

M.3.3.2 Programmation Objet Avancée

RMI et objets distribués

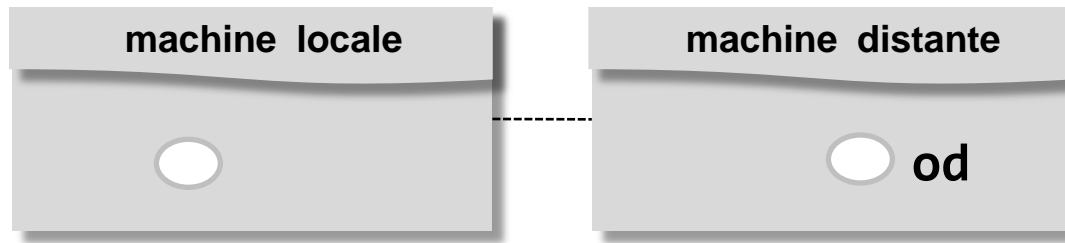




- **Pourquoi RMI ?**
- **Principe de fonctionnement**
- **Développement RMI**

■ Contexte

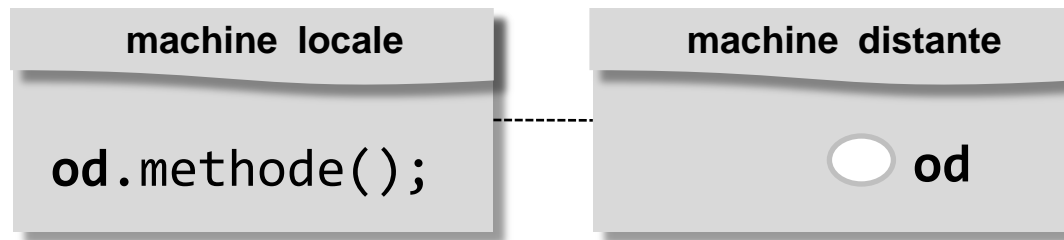
- ➔ Un objet peut demander à un autre objet de faire un travail (généralement spécialisé)



- ➔ Le **problème** est que l'objet qui fait le travail n'est pas localisé sur la même MV. Et peut **ne pas être** implémenté en Java.

Système distribué objet : l'idéal serait de pouvoir

- ➔ Invoquer une méthode d'un **objet distant** (od) de la même manière que *s'il était local*.



- ➔ Demander à un service « dédié » de renvoyer l'adresse de l'od ...

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

... sans savoir où l'objet se trouve.

Systeme distribue objet : l'idéal serait de pouvoir

- ➔ Passer un **od** en paramètre à une méthode (distante ou locale)

```
objetLocal.methode(od);  
oDistant.methode(od);
```

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

```
od = oDistant.methode() ;
```

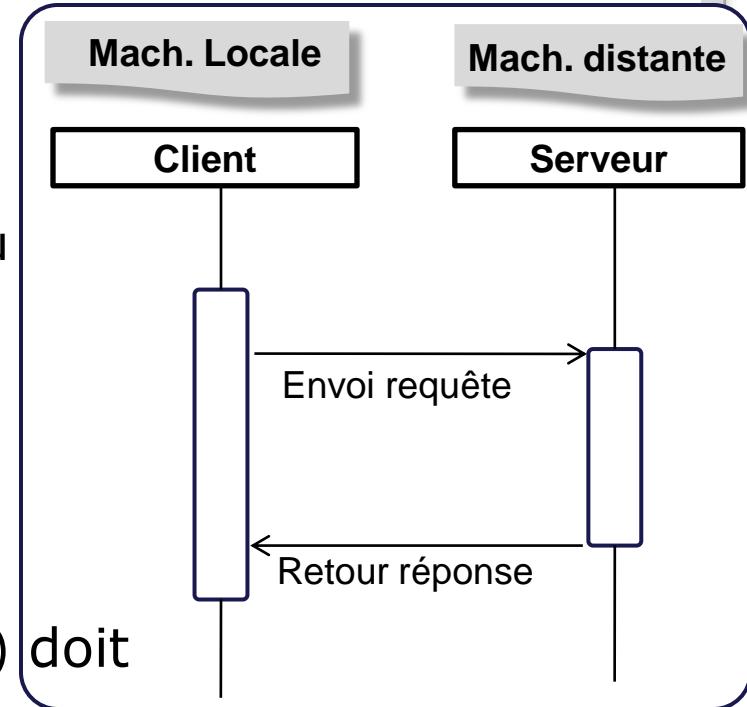
Systeme distribué objet : la difficulté ?

→ 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

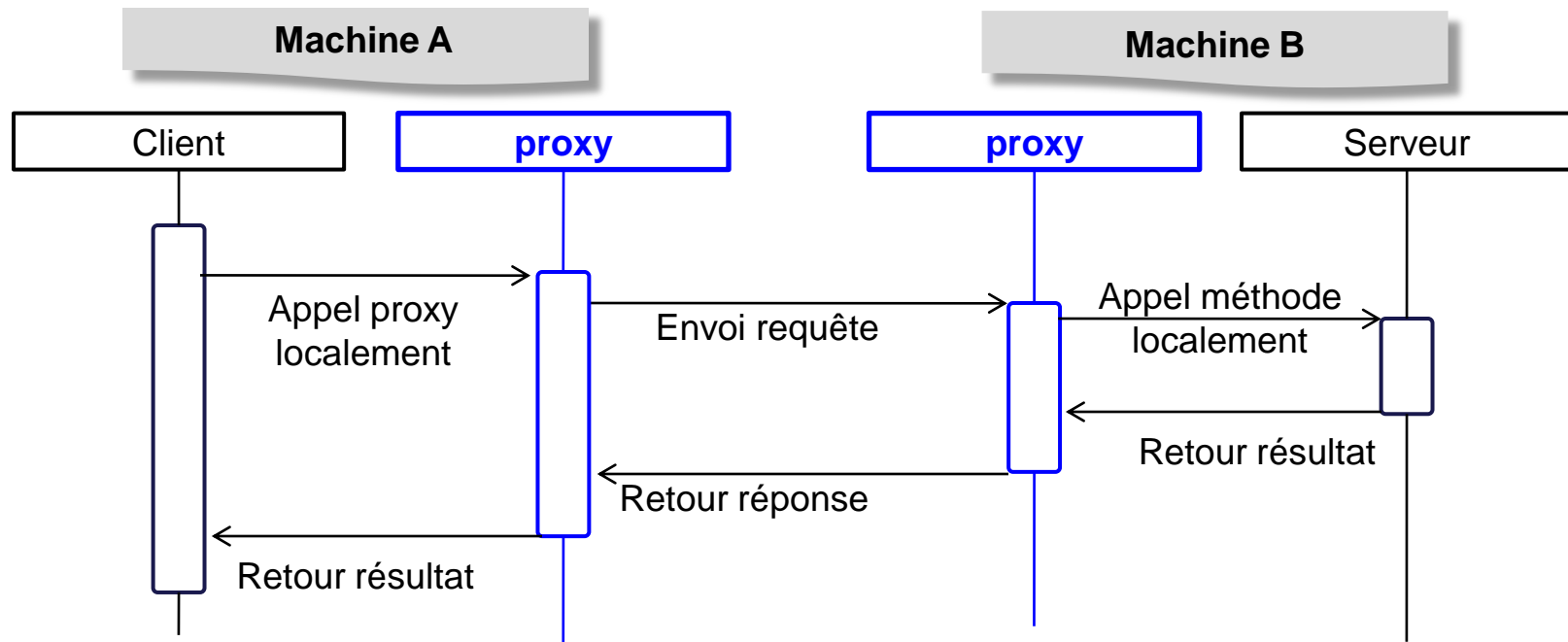
→ A distance, le programme (serveur) doit

- *analyser* la requête,
- *évaluer* la réponse,
- *formater* la réponse pour la transmettre au client



Solution: **proxys**

- ➔ 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



■ Comment les proxys communiquent-il ?

➔ Cela dépend de la technologie d'implémentation.

Généralement trois choix :

- ▶ **CORBA** (**C**ommon **O**bject **R**equest **B**roker **A**rchitecture)
- ▶ **SOAP** (**S**imple **O**bject **A**ccess **P**rotocol)
- ▶ **RMI** (Java **R**emote **M**ethod **I**nvocation)
- ▶ **COM** de Microsoft («oublié » au profit de **SOAP**)

■ Comment les proxys communiquent-il ?

➔ 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)

■ Comment les proxys communiquent-ils ?

➔ SOAP

- est aussi neutre vis-à-vis des langages de programmation.
- Utilise un format de transmission basé sur **XML**
- L'interface de description est spécifiée dans un langage spécial : **WSDL** (Web Services Description Language)

■ Comment les proxys communiquent-ils ?

- ➔ **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**.*

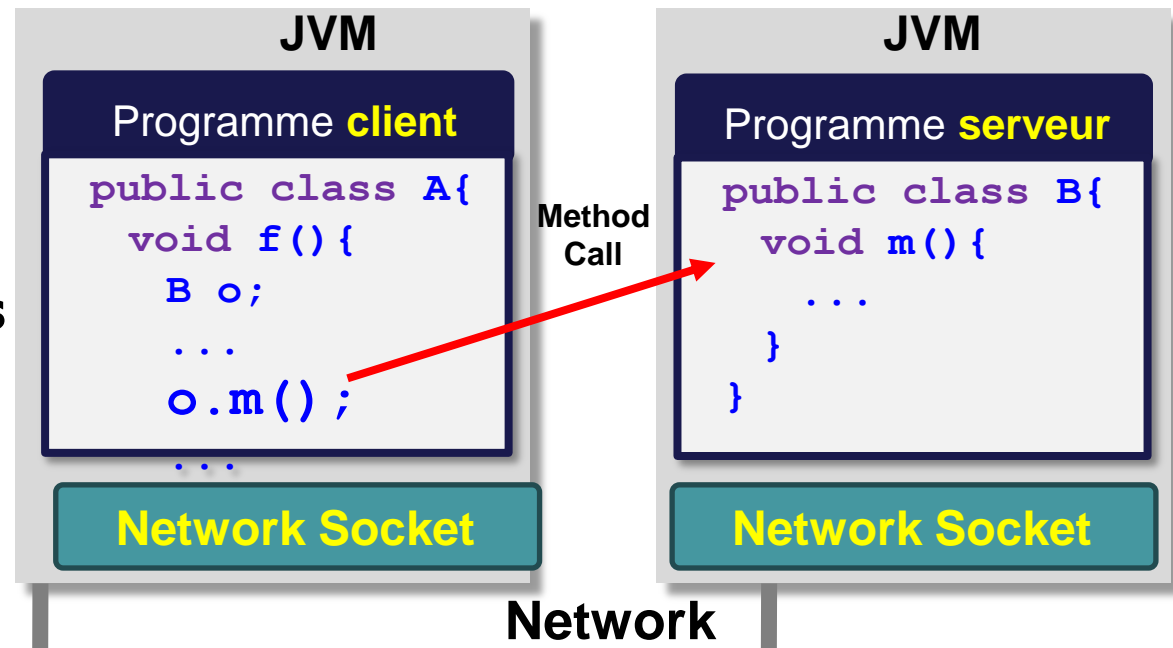
■ Comment les proxys communiquent-ils ?

➔ RMI

- RMI utilise directement les **sockets**
- RMI code ses échanges avec un protocole propriétaire :
RMP (**R**emote **M**ethod **P**rotocol)

- ➔ 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.

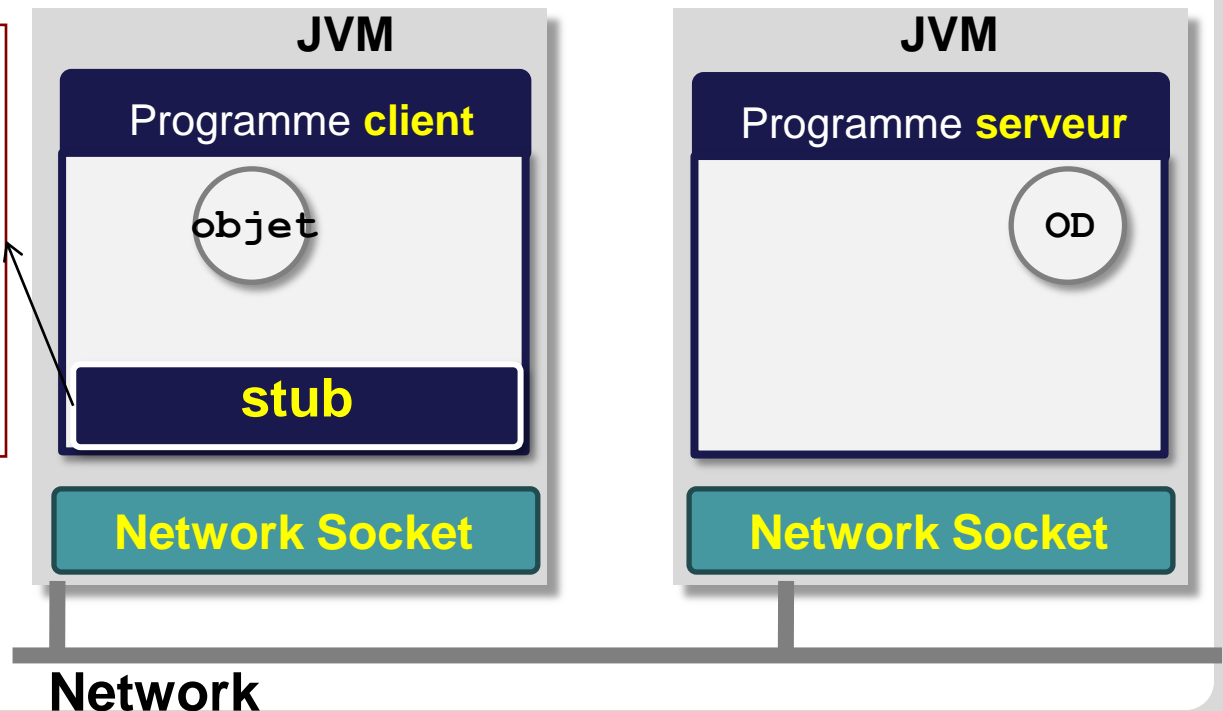
RMI prend en charge ces détails



- 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'OD



→ Le **client** utilise toujours des variables de type interface

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

Le **client** n'a pas de connaissance sur le type d'implémentation

→ A l'appel la variable fait référence au **stub**

```
Warehouse centralWarehouse = ...
```

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

→ La syntaxe est la même que pour un appel local

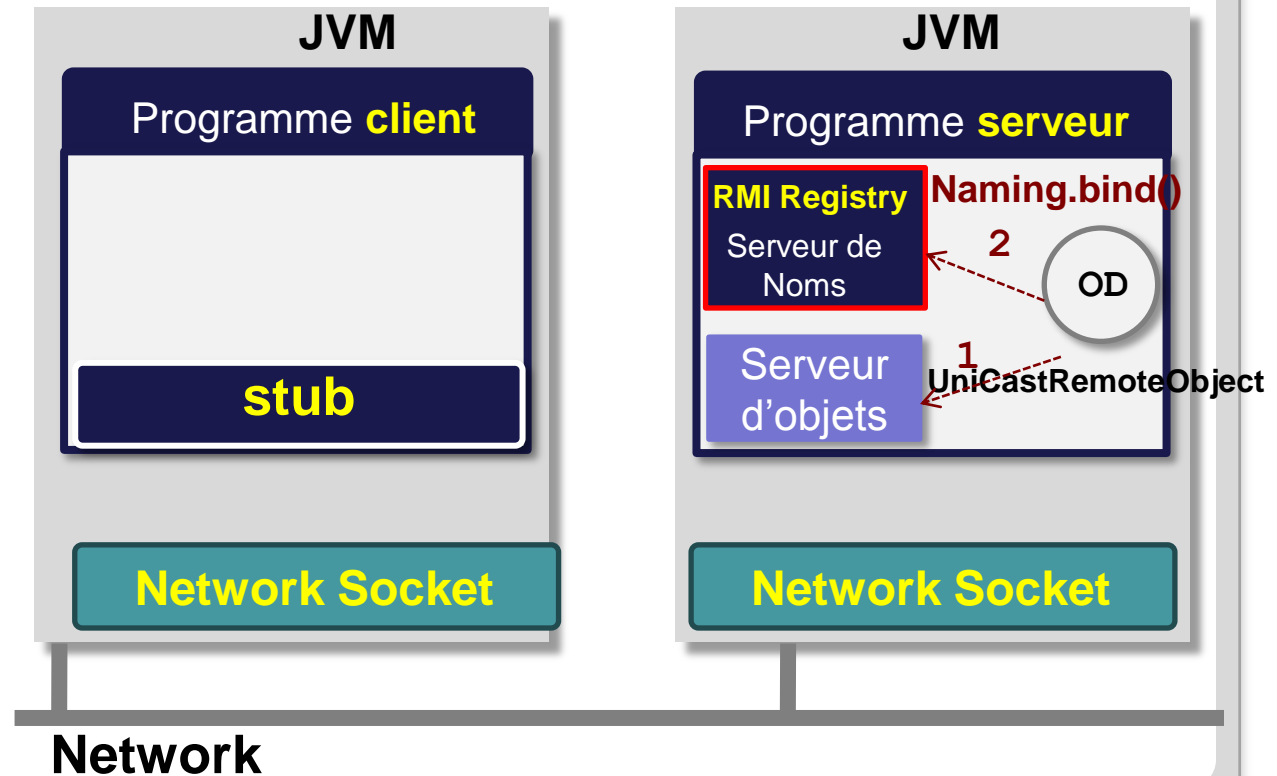
```
int q = centralWarehouse.getQuantity("Super Cleaner");
```

→ 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 maîtrisées

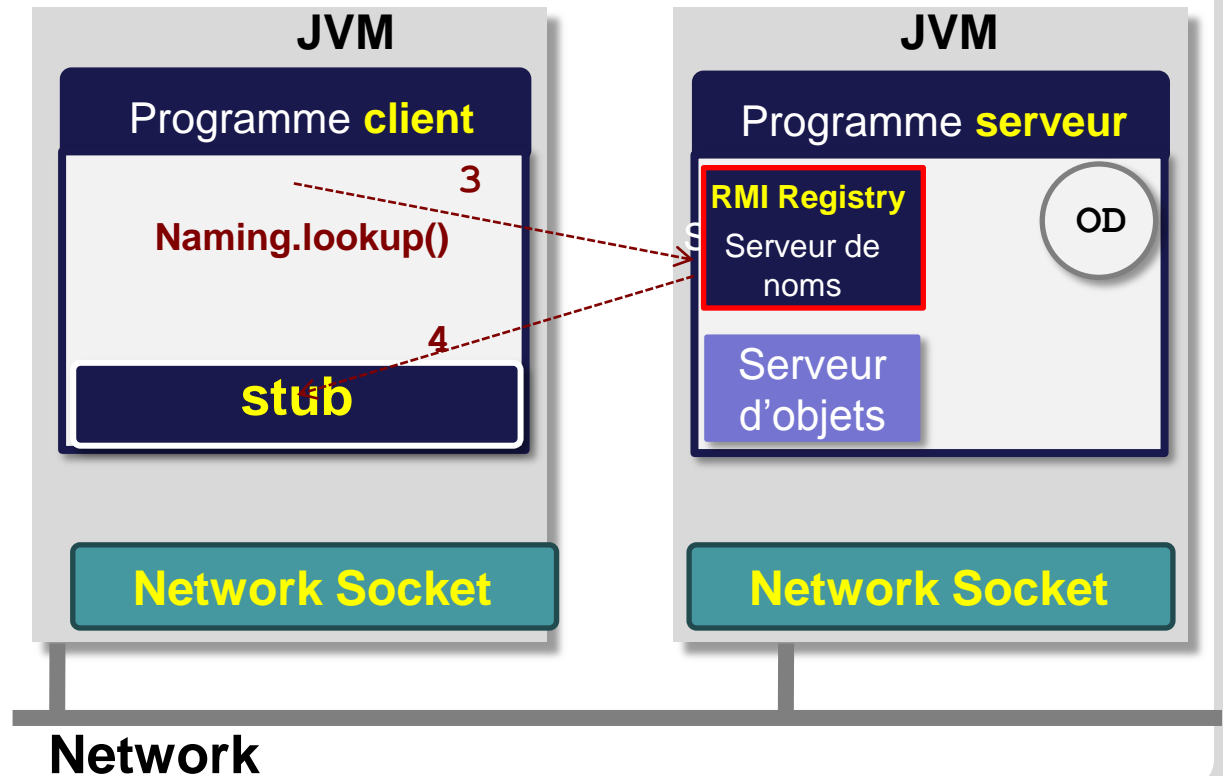
■ Entités : Client – Serveur– RMI Registry

1. Exposer l'OD via `UniCastRemoteObject`
2. L'OD doit s'enregistrer auprès du **Serveur de Noms**



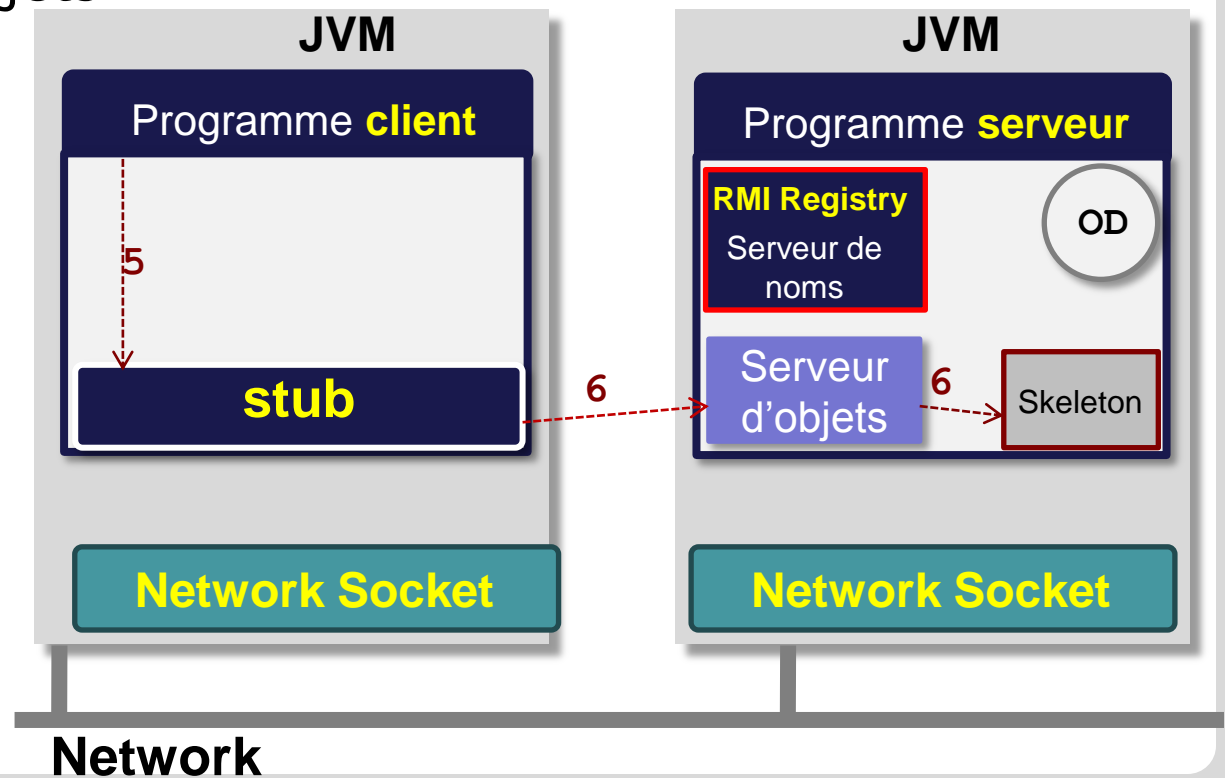
■ Entités : Client – Serveur– RMI Registry

3. Demander l'existence de l'OD
4. Récupérer une instance de l'OD



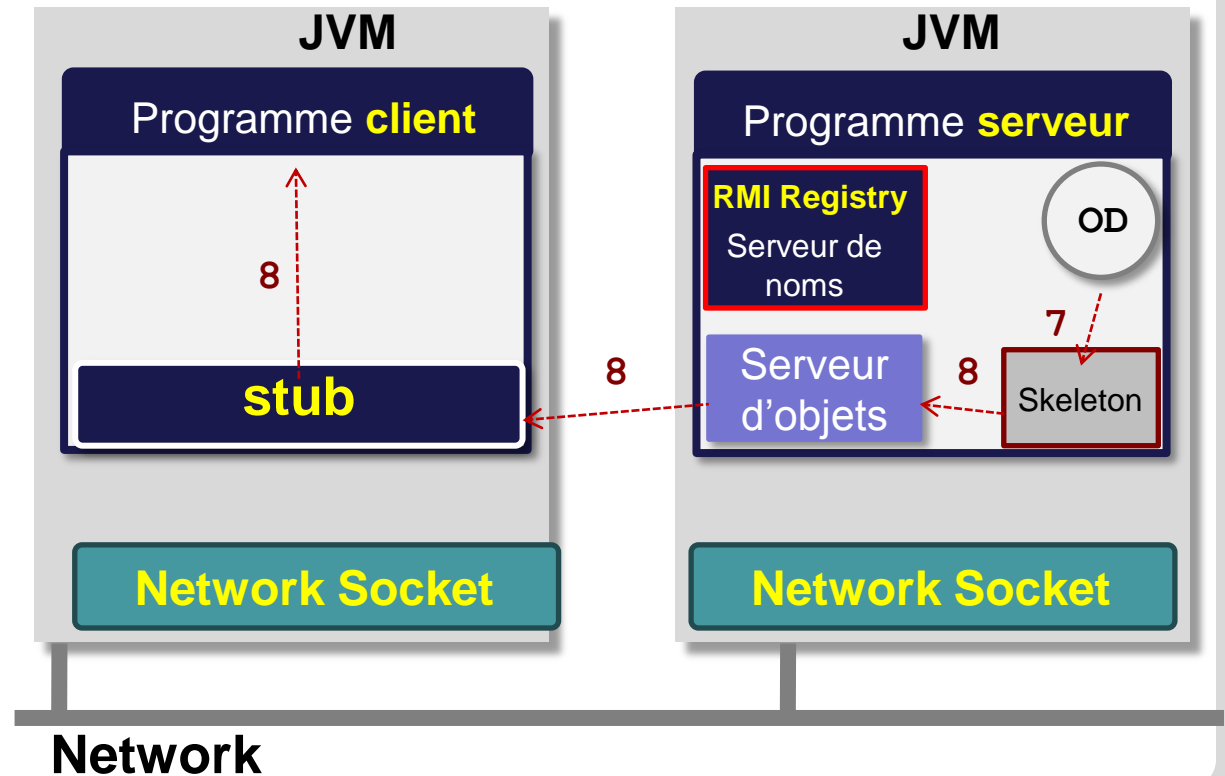
■ Entités : Client – Serveur– RMI Registry

5. Le **stub** récupère les paramètres de la méthode
6. Le **stub** emballe les paramètres qu'il envoie au **skeleton** via le serveur d'objets



■ Entités : Client – Serveur– RMI Registry

7. Le **skeleton** déballe les infos et fait appel à la méthode de l'OD
8. Il renvoie le résultat après emballage



■ Interface pour 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.

■ Implémentation de l'objet distant

ProduitImpl.java

```
import java.rmi.*;
import java.rmi.server.*;
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 + ". Achete moi!";
    }
}
```

■ Génération de la classe stub nécessaire au client

```
javac ProduitImpl.java
```

— JDK 1.2

- `rmic -v1.2 ProductImpl`
- Deux fichiers sont générés

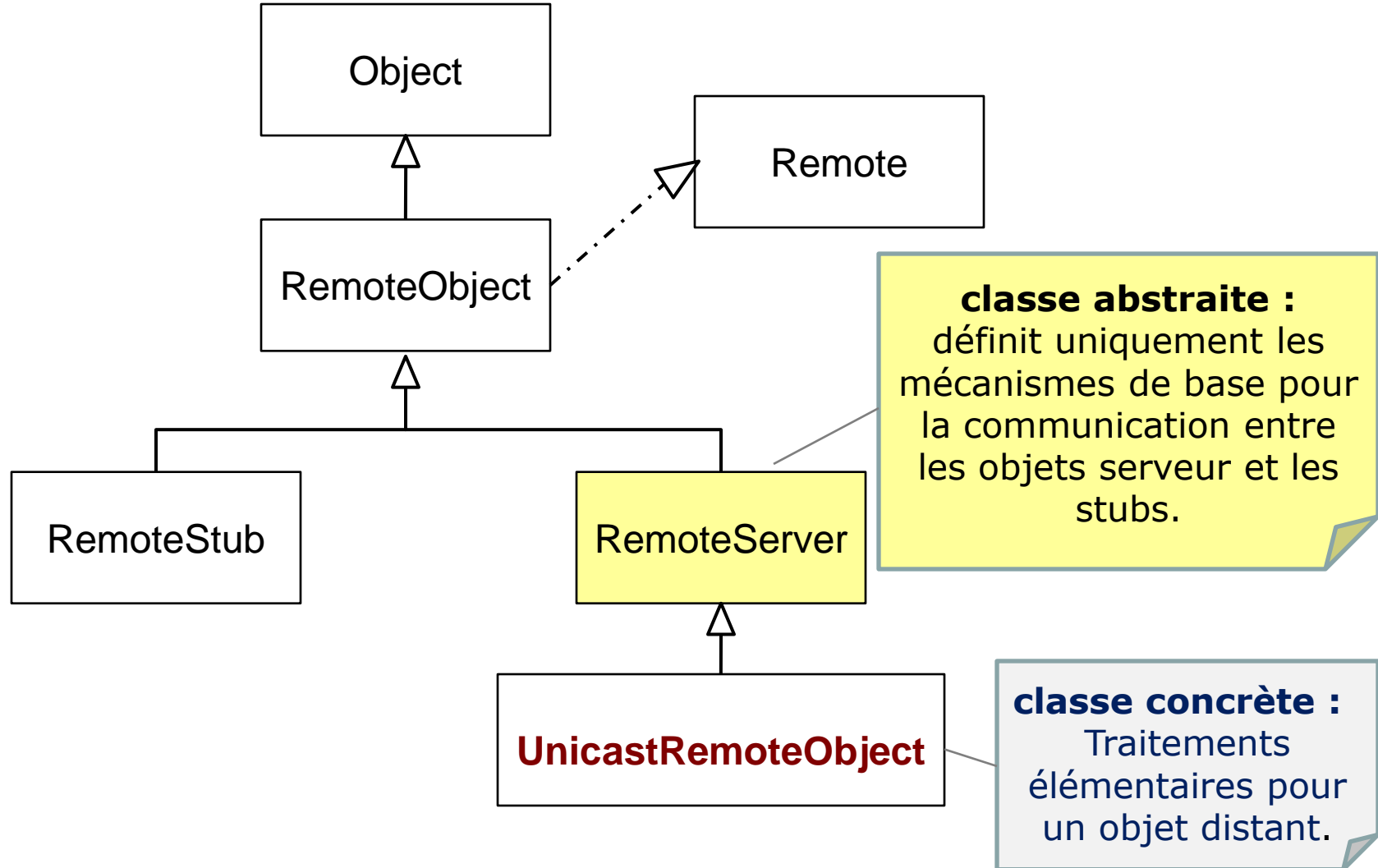
Fichier stub

`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*

■ Les classes RMI



■ 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'importe quel port peut être utilisé pour écouter les connexions client)

■ Le 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");
    }
}
```

- ◆ Le serveur crée les objets **distants**

■ Le programme serveur

ProduitServer.java

```
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

■ Le programme serveur

API

java.rmi.Naming 1.1

- `static void bind(String name, Remote obj)`
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, Remote obj)`
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.

■ Le programme serveur

- ◆ 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 **rmiregistry**
 - Ou dynamiquement dans la classe qui enregistre l'objet.

```
java.rmi.registry.LocateRegistry.createRegistry(1099);  
Naming.rebind(url, od);
```

■ Lancer le serveur

- 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

- **Où est donc l'attente ?**

En fait nous créons des objets d'une classe qui étend `UnicastRemoteObject`, un **thread séparé** est alors lancé, il garde le programme **indéfiniment** en vie .

- 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.RMIBClassLoader` (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.

- ➔ 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.**

■ Le programme client

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

ProduitClient.java

- ◆ 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***"

■ 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
```

■ Le 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

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
```

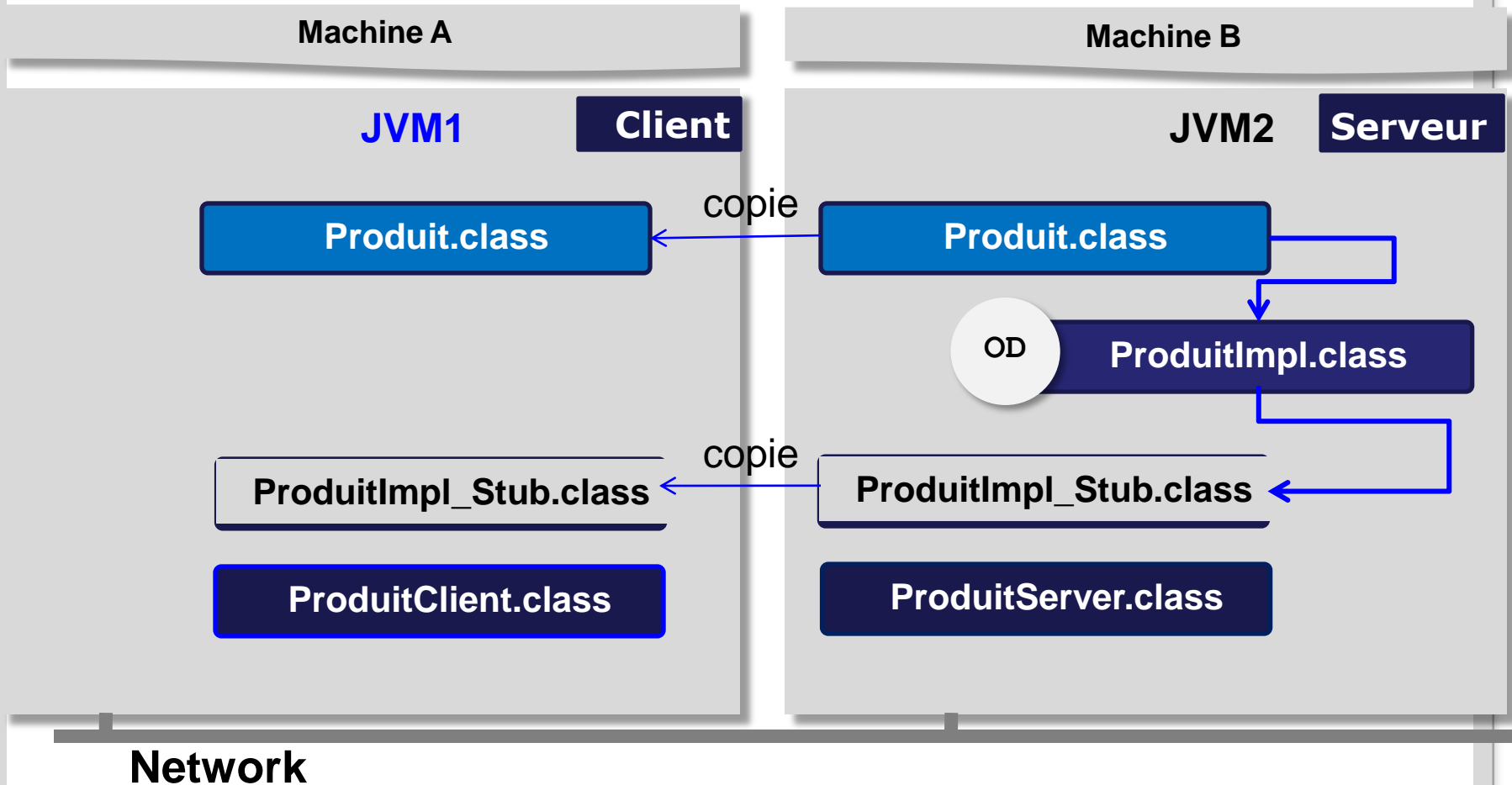
4. Lancer le **serveur**

```
start java ProduitServer
```

5. Exécuter le **client**

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

Fichiers en jeu



■ 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**.



M.3.3.2 Programmation Objet Avancée