

# MRI-COURS 5: JAVA-RMI

yoann.maurel@irisa.fr



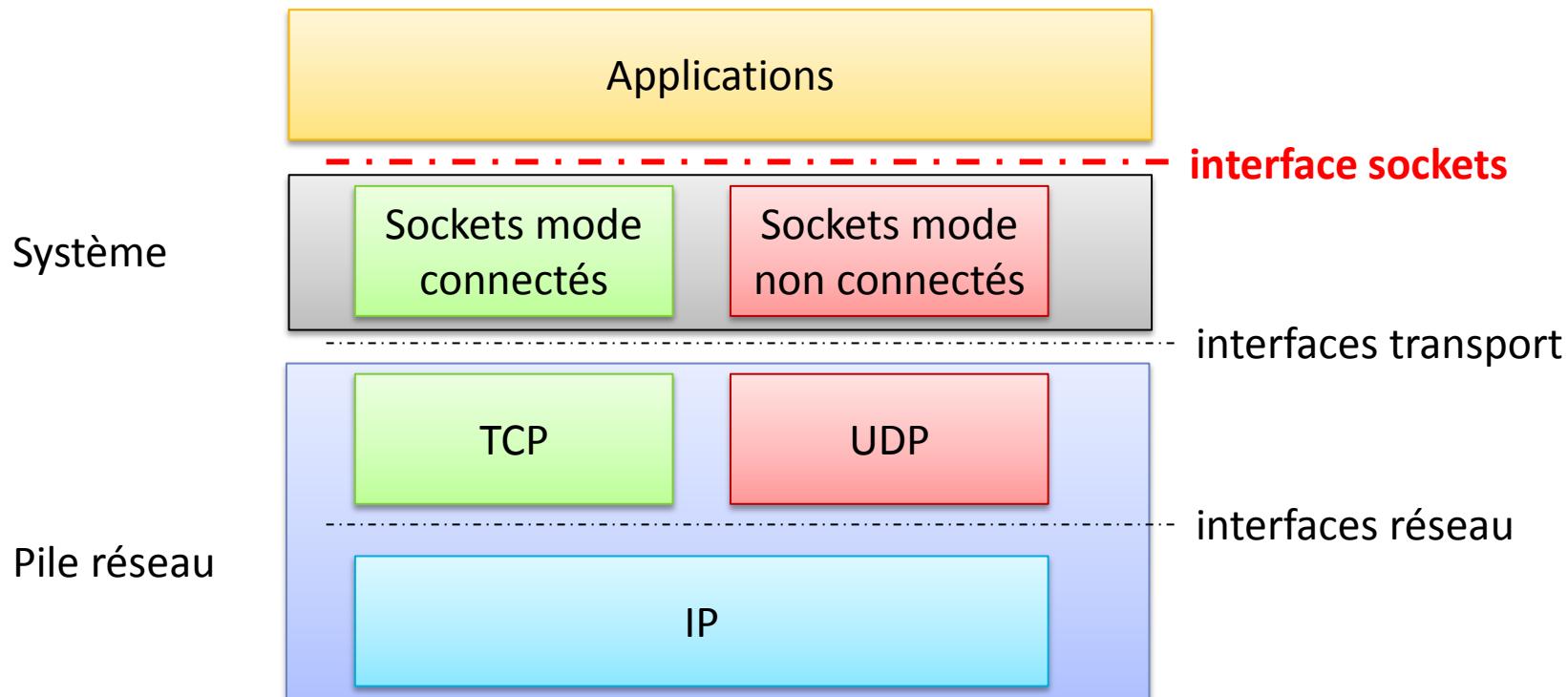
# Les objets répartis

Motivations et Principes

# Les sockets (package java.net)



- ▶ Les sockets = API du package java.net
  - ▶ font l'interface entre les programmes d'applications et les couches réseaux



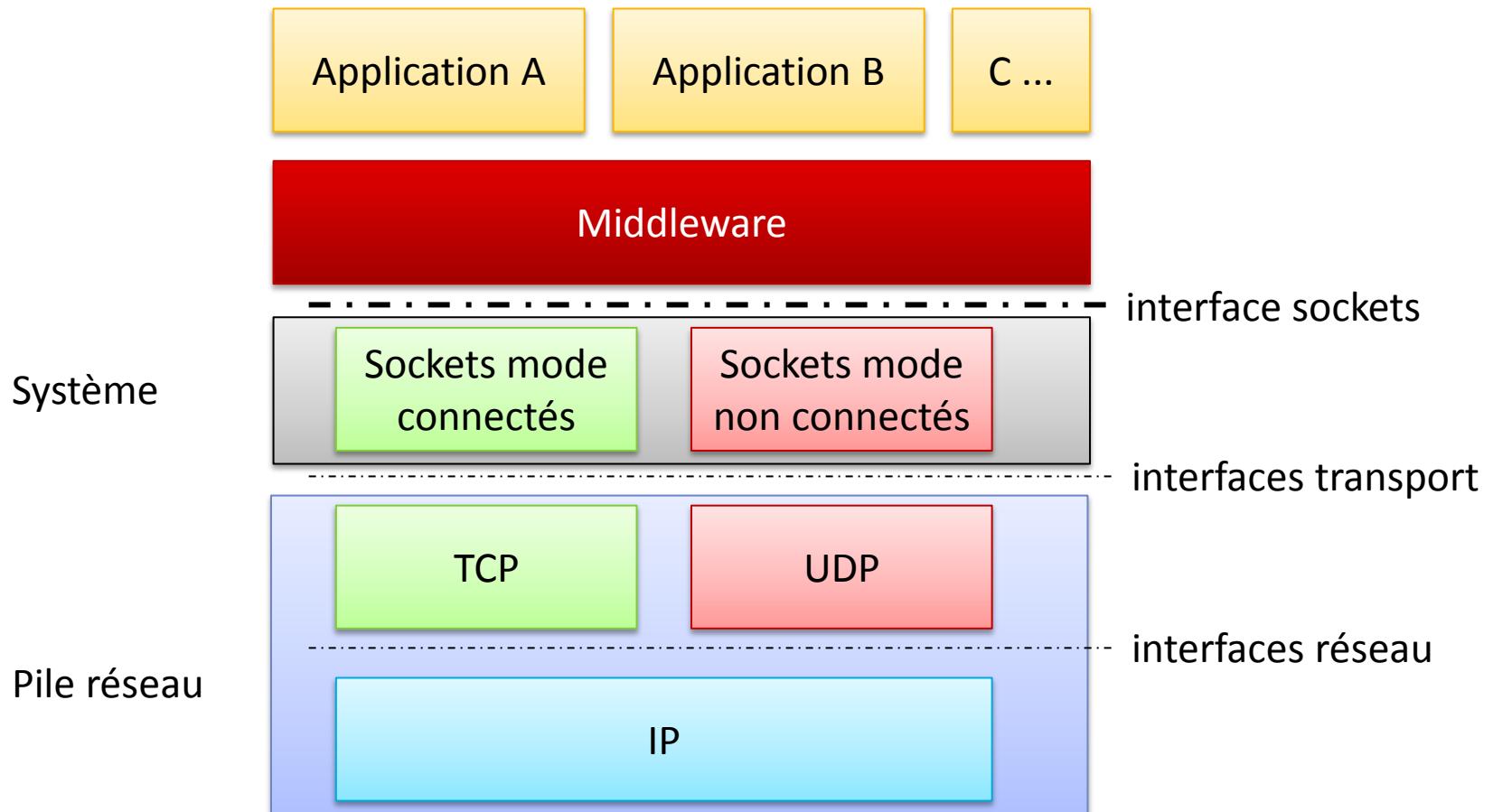
# Les sockets sont complexes

---

- ▶ Les sockets sont trop bas niveau :
  - ▶ allocation des ressources manuelle
    - ▶ sockets, threads, ...
  - ▶ gestion de la concurrence pour répondre à plusieurs clients
    - ▶ pool de threads, Java NIO, ...
  - ▶ implémentation d'un protocole de communication
    - ▶ gérer le format des données
      - endianess, charset, ....
    - ▶ gérer l'ordre des appels
    - ▶ ...
  - ▶ emballage/déballage des données des messages (sérialisation)
  - ▶ pas de découverte de ressources

# Notion de Middleware

- On ajoute une couche entre le système et les applications



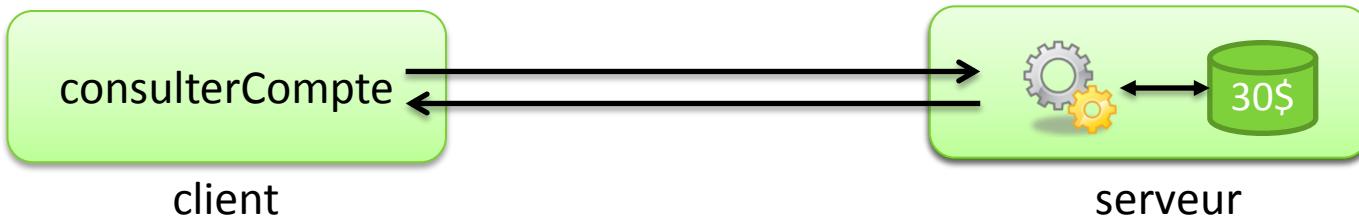
# Objectif de la Couche middleware

---

- ▶ Middle - Software : logiciels du milieux
  - ▶ on parle de la couche middleware ou des logiciels middlewares
  - ▶ en français intergiciels
- ▶ Masque l'hétérogénéité des machines et systèmes
- ▶ Masque la répartition des traitements de données
- ▶ Définissent une API + modèle de programmation

# Les appels distants (RPC)

- ▶ RPC : Remote Call Procédure.



- ▶ Années 80 : appel de procédures à distance d'un ordinateur à un autre
  - ▶ exemple : Sun-RPC
- ▶ Problèmes :
  - ▶ savoir qui traite la méthode au sein du serveur (namespace)
  - ▶ paramètres retour et appel = types primitifs
  - ▶ programmation procédurale
  - ▶ gestion de l'hétérogénéité des formats (cf XDR de Sun)



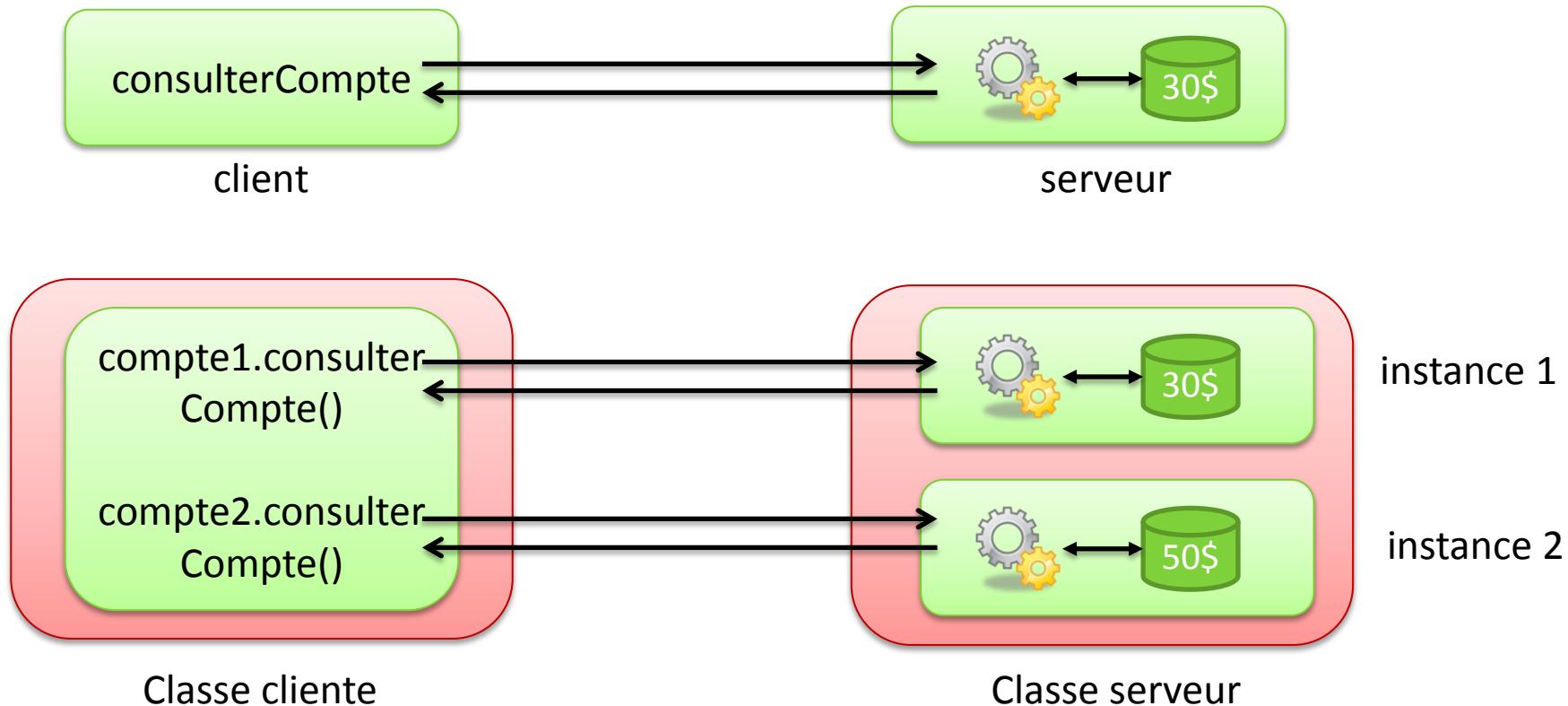
# Distributed Objects to the rescue ...



- ▶ Les applications modernes sont de plus en plus conçues en utilisant les principes de l'OOP:
  - ▶ modularité, encapsulation, composition, polymorphisme, "réutilisation"
  - ▶ les objets définissent la sémantique des données (typage)
  - ▶ la communication se fait par appel de méthodes décrites par des interfaces
- ▶ On souhaite utiliser les mêmes principes pour la communication entre applications distantes
  - ▶ appel de méthodes distantes sur un objet
  - ▶ l'objet devient l'unité de désignation et de distribution

# Les appels de méthodes sur des objets distants

- ▶ Mis en œuvre par CORBA et RMI par exemple
- ▶ Extension du concept aux objets



# Les appels de méthodes sur des objets distants

---

- ▶ Il faut :
  - ▶ avoir une référence sur l'objet
    - ▶ contient ce qui est nécessaire à l'appel distant (localisation, protocoles, ...)
  - ▶ connaître son interface
  - ▶ faire attention aux passage de paramètres et de valeur de retour
    - ▶ passage par référence ?
    - ▶ passage par valeur ?
- ▶ On distingue
  - ▶ les **objets locaux** qui seront passés par copie.
  - ▶ les **objets distants** qui seront accédés via une référence.

# Les appels de méthodes sur des objets distants

---

- ▶ Selon technologie, on manipule :
  - ▶ des **objets du langage** :
    - ▶ instance de classe Java par exemple
    - ▶ cas de **RMI**
    - ▶ Avantage : facilité
    - ▶ Inconvénient : mono-langage
  - ▶ des **objets "universels"** :
    - ▶ définis par le middleware utilisé, souvent à vocation multi-langages
    - ▶ cas de **CORBA**
    - ▶ Notion de langage pivot entre les différents langages
      - Description de l'interface des objets distants (**IDL : interface définition language**)
    - ▶ Avantage : multi-langage
    - ▶ Inconvénient : plus difficile à mettre en œuvre

# Emballage/Déballage (Marshalling/Unmarshalling)

---

- ▶ On parle aussi de sérialisation/désérialisation, linéarisation
- ▶ Encoder l'état mémoire d'un objet pour permettre :
  - ▶ son stockage (persistance)
  - ▶ **sa transmission**
- ▶ Ensemble des règles :
  - ▶ représentation internes des données
  - ▶ structure de l'objet
  - ▶ format binaire/textuel (ex. XML)
- ▶ Java propose une API pour la persistance
  - ▶ **Les objets doivent implémenter `java.io.Serializable`**



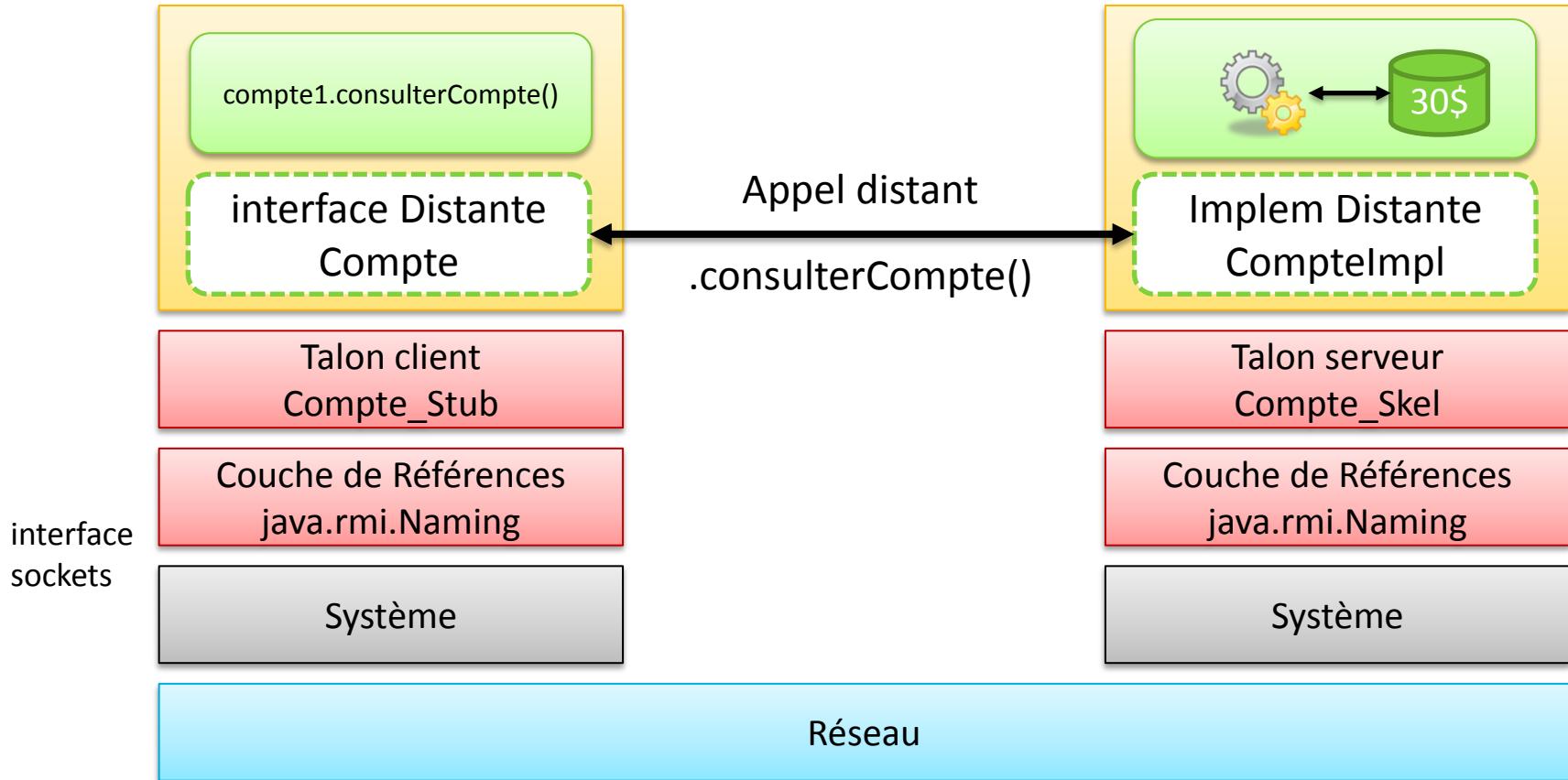
RMI

# Principes de RMI

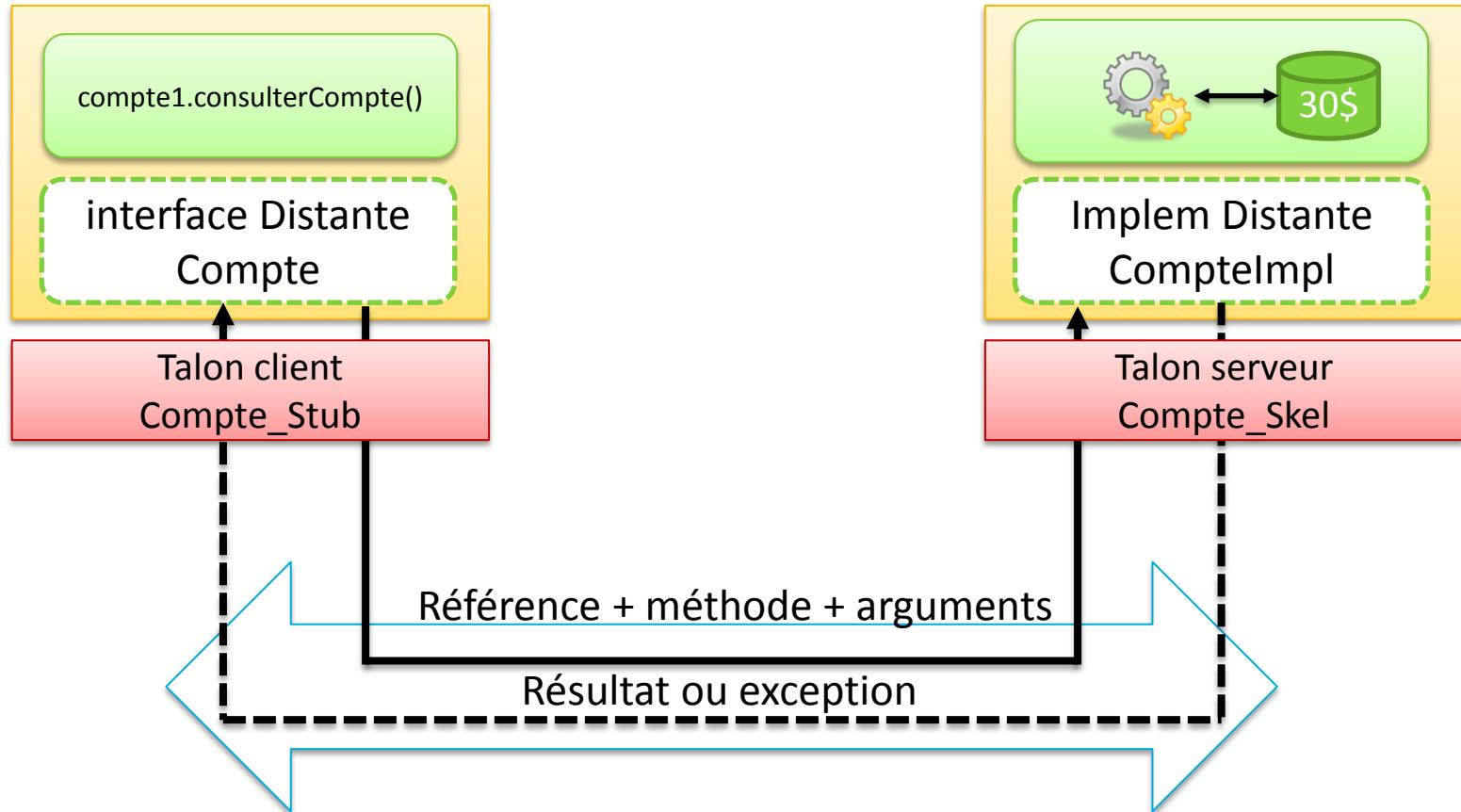
---

- ▶ RPC étendu aux objets
  - ▶ invocation de méthodes sur des objets distribués
- ▶ Dans Java depuis java 1.1 (package java.rmi)
  - ▶ peut interagir avec CORBA et DCOM
- ▶ Mono-langage, Multiplateformes
  - ▶ utilisable avec JAVA seulement
  - ▶ utilisable avec n'importe quel OS
    - ▶ la JVM abstrait le système
    - ▶ communication de JVM à JVM
- ▶ Outils :
  - ▶ annuaire de références, ramasse-miette, génération automatique des stubs, ...
- ▶ Simple en principe mais peu devenir complexe à cause de la sécurité JAVA
- ▶ Alternatives : CORBA, DCOM

# Principes de RMI

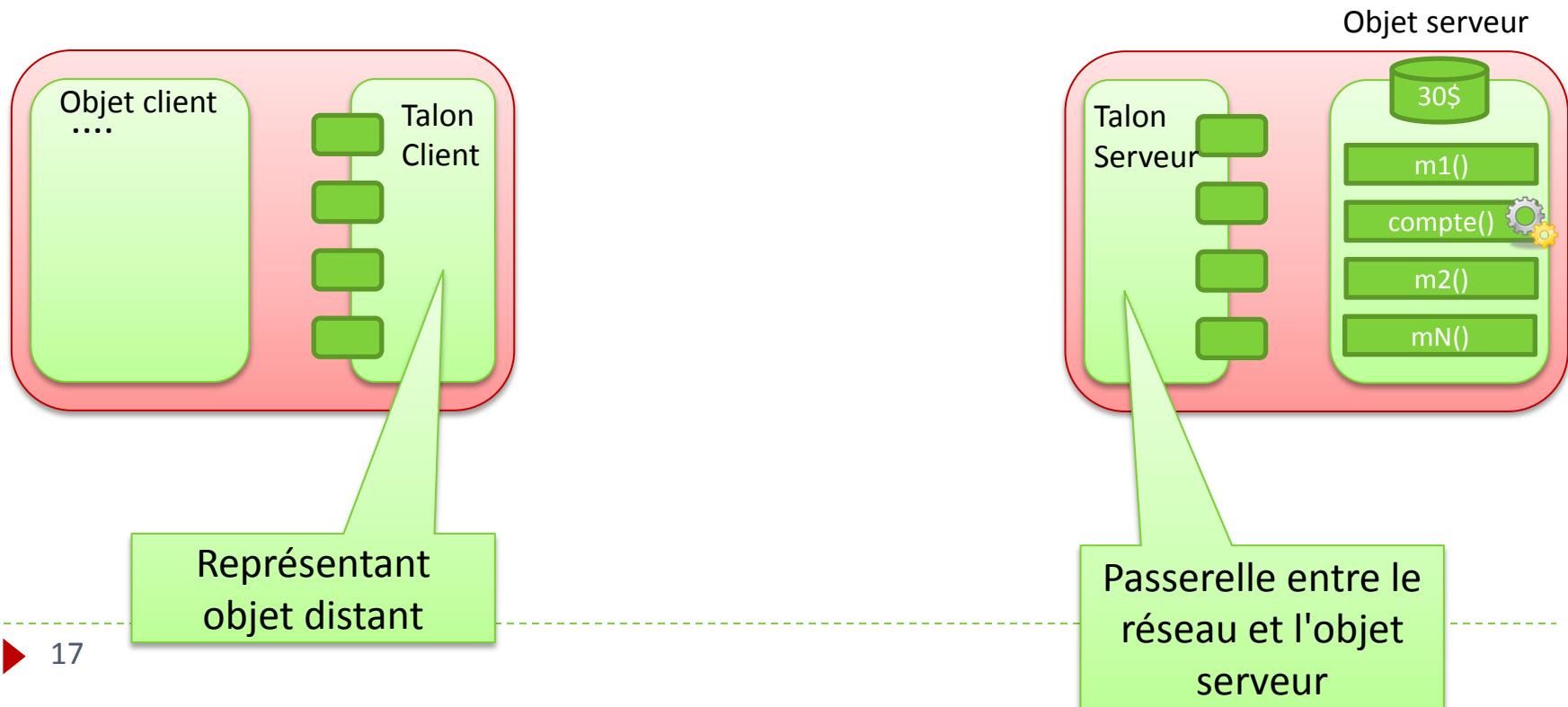


# Principes de RMI



# Mandataires d'Objets

- ▶ on parle de proxy, talon, souche, stub, skeleton, ...
  - ▶ réalisent les opérations nécessaires au transfert/réception des données sur le réseau lors d'une invocation
  - ▶ talon client (stub), talon serveur (skeleton)



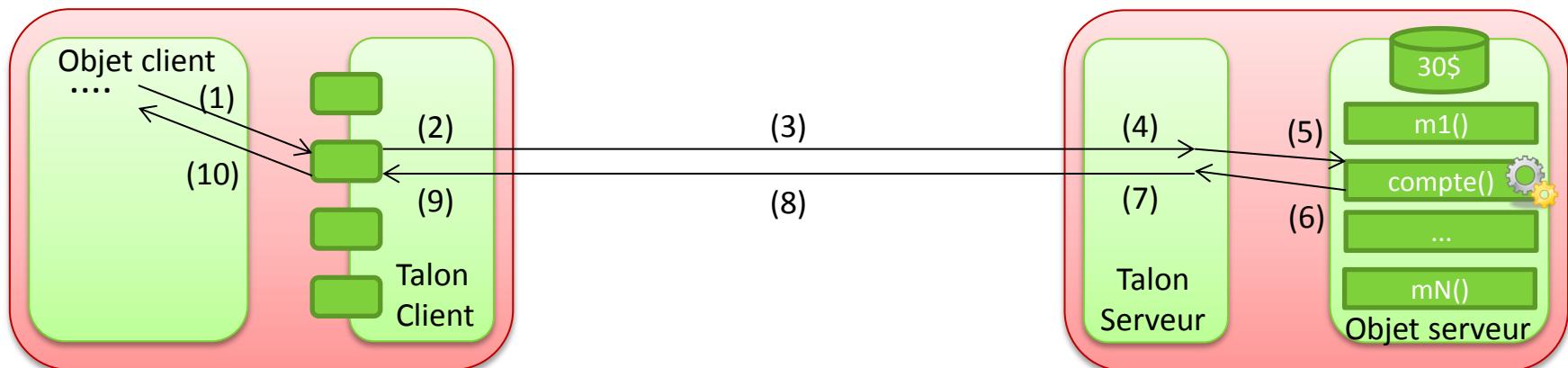
# Mandataires d'Objets

## Coté client

- ▶ 1. Invocation de la méthode
- ▶ 2. Emballage des paramètres
- ▶ 3. Transport de l'invocation
- ▶ 9. Déballage des résultats
- ▶ 10. Retour à l'objet client

## Coté serveur

- ▶ 4. Déballage des paramètres
- ▶ 5. Invocation sur l'objet distant
- ▶ 6. Retour de la méthode
- ▶ 7. Emballage des résultats
- ▶ 8. Transport des résultats



# Rôles des talons

---

## Talon client (Stub)

- ▶ se fait passer pour l'objet distant
- ▶ Emballage vers des messages vers le talon serveur
- ▶ Déballage des résultats ou exceptions retournés vers le serveur
- ▶ Contrôles (signatures, ...)

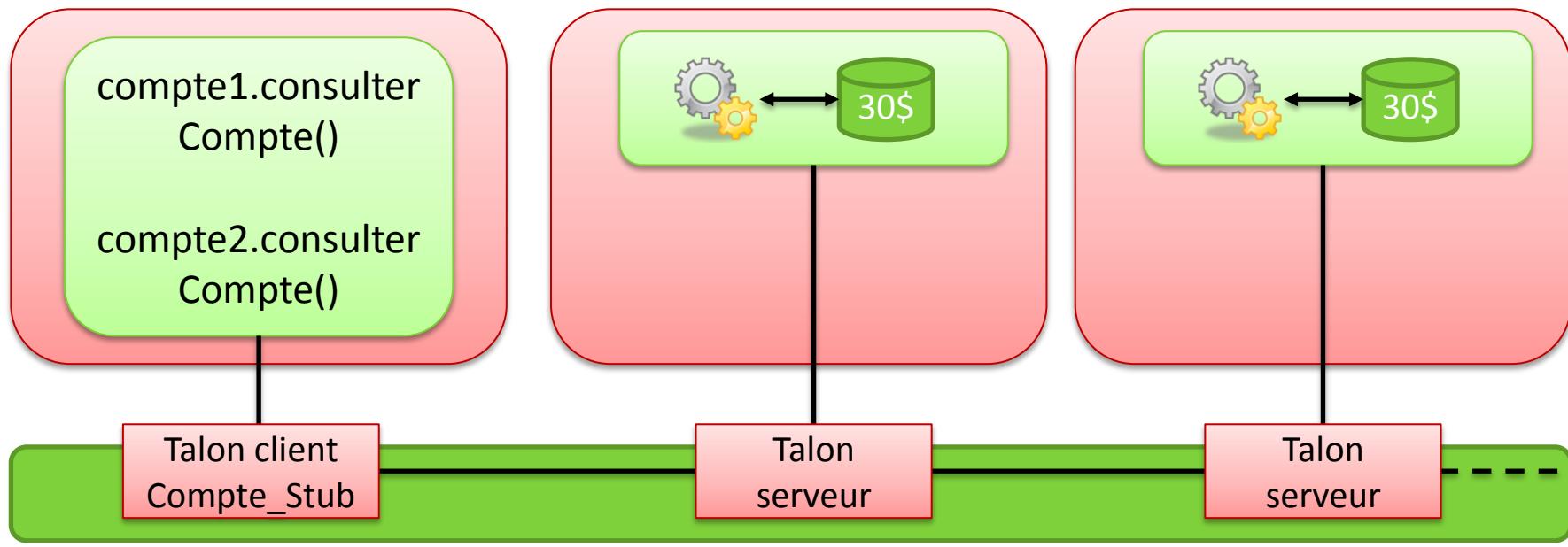
## Talon serveur (skeleton)

- ▶ réalise les appels de méthodes sur l'objet distant
- ▶ Déballage des messages en provenance du talon client
- ▶ Appel de la méthode locale
- ▶ Emballage des résultats ou exceptions à destination du client

Java fournit un générateur de talons (automatique depuis Java 5)

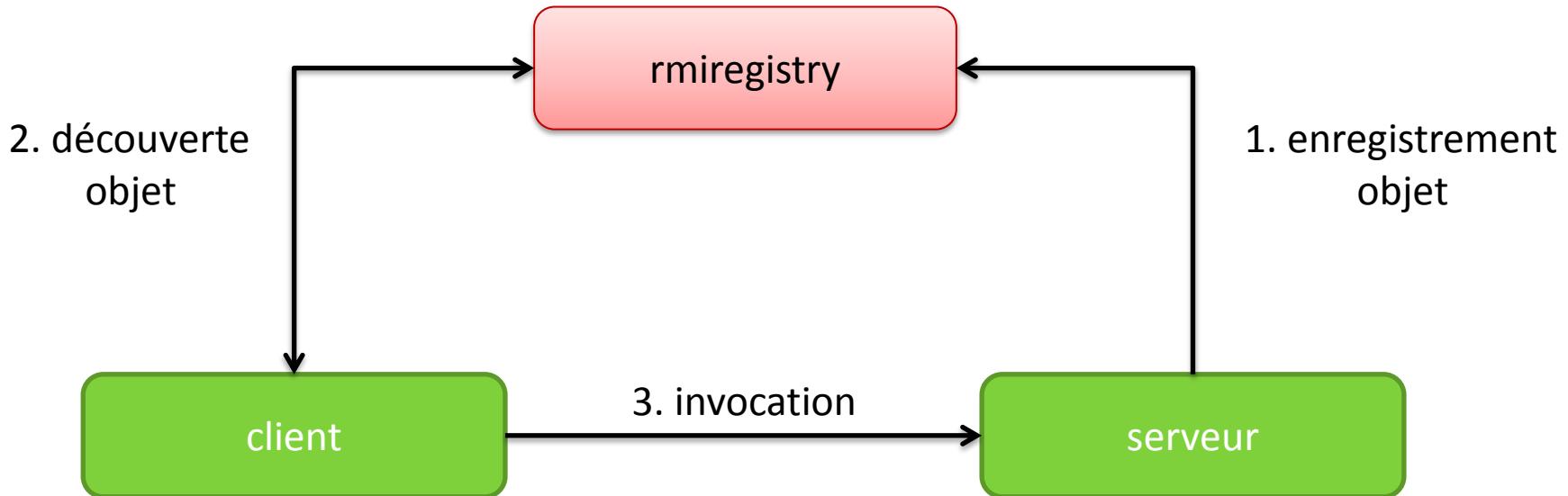
# Bus d'objets répartis

- ▶ Vue logique composée de l'ensemble des talons
- ▶ Propose souvent des services :
  - ▶ résolution de noms
  - ▶ ...

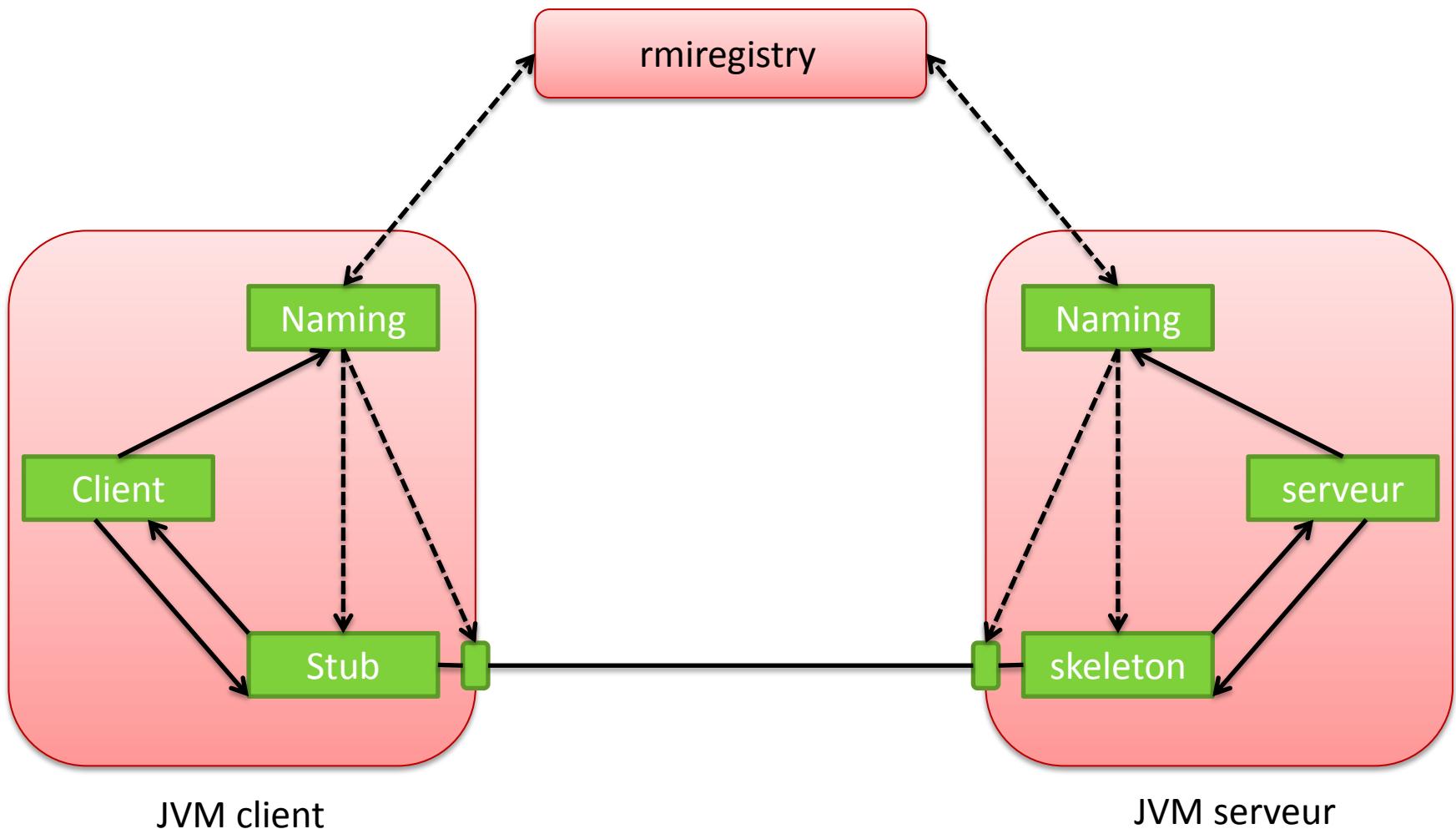


# Découverte d'objets répartis

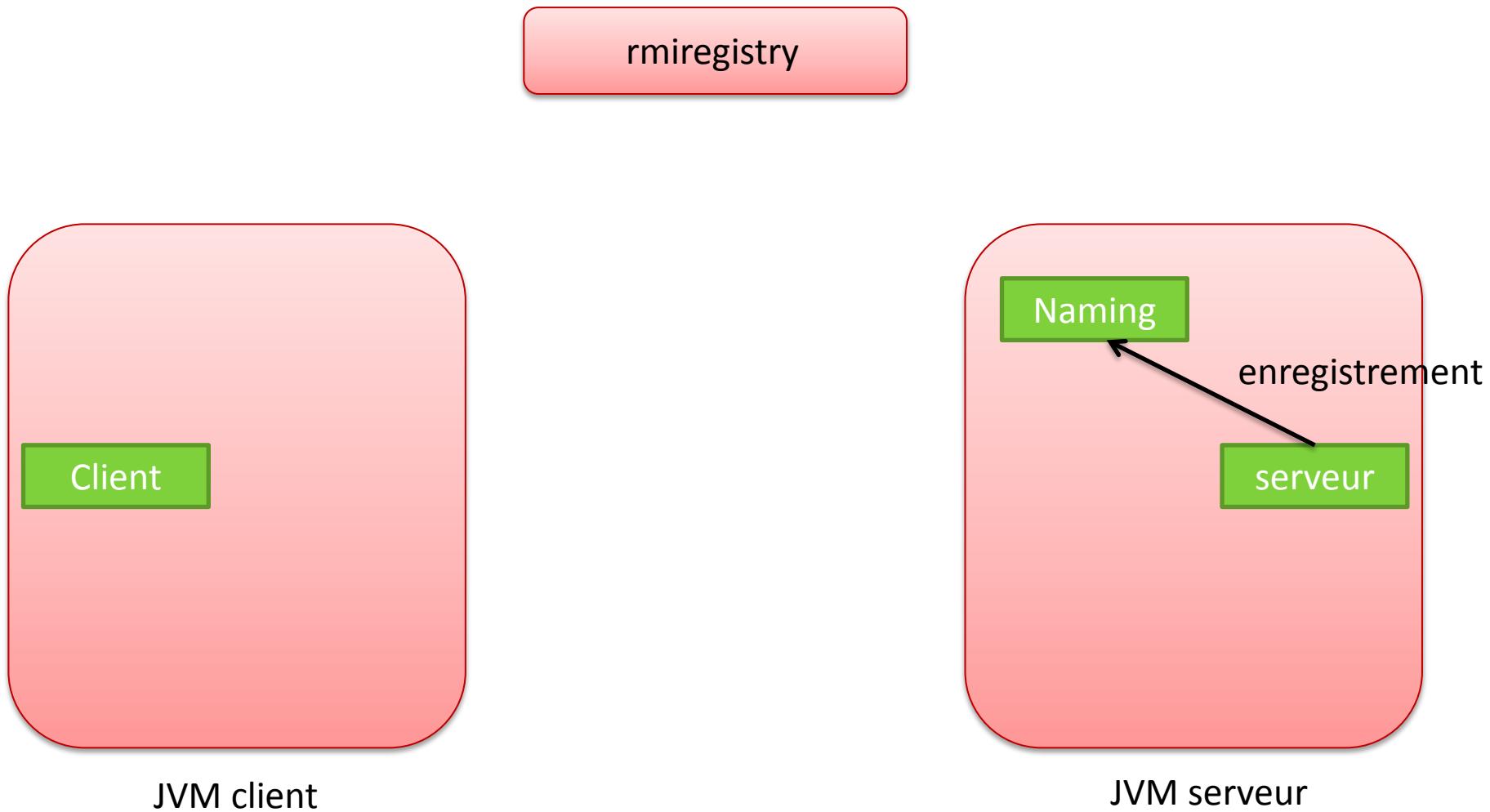
- ▶ RMI propose un annuaire pour la découverte d'objets distants (ne tourne pas sur la même JVM)



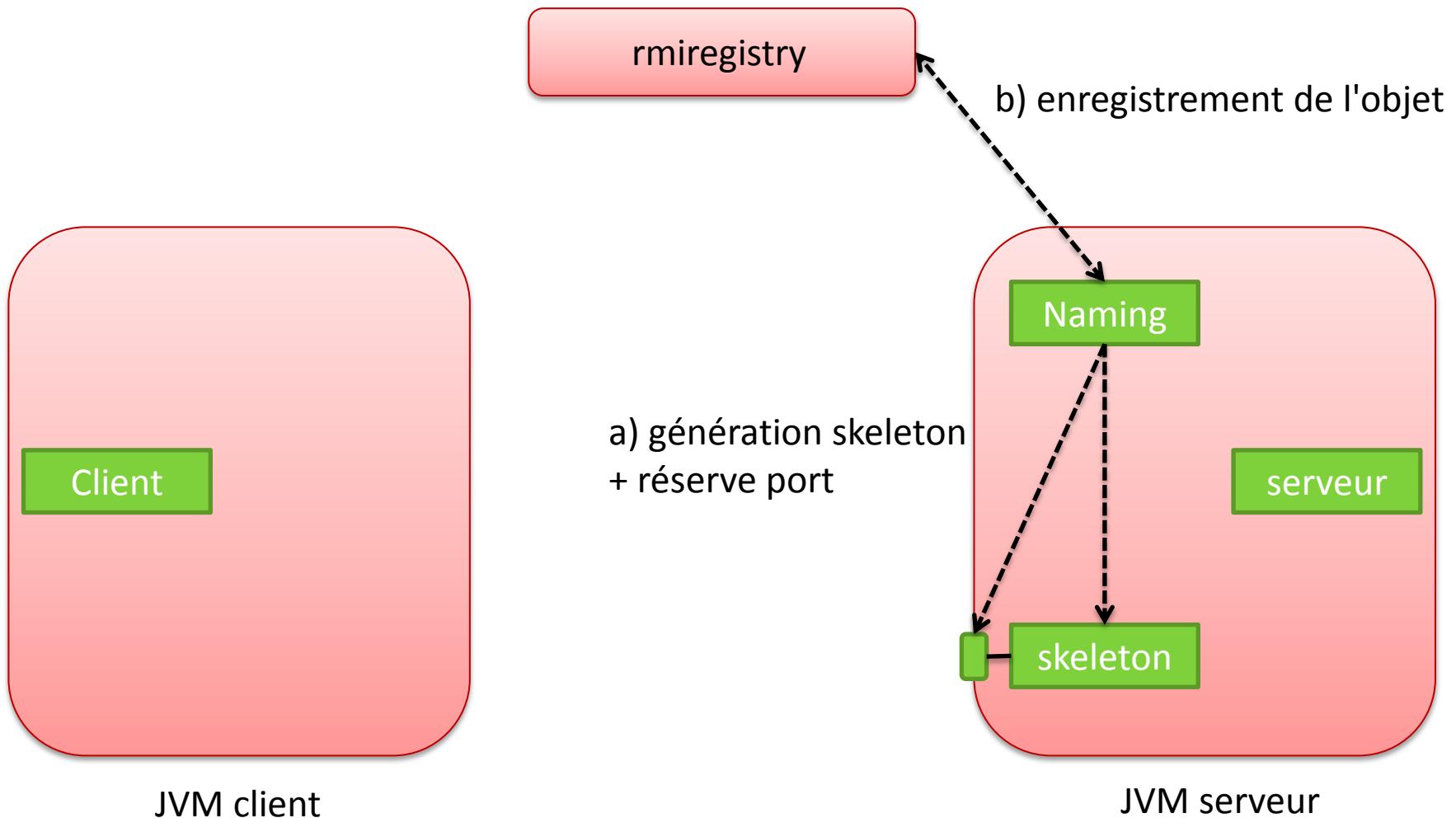
# Découverte d'objets répartis



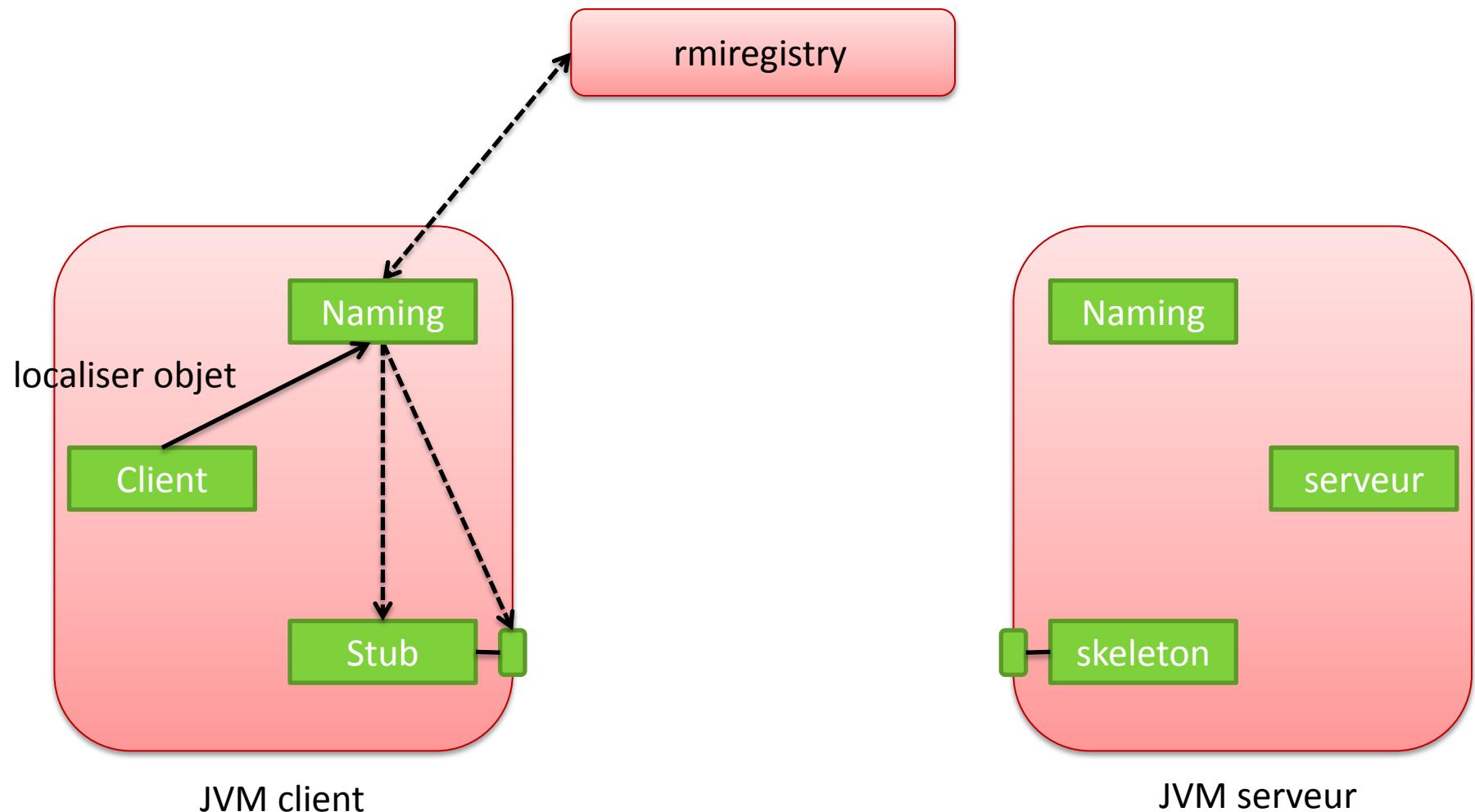
# Découverte d'objets répartis



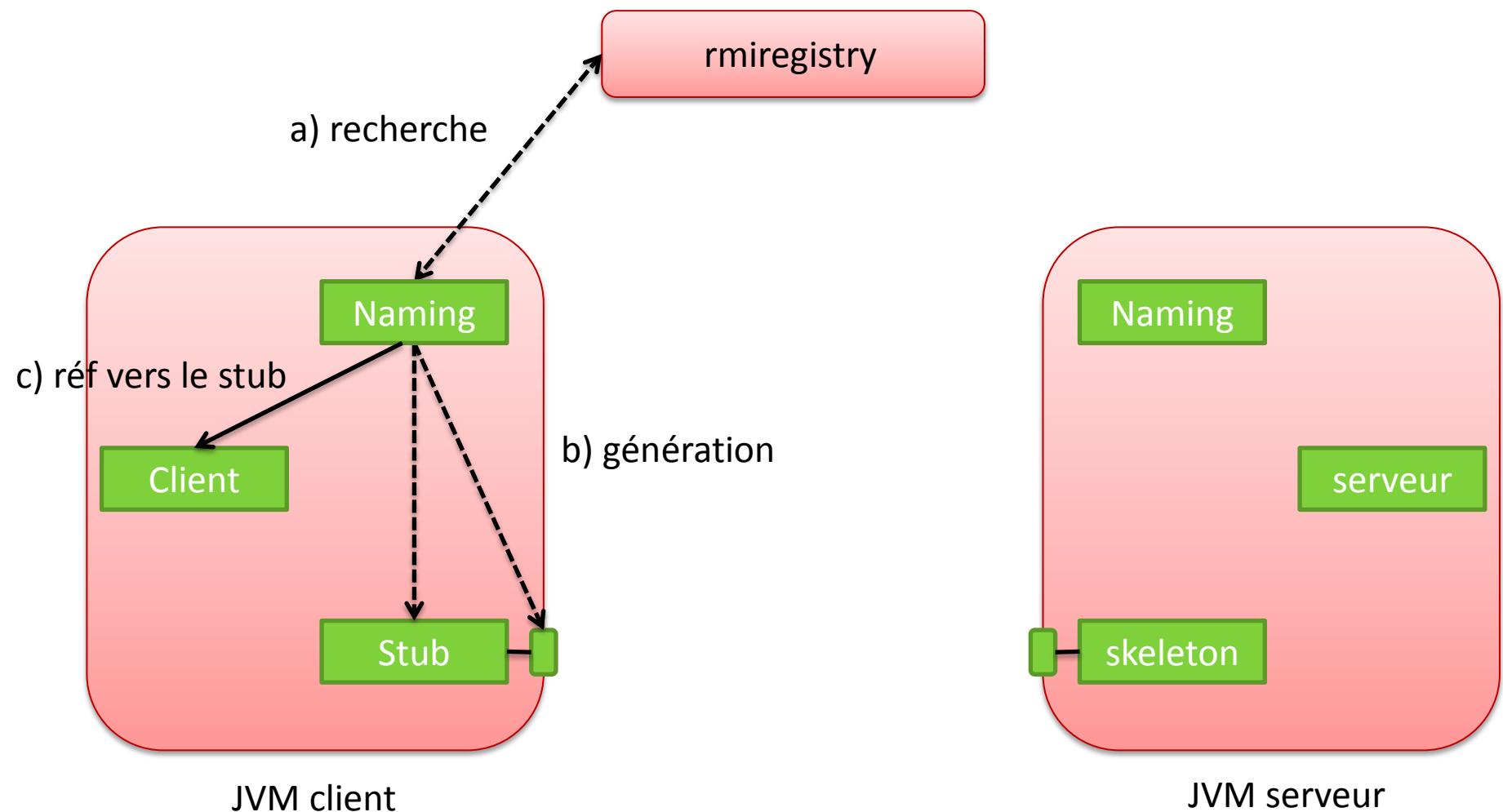
# Découverte d'objets répartis



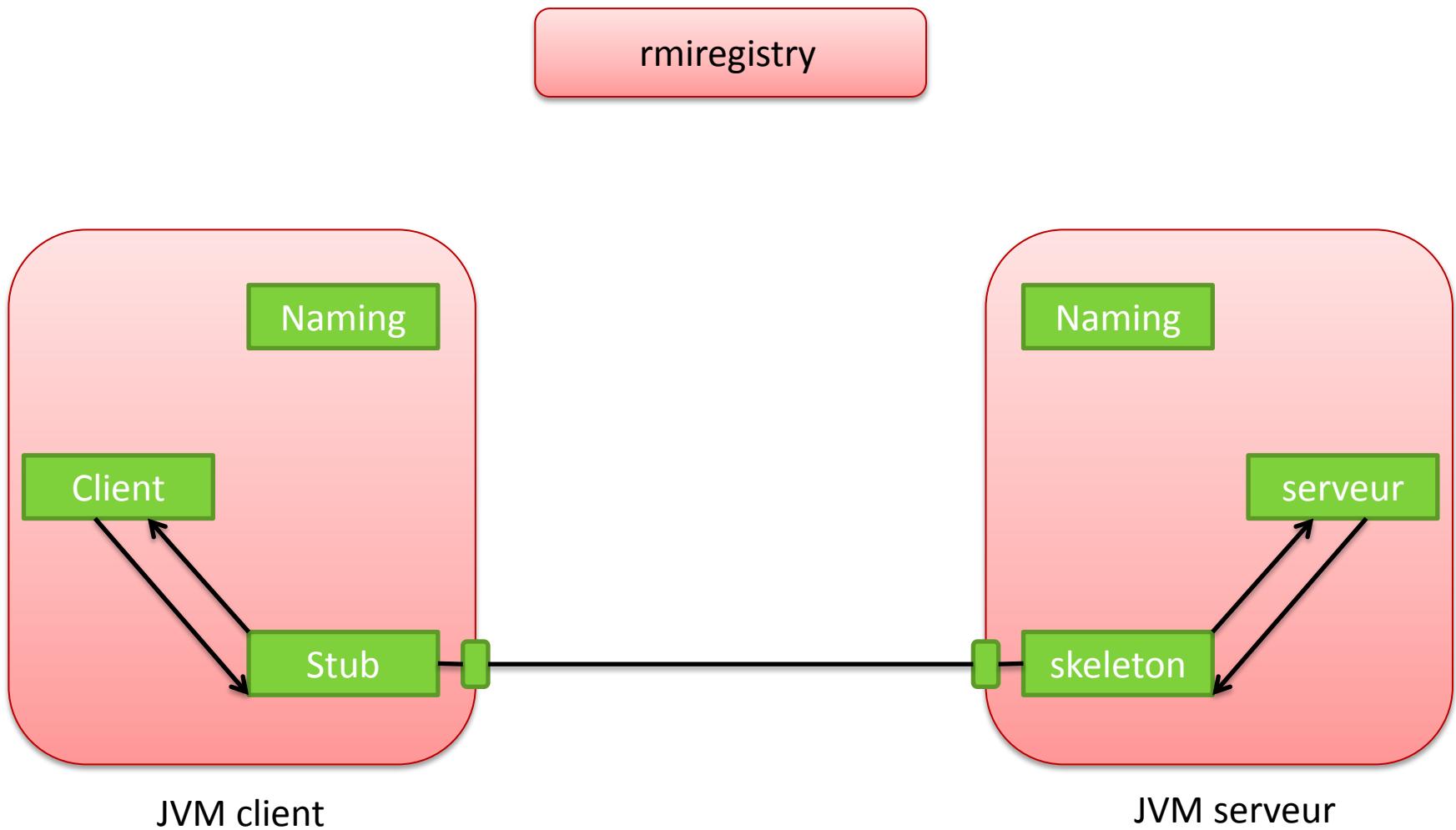
# Découverte d'objets répartis



# Découverte d'objets répartis



# Découverte d'objets répartis





# Sérialisation en Java

# Sérialisation/Désérialisation en Java

---

- ▶ La sérialisation transforme un graphe d'objets (objet et les objets sur lequel il pointe) en flux d'octets :
  - ▶ on passe l'objet
  - ▶ les objets qu'il référence
  - ▶ de façon récursive (copie profonde)
  - ▶ on récupère le nom de la classe et la valeur des attributs
    - ▶ sauf les attributs déclarés avec le mot-clef ***transient***
      - transient utile notamment pour les mots-de-passes
- ▶ La désérialisation est l'opération symétrique :
  - ▶ on lit le flux d'octets
  - ▶ on récupère le nom de la classe
  - ▶ on instancie un objet et on met-à-jour la valeur des attributs.
    - ▶ un attribut *transient* prend la valeur par défaut.

# Créer une classe sérialisable

---

- ▶ Si un objet n'est pas sérialisable (par défaut), une exception NotSerializableException sera levée
- ▶ On doit marquer les classes Serializable
  - ▶ les sous-classes d'une classe sérialisable seront sérialisables
- ▶ Java propose une API pour la persistance
  - ▶ **Les objets doivent implémenter java.io.Serializable**
    - ▶ Cette interface n'a pas de méthodes, c'est un marqueur
- ▶ Java s'occupe de la sérialisation pour vous.
- ▶ On peut écrire et lire des objets sérialisables via des ObjectInputStream, ObjectOutputStream
  - ▶ pour les sauvegarder dans des fichiers par exemple
  - ▶ pas nécessaire pour RMI, les mandataires le font pour vous.

# java.io.Serializable

```
package cours.rmi.example;

import java.io.Serializable;

public class Message implements Serializable {

    private static final long serialVersionUID = -5765881822277065451L;

    private String message;
    private transient String id;

    public Message(String message, String id) {
        super();
        this.message = message;
        this.id = id;
    }

    public String getMessage() {
        return message;
    }
    public String getId() {
        return id;
    }
}
```

Permet de comparer les versions de classes (pas obligatoire=> généré)

Indique que ce champs ne doit pas être sérialisé



## Exemple de serveur temps simple

# Ordre à suivre :

---

- ▶ On écrit une interface d'Objet commune
- ▶ On implémente le service du côté serveur
- ▶ On écrit le client
- ▶ Compilation (javac)
- ▶ Eventuelle génération des skeletons
  - ▶ rmic pour java <5
- ▶ On démarre un rmiregistry
- ▶ On démarre le serveur
  - ▶ enregistrement dans le registry
  - ▶ création du skeleton (avant on utilisait rmic)
- ▶ On démarre les clients
  - ▶ recherche dans le registry
  - ▶ génération du stub
  - ▶ invocation

# Création/manipulation

---

- ▶ 5 packages :
  - ▶ java.rmi : accès à des objets distants
  - ▶ java.rmi.server : créer des objets distants
  - ▶ java.rmi.registry : localisation et nommage
  - ▶ java.rmi.dgc : ramasse-miette
  - ▶ java.rmi.activation : activation à la demande
- ▶ Le serveur de nom
  - ▶ programme externe : rmiregistry ou lancement dans le code

# Création d'un objet distant

---

- ▶ Il faut définir une interface
  - ▶ seules les méthodes de cette interface pourront être appelées à distance
  - ▶ qui spécialise `java.rmi.Remote`
  - ▶ dont les méthodes contiennent une clause
    - ▶ `throws java.rmi.RemoteException`
  - ▶ Les types de retour et les paramètres doivent être `Serializable` !
    - ▶ cas de la majorité des classes de Java
    - ▶ les types primitifs doivent être convertis (auto-boxing)
    - ▶ les autres classes doivent implémenter `java.io.Serializable`

# Exemple d'interface d'objet distant

```
package cours.rmi.example;

import java.rmi.Remote;
import java.rmi.RemoteException;
import java.util.Date;

public interface DateRemote extends Remote {
    public Date getServerDate() throws RemoteException;
    public int getCalls() throws RemoteException;
}
```

Date est Serializable

int est converti vers  
Integer (Serializable)

# Implémentation de l'objet distant

- ▶ L'implémentation de l'objet distant doit :
  - ▶ implémenter l'interface de l'objet (ici DateRemote)
  - ▶ étendre la classe UnicastRemoteObject (donc être Serializable)
    - ▶ impose d'avoir un constructeur qui gère RemoteException

```
public class DateRemotelImpl_extends UnicastRemoteObject implements DateRemote {  
    private int calls = 0;  
    protected ServeurTemps() throws RemoteException {  
        super();  
    }  
    @Override  
    public Date getServerDate() throws RemoteException {  
        calls++;  
        return new Date();  
    }  
    @Override  
    public int getCalls() throws RemoteException {  
        return calls;  
    }  
}
```

# Instanciation et enregistrement de l'objet

---

- ▶ On utilise l'API Naming :
  - ▶ **bind(nom, objet)** : enregistre auprès du registry
    - ▶ AlreadyBoundException si déjà lié au nom
  - ▶ plus couramment **rebind(nom, objet)** : même chose mais pas d'exception AlreadyBoundException.
  - ▶ **unbind(nom)**
- ▶ Format d'un nom (URL) :
  - ▶ //machine:port/nom : enregistre nom comme nom
  - ▶ //machine:port/a/b/c/nom : enregistre a/b/c/nom comme nom
  - ▶ //machine/nom : port par défaut
- ▶ Exemple : **//localhost/date** enregistre date sur la machine localhost

# Exemple de ServeurTemps

```
package cours.rmi.example;

import java.net.MalformedURLException;
import java.rmi.Naming;
import java.rmi.RemoteException;

public class ServeurTemps {

    public static void main(String[] args) throws RemoteException, MalformedURLException {
        DateRemote objetDistant = new DateRemotelImpl();
        Naming.rebind("//localhost/date", objetDistant);
        System.out.println(d + " has been registered");
    }
}
```

# Alternative pour l'enregistrement

- ▶ On utilise LocateRegistry pour trouver l'annuaire qui nous intéresse :
  - ▶ méthode getRegistry donne un registry
  - ▶ puis registry.rebind(..)

```
public class ServeurTempsAvecRecherche {  
  
    public static void main(String[] args) throws RemoteException, MalformedURLException {  
        DateRemote d = new DateRemoteImpl();  
        Registry registry = LocateRegistry.getRegistry("localhost", Registry.REGISTRY_PORT);  
        registry.rebind("//localhost/date", d);  
        System.out.println(d + " has been registered");  
    }  
}
```

On veux le rmiregistry sur la machine localhost avec le port par défaut

# On lance ...

---

```
Exception in thread "main" java.rmi.ConnectException: Connection refused to host:
localhost; nested exception is:
```

```
java.net.ConnectException: Connection refused: connect
```

```
at sun.rmi.transport.tcp.TCPEndpoint.newSocket(Unknown Source)
```

```
at sun.rmi.transport.tcp.TCPChannel.createConnection(Unknown Source)
```

```
at sun.rmi.transport.tcp.TCPChannel.newConnection(Unknown Source)
```

```
at sun.rmi.server.UnicastRef.newCall(Unknown Source)
```

```
at sun.rmi.registry.RegistryImpl_Stub.rebind(Unknown Source)
```

```
at java.rmi.Naming.rebind(Unknown Source)
```

# On dois avoir un rmiregistry !

---

- ▶ On dois lancer le rmiregistry avec la commande rmiregistry :

```
$>rmiregistry
```

- ▶ On teste de nouveau :

```
Exception in thread "main" java.rmi.ServerException: RemoteException occurred
in server thread; nested exception is:
java.rmi.UnmarshalException: error unmarshalling arguments; nested exception
is:
java.lang.ClassNotFoundException: cours.rmi.example.DateRemote
```

# rmiregistry doit avoir accès aux classes !

---

- ▶ soit on le lance dans le bon répertoire :
  - ▶ sous eclipse on se place dans le répertoire bin
    - ▶ c'est là qu'Eclipse compile les classes avant de les lancer
  - ▶ plutôt contraignant
- ▶ soit on indique le chemin avec codabase
  - ▶ on verra un peu plus loin
- ▶ <http://docs.oracle.com/javase/6/docs/technotes/guides/rmi/codebase.html>

# Nouveau test

On bricole pour indiquer où sont les classes

```
java -Djava.rmi.server.codebase=file:/C:/Users/ymaurel/workspace/cours.rmi.example/bin/ ...
```

```
DateRemoteImpl[UnicastServerRef [liveRef:  
[endpoint:[192.168.149.1:57937](local),objID:[17eb3afe:143f36b40e7:-7fff, -7174125592905436095]]]  
has been registered
```

L'objet a bien été enregistré

# Code du client

---

- ▶ Le client doit trouver l'objet :
  - ▶ Naming.lookup(nom)
  - ▶ ou LocateRegistry.getRegistry("adresse") puis lookup ...

```
public class Client {  
  
    public static void main(String[] args) throws MalformedURLException,  
    RemoteException, NotBoundException {  
        DateRemote date = (DateRemote) Naming.lookup("//localhost/date");  
        System.out.println(date.getServerDate());  
        System.out.println(date.getCalls());  
    }  
}
```

# Alternative code client

```
package cours.rmi.example;

import java.net.MalformedURLException;
import java.rmi.NotBoundException;
import java.rmi.RemoteException;
import java.rmi.registry.LocateRegistry;
import java.rmi.registry.Registry;

public class Client {

    public static void main(String[] args) throws MalformedURLException, RemoteException,
    NotBoundException {
        Registry registry = LocateRegistry.getRegistry("localhost", Registry.REGISTRY_PORT);
        System.out.println(registry);
        DateRemote date = (DateRemote) registry.lookup("date");
        System.out.println(date.getServerDate());
        System.out.println(date.getCalls());
    }
}
```

# Test

---

- ▶ On lance le client (après rmiregistry et serveur)

```
Sun Feb 01 17:36:43 CET 2014  
1
```

- ▶ Ca fonctionne parce que le client sait où trouver les bonnes classes.
  - ▶ à nouveau l'histoire du codebase (voir plus loin)

# Lancement du registre depuis le serveur

```
package cours.rmi.example;

import java.net.MalformedURLException;
import java.rmi.AlreadyBoundException;
import java.rmi.RemoteException;
import java.rmi.registry.LocateRegistry;
import java.rmi.registry.Registry;

public class ServeurRegistre {

    public static void main(String[] args) throws RemoteException, MalformedURLException,
    AlreadyBoundException {
        DateRemote d = new DateRemoteImpl();
        Registry registry = LocateRegistry.createRegistry(Registry.REGISTRY_PORT);
        registry.rebind("date", d);
        System.out.println(d + " has been registered");
    }
}
```

Créer et lancer un rmiregistry sur le port par défaut

# Passage de paramètres et valeur de retour

# Objets locaux

- ▶ Les **objets locaux** sont passés par valeur :
  - ▶ ils n'implémentent pas UnicastRemoteObject
  - ▶ ils doivent être **Sérializable**
  - ▶ une modification par le client n'implique pas de modification côté serveur
  - ▶ c'est le cas par défaut des paramètres et valeurs de retour
- ▶ Exemple



# Implémentation de CompteLocal

```
package exemple.compte;

import java.io.Serializable;

public class CompteLocallImpl implements Serializable {
    private static final long serialVersionUID = 1L;
    private int value = 0;

    public void addValue(int value) {
        this.value += value;
    }

    public int getValue() {
        return value;
    }

}
```

# Interface BanqueRemote

```
package exemple.compte;

import java.rmi.Remote;
import java.rmi.RemoteException;

public interface BanqueRemote extends Remote {
    CompteLocalImpl getCompte() throws RemoteException;
}
```

retourne des objets CompteLocalImpl locaux

# Implémentation de l'interface BanqueRemote

```
package exemple.compte;

import java.rmi.RemoteException;
import java.rmi.server.UnicastRemoteObject;

public class BanqueImpl extends UnicastRemoteObject implements BanqueRemote {

    private CompteLocalImpl compte = new CompteLocalImpl();

    protected BanqueImpl() throws RemoteException {
        super();
    }

    @Override
    public CompteLocalImpl getCompte() throws RemoteException {
        return compte;
    }
}
```

# Implémentation du serveur

```
package exemple.compte;

import java.net.MalformedURLException;
import java.nio.file.Paths;
import java.rmi.Naming;
import java.rmi.RemoteException;

public class Serveur {
    public static void main(String[] args) throws MalformedURLException,
RemoteException {
        BanqueImpl banque = new BanqueImpl();
Naming.rebind("//localhost/banque", banque);
        System.out.println(banque + " has been registered");
    }
}
```

# Implémentation du client

```
package exemple.compte;

import java.net.MalformedURLException;
import java.rmi.Naming;
import java.rmi.NotBoundException;
import java.rmi.RemoteException;

public class Client {

    public static void main(String[] args) throws MalformedURLException, RemoteException,
NotBoundException {
        BanqueRemote banque = (BanqueRemote) Naming.lookup("//localhost/banque");
        CompteLocallImpl compte = banque.getCompte();
        System.out.println("old value =" + compte.getValue());
        compte.addValue(1);
        System.out.println("new value =" + compte.getValue());
    }
}
```

# Exécution de plusieurs clients

```
old value =0  
new value =1
```

```
old value =0  
new value =1
```

```
old value =0  
new value =1
```

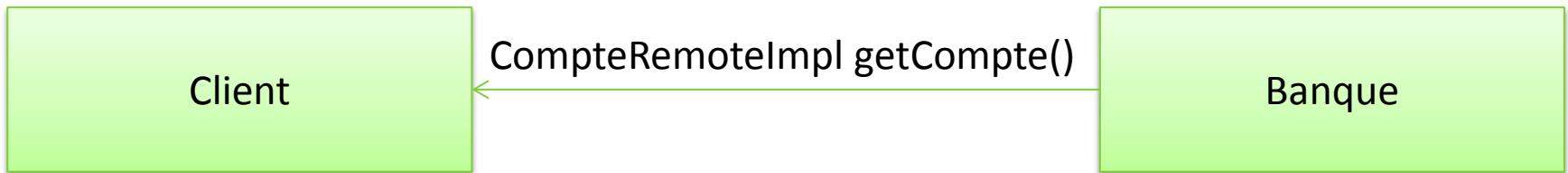
La valeur n'a pas été changée côté serveur et on se retrouve encore avec un compte = 0

- ▶ CompteLocalImpl est un objet local et donc sa modification n'est pas propagée au serveur :
  - ▶ pas de mandataires pour propager les modifications

# Objets distants

---

- ▶ Les **objets distants** sont passés par référence:
  - ▶ ils étendent UnicastRemoteObject
  - ▶ ils implémentent une interface ObjetDistantRemote qui étends Remote
  - ▶ une modification par le client est propagée coté serveur
- ▶ Exemple



# Interface CompteRemote

---

```
package exemple.compte.distant;

import java.rmi.Remote;
import java.rmi.RemoteException;

public interface CompteRemote extends Remote {

    public void addValue(int value) throws RemoteException;

    public int getValue() throws RemoteException;

}
```

# Implémentation de CompteRemote

```
package exemple.compte.distant;

import java.rmi.RemoteException;
import java.rmi.server.UnicastRemoteObject;

public class CompteRemoteImpl extends UnicastRemoteObject implements CompteRemote {

    private static final long serialVersionUID = 1L;
    private int value = 0;

    protected CompteRemoteImpl() throws RemoteException {
        super();
    }

    public void addValue(int value) throws RemoteException {
        this.value += value;
    }

    public int getValue() throws RemoteException {
        return value;
    }

}
```

# Modification de Banque en conséquence

```
package exemple.compte.distant;

import java.rmi.Remote;
import java.rmi.RemoteException;

public interface BanqueRemote extends Remote {
    CompteRemote getCompte() throws RemoteException;
}
```

```
public class BanqueImpl extends UnicastRemoteObject implements
BanqueRemote {
    CompteRemote compte = new CompteRemotelImpl();
    protected BanqueImpl() throws RemoteException {
        super();
    }

    public CompteRemote getCompte() throws RemoteException {
        return compte;
    }
}
```

# Modification du serveur et client

```
public class Serveur {  
    public static void main(String[] args) throws MalformedURLException, RemoteException {  
        BanqueImpl banque = new BanqueImpl();  
        Naming.rebind("//localhost/banque", banque);  
        System.out.println(banque + " has been registered");  
    }  
}
```

pas de modifications

```
public class Client {  
    public static void main(String[] args) throws MalformedURLException, RemoteException,  
NotBoundException {  
    BanqueRemote banque = (BanqueRemote) Naming.lookup("//localhost/banque");  
    CompteRemote compte = banque.getCompte();  
    System.out.println("old value =" + compte.getValue());  
    compte.addValue(1);  
    System.out.println("new value =" + compte.getValue());  
}  
}
```

# Exécution de plusieurs clients séquentiellement

old value =0

new value =1

**old value =1**

new value =2

**old value =3**

new value =4

La valeur **a été changée** coté serveur et on se retrouve avec un compte = 1

- ▶ CompteRemote est un objet distant et donc sa modification est propagée au serveur :
  - ▶ génération de mandataires
  - ▶ propagation des modifications
  - ▶ lenteur

# Conclusion valeur/référence

---

- ▶ Le passage par valeur:
  - ▶ plus simple à mettre en œuvre
  - ▶ plus rapide (pas de propagation)
  - ▶ pas de concurrence à gérer
- ▶ Le passage par référence:
  - ▶ nécessite l'écriture d'une interface et l'implem, ...
  - ▶ plus lent (latence réseau)
  - ▶ on doit gérer la concurrence (à vous de regarder comment)
- ▶ Très utile pour mettre en place des Callbacks et faire du publish/subscribe !
  - ▶ le client donne des objets RemoteCallback au serveur qui seront appelés par le serveur pour envoyer des notifications

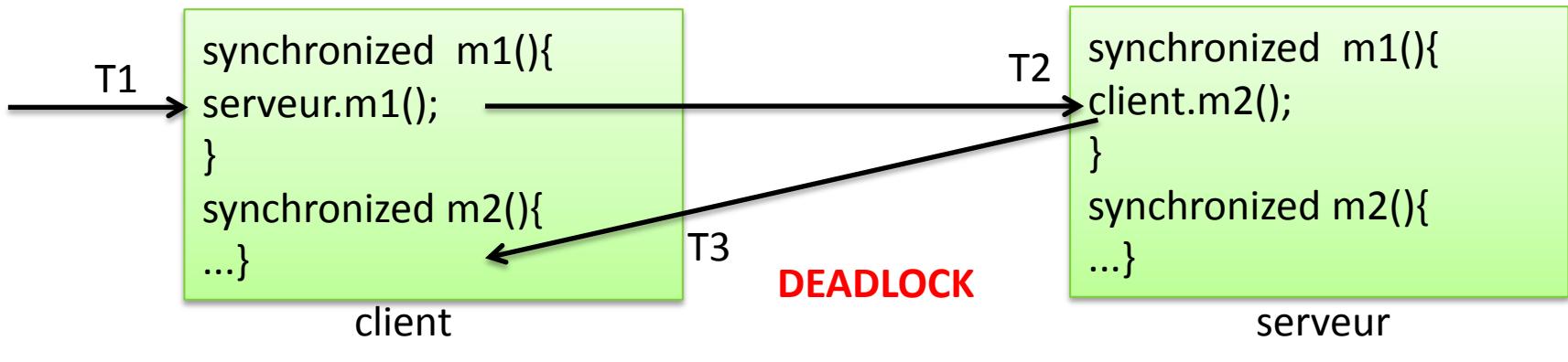
# Un mot sur la synchro ....

## ► En pratique :

- ▶ utilisation d'un Thread pool de façon transparente par RMI
- ▶ un Thread est utilisé à chaque nouvel appel à une méthode
- ▶ en cas d'appels multiples, plusieurs threads sont utilisés

## ► Pose des problèmes de synchro

- ▶ notamment pour la réentrance



**T3 attend fin T1 qui attend fin T2 qui attend fin T3 qui ...**

# Options de débogage pour le registre

---

- ▶ On peut lancer le registre avec les options suivantes pour activer les traces et permettre le débogage (sur la même ligne) :

```
rmiregistry
-J-Dsun.rmi.log.debug=true
-J-Dsun.rmi.server.exceptionTrace=true
-J-Dsun.rmi.loader.logLevel=verbose
-J-Dsun.rmi.dgc.logLevel=verbose
-J-Dsun.rmi.transport.logLevel=verbose
-J-Dsun.rmi.transport.tcp.logLevel=verbose
```



# Chargement dynamique et codebase

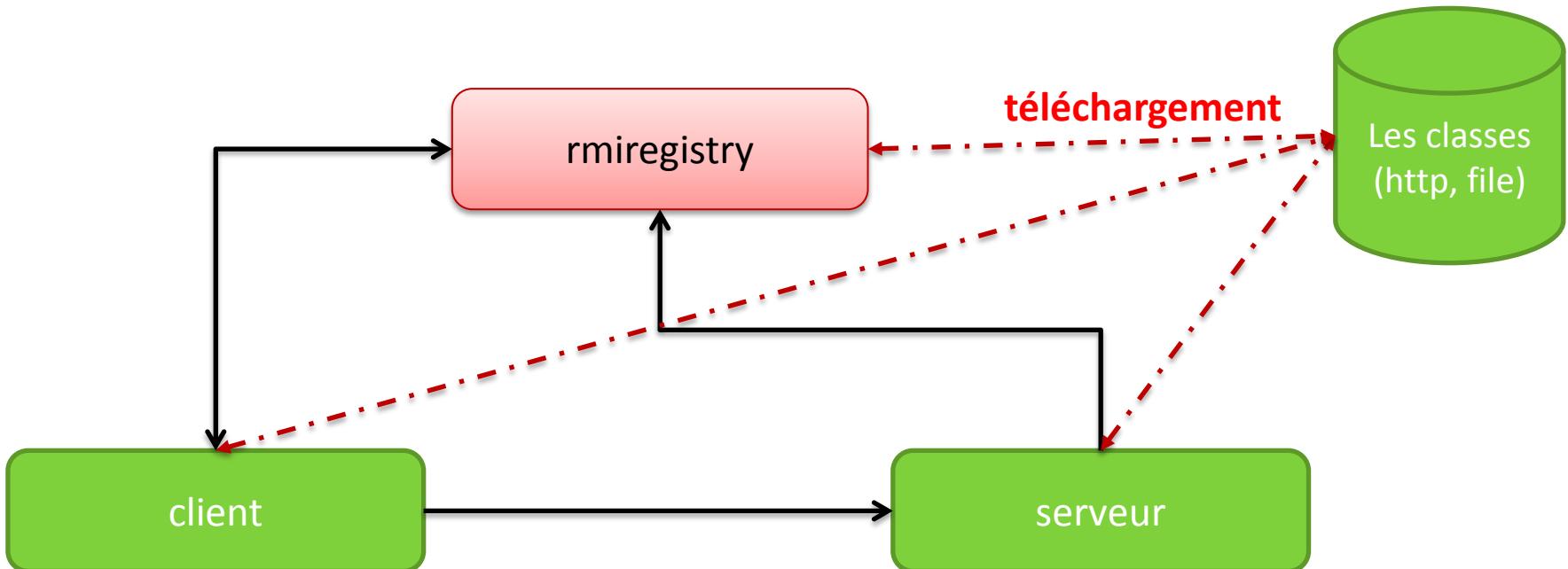
# Notion de chargement dynamique

---

- ▶ Pour pouvoir fonctionner correctement, le rmiregistry et le client doivent savoir où trouver les classes nécessaires.
- ▶ C'est encore plus important lorsque le client ne possède pas toutes les classes.
  - ▶ on a compilé le client avec toutes les classes
    - ▶ mais le client.jar possède uniquement les classes nécessaires aux client sans les implémentations des objets distants.
  - ▶ on a compilé le serveur avec toutes les classes
    - ▶ mais le serveur.jar possède uniquement la classe Serveur sans les implémentations des classes clients (et objets distants potentiellement fournis par le client).
  - ▶ on a mis l'ensemble des classes communes sur un serveur http ou un serveur ftp
    - ▶ le client, le serveur et le rmiregistry doivent pouvoir y avoir accès.

# Codebase

- ▶ RMI utilise la notion de codebase pour indiquer où trouver les classes
- ▶ Les jvm commencent par chercher les classes dans leur classpath et regardent dans le codebase si non trouvées



# Codebase

---

- ▶ Deux propriétés java sont importantes :
  - ▶ java.rmi.server.codebase :
    - ▶ indique où les classes communes sont disponibles
    - ▶ URL qui pointe sur :
      - un lien vers le disque dur : file:/<lien-vers-les-classes>
      - un http : http:/
      - ...
  - ▶ java.rmi.server.useCodebaseOnly :
    - ▶ si **false**, indique au Client et à rmiregistry qu'il doit utiliser le codebase donné par le serveur
    - ▶ si **true**, alors il faut préciser le codebase pour chaque jvm : le client, le registre et le serveur.
      - valeur par défaut depuis JAVA 7.

# En pratique, méthode 1

---

- ▶ Soit on lance le **client**, le **serveur** et le **rmiregistry** avec l'option suivante :

```
java -Djava.rmi.server.codebase=<url> ....
```

- ▶ Exemple :

```
java -Djava.rmi.server.codebase=file:/C:/Users/ymaurel/workspace/cours.rmi.example/bin/ ....
```

- ▶ **Attention pour rmiregistry on doit ajouter -J**

```
rmiregistry -J-Djava.rmi.server.codebase=<url>
```

# En pratique, méthode 2

---

- ▶ Définition programmatique pour le client et serveur
  - ▶ **à faire avant les appels et enregistrements !**

```
public class Serveur {  
    public static void main(String[] args) throws MalformedURLException, RemoteException {  
        String pathToClasses = Paths.get("bin").toUri().toURL().toString();  
        System.setProperty("java.rmi.server.codebase", pathToClasses);  
  
        BanqueImpl banque = new BanqueImpl();  
        Naming.rebind("//localhost/banque", banque);  
        System.out.println(banque + " has been registered");  
    }  
}
```

- ▶ Et on lance rmiregistry comme avec la méthode 1

# En pratique, méthode 3

---

- ▶ On lance le serveur avec une des méthodes décrites précédemment en 1 et 2
- ▶ On lance les clients et rmiregistry avec l'option
  - ▶ -Djava.rmi.server.useCodebaseOnly=false

```
java -Djava.rmi.server.useCodebaseOnly=false ...
```

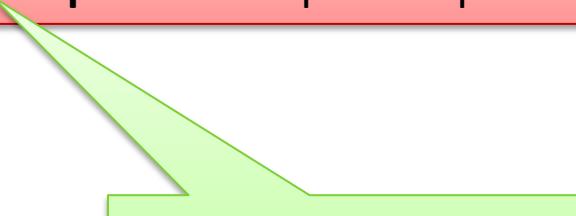
```
rmiregistry -J-Djava.rmi.server.useCodebaseOnly=false
```

- ▶ L'option peut également être positionnée de façon programmatique pour le client.

## Si mal fait :

---

```
Exception in thread "main" java.rmi.ServerException: RemoteException occurred in server  
thread; nested exception is:  
java.rmi.UnmarshalException: error unmarshalling arguments; nested exception is:  
java.lang.ClassNotFoundException: exemple.compte.local.BanqueRemote
```



Les classes n'ont pas  
pu être trouvées

# Hostname

---

- ▶ Le hostname permet d'indiquer comment contacter le serveur pour utiliser ses objets distants.
- ▶ Le serveur doit positionner `java.rmi.server.hostname` de façon à ce qu'on puisse le trouver :

```
java -Djava.rmi.server.codebase=file:/url/ -Djava.rmi.server.hostname="w.x.y.z"
```

- ▶ ou `w.x.y.z` est l'IP ou le nom de domaine qui pointe sur le serveur.



## La sécurité

(très sommairement)

# La sécurité en java

---

- ▶ L'utilisation de RMI sur plusieurs machines active la sécurité dans java
  - ▶ c'est particulièrement le cas si vous utilisez les objets activables.
- ▶ Problème : Java est parano
  - ▶ Tout est interdit par défaut
    - ▶ lecture/écriture dans des fichiers
    - ▶ les appels réseaux
    - ▶ le téléchargement distant
    - ▶ les API reflection
  - ▶ et c'est bien pour les applets web par exemple ...
  - ▶ En revanche c'est mauvais pour vous :
    - ▶ votre programme ne fonctionnera pas si votre client, votre serveur et votre registre ne sont pas sur la même machine ....

# Le SecurityManager

---

- ▶ La sécurité en Java est géré par un Security Manager
  - ▶ objet de la classe SecurityManager ou d'une sous-classe.
- ▶ Par défaut, les programmes tournent sans Security Manager
  - ▶ donc tout est permis
- ▶ Le SecurityManager fourni par la classe SecurityManager interdit tout par défaut.
  - ▶ mais on peut le configurer via des politiques
- ▶ RMI utilise un SecurityManager : RMISecurityManager qui lui aussi interdit tout par défaut

# La classe SecurityManager

---

- ▶ Une classe avec de nombreuses méthodes check\* du format checkPermission qui déclenchent une exception si la permission n'est pas accordée
- ▶ Exemple :
  - ▶ public void checkRead(**String** file) déclenche une exception si on n'a pas le droit de lire le fichier "file"
- ▶ Les appels au SecurityManager sont faits par Java automatiquement et de façon transparente
  - ▶ il n'est pas possible de le contourner

# Les politiques de sécurités

---

- ▶ Le gestionnaire de sécurité est capable de lire et interpréter sa configuration via des fichiers politiques.
  - ▶ fichier de la forme nom.policy
- ▶ Il est possible de les éditer avec policytool (outil graphique), mais à la main c'est aussi efficace.
- ▶ Exemple de fichier politique :

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

# Mettre en place un SecurityManager

---

## ► On utilise les options :

- ▶ -Djava.security.manager=*nom de la classe*
  - ▶ la classe doit être dans le classpath
  - ▶ si pas de classe spécifiée alors Java utilise le SecurityManager par défaut
- ▶ -Djava.security.policy=*lien vers le fichier policy*
  - ▶ pour utiliser une politique de sécurité particulière

## ► Exemple :

```
java -Djava.security.manager -Djava.security.policy=date.policy ....
```

# Dans notre cas pour que ça fonctionne

---

- ▶ On utilise soit une politique laxiste :

```
// Do what you will. Totally permissive policy file.  
grant {  
    permission java.security.AllPermission;  
};
```

- ▶ Soit une politique plus "raisonnable" :
  - ▶ n'autorise que les connections réseaux

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



Et plus ...

# RMI permet encore entre autre ...

---

- ▶ De n'activer les objets que si nécessaires
  - ▶ implémentation d'objets Activatable
    - ▶ java.rmi.activation.Activatable
  - ▶ utilisation de rmid
- ▶ De gérer les références avec un garbage collector distribué :
  - ▶ comptage de références et lease
  - ▶ voir à ce sujet dgc
- ▶ Changer le protocole de communication sous-jacent
  - ▶ JRMP(historique RMI), IIOP (protocole de Corba)