Université Mohamed V - Rabat
Ecole Nationale Supérieure d'Informatique
et d'Analyse des Systèmes

Programmation Objet Avancée

PROF. GUERMAH HATIM

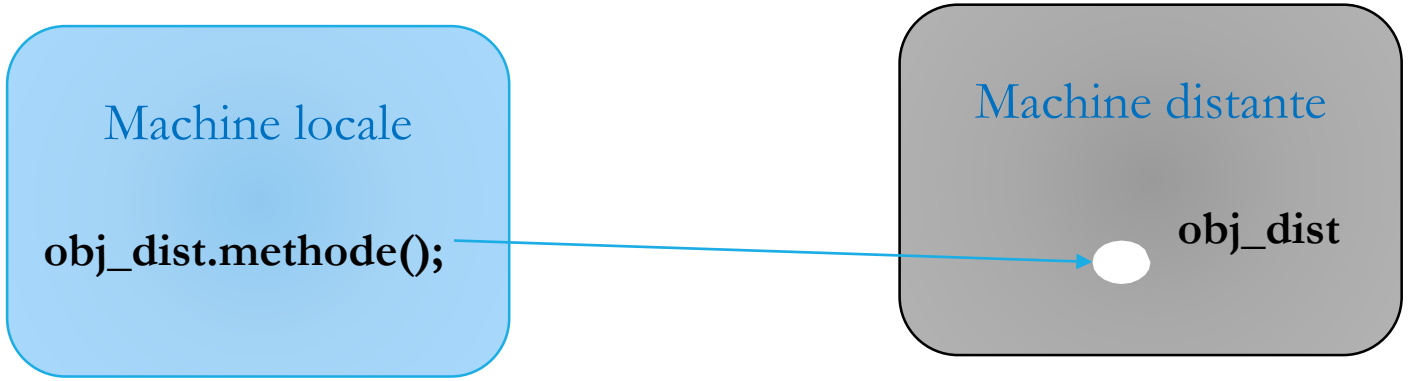
EMAIL: GUERMAH.ENSIAS@GMAIL.COM

Année Universitaire: 2019-2020

RMI Et les objets distribués

Introduction

- Un objet peut demander à un autre objet de faire un travail (généralement spécialisé).



- Contraintes:
  - L'objet qui fait le travail n'est pas localisé sur la même Machine.
  - L'objet peut ne pas être implémenté en Java.

Introduction

- Objectifs : créer un système distribué capable de:
  - Invoquer une méthode d'un objet distant (**obj_dist**) de la même manière que s'il était local.
  - Demander à un service « dédié » de renvoyer l'adresse de l'**obj_dist** sans savoir où l'objet se trouve

```
obj_dist =ServiceDeNoms.rechercher("monObjet");
```

Service de noms

  - Passer un obj_dist en paramètre à une méthode (distante ou locale).

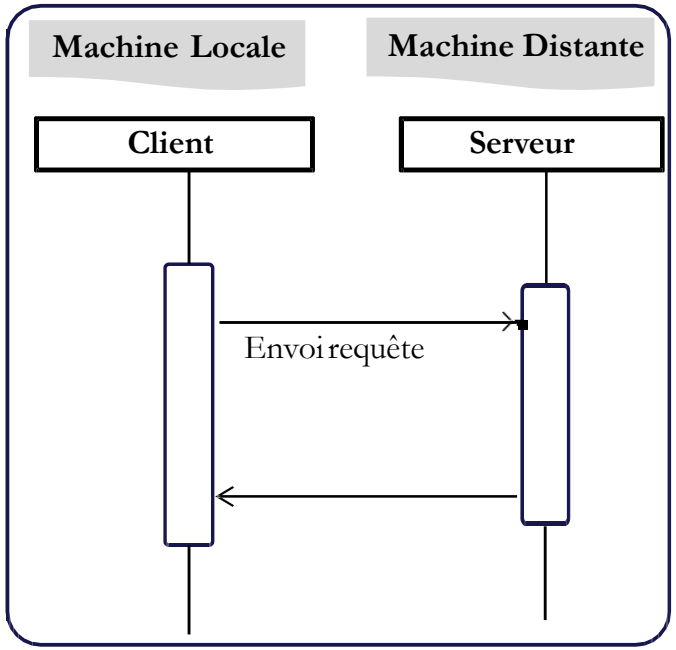
```
objetLocal.methode(obj_dist);
objDistant.methode(obj_dist);
```

  - Récupérer le résultat d'un appel distant sous forme d'un nouvel objet qui aurait été créé sur la machine distante

```
obj_dist = objDistant.methode() ;
```

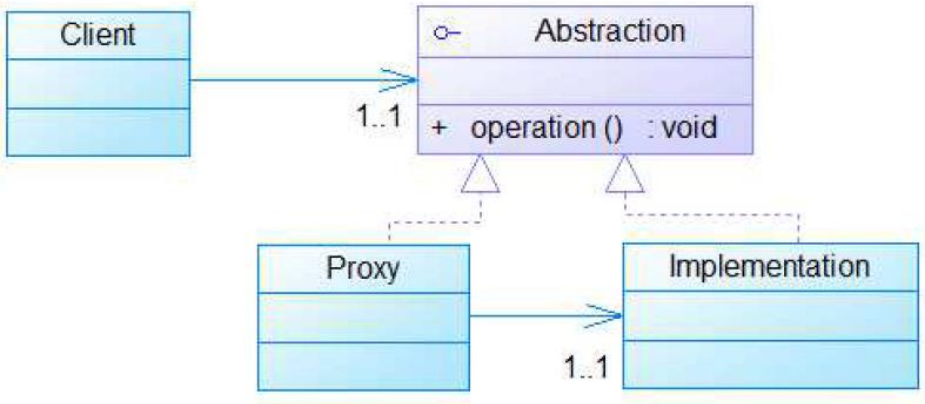
Introduction

- Difficultés : comment implémenter ?
  - En local : le programme « client » doit
    - Traduire la requête dans un format intermédiaire pour la transmission.
    - Envoyer les données de la requête au serveur.
    - Analyser la réponse et l'afficher à l'utilisateur.
  - A distance : le programme « serveur » doit
    - Analyser la requête.
    - Evaluer la réponse.
    - Formater la réponse pour la transmettre au client.



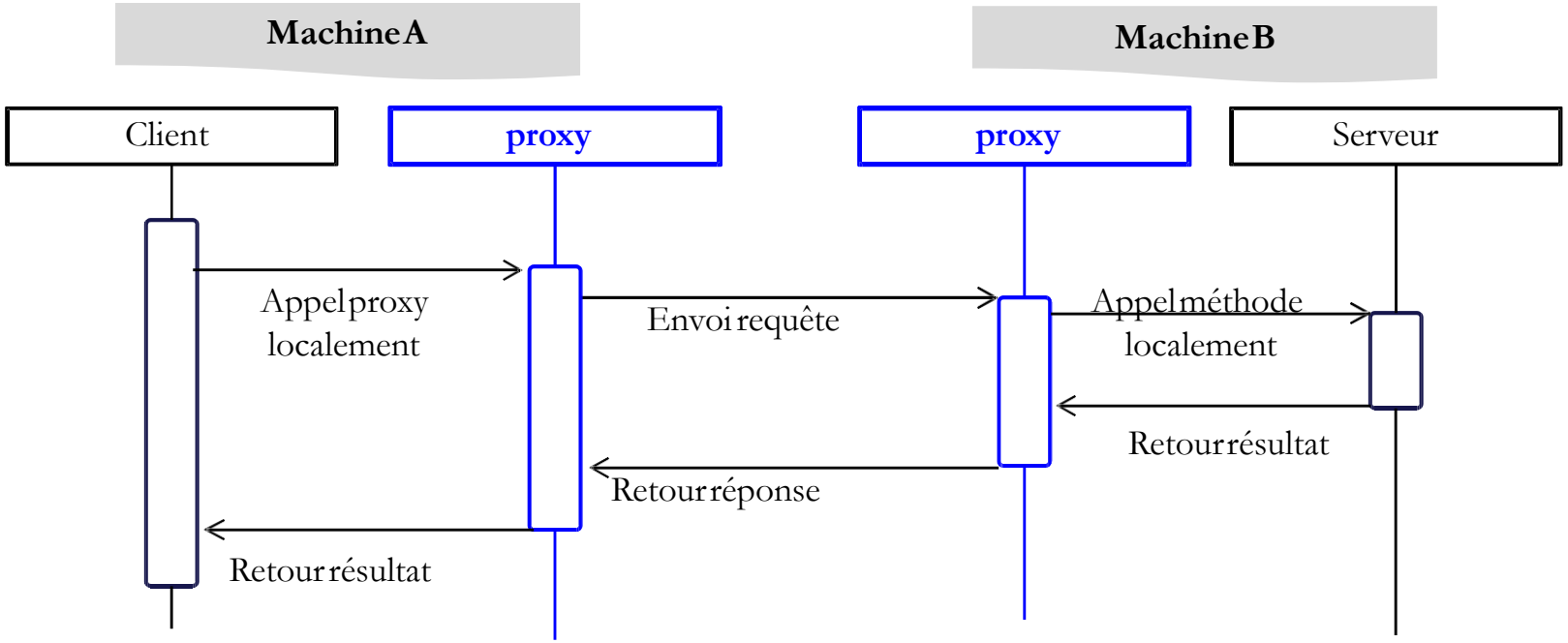
Pattern Proxy

- Objectif : Fournir un intermédiaire entre la partie cliente et un objet pour contrôler les accès à ce dernier.
- Résultat : Le Design Pattern permet d'isoler le comportement lors de l'accès à un objet.
- Responsabilités
  - Abstraction : définit l'interface des classes Implémentation et Proxy.
  - Implémentation : implémente l'interface. Cette classe définit l'objet que l'objet Proxy représente.
  - Proxy : fournit un intermédiaire entre la partie cliente et l'objet Implémentation. Cet intermédiaire peut avoir plusieurs buts (synchronisation, contrôle d'accès, cache, accès distant, ...).
  - La partie cliente appelle la méthode operation() de l'objet Proxy.



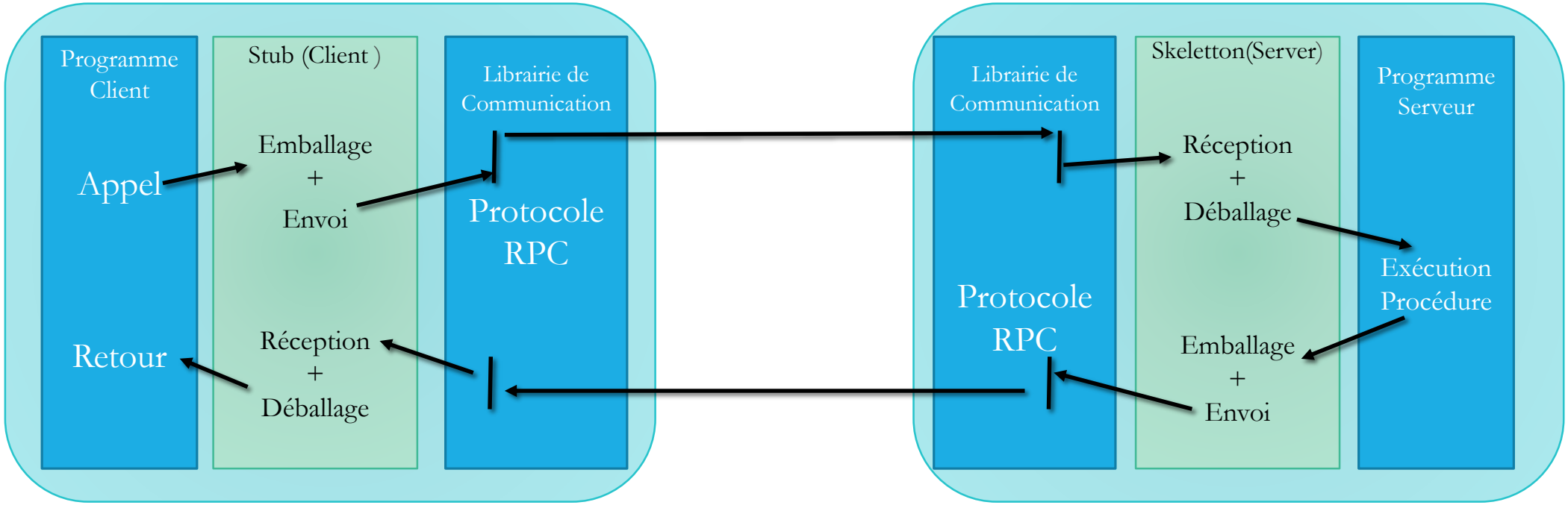
Pattern Proxy

- Ajouter des représentants (proxys) pour faire ce travail
- Ni le client ni le serveur n'ont à se soucier de l'envoi des données dans le réseau ni de leur analyse



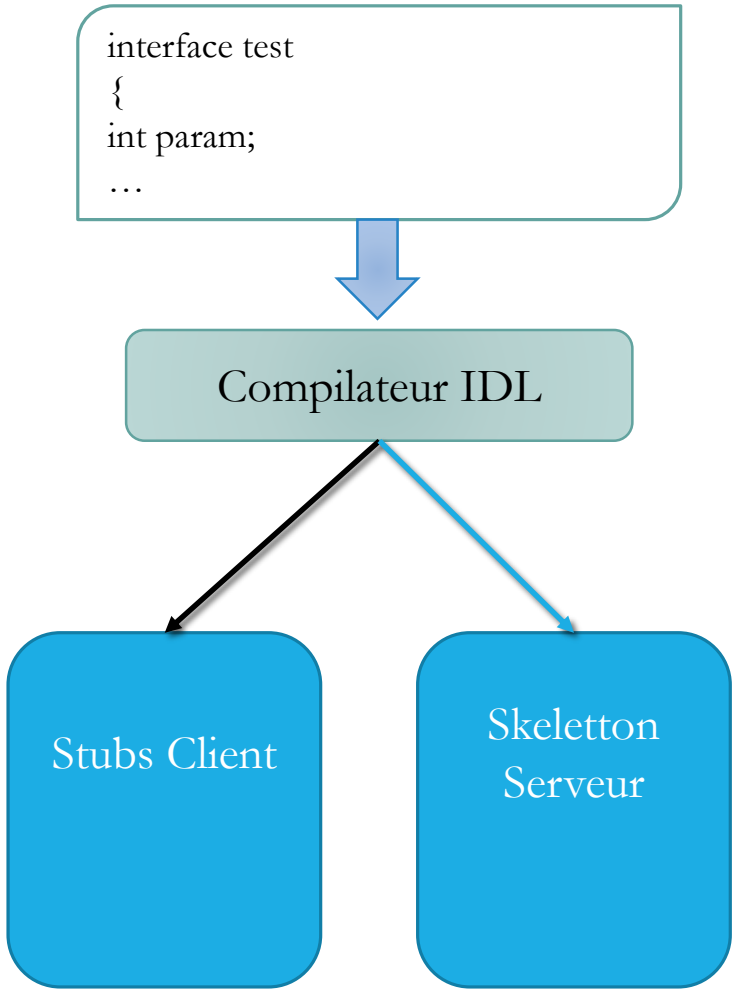
RPC Middleware

- Le client appelle une procédure locale (souche – stub / proxy)
- La procédure locale utilise une connexion socket pour envoyer un identifiant de procédure et les paramètres au serveur
- Le serveur reçoit la requête grâce à un socket, extrait l'id de procédure et les paramètres (ceci est fait par un squelette – skeletton)
- Le squelette exécute la procédure et renvoie le résultat via le socket



RPC Middleware : Contrat

- Le contrat : formalise le dialogue entre le client et le serveur et permet au client et au serveur d'avoir la même compréhension des échanges effectués. Il permet de répondre aux questions :
  - que transmet-on ?
  - où envoie-t-on les données ?
  - qui reçoit les données ?
  - comment sait-on que le travail est terminé ?
- Compilateur IDL : Génère le stub et le skeletton à partir d'un fichier présentant l'interface des méthodes dans un format indépendant du langage [IDL Interface Definition Language (OSF IDL RPC pour la technologie RPC.)
- Couche de présentation XDR (eXternal Data Representation) : Format pivot de représentation des données de types primitifs et structurés (tableaux, tableaux de taille variable, structures…)

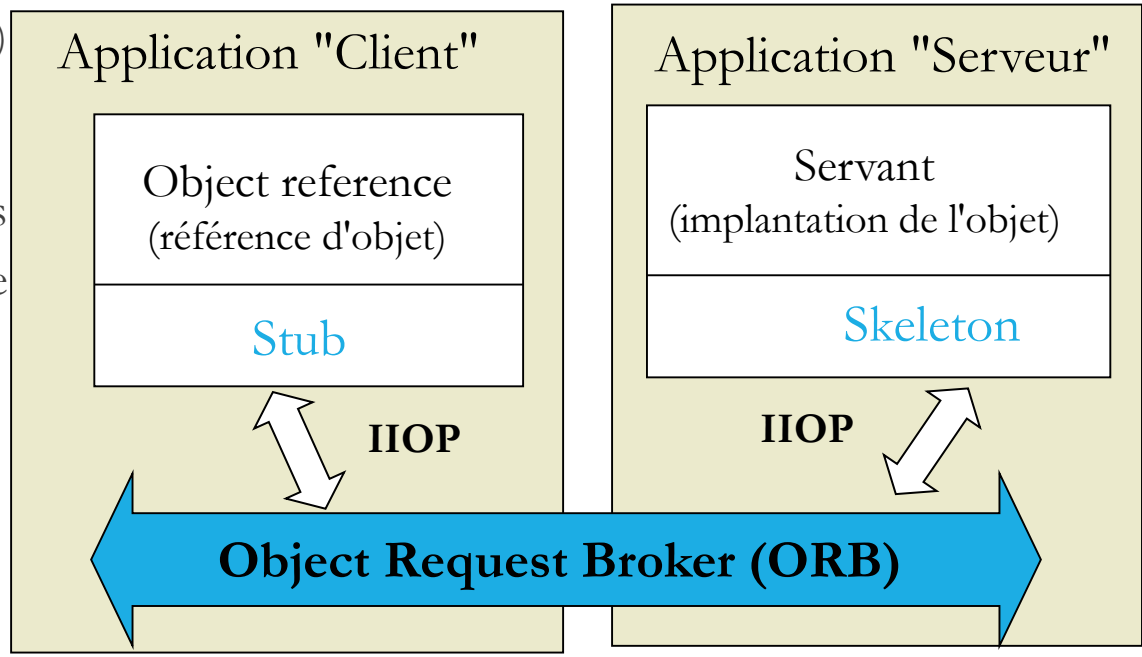


RPC Middleware

- Limitations :
  - Pas de gestion des concepts objets (encapsulation, héritage, polymorphisme)
  - Pas de services évolués : nommage …
- Successeurs :
  - RMI : mono langage, multi plateforme
  - CORBA : multi langage, multi plateforme
  - COM : multi langage, mono plateforme (multi pour DCOM)
  - SOAP / .NET / web services : multi langage, multi plateforme

CORBA

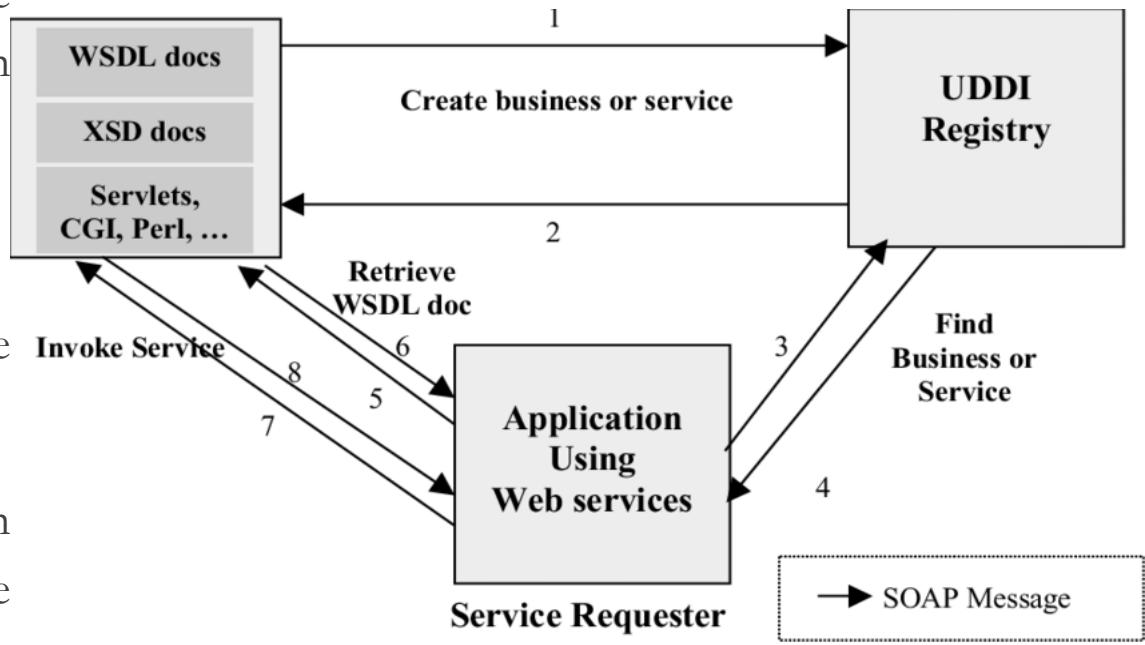
- Client et serveur peuvent être écrits en C, C++, Java ou tout autre langage
- Utilise le protocole IIOP (Internet Inter-ORB Protocol) pour communiquer entre les objets
- Interface de description qui spécifie les signatures des méthodes et les types de données des objets. Un langage spécial : IDL (Interface Definition Language)
- Etapes de développement :
  - Ecriture compilation de l'interface de l'objet en IDL
  - Implantation de l'objet
  - Réalisation et Compilation de l'application serveur
  - Réalisation et Compilation de l'application cliente



Le dialogue CORBA : client, ORB et serveur

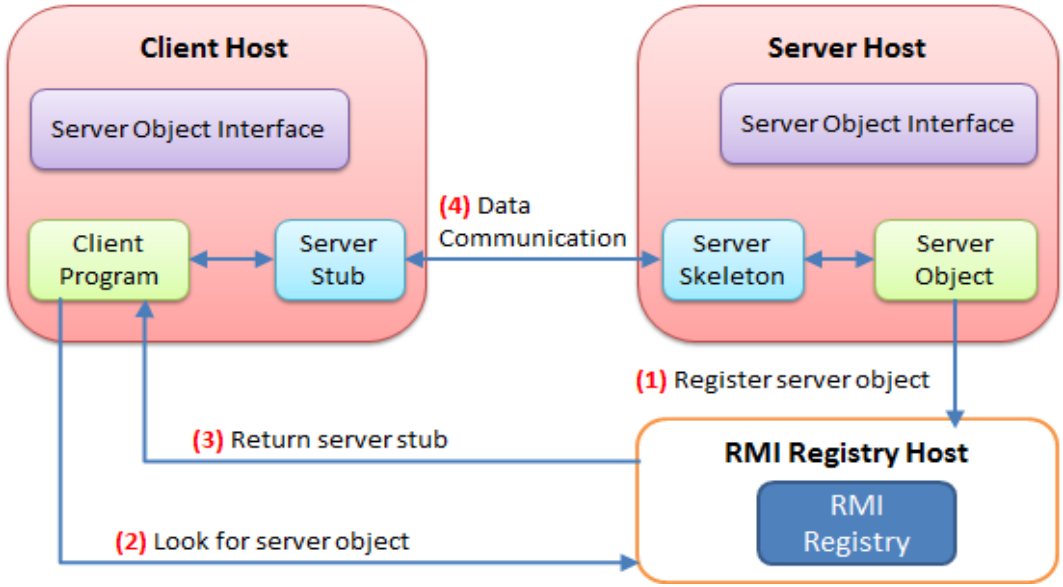
SOAP

- SOAP (Simple Object Access Protocol) est un protocole d'échange d'information structurée dans l'implémentation de services web bâti sur XML.
- Neutre vis-à-vis des langages de programmation
- L'interface de description est spécifiée dans un langage spécial : WSDL
- Le WSDL ou Web Services Description Language (prononcez en sigle ou « Whiz-Deul ») est une grammaire XML permettant de décrire un Service Web.



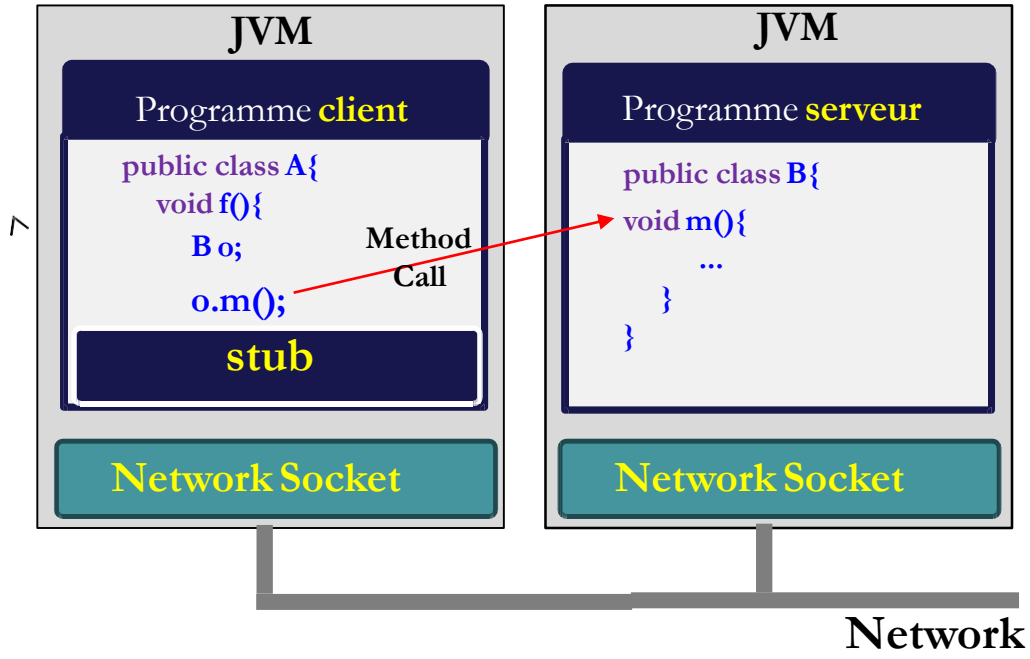
RMI

- RMI est une core API (intégré au JDK 1.1)
  - 100 % Java, gratuit (différent de CORBA)
  - Une version "orientée objet" de RPC
  - Permet aux Objets Java Distants de communiquer Mais sans écrire une seule ligne de code réseau
  - RMI est suffisant car plus simple : Lorsque les objets communicants sont implémentés en Java, la complexité et la généralité de CORBA ou de SOAP les rendent plus difficile que RMI.
- Caractéristiques de RMI
  - RMI utilise directement les sockets
  - RMI code ses échanges avec un protocole propriétaire : RMP (Remote Method Protocol)



RMI : Principe d'utilisation

- L'objet client appelle les méthodes de l'objet distant
  - Bien entendu les paramètres doivent être envoyés d'une façon ou d'une autre à l'autre machine
  - Le serveur doit en être informé pour exécuter la méthode et la valeur de retour doit être renvoyés.
- La terminologie objet client, objet serveur concerne un appel distant : Un ordinateur peut être client pour un appel et serveur pour un autre
  - L'objet proxy qui se trouve sur le client est appelé **stub : Représentant local de l'objet distant.**



RMI : Principe d'utilisation

- Le client utilise toujours des variables de type interface : Le **client** n'a pas de connaissance sur le type d'implémentation.

```
interface Warehouse {
    int getQuantity (String description) throws RemoteException;
    Product getProduct(Customer cust) throws RemoteException; }
```

- A l'appel la variable fait référence au stub

```
Warehouse   centralWarehouse      = …

//doit   être   lié   à   un   objet   courant   d'un   certain type
```

- La syntaxe reste la même que pour un appel local

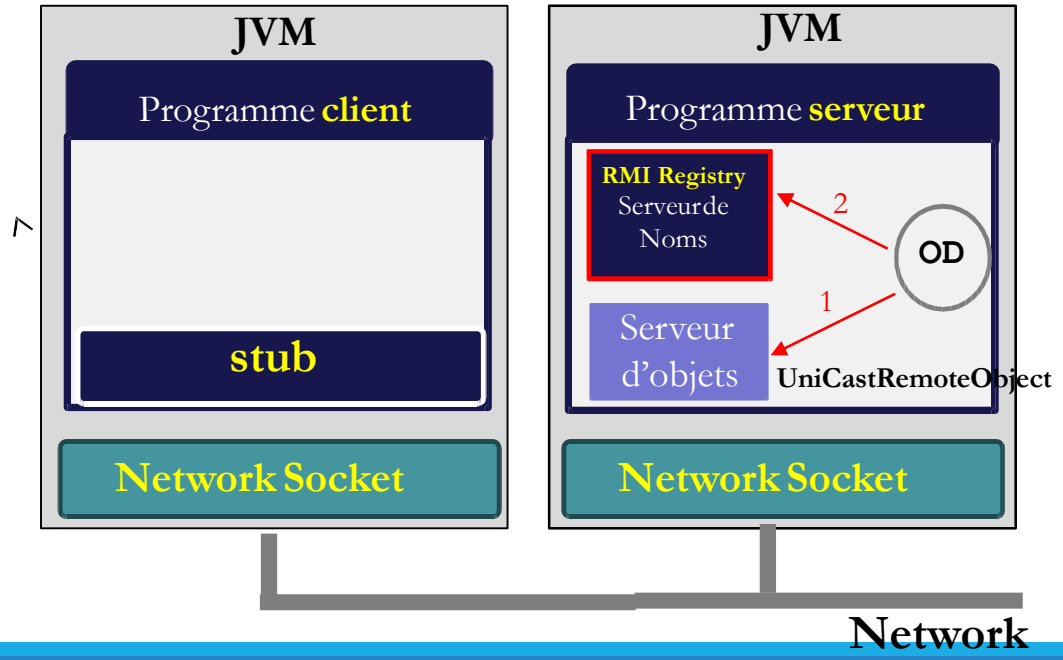
```
int   q  =  centralWarehouse.getQuantity("Super Cleaner");
```

- Les classes du stub et les objets associés sont créés automatiquement : La plupart des détails sont cachés au programmeur, mais un certain nombre de techniques doivent être maitrisées

RMI : Fonctionnement

1. L'objet serveur est créé, si nécessaire il crée l'objet squelette (avant Java 2), puis le port de communication est ouvert : Il faut Exposer l'OD via **UniCastRemoteObject**

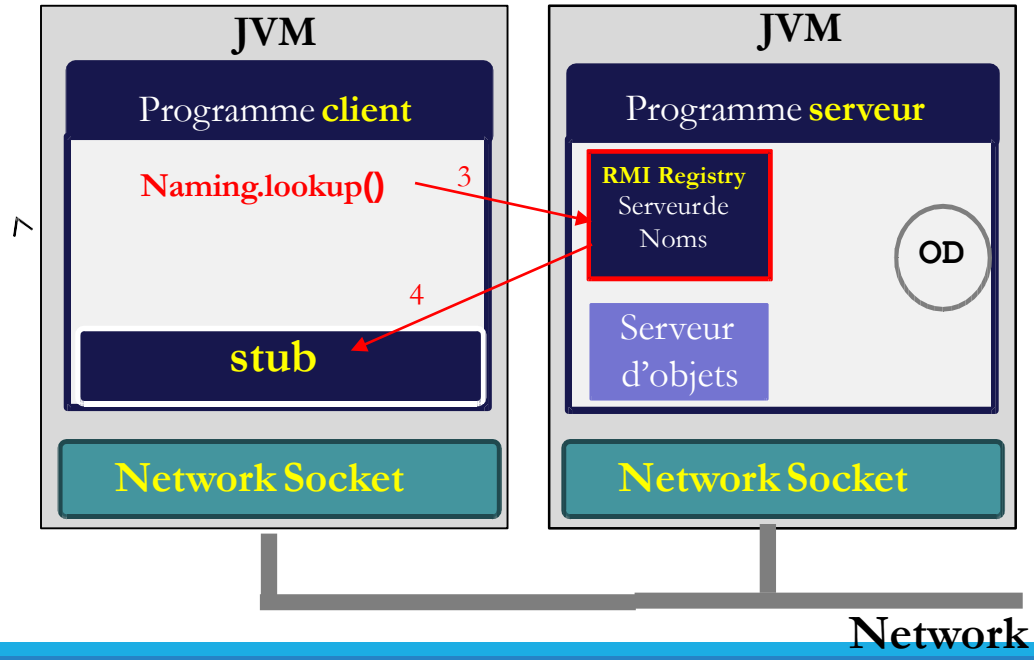
2. L'objet serveur s'enregistre auprès du service de noms RMI via la classe Naming de sa JVM (méthode bind ou rebind): Le Naming enregistre le nom de l'objet serveur et une souche client (contenant l'@IP et le port de comm.) dans le rmiregistry



RMI : Fonctionnement

3. L'objet client fait appel au Naming de sa JVM pour localiser l'objet serveur (méthode lookup)

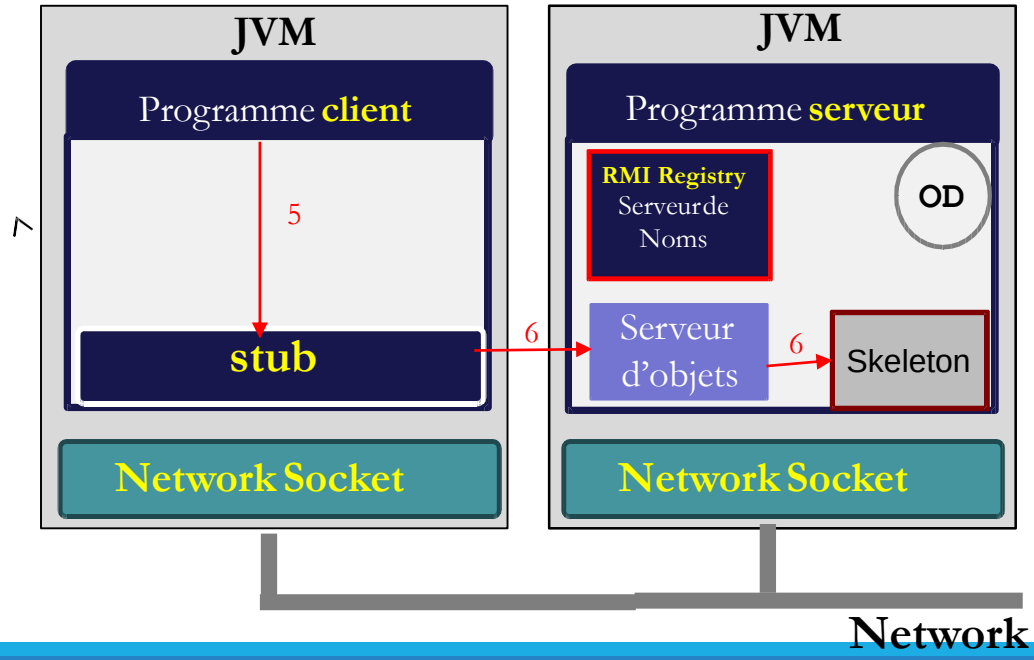
4. Le Naming récupère la souche client de l'objet serveur, …



RMI : Fonctionnement

5. Le stub récupère les paramètres de la méthode

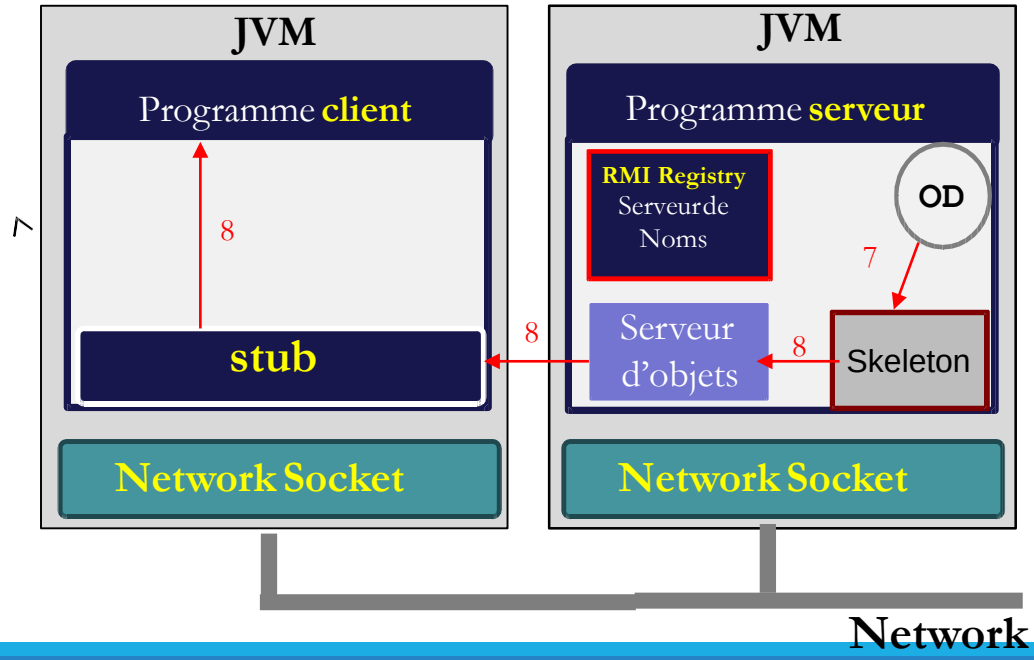
6 . Le stub emballe les paramètres qu'il envoi au skeleton via le serveur d'objets



RMI : Fonctionnement

7. Le skeleton déballe les infos et fait appel à la méthode de l'OD

8 . Il renvoie le résultat après emballage



Etapes de développement

- Etape 1 : Définition de l'interface de l'objet distant :
  - interface héritant de java.rmi.Remote
  - Utiliser pour les méthodes : "throws java.rmi.RemoteException"
  - paramètres de type simple, objets Sérialisables (implements Serializable) ou Distants (implements Remote)
- Etape 2: Ecrire une implémentation :
  - Classe héritant de java.rmi.server.UnicastRemoteObject et implémentant l'interface précédente.
  - Ecrire un main permettant l'enregistrement auprès du Naming
- Etape 3: Génération de la classe stub nécessaire au client et Ecriture du programme client
  - utilisation du Naming pour trouver l'objet distant
  - appel(s) de méthodes.

Remote Method Invocation

- Etape 1 : Définition de l'interface de l'objet distant

Produit.java

```
import java.rmi.*;
public interface Produit extends Remote {
        String getDescription() throws RemoteException;
}
```

- Cette interface doit résider sur le client et le serveur
- Elle doit étendre "Remote"
- Ces méthodes doivent lancer une RemoteException puisque des problèmes réseau peuvent survenir.

Remote Method Invocation

- Etape 2 : Implémentation de l'objet distant

Produit.java

```
import java.rmi.*;

public class ProduitImpl   extends UnicastRemoteObject   implements Produit {
   private String name;
   public ProduitImpl(String n) throws RemoteException
   {  name = n;
   }
   public String getDescription() throws RemoteException {
                        return " Je suis " + name + ". Achetemoi!";
            }
}
```

Remote Method Invocation

Les classes RMI

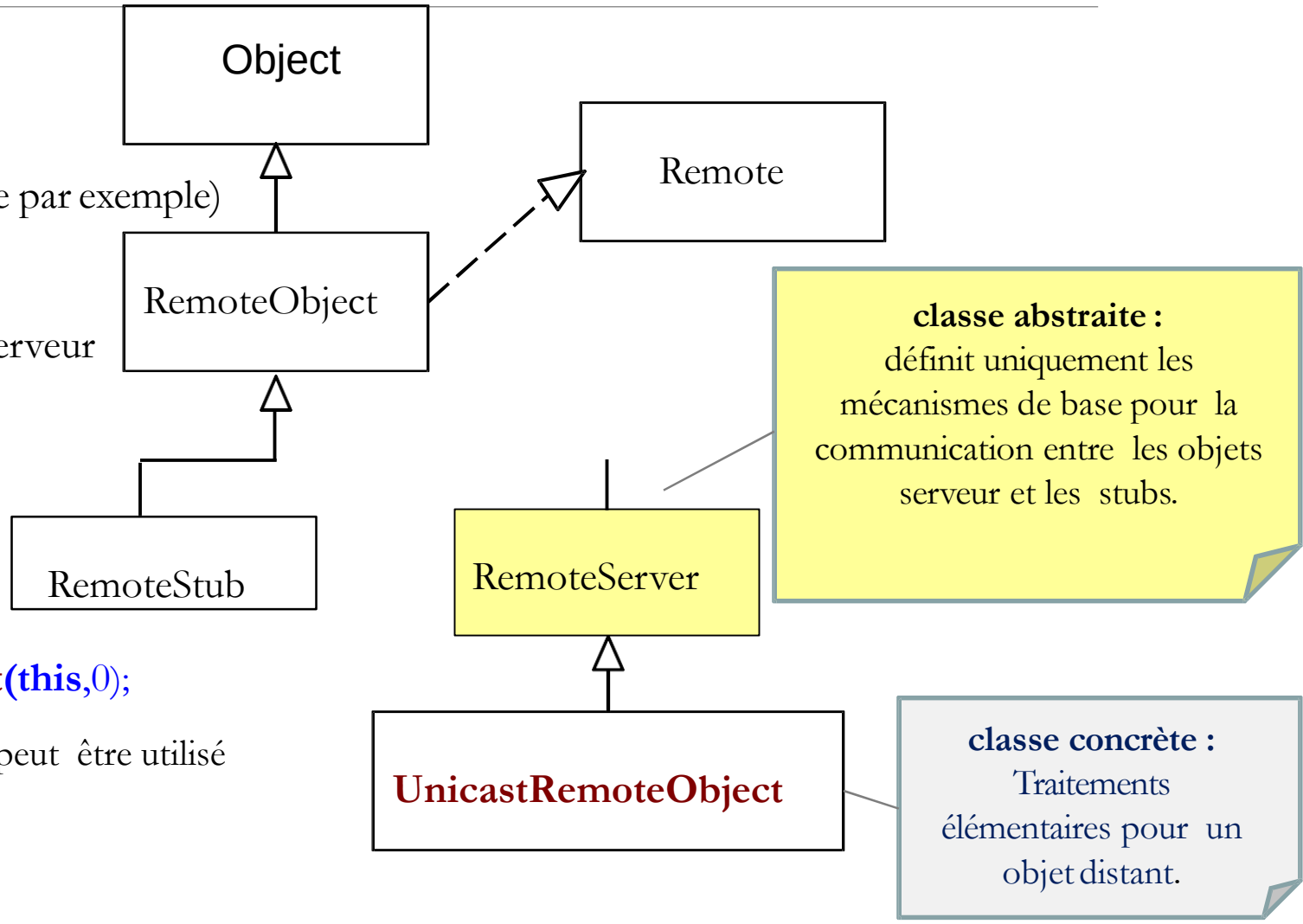
- Il est possible de ne pas étendre **UnicastRemoteObject** (héritage multiple par exemple)
- Dans ce cas, il faut :
  - instancier manuellement les objets serveur

    et les passer à la méthode statique

    exportObject()
  - Dans le **constructeur** de l'objet serveur par exemple :

    UnicastRemoteObject.**exportObject(this**,0);

→ (0 pour indiquer que n'importes quel port peut être utilisé pour écouter les connexions client)



Remote Method Invocation

- Etape 3 : Génération de la classe stub nécessaire au client

javac ProduitImpl.java

- **JDK 1.2**
  - rmic –v1.2 ProductImpl
  - Deux fichiers sont générés Fichier stub Product_Skel.class
- **JDK 5.0**
  - *Toutes les classes stub sont générées automatiquement*
  - *La classe skeleton n'est plus nécessaire à partir de JDK1.2*
- Lancement du serveur de nom (rmiregistry) : on peut aussi appeler dans le main serveur

LocateRegistry.createRegistry(1099);

RMI : Programme Serveur

ProduitServer.java

```
import java.rmi.*;  import
java.rmi.server.*;

public class ProduitServer {

 public static void main(String args[]) {

 System.out.println("Construction des implémentations");

    ProduitImpl ref1 = new ProduitImpl("Sony 40p");

    ProduitImpl ref2 = new ProduitImpl("ZapXpress Microwave");

    System.out.println("Binding implementations to registry");

    Naming.rebind("television", ref1);

    Naming.rebind("microwave", ref2);

    System.out.println("Enregistrement effectué attente de clients...");
```

- Le serveur **enregistre** les objets auprès du serveur de noms en donnant un nom unique et une reference à chaque objet.
- **rebind()** à la place de **bind()** pour éviter l'erreur **AlreadyBoundException** lorsque l'entrée existe déjà

RMI : Programme Serveur

A P I **java.rmi.Naming 1.1**

- static void bind(String name, Remoteobj)
  binds name to the remote object obj. Throws a AlreadyBoundException if the object is already bound.
- static void unbind(String name)
  unbinds the name. Throws a NotBoundException if the name is not currently bound.
- static void rebind(String name, Remoteobj)
  binds name to the remote object obj. replaces an existing binding.
- static String[] list(String url)
  returns an array of strings of the URLs in the registry located at the given URL. The array contains a snapshot of the names present in the registry.

RMI Registry

- Le registre de noms RMI doit s'exécuter avant de pouvoir enregistrer un objet ou obtenir une référence.
- Le registre de noms RMI peut être lancé :
  - par **start rmiregistry**
  - Ou dynamiquement dans la classe qui enregistre l'objet.

java.rmi.registry. LocateRegistry.createRegistry(1099);

Naming.rebind(url, od);

- L'objet UnicastRemoteObject réside sur le serveur. Il doit être actif lorsqu'un service est demandé et doit être joignable à travers le protocole TCP/IP : Nous créons des objets d'une classe qui étend **UnicastRemoteObject**, un **thread séparé** est alors lancé, il garde le programme **indéfiniment** en vie.

Passage de paramètres

- Une référence à un **od** peut être passée en **argument** ou retournée en **résultat** d'un appel dans toutes les invocations (**locales ou distantes**)
- Les arguments locaux et les résultats d'une invocation distante sont toujours **passés par copie** et non par référence : leurs classes doivent implémenter java.io.Serializable
- Si jamais le type n'est pas disponible localement, il est chargé dynamiquement : C'est **java.rmi.server.RMIClassLoader** (un chargeur de classes spécial RMI) qui s'en charge.
- Le processus est le même que pour les applets qui s'exécutent dans un navigateur
- A chaque fois qu'un programme charge du code à partir d'une autre machine sur le réseau, le **problème de la sécurité** se pose.

Gestionnaire de Sécurité

- Il faut donc utiliser un **gestionnaire de sécurité** dans les applications clientes RMI.
- Le comportement par défaut lors de l'exécution d'une application Java est qu'**aucun gestionnaire** de sécurité n'est installé.
- Le gestionnaire de sécurité par défaut pour RMI est
  - java.rmi.**RMISecurityManager**
  - System.setSecurity(**RMISecurityManager**)
- Les Applets, elles, installent un gestionnaire de sécurité assez restrictif :

  AppletSecurityManager
- Pour des applications spécialisées, les programmeurs peuvent utiliser leur propre

  «**ClassLoader** » et « **SecurityManager** » mais pour un usage normal, ceux fournis par

  RMI suffisent

RMI : Programme Client

ProduitClient.java

```
import java.rmi.*;
public class ProduitClient {
 public static void main(String args[]) {
    System.setSecurityManager( new RMISecurityManager());
```

- Par défaut, RMISecurityManager empêche tout le code dans le programme d'établir des connexions réseau.
- Mais ce programme a besoin de connexions réseau :
  - Pour atteindre le « RMI Registry »
  - Pour contacter les « objets serveur »
  - Lorsque le client est déployé, il a aussi besoin de permissions pour charger ces classes de stub.
- Donc, Java exige que nous écrivons un "policy file"

RMI : Programme Client

Policy File : « client.policy »

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

- Autorise l'application à faire des connections réseau sur un port supérieur à 1024. (Le port RMI est 1099 par défaut)

- A l'exécution du client, on doit fixer une propriété système :

```
javac ProduitClient.java
java -Djava.security.policy=client.policy ProduitClient
```

RMI : Programme Client

ProduitClient.java

```
String url  = "rmi://localhost/";
//stub
Produit c1  = (Produit)Naming.lookup(url + "television");
Produit c2  = (Produit)Naming.lookup(url + "microwave");
System.out.println(c1.getDescription());
System.out.println(c2.getDescription());
```

- Le serveur de noms fournit la méthode statique lookup(url) pour localiser un objet serveur. L'URL RMI : "rmi://serveur:[port]/objet"
- c1 et c2 ne font pas référence à des objets sur le serveur. Ils font plutôt référence à un stub qui doit exister sur le client.

RMI : Recapitulatif

Récapitulatif des activités

1. Compiler les fichiers java

```
javac *.java
```

2. Avant JDK1.5: générer les stub

```
rmic –v1.2 ProduitImpl
```

3. Lancer le RMI **registry** (serveur de nom)

```
start rmiregistry
```

4. Lancer le **serveur**

```
start java ProduitServer
```

5. Exécuter le **client**

```
java –Djava.security.policy=client.policy ProduitClient
```

RMI : Recapitulatif

Déploiement

- Préparer le déploiement (JDK1.5)
- Trois dossiers

server/
    ProductServer.class
    ProductImpl.class
    Product.class
client/
    ProductClient.class
    Product.class
    client.policy
download/
    Product.class

download contient les classes utilisées par **RMI registry**, le **client** et le **serveur**.

Exercice :

- On souhaite rendre une Méthode HelloWorld accessible à distance de manière à ce qu'elles définissent l'interface entre le client et le serveur : Ecrire cette interface. (Respectez les règles syntaxiques que doit suivre une interface Java RMI) et les programmes client et Serveur.