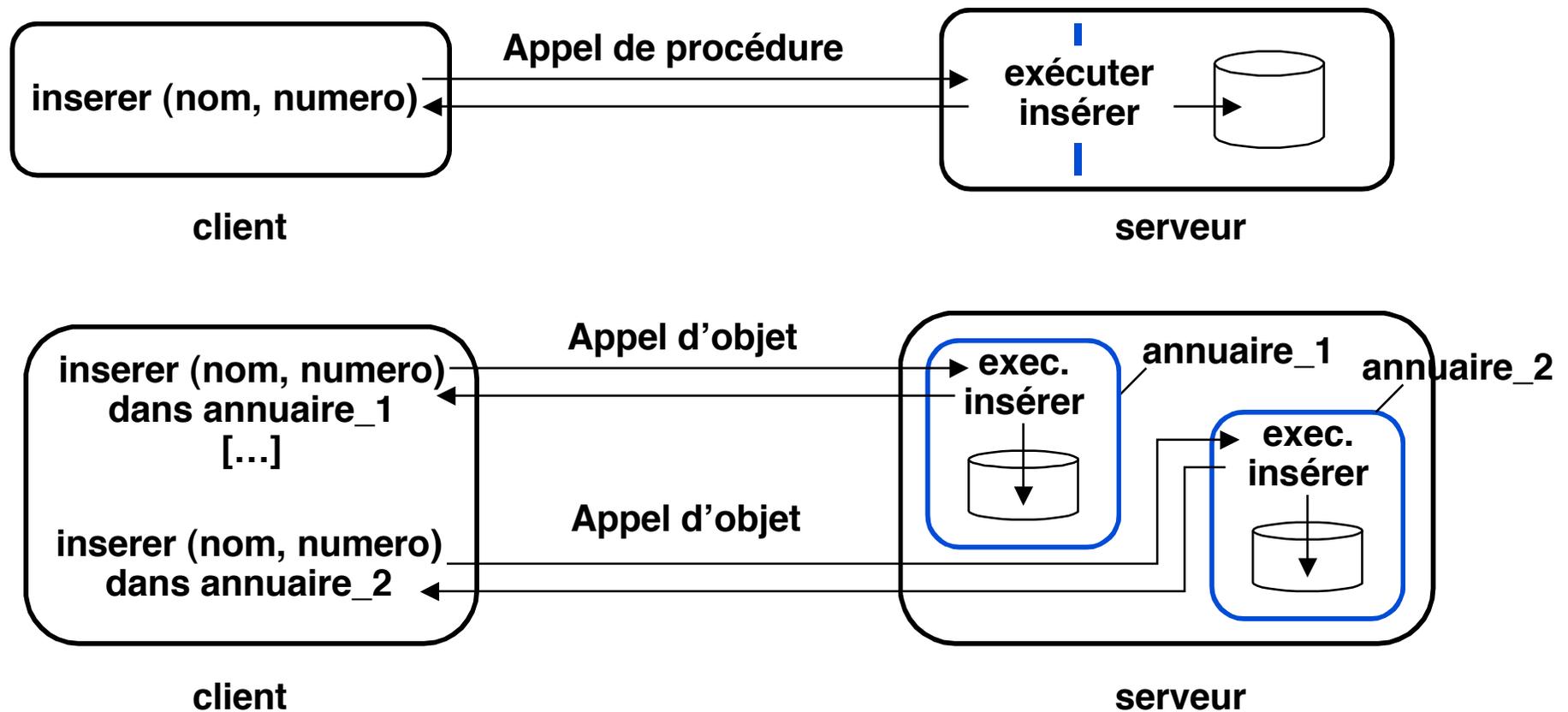
- ◆ Mécanisme de spécialisation : facilite récupération et réutilisation de l'existant

## ■ Polymorphisme

- ◆ Mises en œuvre diverses des fonctions d'une interface
- ◆ Remplacement d'un objet par un autre si interfaces "compatibles"
- ◆ Facilite l'évolution et l'adaptation des applications

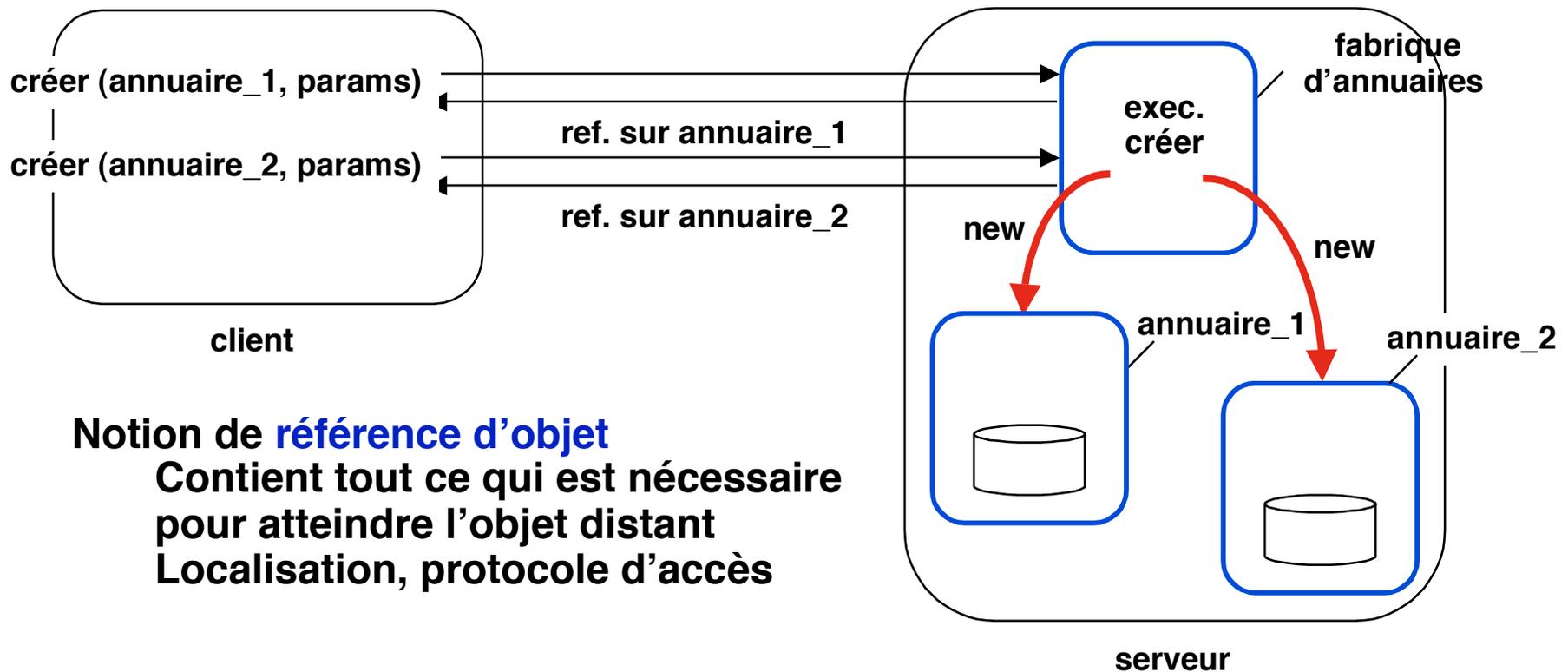
# Extension du RPC aux objets (1)

- Appel de procédure vs appel de méthode sur un objet
  - ◆ Exemple : insérer une entrée dans un annuaire



## Extension du RPC aux objets (2)

- Phase préalable : création d'instances d'une classe d'objects
  - ◆ Notion de fabrique (factory)



Notion de **référence d'objet**  
Contient tout ce qui est nécessaire  
pour atteindre l'objet distant  
Localisation, protocole d'accès

# Références d'objet

---

## ■ Notion clé pour les objets répartis

- ◆ Référence = moyen d'accès à un objet distant
- ◆ En général une référence est "opaque" (son contenu ne peut pas être directement exploité) ; elle n'est utilisable que via un mécanisme d'appel à distance

## ■ Contenu d'une référence

- ◆ Toutes les informations nécessaires pour atteindre physiquement l'objet
  - ❖ Site de résidence
  - ❖ Numéro de porte sur le site
  - ❖ Localisation interne au serveur
  - ❖ ...

## ■ Exemples (cf détails plus loin)

- ◆ En Java RMI : la référence est simplement le talon client (*stub*)
- ◆ En CORBA : format de référence spécifié (IOR : *Interoperable Object Reference*)
- ◆ En DCOM : format "propriétaire"

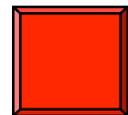
# Java RMI (*Remote Method Invocation*)

---

---

- **Motivation : construction d'applications réparties avec Java**
  - ◆ Appel de méthode au lieu d'appel de procédure
- **Principe : même schéma que RPC**
  - ◆ Le programmeur fournit
    - ❖ Une (ou plusieurs) description(s) d'interface
      - ▲ Ici pas d'IDL séparé : Java sert d'IDL
    - ❖ Le programme du serveur
      - ▲ Objets réalisant les interfaces
      - ▲ Serveur
    - ❖ Le programme du client
  - ◆ L'environnement Java fournit
    - ❖ Un générateur de talons (rmic)
    - ❖ Un service de noms (Object Registry)

voir <http://java.sun.com/docs/books/tutorial/rmi/>



# Java RMI : règles d'usage (1/2)

---

---

## ■ Interface

- ◆ L'interface d'un objet distant (**Remote**) est celle d'un objet Java, avec quelques règles d'usage :
- ◆ L'interface distante doit être publique
- ◆ L'interface distante doit étendre l'interface **java.rmi.Remote**
- ◆ Chaque méthode doit déclarer au moins l'exception **java.rmi.RemoteException**

## ■ Passage d'objets en paramètre

- ◆ Les objets **locaux** sont passés **par valeur** (copie) et doivent être sérialisables (étendent l'interface **java.io.Serializable**)
- ◆ Les objets **distants** sont passés **par référence** et sont désignés par leur interface

## Java RMI : règles d'usage (2/2)

---

---

### ■ Réalisation des classes distantes (Remote)

- ◆ Une classe distante doit implémenter une interface elle-même distante (**Remote**)
- ◆ Une classe distante doit étendre la classe **java.rmi.server.UnicastRemoteObject** (d'autres possibilités existent)
- ◆ Une classe distante peut aussi avoir des méthodes appelables seulement localement (ne font pas partie de son interface Remote)

# Java RMI : règles d'écriture du serveur

---

---

- **Un serveur est une classe qui implémente l'interface de l'objet distant**
  - ◆ **Spécifier les références distantes qui doivent être implémentées (objets passés en paramètres)**
  - ◆ **Définir le constructeur de l'objet distant**
  - ◆ **Fournir la réalisation des méthodes appelables à distance**
  - ◆ **Créer et installer le gestionnaire de sécurité**
  - ◆ **Créer au moins une instance de la classe serveur**
  - ◆ **Enregistrer au moins une instance dans le serveur de noms**

# Java RMI : exemple (*Hello world*)

## Définition d'interface

```
import java.rmi.*;
public interface HelloInterface
    extends Remote {

    /* méthode qui imprime un message
       prédéfini dans l'objet appelé */

    public String sayHello ()
        throws java.rmi.RemoteException;
}
```

## Classe réalisant l'interface

```
import java.rmi.*;
import java.rmi.server.*;
public class Hello
    extends java.rmi.server.UnicastRemoteObject
    implements HelloInterface {
    private String message;

    /* le constructeur */
    public Hello (String s)
        throws RemoteException
    {
        message = s ;
    };

    /* l'implémentation de la méthode */
    public String sayHello ()
        throws RemoteException
    {
        return message ;
    };
}
```

# Java RMI : exemple (Hello world)

## Programme du client

```
import java.rmi.*;
public class HelloClient {
    public static void main (String [ ] argv) {

        /* lancer SecurityManager */

        System.setSecurityManager (
            new RMISecurityManager ());

        try {

            /* trouver une référence vers l'objet distant */
            HelloInterface hello =
                (HelloInterface) Naming.lookup
                    ("rmi://goedel.imag.fr/Hello1");

            /* appel de méthode à distance */
            System.out.println (hello.sayHello());

        } catch (Exception e) {
            System.out.println
                ("Erreur client : " + e);
        }
    }
}
```

## Programme du serveur

```
import java.rmi.*;
public class HelloServer {
    public static void main (String [ ] argv) {

        /* lancer SecurityManager */

        System.setSecurityManager (
            new RMISecurityManager ());

        try {

            /* créer une instance de la classe Hello et
            l'enregistrer dans le serveur de noms */
            Naming.rebind ("Hello1",
                new Hello ("Hello world !"));
            System.out.println ("Serveur prêt.");

        } catch (Exception e) {
            System.out.println
                ("Erreur serveur : " + e);
        }
    }
}
```

# Java RMI : Étapes de la mise en œuvre (1/2)

---

---

## ■ Compilation

- ◆ Sur la machine serveur : compiler les interfaces et les programmes du serveur

```
javac HelloInterface.java Hello.java HelloServer.java
```

- ◆ Sur la machine serveur : créer les talons client et serveur pour les objets appelés à distance (à partir de leurs interfaces) - ici une seule classe, **Hello**

```
rmic -keep Hello
```

N.B. cette commande construit et compile les talons client **Hello\_Stub.java** et serveur **Hello\_Skel.java**. L'option **-keep** permet de garder les sources de ces talons

- ◆ Sur la machine client : compiler les interfaces et le programme client

```
javac HelloInterface.java HelloClient.java
```

N.B. il est préférable de regrouper dans un fichier **.jar** les interfaces des objets appelés à distance, ce qui permet de les réutiliser pour le serveur et le client - voir TP

# Java RMI : Étapes de la mise en œuvre (1/2)

---

## ■ Exécution

- ◆ Lancer le serveur de noms (sur la machine serveur)

`rmiregistry &`

N.B. Par défaut, le registry écoute sur le port 1099. Si on veut le placer sur un autre port, il suffit de l'indiquer, mais il faut aussi modifier les URL en conséquence :

`rmi://<serveur>:<port>/<répertoire>`

- ◆ Lancer le serveur

`java -Djava.rmi.server.codebase=http://goedel.imag.fr/<répertoire des classes>  
-Djava.security.policy=java.policy HelloServer &`

N.B. Signification des propriétés (option -D) :

- Le contenu du fichier `java.policy` spécifie la politique de sécurité, cf plus loin.
- L'URL donnée par `codebase` sert au chargement de classes par le client

- ◆ Lancer le client

`java -Djava.security.policy=java.policy HelloClient`

N.B. Le talon client sera chargé par le client depuis le site du serveur, spécifié dans l'option `codebase` lors du lancement du serveur

# Sécurité

---

## ■ Motivations

- ◆ La sécurité est importante lorsqu'il y a téléchargement de code (il peut être dangereux d'exécuter le code chargé depuis un site distant)

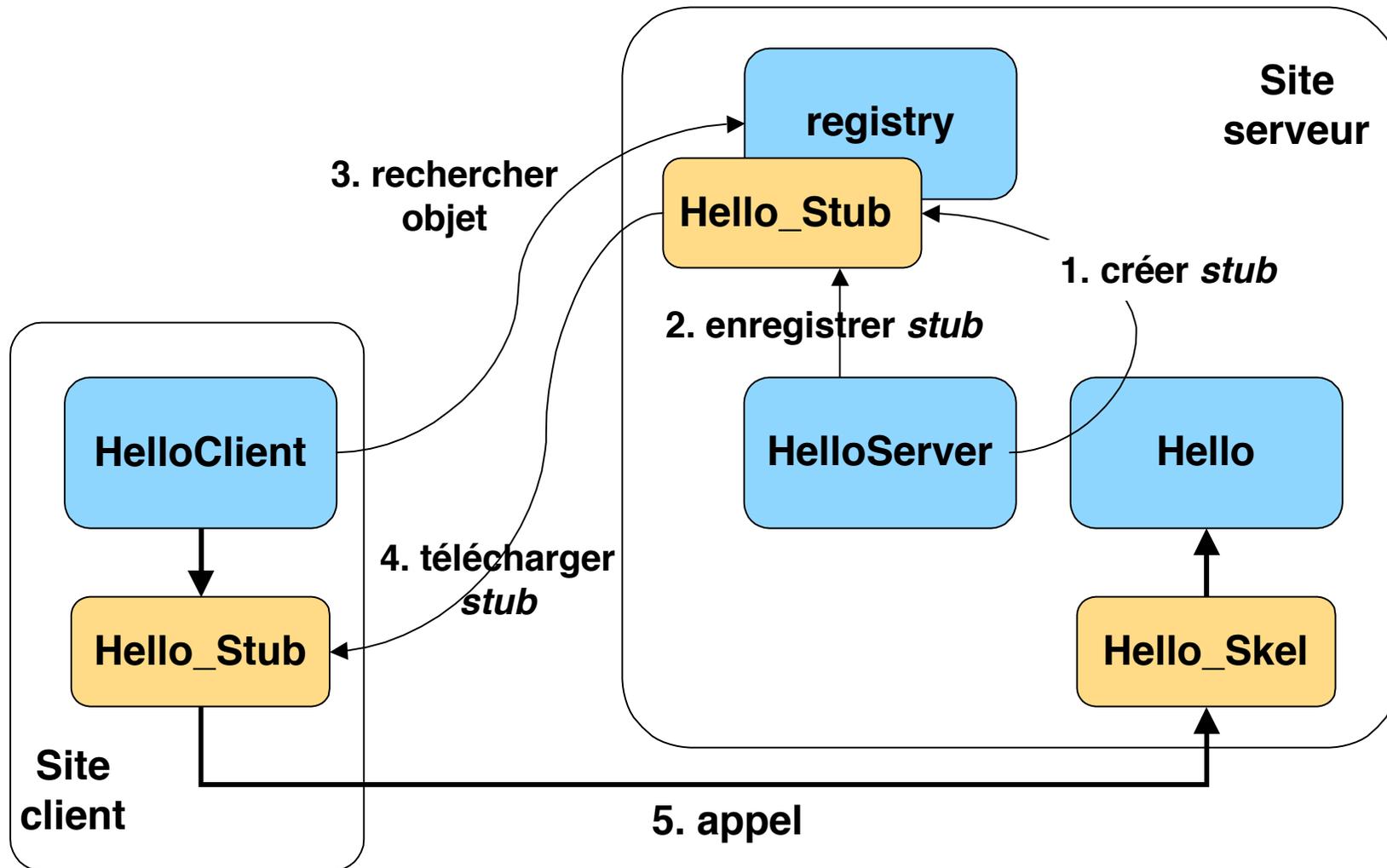
## ■ Mise en œuvre

- ◆ La politique de sécurité spécifie les actions autorisées, en particulier sur les *sockets*
- ◆ Exemple de contenu du fichier `java.policy`

```
grant {  
    permission java.net.SocketPermission "*:1024-65535",  
        "connect,accept";  
    permission java.net.SocketPermission "*:80", "connect";  
};
```

- ◆ Permet d'utiliser les *sockets* comme indiqué. Toute autre utilisation est interdite

# Fonctionnement d'ensemble de Java RMI



# Fabrique d'objets (*Factory*)

## ■ Motivation

- ◆ Permettre au client de construire des instances multiples d'une classe *C* sur le site serveur
- ◆ Le *new* n'est pas utilisable tel quel (car il ne gère que la mémoire locale, celle du client)
- ◆ Solution : appel d'un objet *FabriqueC*, qui crée localement (sur le serveur) les instances de *C* (en utilisant *new C*)

## ■ Exemple

```
public interface Annuaire extends Remote{
    public String titre;
    public boolean inserer(String nom, Info info)
        throws RemoteException, ExisteDeja;
    public boolean supprimer(String nom)
        throws RemoteException, PasTrouve;
    public Info rechercher(String nom)
        throws RemoteException, PasTrouve;
}
```

```
import java.rmi.*
public interface FabAnnuaire extends Remote{
    public Annuaire newAnnuaire(String titre)
        throws RemoteException ;
}
```

```
public class Info implements Serializable {
    public String adresse;
    public int num_tel ;
}
public class ExisteDeja extends Exception{} ;
public class PasTrouve extends Exception{} ;
```

# Mise en œuvre d'une fabrique

```
public class AnnuaireImpl implements Annuaire
    extends UnicastRemoteObject{
    private String letitre;
    public Annuaire(String titre) {
        this.letitre=titre};
    public String titre {return letitre};
    public boolean inserer(String nom, Info info)
        throws RemoteException, ExisteDeja{
    ...};
    public boolean supprimer(String nom)
        throws RemoteException, PasTrouve{
    ...};
    public Info rechercher(String nom)
        throws RemoteException, PasTrouve{
    ...};
}
```

La classe annuaire

```
public class FabAnnuaireImpl implements FabAnnuaire
    extends UnicastRemoteObject{
    public FabAnnuaireImpl{};
    public Annuaire newAnnuaire(String titre)
        throws RemoteException {
        return new AnnuaireImpl(titre)};
}
```

```
import java.rmi.*;
public class Server {
    public static void main (String [ ] argv)
    {
        /* lancer SecurityManager */
        System.setSecurityManager (
            new RMISecurityManager ());
        try {
            Naming.rebind ("Fabrique",
                new (FabAnnuaireImpl));
            System.out.println ("Serveur prêt.");
        } catch (Exception e) {
            System.out.println
                ("Erreur serveur : " + e);
        }
    }
}
```

Le serveur

← La classe fabrique

# Utilisation de la fabrique

## Programme client

```
import java.rmi.*;
public class HelloClient {
    public static void main (String [ ] argv) {

        /* lancer SecurityManager */

        System.setSecurityManager (
            new RMISecurityManager ());

        try {

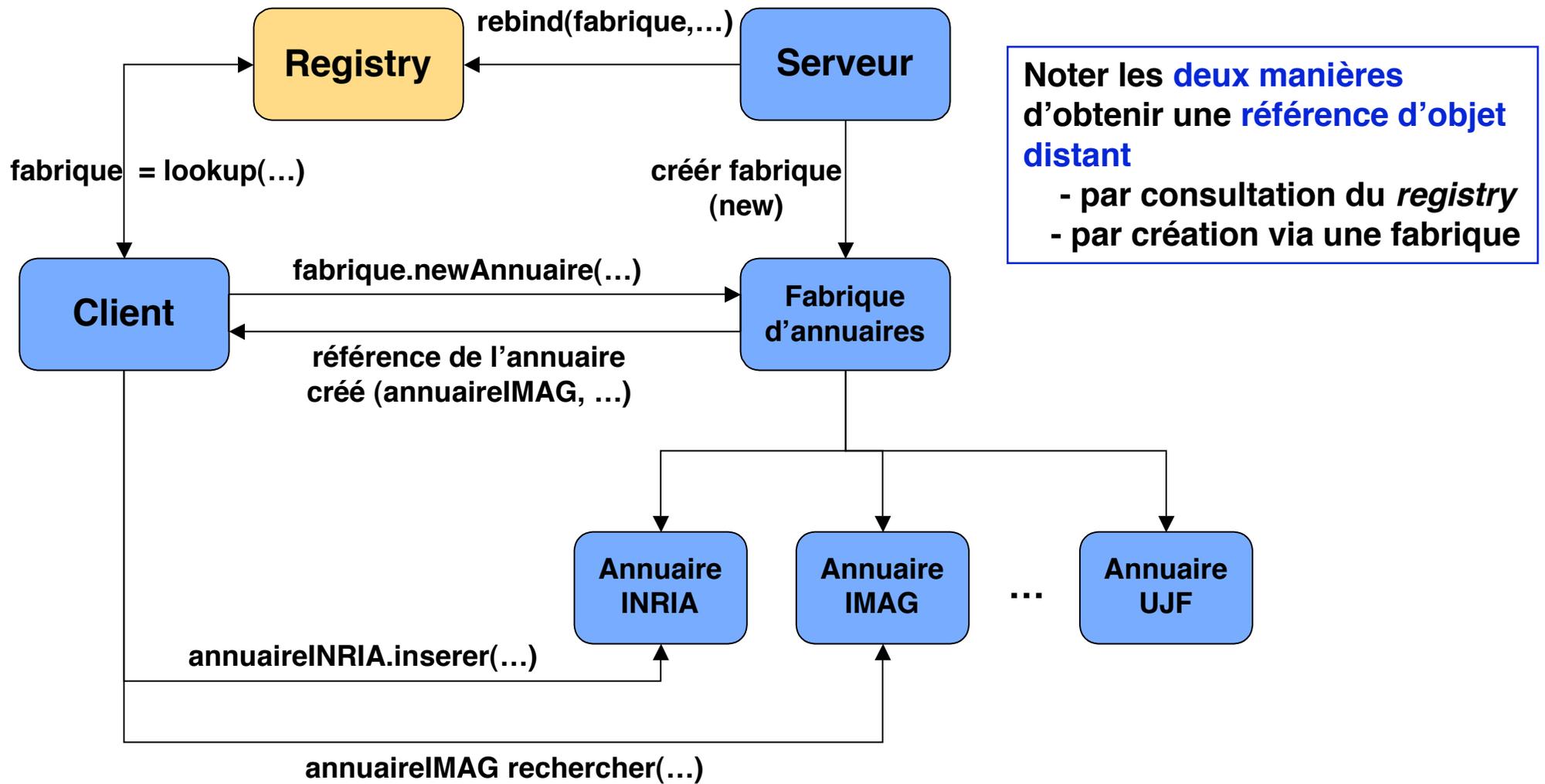
            /* trouver une référence vers la fabrique */
            FabAbannuaire fabrique =
                (FabAbannuaire ) Naming.lookup
                    ("rmi://goedel.imag.fr/Fabrique");

            /* créer un annuaire */
            annuaireIMAG = fabrique.newAnnuaire("IMAG");
            /* créer un autre annuaire */
            annuaireINRIA= fabrique.newAnnuaire("INRIA");
```

```
        /* utiliser les annuaires */
        annuaireIMAG.inserer(..., ...);
        annuaireINRIA.inserer(..., ...);
        ....

        } catch (Exception e) {
            System.out.println
                ("Erreur client : " + e);
        }
    }
}
```

# Fonctionnement d'ensemble de la fabrique



# Passage d'objets en paramètre (1)

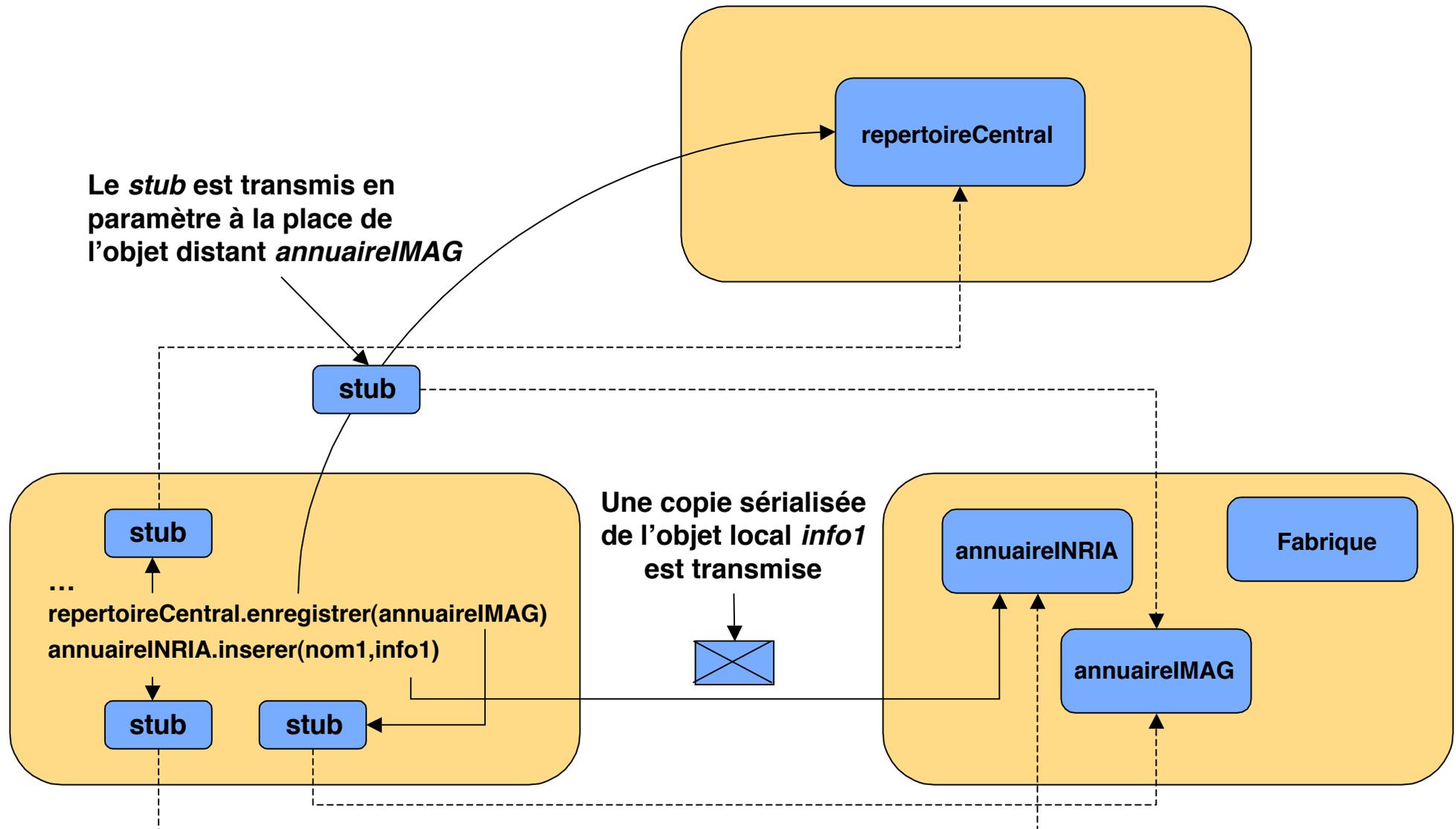
---

---

## ■ Deux cas possibles

- ◆ Passage en paramètre d'un objet **local** (sur la JVM de l'objet appelant)
  - ❖ Passage par **valeur** : on transmet une **copie** de l'objet (plus précisément : une copie de l'ensemble ses variables d'état) Pour cela l'objet doit être **sérialisable** (i.e. implémenter l'interface *java.io.Serializable*)
  - ❖ Exemple de l'annuaire : le client transmet un objet de la classe locale *Info*
- ◆ Passage en paramètre d'un objet **non-local** (hors de la JVM de l'objet appelant, par ex. sur un site distant)
  - ❖ Passage par **référence** : on transmet une **référence** sur l'objet (plus précisément : un *stub* de l'objet). Le destinataire utilisera ce *stub* pour appeler les méthodes de l'objet.
  - ❖ Exemple de l'annuaire : le client passe un objet de type *Annuaire* (localisé sur le serveur distant)

# Passage d'objets en paramètre : illustration



# Passage d'objets en paramètre (2)

---

## ■ Notions sur les objets sérialisables

- ◆ Un objet sérialisable (transmissible par valeur hors de sa JVM) doit implémenter l'interface *java.io.Serializable*. Celle-ci est réduite à un marqueur (pas de variables ni d'interface)
- ◆ Les objets référencés dans un objet sérialisable doivent aussi être sérialisables
- ◆ Comment rendre effectivement un objet sérialisable ?
  - ❖ Pour les variables de types primitifs (int, boolean, ...), rien à faire
  - ❖ Pour les objets dont les champs sont constitués de telles variables : rien à faire
  - ❖ On peut éliminer une variable de la représentation sérialisée en la déclarant *transient*
  - ❖ Pour un champ non immédiatement sérialisable (ex. : *Array*), il faut fournir des méthodes *readObject()* et *writeObject()*
- ◆ Exemples de sérialisation : passage en paramètres, écriture sur un fichier. Le support de sérialisation est un *stream* (flot) : classes *java.io.ObjectOutputStream* et *java.io.ObjectInputStream*.

Pour des détails techniques : voir javadoc de l'interface *Serializable* et des classes *ObjectOutputStream* et *ObjectInputStream*

# Notions sur le fonctionnement interne de Java RMI (1)

---

## ■ La classe *UnicastRemoteObject*

- ◆ Rappel de la règle d'usage : une classe d'objets accessibles à distance étend la classe *java.rmi.UnicastRemoteObject*.
  - ❖ Le principale fonction de cette classe se manifeste lors de la création d'une instance de l'objet.

```
public class AnnuaireImpl extends UnicastRemoteObject {...}

AnnuaireImpl (...) {      // le constructeur d'AnnuaireImpl appelle automatiquement
                          // le constructeur de la superclasse UnicastRemoteObject

    ...
    monAnnuaire = new AnnuaireImpl (...);
}
```

- ❖ Le constructeur de *UnicastRemoteObject* crée une instance de *stub* pour l'objet (en utilisant la classe *Annuaire\_Stub* engendrée par *rmic*, et retourne ce *stub* comme résultat de la création

## ■ Le contenu d'un *stub*

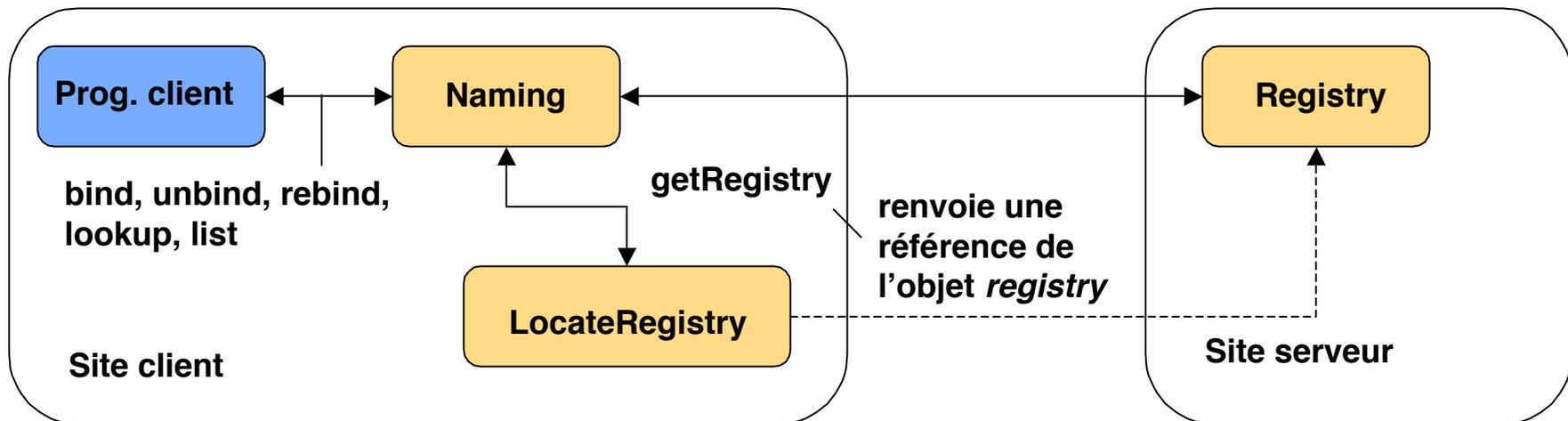
- ◆ Un *stub* contient essentiellement une variable *ref* de type *RemoteRef* qui contient la localisation de l'objet (adresse IP, port)
- ◆ Un appel de méthode se fait par appel de *ref.invoke(...)* qui utilise les sockets pour la communication

# Notions sur le fonctionnement interne de Java RMI (2)

## ■ Le serveur de noms (*registry*)

### ◆ Classes utiles (fournies par *java.rmi*)

- ❖ ***Naming*** : sert de représentant local du serveur de noms. Permet d'utiliser les méthodes *bind()*, *rebind()*, *lookup()*, *unbind()*, *list()*
- ❖ ***LocateRegistry*** : permet de localiser un serveur de noms (*rmiregistry*) et éventuellement d'en créer un. En général invisible au client (appelé en interne par *Naming*)



# Appel en retour (*callback*)

---

## ■ Le problème

- ◆ Réaliser un appel du serveur vers le client

## ■ Intérêt

- ◆ Augmenter l'asynchronisme

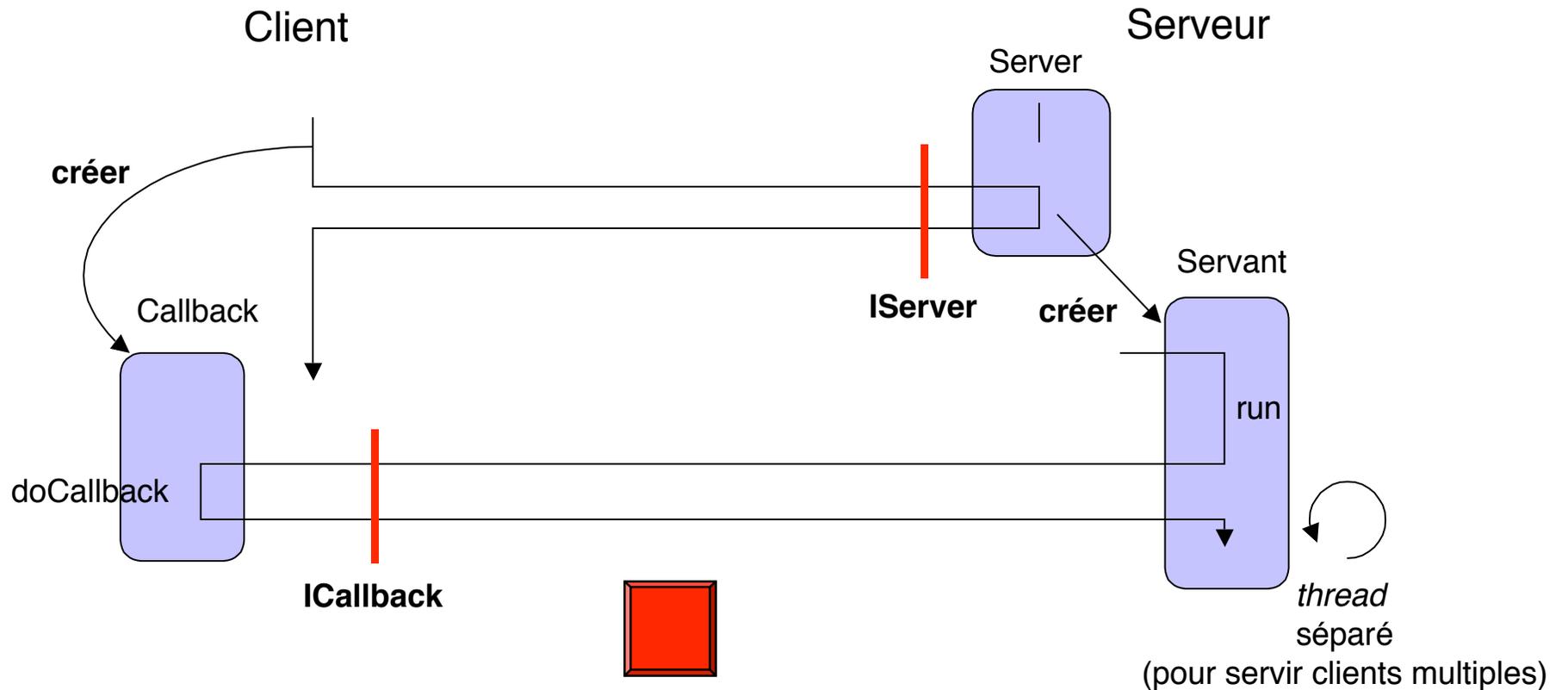
- ❖ 1) appel client-serveur avec retour immédiat (demande service)
- ❖ 2) rappel serveur-client quand le service est exécuté

- ◆ Augmenter le dynamisme

- ❖ Le serveur peut demander des informations complémentaires au client en cours d'exécution
- ❖ L'exécution du service peut faire appel à des services fournis par le client
  - ▲ Exemple : traitement d'exceptions

# Appel en retour : principe

- Le client passe en paramètre au serveur l'objet à rappeler
- Le serveur exécute un appel asynchrone sur cet objet



# Appel en retour : réalisation interfaces des objets distants)

---

// les deux interfaces des objets appelés à distance (doivent étendre Remote)

```
import java.rmi.*;
public interface ICallback extends Remote {           // interface de l'objet callback
    public void doCallback(String message) throws RemoteException;
}
```

```
import java.rmi.*;                                   // interface du serveur
public interface IServer extends Remote {
    public void callMeBack(int time, String param, ICallback callback)
        throws RemoteException;
}
```

## Appel en retour : réalisation (le servent)

```
import java.rmi.*;
public class Servant extends Thread {
    private int time;
    private String param;
    private ICallback callback;
    public Servant(int time, String param, ICallback callback) {           // le constructeur
        this.time = time;
        this.param = param;
        this.callback = callback;
    }
    public void run() {                                                   // exécution du servent comme thread séparé
        try {
            Thread.sleep(1000*time);                                     // attend le délai fixé (time exprimé en secondes)
        } catch (InterruptedException e) { }
        try {
            callback.doCallback(param);                                 // exécute l'appel en retour
        } catch (RemoteException e) {
            System.err.println("Echec appel en retour : "+e);
        }
        callback = null;                                               // nettoyage
        System.gc();
    }
}
```



## Appel en retour : réalisation (le serveur)

---

```
import java.rmi.*;
import java.rmi.server.*;
public class Server extends UnicastRemoteObject implements IPushServer {
    public Server() throws RemoteException {
        super();
    }
    public void callMeBack(int time, String param, ICallback callback)
        throws RemoteException {
        Servant servant = new Servant(time, param, callback); // création du servent
        servant.start(); // démarrage du servent
    }
    public static void main(String[] args) throws Exception {
        // démarrage du serveur
        Server serveur = new Server();
        Naming.rebind("Server", serveur);
        System.out.println("Serveur pret");
    }
}
```

## Appel en retour : réalisation (le *callback*)

---

```
import java.rmi.*;
import java.rmi.server.*;
public class Callback extends UnicastRemoteObject
    implements ICallback {
    public Callback() throws RemoteException {
        super();
    }
    public void doCallback(String message) throws RemoteException {
        System.out.println(message);
    }
}
```

## Appel en retour : réalisation (un client)

---

```
import java.rmi.*;
public class Client {
    public static void main(String[] args) throws Exception {

        Callback callback = new Callback();           // création de l'objet callback
        IServer serveur = (IServer) Naming.lookup("Server");
        System.out.println("démarrage de l'appel");
        serveur.callMeBack(5, "coucou", callback);    // demande de rappel
        for (int i=; i<=5; i++) {
            System.out.println(i);
            try {
                Thread.sleep(2000);
            } catch (InterruptedException e) { }
        }
        System.out.println("fin du main");
    }
}
```

# Conclusion sur Java RMI

---

---

## ■ Extension du RPC aux objets

- ◆ Permet l'accès à des objets distants
- ◆ Permet d'étendre l'environnement local par chargement dynamique de code
- ◆ Pas de langage séparé de description d'interfaces (IDL fourni par Java)

## ■ Limitations

- ◆ Environnement **restreint à un langage unique (Java)**
  - ❖ **Mais passerelles possibles, en particulier RMI/IIOP**
- ◆ Services réduits au minimum
  - ❖ **Service élémentaire de noms (sans attributs)**
  - ❖ **Pas de services additionnels**
    - ▲ Duplication d'objets
    - ▲ Transactions
    - ▲ ...

# Introduction à CORBA

---

**Sacha Krakowiak**  
Université Joseph Fourier  
Projet Sardes (INRIA et IMAG-LSR)  
<http://sardes.inrialpes.fr/~krakowia>

# Problème et voie d'approche

---

---

## ■ Situation et besoins

- ◆ Extension du domaine des applications réparties
- ◆ Interconnexion des systèmes d'information et des réseaux
- ◆ Réutilisation et adaptation d'applications existantes

## ■ Contraintes

- ◆ Il n'y aura **pas de consensus** sur les plates-formes matérielles, les langages, les systèmes, les protocoles de réseau, les formats d'applications → nécessité de **l'interopérabilité**

## ■ Voie d'approche

- ◆ Proposer un standard pour une architecture d'applications réparties
  - ❖ **Permettant l'interopérabilité entre plates-formes et systèmes hétérogènes**
  - ❖ **Fondé sur un modèle à objets**

# L'OMG (*Object Management Group*)

---

---

<http://www.omg.org/>

## ■ Consortium à but non lucratif, créé en 1989

- ◆ Plus de 850 membres en 1999 (constructeurs, SSII, utilisateurs, recherche)

## ■ Objectif

- ◆ Faire émerger des standards pour l'intégration d'applications distribuées hétérogènes et la réutilisation de composants logiciels

## ■ Fonctionnement

- ◆ Appel à propositions (appuyées par des réalisations prototypes)
- ◆ Examen des réponses, discussion et amélioration, proposition de standards
- ◆ Fournit des spécifications, non des produits

# OMG : vision globale

---

---

- **L'OMG prône l'utilisation d'une approche fondée sur les objets pour fournir une vue unique d'un système distribué hétérogène**
  - ◆ **OMA (*Object Management Architecture*) pour l'architecture logicielle globale**
  - ◆ **CORBA (*Common Object Request Broker Architecture*) pour les outils et services *middleware***
  - ◆ **UML (*Unified Modeling Language*) pour la modélisation**
  - ◆ **MOF (*Meta-Object Facility*) pour la méta-modélisation**

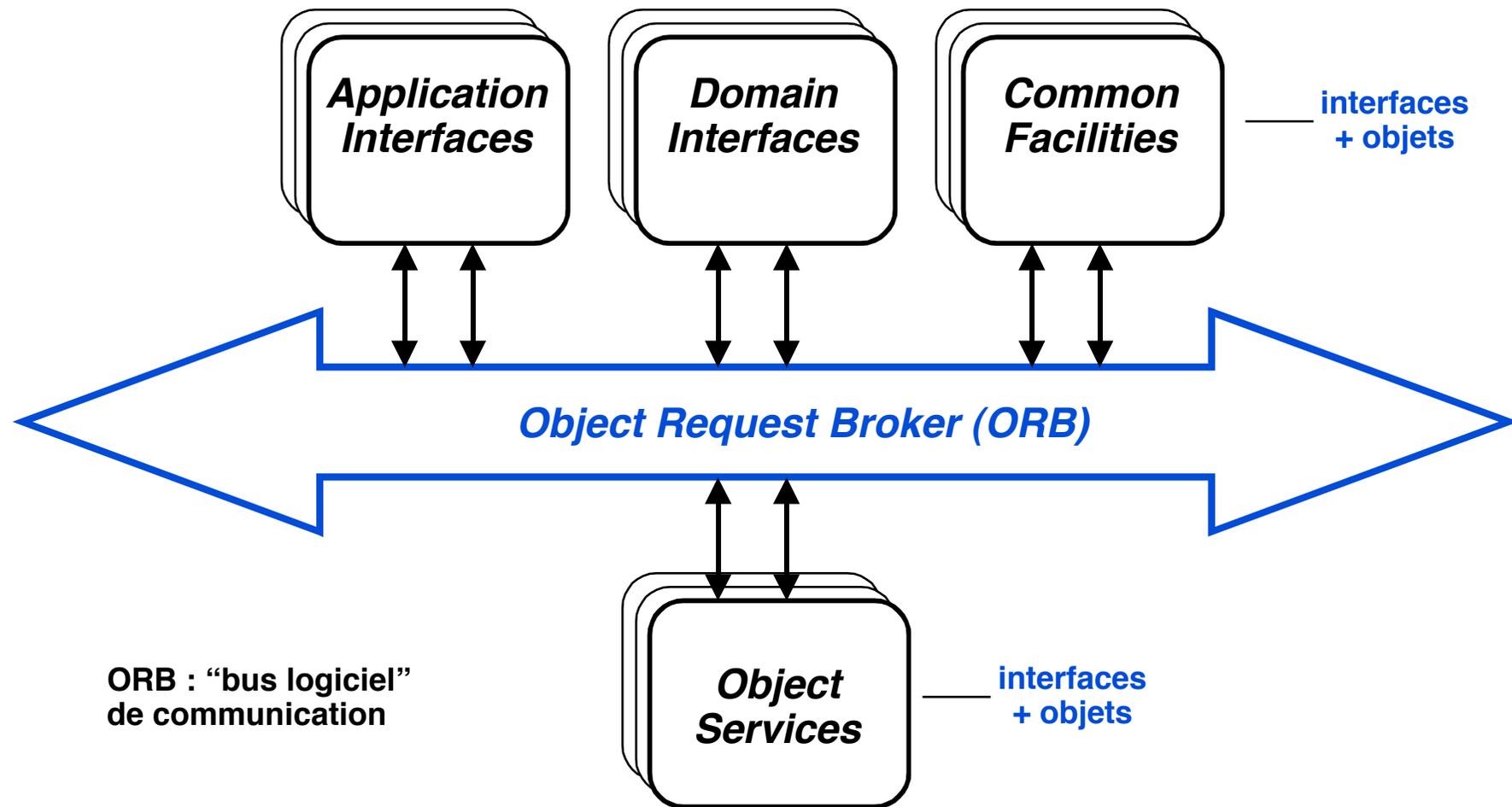
# CORBA : objectifs

---

---

- Permettre l'interopérabilité des composants et des applications répartis par l'intermédiaire d'un mode de coopération unifié : l'appel à des objets distants
- Gérer l'hétérogénéité des machines, systèmes et langages par l'utilisation d'un langage pivot commun : OMG IDL
- Proposer un support *middleware* technique ouvert (non propriétaire) et multi-fournisseurs
- Promouvoir la réalisation de services communs utilisant eux-mêmes le standard CORBA

# Vue d'ensemble de l'OMA (*Object Management Architecture*)



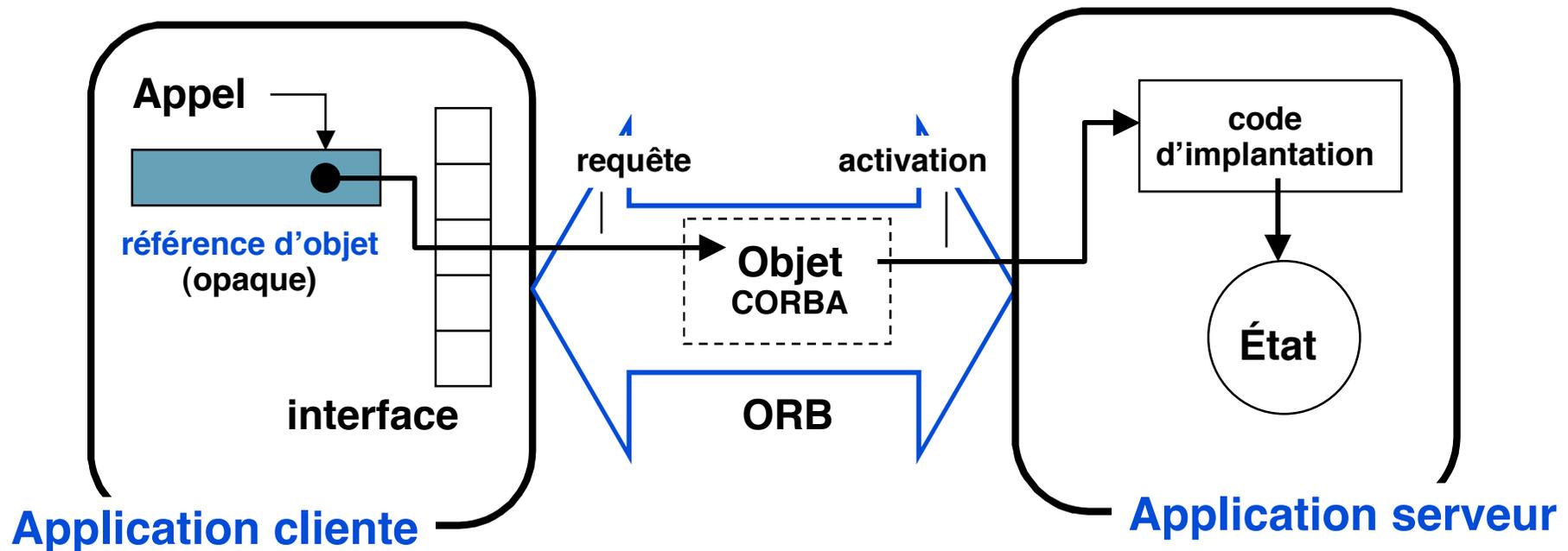
# Composants de l'architecture OMA

---

---

- **ORB (*Object Request Broker*)**
  - ◆ “bus logiciel” de communication entre les autres composants
- ***Application Interfaces***
  - ◆ Interfaces développées pour une application spécifique (non standardisées)
- ***Object Services***
  - ◆ Interfaces de services communs au niveau “système”, utilisés par les applications.
    - ❖ Service de noms, de transactions, d'événements, de persistance, etc.
- ***Common Facilities***
  - ◆ Interfaces de services communs au niveau “applications”
    - ❖ Service d'échange et de liaison de documents
- ***Domain Interfaces***
  - ◆ Interfaces de services spécifiques à un domaine d'application (“objets métier”).
    - ❖ Télécommunications, finance, médecine, etc.

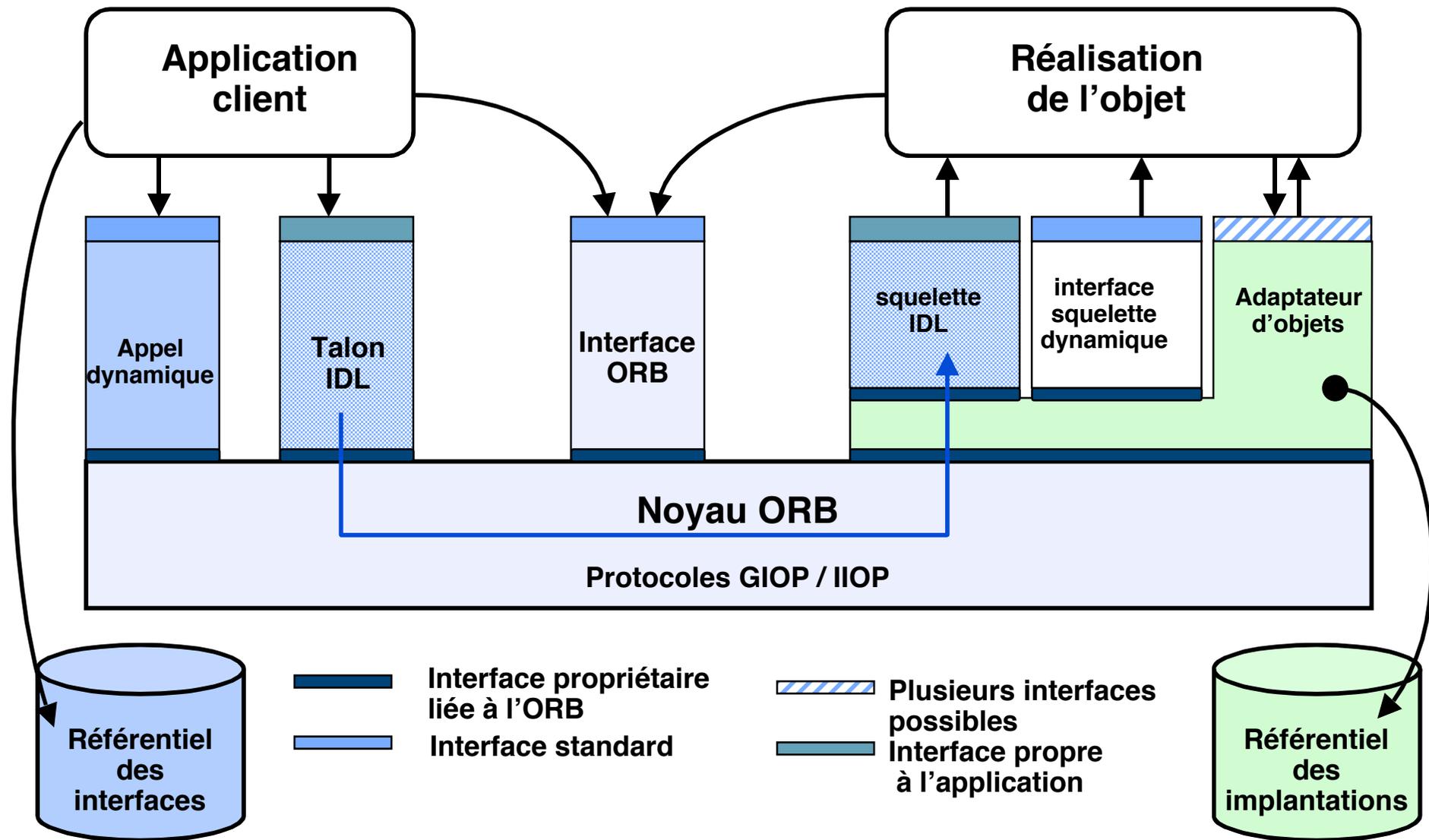
# Vue schématique du fonctionnement de CORBA



## Fonctions de l'ORB

- localiser les objets
- transmettre les requêtes
- activer les objets

# Schéma d'ensemble de CORBA



# Références d'objets

---

---

- Désignation unique d'un objet
  - ◆ Opaque (contenu non directement exploitable par application)
- Comment obtenir une référence d'objet ?
  - ◆ Création (notion de “fabrique” d'objets, cf plus loin)
  - ◆ Recherche à partir d'un nom symbolique ou d'attributs
    - ❖ Services de noms (font partie des *Object Services*)
      - ▲ *Naming* (pages blanches)
      - ▲ *Trading* (pages jaunes)
  - ◆ Conversion *string* <--> *reference*
    - ❖ Service de l'ORB
- À l'initialisation
  - ◆ Un service de l'ORB contient les références de quelques objets connus, dont un service de noms

# Normalisation des communications

---

---

## ■ Protocoles d'interopérabilité entre bus de la norme CORBA 2

- ◆ GIOP : *General Inter-ORB Protocol*
- ◆ IIOP (*Internet Inter-ORB Protocol*) : instantiation de GIOP sur les protocoles de l'Internet (TCP/IP)

## ■ Composants du protocole

- ◆ CDR : *Common Data Representation* (format de données)
  - ❖ **Décrit la représentation de tous les types de l'IDL**
- ◆ IOR : *Interoperable Object References* (références d'objets)
  - ❖ **Contient au moins**
    - ▲ Le numéro de version de la couche transport acceptée par le serveur de l'objet
    - ▲ L'adresse de la machine destinataire
    - ▲ Une clé interne permettant de localiser l'objet dans le serveur
- ◆ **Format des messages échangés**

# L'interface ORB

---

---

## ■ Utilité

- ◆ Fournit des fonctions de service utiles au client et au serveur
  - ❖ Conversion entre références et chaîne de caractères (service élémentaire de désignation)
  - ❖ Ensemble de références initiales vers des objets connus
  - ❖ Initialisation de l'adaptateur d'objets

## ■ Fonctionnement

- ◆ L'ORB est lui-même accessible via une référence d'objet
- ◆ La première opération que doit faire un client ou un serveur est d'obtenir cette référence (soit `orb`)
  - ❖ Opération `CORBA.orb.init` (ou équivalent) fournie par l'environnement CORBA
    - ▲ `orb = CORBA.orb.init (args)`
  - ❖ Les fonctions de l'ORB sont accessibles par `orb.operation`

# L'adaptateur d'objets

## notions de base

---

### ■ Fonction

- ◆ **Facilite la construction de serveurs en permettant d'isoler et de mettre en commun différentes fonctions utiles**
  - ❖ **Enregistrement et recherche des implémentations d'objets**
  - ❖ **Génération de références pour les objets**
  - ❖ **Activation des processus dans le serveur**
  - ❖ **Adaptation aux spécificités des langages de programmation**
  - ❖ **Adaptation à différentes formes de serveurs (SGBDOO, carte à puce, etc.)**

### ■ Mise en œuvre

- ◆ **Plusieurs adaptateurs peuvent coexister**
- ◆ **Sélection via un service de l'ORB**

# OMG IDL (*Interface Definition Language*)

---

---

## ■ Fonctions

- ◆ Description des interfaces des objets
  - ❖ **Types et opérations**
    - ▲ Généralisation de l'IDL utilisé pour le RPC
- ◆ Langage pivot entre plusieurs applications
  - ❖ **Définit un "contrat"**
- ◆ Base pour la génération automatique des talons et squelettes

## ■ Propriétés

- ◆ Langage déclaratif
  - ❖ **Séparation entre interfaces et implémentations**
- ◆ Indépendant d'un langage de programmation
  - ❖ **"projections" possibles vers différents langages d'écriture d'applications**

# OMG IDL : exemple de définition de types

---

```
#include <date.idl>
module annuaire {
    typedef string Nom;
    typedef sequence<Nom> DesNoms;
    struct Personne {
        Nom nom;
        string<200> informations;
        string<16> telephone;
        string<60> email;
        string<60> url;
        ::date::Date date_naissance;
    }; // fin Personne;
    typedef sequence<Personne> DesPersonnes
```

# OMG IDL : exemple de définition d'interface

---

```
Interface Repertoire {
    readonly attribute string libelle;

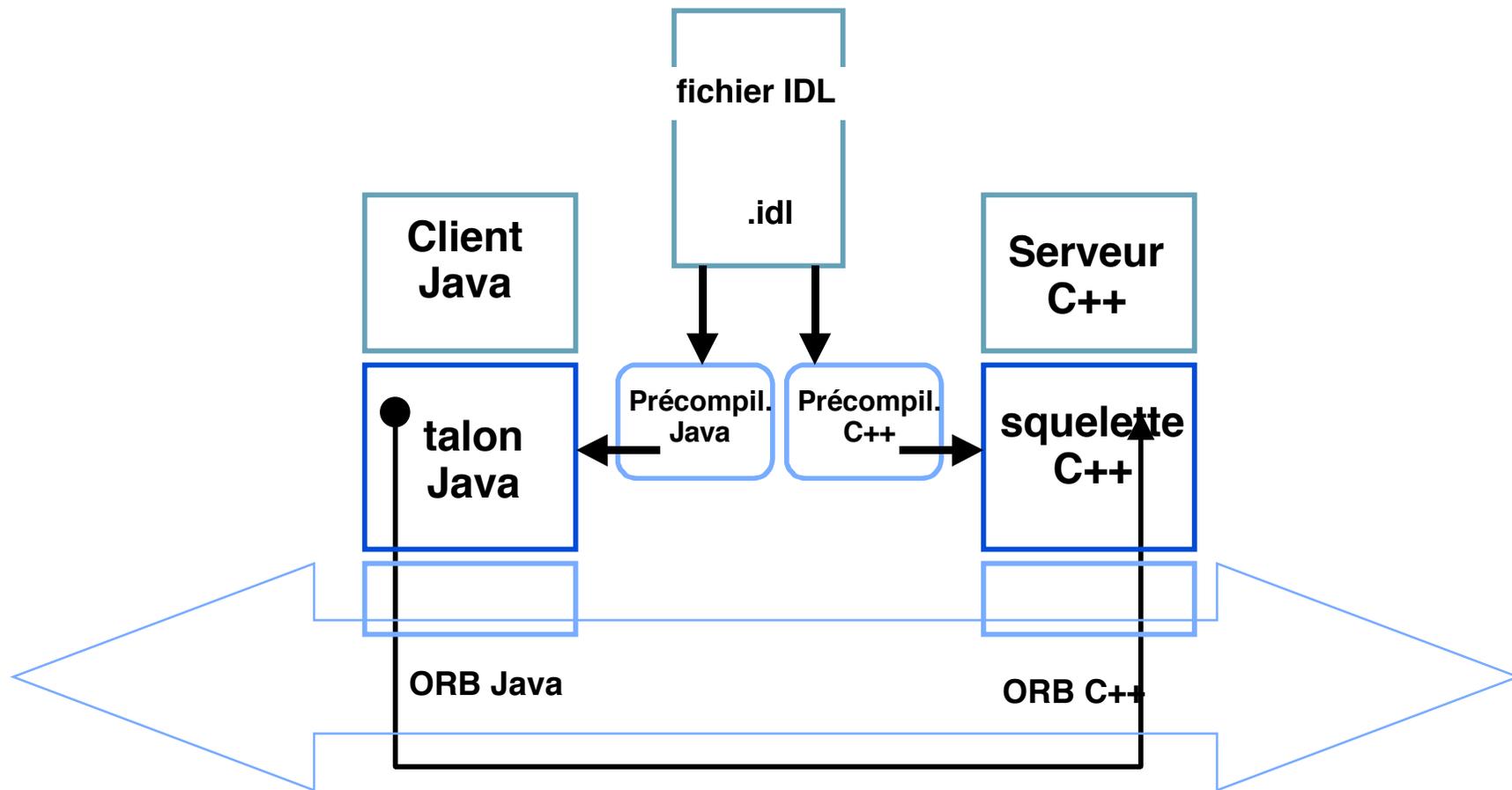
    exception ExisteDeja { Nom nom; };
    exception Inconnu { Nom nom; };

    void ajouterPersonne (in Personne personne)
        raises (ExisteDeja);
    void retirerPersonne (in Nom nom)
        raises({Inconnu});
    void modifierPersonne {in Nom nom, in Personne p}
        raises (Inconnu);
    Personne obtenirPersonne (in Nom nom)
        raises {Inconnu};
    DesNoms listerNoms () ;
}; // fin Repertoire
}; // fin annuaire
```

## ■ Éléments

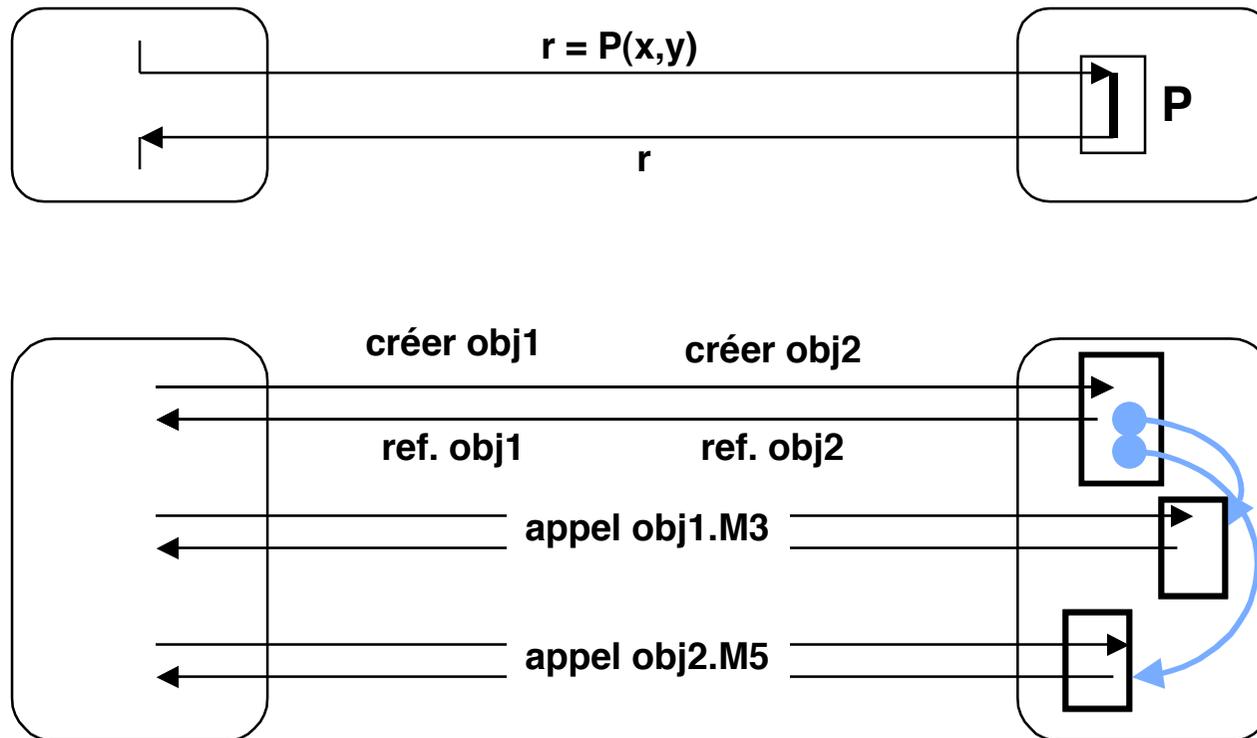
- ◆ Pragma (aide à la désignation ; exemple : version, etc)
- ◆ Module (ensemble de définitions d'intérêt commun)
- ◆ Types de base, format binaire normalisé
- ◆ Types construits
  - ❖ typedef, enum, struct, union, array, sequence, string
- ◆ Exceptions
- ◆ Interface (avec héritage multiple, sans **surcharge**)
  - ❖ une interface est associée à une référence d'objet
- ◆ Opération (synchrone, ou asynchrone sans résultat)
  - ❖ description d'une signature
- ◆ Attribut (possibilité de lecture seule)
- ◆ types de méta-données (TypeCode, any)

## Principe de l'utilisation de l'IDL pour la génération (mode statique)



Principe analogue à celui de *rpcgen* pour le RPC

# Différences entre RPC et CORBA



Pour CORBA, la norme spécifie que la sémantique en cas d'erreur est **“au plus une fois”** (appel exécuté 0 ou 1 fois)

## À venir : compléments sur CORBA

---

---

- **La construction d'une application CORBA**
- **Appel dynamique et référentiel des interfaces**
- **Adaptateur d'objets**
  - ◆ **POA (Portable Object Adaptor)**
- **Détails sur les références d'objets**
- **Interopérabilité entre bus CORBA**