Objets distribués

Tour d'horizon, application avec java

partie B- RMI

B- RMI

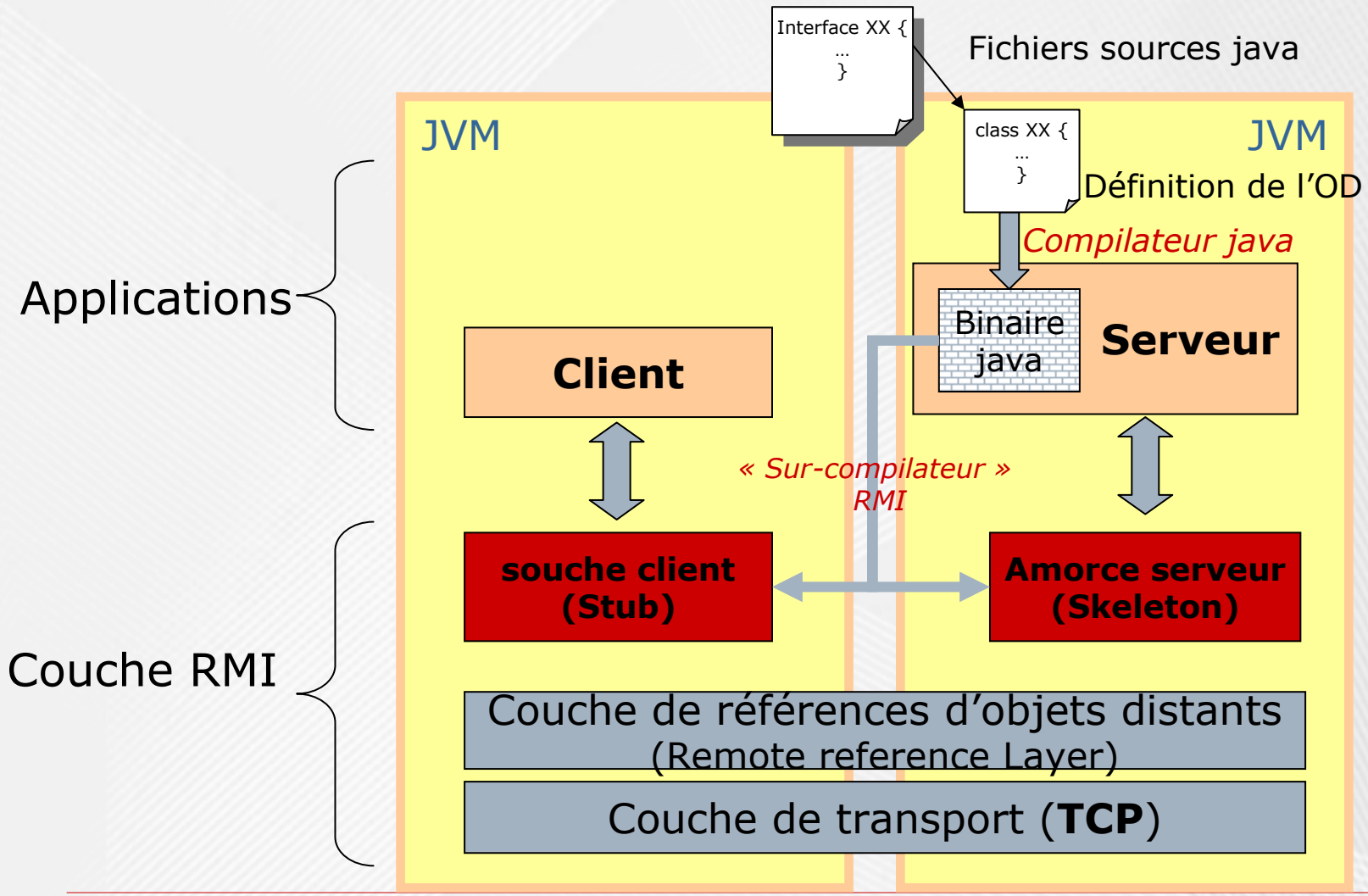
RMI: Présentation

- RMI = Remote Method Invocation
  - gratuit (différent de CORBA)
  - 100 % Java
- RMI propose un système d'objets distribués plus simple que CORBA
- RMI est uniquement réservé aux objets Java
- RMI…
  - fournit un mécanisme permettant l'appel de méthodes entre objets Java s'exécutant sur des machines virtuelles différentes (espaces d'adressage distincts)
  - utilise directement des sockets TCP
  - code ses échanges avec un protocole propriétaire

RMI: Objectifs

- Un objet sur une JVM peut avoir une référence sur un autre dans une autre JVM.
- Rendre transparent l'accès à des objets distribués sur un réseau
- Faciliter la mise en œuvre et l'utilisation d'objets distants Java
- Préserver la sécurité (inhérent à l'environnement Java)
  - RMISecurityManager
  - Distributed Garbage Collector (DGC)

RMI: Architecture



RMI: Principes

- Un objet distant (objet serveur) : ses méthodes sont invoquées depuis une autre JVM
  - dans un processus différent (même machine)
  - ou dans une machine distante (via réseau)
- Un OD (sur un serveur) est décrit par une interface distante Java (ou plus)
  - déclare les méthodes distantes utilisables par le client
- Une invocation distante (RMI) est l'action d'invoquer une méthode d'une interface distante d'un objet distant.
- Une invocation de méthode sur un objet distant a la même syntaxe qu'une invocation sur un objet local.

Un OD se manipule comme un objet local

RMI: La distribution d'objets

- Une référence à un OD peut être passée en argument ou retournée en résultat d'un appel dans toutes les invocations (locales ou distantes)
- Un OD peut être transformé (*cast*) en n'importe quelles interfaces distantes supportées par l'implémentation de l'objet

RMI: Le passage de paramètres

- 3 cas à gérer pour transmettre des paramètres ou la valeur de retour d'une méthode distante :
  - Types simples
  - Objets locaux
  - Objets distants

(Objet distant=Sous-classe de « RemoteObject »)

- RMI fournit une solution pour les 3

RMI: Le passage de paramètres

- Mécanismes:
  - les paramètres de types simples (int, float, ...) sont transmis par copie
  - les paramètres de type objet local sont sérialisés et transmis par copie (cad l'ensemble de ses variables est copié par sérialisation)
    - l'objet doit implémenter l'interface **`java.io.Serializable`**
  - les paramètres de type **objet distant** sont transmis par référence
    - Ca veut dire aussi que l'objet amorce (Stub) est transmis
    - Si jamais un type n'est pas disponible localement, il est chargé dynamiquement (RMIClassLoader)

RMI: Interface

- L'interface constitue le contrat - abstrait - liant objets serveurs et objets clients
- elle est destinée à être implémentée par l'OD et constitue la base d'appel pour les objets clients
- elle définie les signatures (noms, types de retours, paramètres) d'un ensemble de méthodes
- seules ces méthodes pourront être invoquées à distance
- Cette interface RMI est une interface Java classique mais dérivant de la classe **`java.rmi.Remote`**

RMI: L'exception *RemoteException*

- Les objets clients doivent traiter des exceptions supplémentaires comme **`RemoteException`**
- Toute invocation distante de méthode doit lever ou traiter cette exception
- Peut se produire si la connexion a été interrompue ou si le serveur distant ne peut être trouvé

RMI: Les amorces (Stub/Skeleton)

- Programmes jouant le rôle d'adaptateurs pour le transport des appels distants
  - réalisent les appels sur la couche réseau
  - « Codage » des paramètres (marshalling)
- A une référence d'OD manipulée par un client correspond une référence d'amorce
- Les amorces sont générées par le compilateur d'amorces : **rmic**

RMI: L'amorce client (Stub)

- Représentant local de l'OD qui implémente ses méthodes « exportées »
- Transmet l'invocation distante à la couche inférieure *Remote Reference Layer*
- Il réalise le « marshalling » des paramètres d'appels des méthodes distantes
- Dans l'autre sens, il réalise le « demarshalling » des valeurs de retour
- Il utilise pour cela la sérialisation *(=marshalling)* des objets et des données
  - Sérialisation=Encoder l'état d'un objet en mémoire sous la forme d'une suite d'octets
  - Le Stub transforme les paramètres en un flux de données transmissible sur le réseau
  - les objets doivent implémenter l'interface **`java.io.Serializable`**

RMI: L'amorce serveur (Skeleton)

- Réalise le « marshalling » des arguments reçus par le flux de données
- Fait un appel local à la méthode de l'objet distant
- Réalise le « marshalling » de la valeur de retour de la méthode le cas échéant

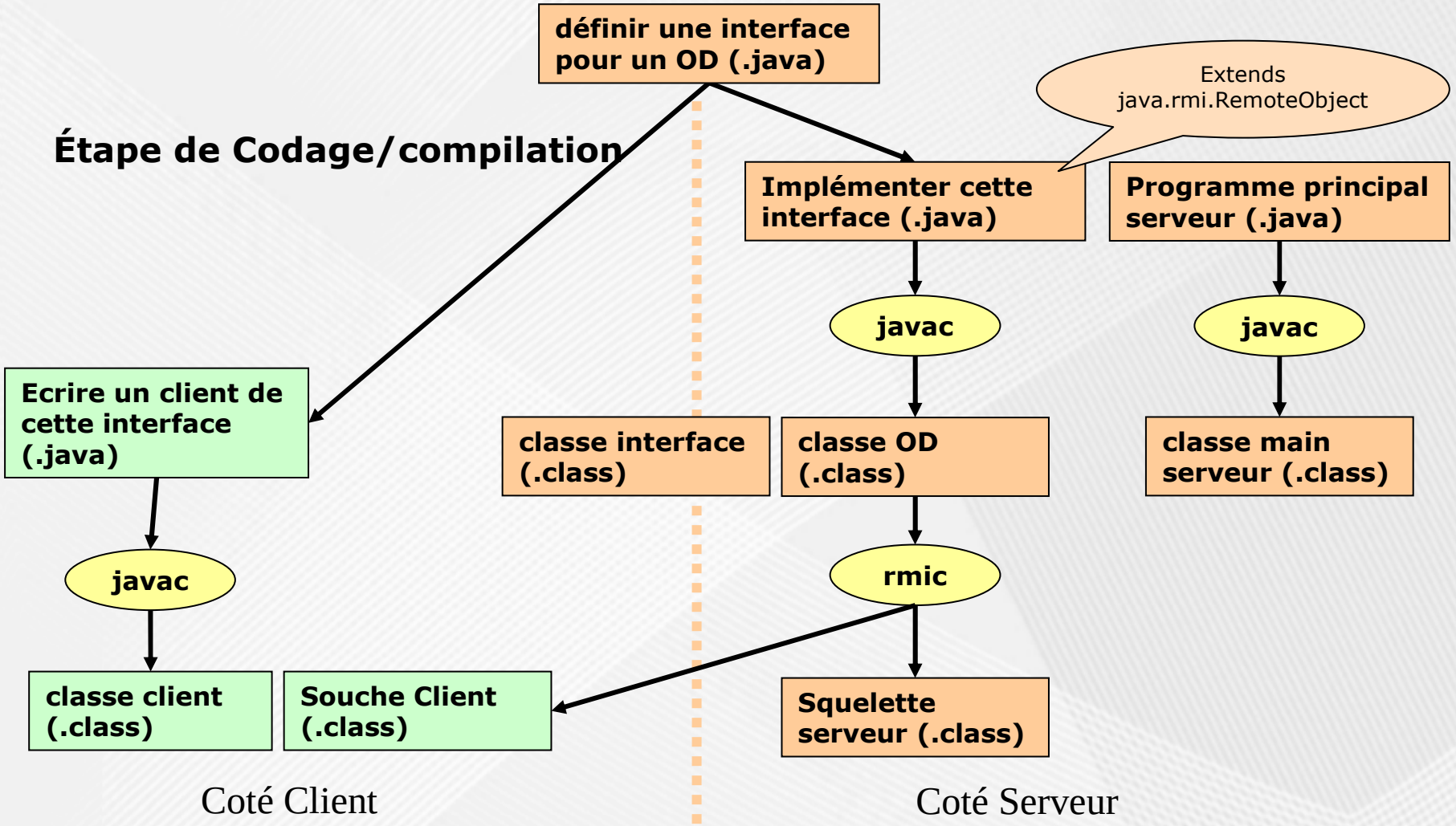
RMI: La couche des références distantes

- Permet l'obtention d'une référence d'objet distant à partir de la référence locale au Stub
- Ce service est assuré par le lancement du programme **`rmiregistry`**
  - à ne lancer qu'une seule fois par JVM, pour tous les objets à distribuer
  - une sorte de service d'annuaire pour les objets distants enregistrés

RMI: La couche de transport

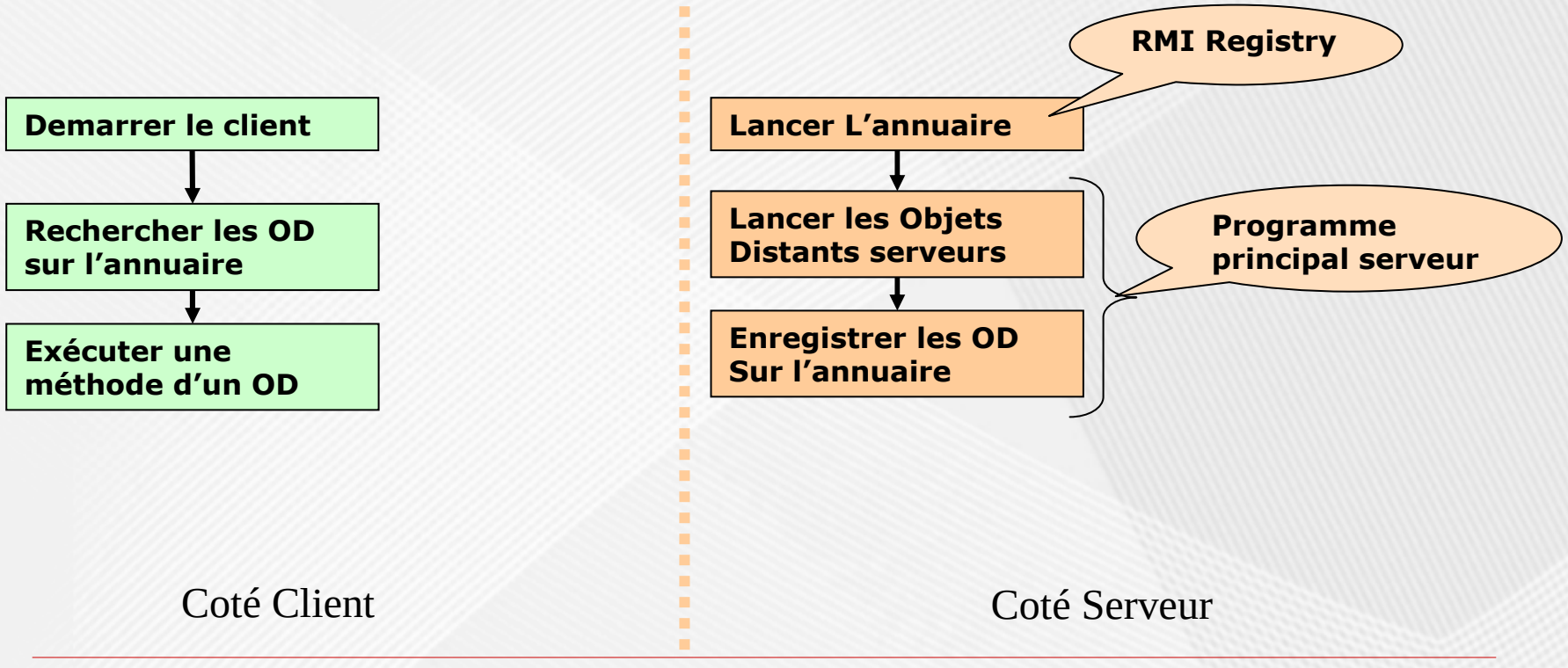
- Connecte les 2 espaces d'adressage (cad 2 JVM)
- Suit les connexions en cours
- Ecoute et répond aux invocations
- Construit une table des OD disponibles
- Réalise l'aiguillage des invocations

RMI: création d'une application (1/2)



RMI: création d'une application (2/2)

Étape d'exécution



RMI: un exemple

- Un « Hello World » distribué :
  - invocation distante de la méthode sayHello() d'un objet distribué avec le mécanisme RMI.
- 4 sources sont nécessaires :
  - Hello.java : l'interface décrivant l'objet distant (OD)
  - HelloImpl.java : l'implémentation de l'OD
  - HelloServer.java : une application principale (main)
  - HelloClient.java : l'application cliente utilisant l'OD

1) définir l'interface pour la classe distante

```
import java.rmi.*;

public interface Hello extends Remote {
    String sayHello() throws RemoteException;
}
```

- les classes d'objets distants sont spécifiées par des interfaces qui doivent dériver de *java.rmi.Remote* et dont **TOUTES** les méthodes lèvent une exception *java.rmi.RemoteException*

2) définir l'implémentation de l'OD

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

public class HelloImpl
       extends UnicastRemoteObject implements Hello {

   int count;

   public HelloImpl() throws RemoteException {super();}

   public String sayHello() {
      count++;
       System.out.println("appel de sayHello() "+count);
        return  "Hello World #"+count+" !";
    }
}
```

Attention : un « piège » ici est l'obligation de re-définir un constructeur qui déclare pouvoir lancer une « RemoteException » - Sinon java va en définir automatiquement un par défaut qui ne surchargera pas celui de la classe parente.

3) définir une application serveur

- C'est le programme qui sera à l'écoute des demandes des clients.
  - il lui faut un SecurityManager spécifique : **`RMISecurityManager`**
  - pour rendre l'objet disponible, il faut l'enregistrer dans l'annuaire « RMIregistry » par la méthode statique :
    - `Naming.rebind("choucou", obj_instance );`

3) définir une application serveur (suite)

- Un objet distant sera accessible par les autres machines par l'url :
  - (On indique son nom et la machine sur laquelle il est exécuté)
  - rmi://nomServeurRMI:port/nomOD

    Exemple (si l'OD est publié en tant que « chouchou »):
    - rmi://serveur.fr/chouchou
    - Ou encore //serveur.fr/chouchou
  - « rmi://» et «:port » facultatifs
  - Port par defaut:1099
  - « nomServeurRMI » vaut localhost par défaut

- Cet "url RMI" sera utilisé par les clients pour interroger le serveur d'annuaire (rmiregistry) grâce à l'appel :

  Naming.lookup(url);

Service de nommage RMI

- Classe java.rmi.Naming
  - Toutes les méthodes sont static

- 1. void bind(String,Remote) enregistre un objet
- 2. void rebind(String,Remote) réenregistre un objet
- 3. void unbind(String) désenregistre un objet
- 4. String[] list(String) liste des noms d'objets enregistrés
- 5. Remote lookup(String) recherche d'un objet

- Pour lesquels:
  - Les paramètres String correspondent à des URL d'objets RMI
  - les paramètres « remote » à des instances d'objets distants
  - Les points 1 2 3 accessibles coté serveur uniquement

3) définir une application serveur (suite)

```
public class HelloServer {
   public static void main(String args[]) {

        // Création et installation du manager de sécurité
        System.setSecurityManager(new RMISecurityManager());

        try {
                // Création de l'OD
                HelloImpl obj = new HelloImpl();

                // Attache cet objet (instance) au nom « chouchou"
                String nom="chouchou";
                Naming.rebind(nom, obj);

                System.out.println(nom+" declaré dans la registry");

        } catch (Exception e) {
                System.out.println(
                "HelloImpl err: "+e.getMessage());
        }
   }
}
```

- Le programme reste actif tant que obj reste enregistré dans le runtime RMI (service de nommage)
- pour le dé-senregistrer: UnicastRemoteObject.unexportObject()

4) créer les amorces

- Compiler les classes

```
> javac *.java
```

- compiler les amorces sur la classe compilée de l'implémentation de l'OD avec RMIC:

```
> rmic HelloImpl
```

- 2 classes sont alors créées représentant la souche (ou Stub) et le Skeleton :

  `HelloImpl_Stub.class`

  `HelloImpl_Skel.class`

5) Lancer **rmiregistry** et l'application serveur

- D'abord lancer ***rmiregistry*** (service d'annuaire d'objets distants implémentant la couche des références d'objets)

```
> rmiregistry& (unix)
> Start rmiregistry ( windows )
```

- Lancer l'application serveur :

```
> java HelloServer
```

  - Cela rend HelloImpl disponible sur le réseau
  - *(la commande est partielle, il manque la spécification du fichier de politique de sécurité – voir plus loin dans le cours)*

6) définir l'application cliente utilisant l'OD

```
import java.rmi.*;

public class HelloClient {
   public static void main(String args[]) {

        //"od" référencera l'objet distant qui implémente l'interafce "Hello"
        Hello od = null;

        try {
                // Création et installation du manager de sécurité
                System.setSecurityManager(new RMISecurityManager());

                // Recherche de l'OD
                od = (Hello)Naming.lookup("rmi://"+args[0]+ "/chouchou");

                String message = od.sayHello();
                System.out.println("message de l'OD:"+message);

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

Donner l'IP !

RMI: Chargement dynamique des amorces

- Rappel : un objet client ne peut utiliser un objet distant qu'au travers des amorces
- RMI permet l'utilisation des OD dont les amorces ne sont pas disponibles au moment de la compilation
- A l'exécution, RMI réclamera au serveur l'amorce cliente manquante et la téléchargera dynamiquement (byte code)

RMI: Chargement dynamique de classes

- Plus généralement, le système RMI permet le chargement dynamique de classes comme les amorces, les interfaces distantes et les classes des arguments et valeurs de retour des appels distants
- C'est un chargeur de classes spécial RMI qui s'en charge : `java.rmi.server.RMIClassLoader`

RMI: gestion de la sécurité

- Si aucun Security Manager n'est installé
  - Tous les chargement de .class doivent se faire à partir du CLASSPATH local.
  - Mauvais pour le déploiement d 'application distribuées!

- Le gestionnaire de sécurité par défaut pour RMI est `java.rmi.`RMISecurityManager
  - Obligatoire pour les applications standalone (faire appel à System.setSecurityManager())
  - Pour les applets, c'est l'AppletSecurityManager (existe par défaut) qui s'en charge
- RMISecurityManager autorisera le téléchargement dynamique de classes (avec *RMIClassLoader*) selon des règles définies dans un fichier de paramètres :
  - Un fichier de « politique de sécurité » (policy) doit etre fournis

RMI: stratégie de sécurité

- Il est conseillé d'utiliser des fichiers de configuration de la securité
- Exemple de fichier (nommé « policy ») :

```
grant {
      // autorise tout les accès
      permission java.security.AllPermission;
};
```

- Puis l'utiliser à l'appel du client et du serveur:
  - `java -Djava.security.policy=policy HelloServeur`
  - `java -Djava.security.policy=policy HelloClient 127.0.0.1`

- Format :

```
grant signedBy "signer_names", codeBase "URL" {
permission permission_class_name "target_name", "action", signedBy "signer_names";
 ....
permission permission_class_name "target_name", "action", signedBy "signer_names";
};
```

RMI: stratégie de sécurité

- Format de fichier « policy » :

```
grant signedBy "signer_names", codeBase "URL" {
permission permission_class_name "target_name", "action", signedBy "signer_names";
 ....
permission permission_class_name "target_name", "action", signedBy "signer_names";
};
```

- Exemple :

```
grant signedBy "user1" { permission java.io.FilePermission "/tmp/*
   "read,write";
};

Ou

grant signedBy "sysadmin", codeBase "file:/home/sysadmin/*" {
permission java.security.SecurityPermission "Security.insertProvider.*";
permission java.security.SecurityPermission "Security.removeProvider.*";
permission java.security.SecurityPermission "Security.setProperty.*";
};
```

RMI: Les packages

- **`java.rmi`** : pour les classes côté client (accéder à des OD)
- **`java.rmi.server`** : pour les classes côté serveur (création des OD) notamment *java.rmi.server.UnicastRemoteObject*
- **`java.rmi.registry`** : lié a l'enregistrement et à la localisation des OD
- **`java.rmi.dgc`** : pour le *Garbage Collector* des OD