Objets répartis

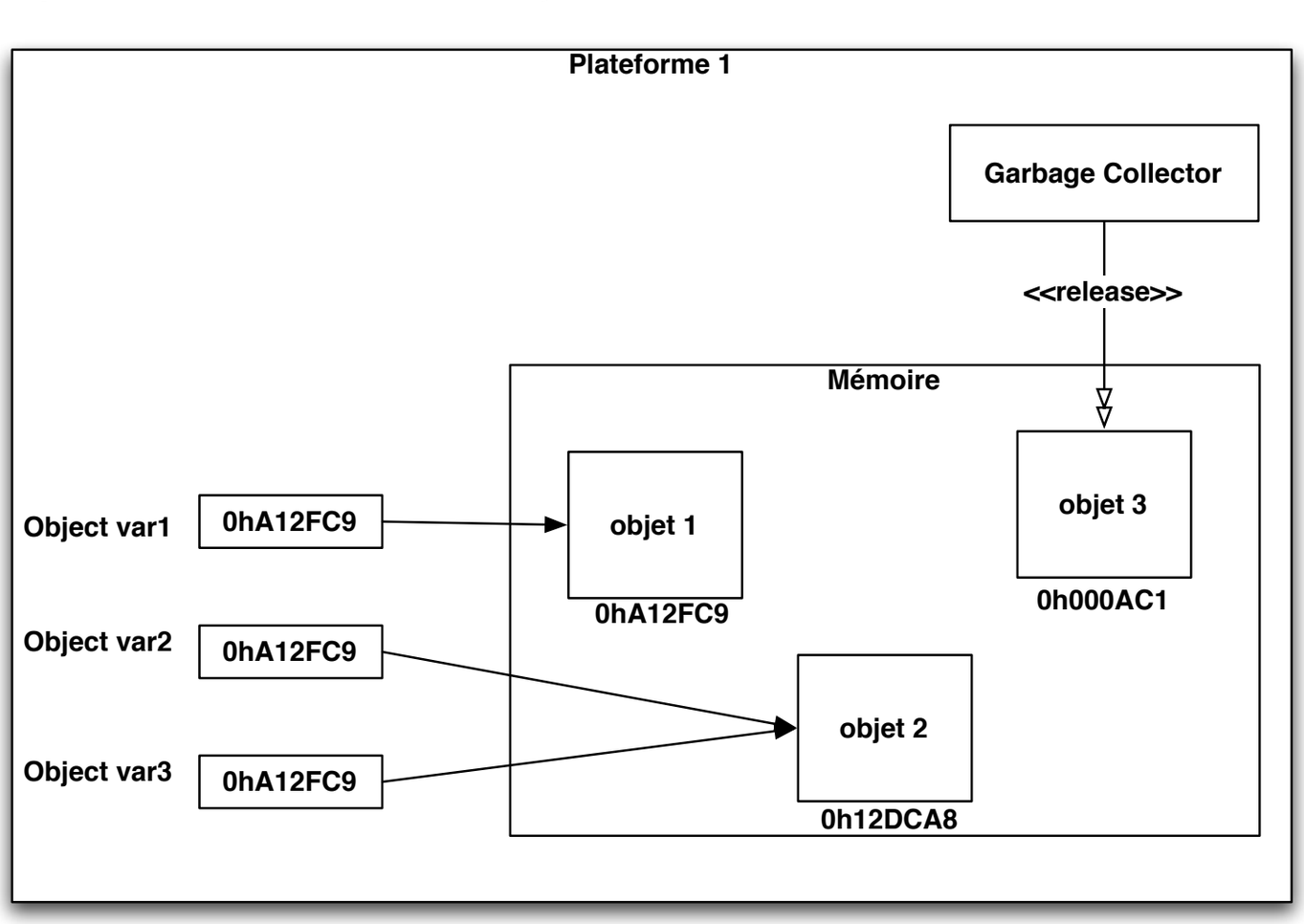
Chap #5

Notion d’objets répartis

- **Une architecture logicielle à base d’objets répartis est constitué d’un ensemble d’objets conçus pour travailler en collaboration.**
- Mais contrairement à une application objet traditionnelle, les objets résident :
  - **Sur différents ordinateurs connectés via un réseau.**
  - Dans différents processus sur la même machine.
- Un objet réparti peut envoyer un message à un autre objet réparti sur une machine distante ou processus pour exécuter une tâche bien précise et ensuite en exploiter le résultat.
- Mais la puissance des objets répartis ne réside pas dans le fait qu’ils soient éparpillés sur un réseau : leur intérêt principal provient du fait qu’un objet distant reste accessible de la même manière qu’un objet local du système par **invocation de méthodes**.
- **On parle de transparence de localisation (“Location Transparency”).**

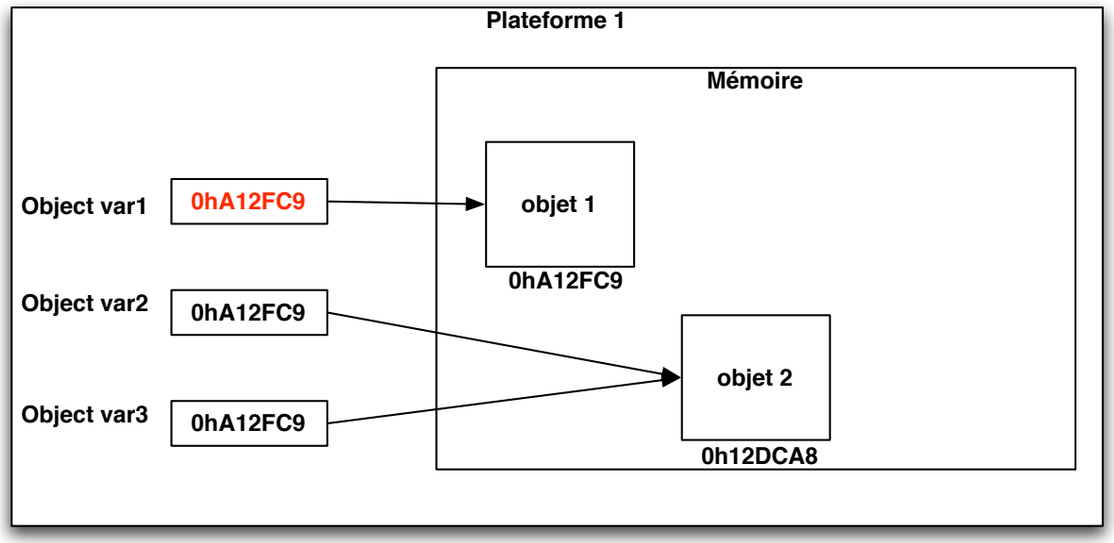
Notion d'objets répartis

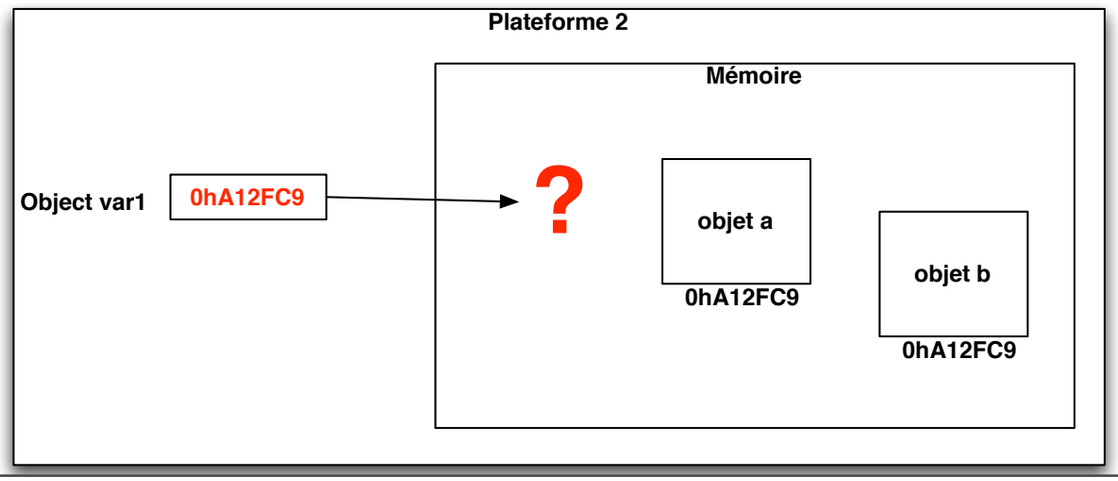
- Dans le paradigme de programmation orienté objets, on accède aux objets via des variables :
  - une variable permet de nommer un objet
  - une variable ne stocke pas l'objet lui même, elle contient une référence mémoire qui pointe vers cet objet



Notion d'objets répartis

- Que se passe-t-il si l'on souhaite accéder à un objet distant depuis notre machine locale ?
  - **Espaces mémoires disjoints** (espaces d'adressage distincts) donc l'adresse de l'objet distant (passé par valeur via le réseau) ne serait pas valide car on ne possède pas l'objet dans la mémoire de la machine locale.





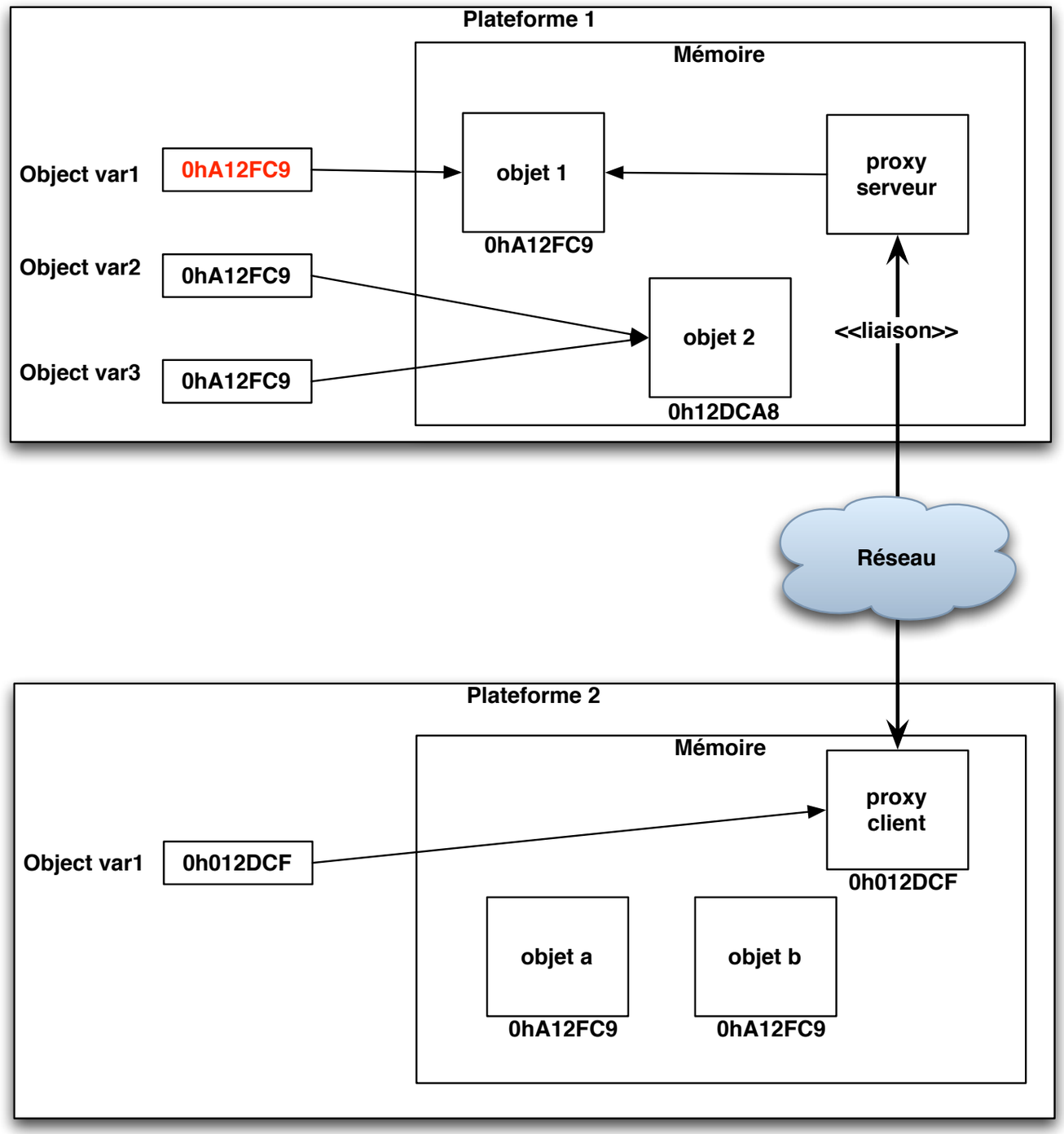
Notion d'objets répartis

- **Une première solution : le passage par valeur de l'objet lui même.**
- Dans ce cas, l'objet de la plate-forme 1 est transmis à la plate-forme 2 **par copie** (copie récursive en profondeur des valeurs des champs de l'objet).
- La variable "var1" de la plate-forme 2 ne pointe pas sur l'objet situé sur la plate-forme 1 mais bien sur une copie de cet objet.
    - L'adresse stockée dans plate-forme1.var1 n'est pas la même que celle sur plate-forme2.var2
- **Mais est-ce vraiment une bonne solution ?**
    - Puisqu'il s'agit d'une copie, si les champs contenus dans l'objet sont mis à jours sur la plate-forme 1, alors la plate-forme 2 n'en a aucunement connaissance (et inversement).
    - Comportement tout à fait normal, puisqu'il s'agit en fait d'une copie !
    - Comportement souhaitable dans certains cas.

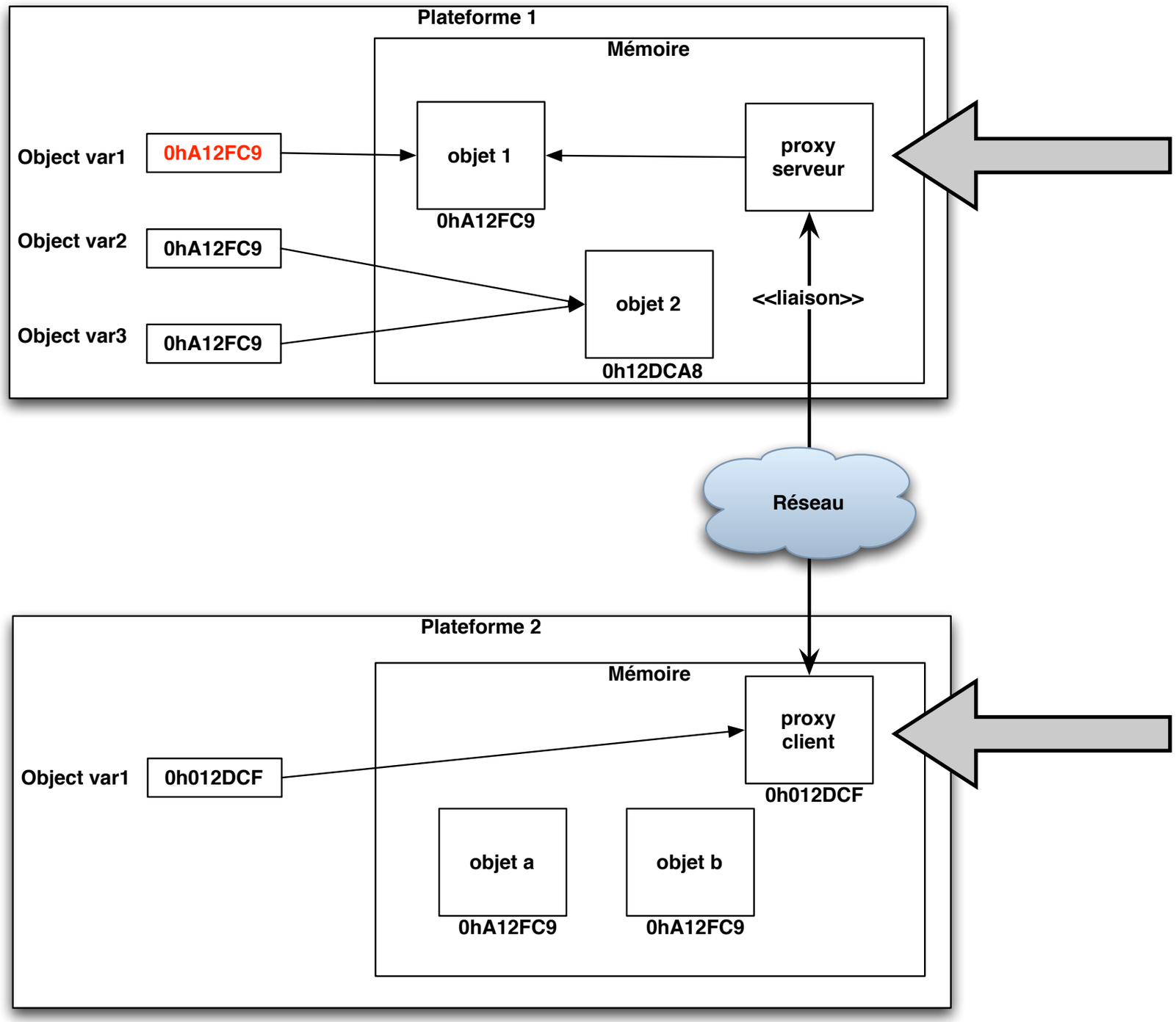
Notion d'objets répartis

- **Une solution plus subtile: les objets répartis.**
- On parle alors d'objet réparti car un objet de ce type est manipulable simultanément par plusieurs clients (en local) bien que situé en fait (à distance) sur le serveur.
- Ces clients peuvent être situés sur différentes plate-formes, dans des espaces d'adressage différents que celui du serveur:
- **Implémentation :**
  - On utilise la notion de "proxy".
  - Le client croit posséder l'objet en local mais il dispose en fait d'un proxy de cet objet.
  - Se faisant passer pour l'objet qu'il n'est pas, le proxy utilise un protocole de liaison (réseau) à sa disposition pour se synchroniser avec l'objet "source" distant.
  - Ce proxy coté client et accompagné de son "miroir" coté serveur.

Notion d'objets répartis



Notion d’objets répartis



Notion d'objets répartis

- Une solution subtile, certes. Mais nettement plus difficile à mettre en oeuvre que le passage par valeur !
- Il n'est pas question pour un développeur de mettre au point toute la mécanique nécessaire à la la création, l'instanciation et la liaison des proxies clients et serveur.
- On se repose donc sur des solution pré-existantes : Java RMI, CORBA, Microsoft DCOM, Borland DDObjects pour le langage Delphi, Pyro pour le langage Python, dRuby pour le langage Ruby,...
- Ce cours en présente 2:
  - **La technologie Java RMI.**
  - **La spécification CORBA.**

Objets répartis

RMI

Chap #5

Un peut d’histoire

- On peut considérer RPC comme étant l'ancêtre de RMI.
- RPC correspond au concept d’appel de procédures à distance. Il a vu le jour en 1976 dans le cadre d’une spécification pour ARPANET puis fut implémenté pour la première fois dans les laboratoires de Xerox en 1981 sous le nom de “Courier”.
- L’implémentation de cette spécification constitue un protocole de communication inter-processus de haut niveau.
- Sun mis au point la première implémentation vraiment populaire pour Unix, elle s’appelle maintenant ONC RPC et constitue la base du protocole NFS de partage de fichiers.
- RPC est un paradigme populaire pour implémenter le modèle client-serveur dans les architectures réparties.
- Il présente des avantages (communications simples, synchrones ou asynchrones, transparence et gestion de l’hétérogénéité) mais aussi des inconvénients (sémantique complexe en cas de panne, restriction sur les paramètres).

Présentation de RMI

- RMI est une implémentation de RPC par SUN pour le langage orienté objet Java. En se sens elle peut être vue comme une évolution de RPC qui intègre directement la notion d'objet dans les communications inter-processus.
  - RPC = "Remote Procedure Call", appel de procédure à distance
  - **RMI = "Remote Method Invocation", invocation de méthodes Java à distance**
- Inclus **par défaut** dans le JDK depuis Java 1.1
- Il existe des implémentations alternatives peu utilisées:
  - NinjaRMI de Berkeley
  - Jeremie de ObjectWeb

Objets répartis en RMI

Stubs et skeletons

- RMI permet la création d'**objets répartis**: un objet RMI est interrogeable à distance par plusieurs clients simultanément.

- Les clients d'un même objet RMI distant **ont l'impression de disposer de l'objet en local**, c'est une illusion rendu possible par la génération de "stubs" et "skeletons" (on parle aussi de souches client et serveur).

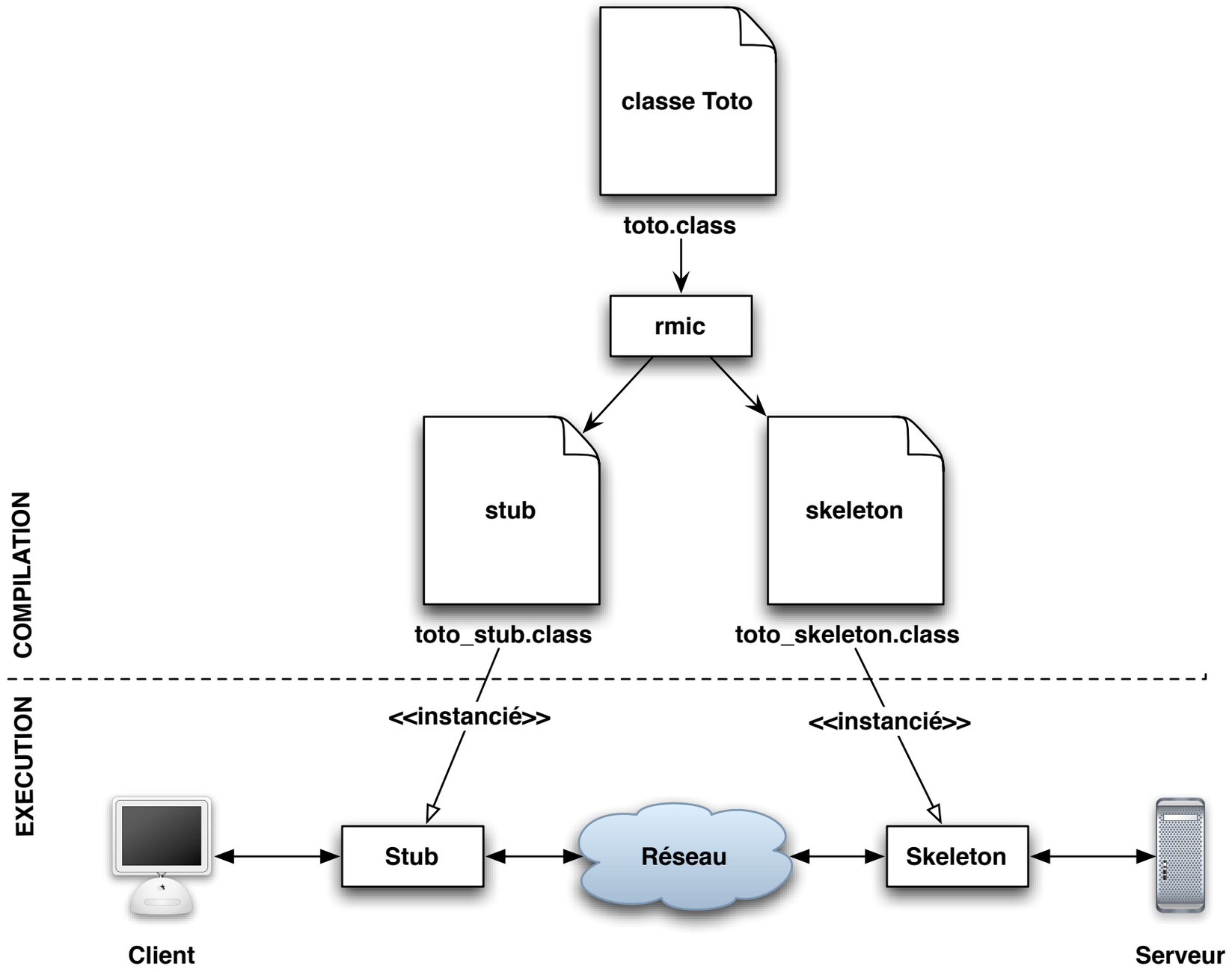
- **Un stub** est un objet coté client qui gère l'encodage et le désencodage des données lors d'un appel à un objet distant RMI.

  Le client croit invoquer l'objet distant **mais il invoque en fait son stub** qui implémente la même interface et gère toute la problématique réseau. Le stub est lié à un skeleton.

- **Un skeleton** est un objet coté serveur qui gère l'encodage et le désencodage des données lors de la réception d'un appel à un objet RMI.

Objets répartis en RMI

Stubs et skeletons



Objets répartis en RMI

Transmission de données

- En fonction de leur type, les données échangées entre clients et serveurs RMI (par exemple lors de l'appel d'une méthode distante) ne seront pas transmises de la même façon.

- On distingue 3 types de données transmissibles dans une architecture RMI:

  - **Les types simples** (int, float, ...): la transmission est effectuée par valeur.

  - **Les objets locaux** (non RMI) implémentants *java.io.Serializable*: la transmission est effectue par valeur (sérialisation java).

  - **Les objets RMI** implémentants *java.rmi.Remote*: la transmission est effectuée par référence (via les stubs et skeletons).

  - Dans les autres cas, une exception *java.rmi.MarshalException* est levée.

Objets répartis en RMI

Remarques

- Les stubs et skeletons étants générés à partir des interfaces des objets RMI, le développeur n'a pas à écrire leur code. Par contre il doit respecter certaines conventions lors du développement de son objet serveur et de ses clients.

- Depuis le JDK 1.2, le skeleton à été intégré au serveur lui-même, il n'existe donc plus d'entités skeleton séparées.

- **Depuis le JDK 1.5, il n'est plus nécessaire de générer la classe du stub à la main avec l'outil *rmic* au moment de la compilation**: cette classe est générée automatiquement lorsque nécessaire à l'exécution.

Modèle de programmation
Ecriture d'une application RMI

- On procède par étapes:

1. **Déclaration des services accessibles à distance**
   → Ecriture d'une interface distante

2. **Définition du code des services**
   → Ecriture d'une classe d'objet serveur implantant l'interface

3. **Instanciation et enregistrement de l'objet serveur**
   → Ecriture du programme serveur

4. **Interaction avec l'objet serveur**
   → Ecriture du programme client

Modèle de programmation
Ecriture d'une interface distante

- Dans l'architecture RMI, un serveur accessible à distance **est un objet** qui expose différents services via ses méthodes.

- Chaque classe d'objet serveur doit donc implémenter **une interface visible depuis l'extérieur**.

  On parle d'interface distante: seules les méthodes de cette interface pourront être invoquées à distance. Les autres, uniquement depuis la JVM locale au serveur.

- Il s'agit d'une interface Java:

  - qui doit étendre *java.rmi.Remote*

  - dont toutes les méthodes doivent lever une exception du type *java.rmi.RemoteException*

Modèle de programmation

Ecriture d'une interface distante

```
import java.rmi.Remote;
import java.rmi.RemoteException;
import java.util.List;

public interface RemoteCompte extends Remote {
    public String getTitulaire() throws RemoteException;
    public float solde() throws RemoteException;
    public void deposer(float montant) throws RemoteException;
    public void retirer(float montant) throws RemoteException;
    public List historique() throws RemoteException;
}
```

Modèle de programmation

Ecriture d'une classe implantant l'interface

- Afin de spécifier le fonctionnement de l'objet serveur, on écrit une classe implémentant l'interface distante définie précédemment.

- Les points suivants doivent être respectés:

    - Les constructeurs doivent lever une exception *java.rmi.RemoteException*

    - Si elle ne possède pas de constructeurs alors il faut quand même en déclarer un vide qui lève une exception *java.rmi.RemoteException*

    - De préférence la classe doit étendre *java.rmi.server.UnicastRemoteObject,* ce qui facilitera le déploiement de l'objet serveur dans le runtime RMI.

Modèle de programmation

Ecriture d'une classe implantant l'interface


```
package cours_rmi;

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

public class RemoteCompteImpl extends UnicastRemoteObject implements RemoteCompte
{

    private String nom;
    //attention, il s'agit de la classe Float
    //pour permettre le synchronized()
    private Float solde;

    protected RemoteCompteImpl(String nom) throws RemoteException {
        super();
        this.nom = nom;
    }

    public void deposer(float montant) throws RemoteException {
        synchronized(solde) {
            solde += montant; //attention aux accès concurrents
        }
    }
(...)
```

Modèle de programmation
Génération des stubs

- Cette manipulation n'est pas nécessaire si **toutes** les JVM de l'architecture répartie >= v1.5.0.
- Sinon, il faut quand même générer manuellement et de manière statique les stub pour chaque classe d'objet serveur.
- Pour cela on utilise la commande `rmic` après avoir compilé les classes de l'application, par exemple:
  `rmic RemoteCompteImpl` → RemoteCompteImpl_stub.class
- Options de la commande `rmic`:
  - `-d path` répertoire pour les fichiers générés
  - `-keep` conserve le code source des souches générées
  - `-v1.1 /-v1.2` souches version 1.1/1.2
  - `-vcompat` par défaut, souches pour JDK 1.1 et 1.2

Modèle de programmation

Ecriture du programme serveur

- On écrit le code du serveur, il s'agit d'une classe qui dispose d'une méthode `main(...)` qui se charge:

1. D'instancier la classe définie précédemment de manière à obtenir l'objet serveur qui fournis les services.

2. D'exporter l'instance dans le runtime RMI, **le stub est créé lors de cette étape.**

   - **Dans le code suivant, cela est fait automatiquement car la classe serveur étend** `UnicastRemoteObject`**.**

   - Dans le cas contraire il faudrait faire un appel explicite à `UnicastRemoteObject.exportObject(Remote obj,...)`

     Attention: si les stubs sont générés automatiquement (pas de rmic) on ne peut pas utiliser la méthode `exportObject(Remote obj)`

3. D'enregistrer l'instance (en fait son stub) dans le serveur de nom RMI.

Modèle de programmation

Ecriture du programme serveur


```
import java.rmi.Naming;
import java.rmi.RemoteException;

public class ServeurCompte {
    public static void main(String [] args) throws Exception {
        RemoteCompte compte = new RemoteCompteImpl("Bob");

        Naming.bind("Bob", compte);
    }
}
```

Modèle de programmation

Ecriture du programme serveur


```
import java.rmi.Naming;
import java.rmi.RemoteException;

public class ServeurCompte {
    public static void main(String [] args) throws Exception {
        RemoteCompte compte = new RemoteCompteImpl("Bob");

        Naming.bind("Bob", compte);
    }
}
```

Instanciation +
export dans le runtime

Modèle de programmation

Ecriture du programme serveur


```
import java.rmi.Naming;
import java.rmi.RemoteException;

public class ServeurCompte {
    public static void main(String [] args) throws Exception {
        RemoteCompte compte = new RemoteCompteImpl("Bob");

        Naming.bind("Bob", compte);
    }
}
```

Instanciation + export dans le runtime

enregistrement registry

Modèle de programmation
Ecriture du programme serveur

- Une fois exportée dans le runtime RMI, l'instance *compte* est prête à recevoir des invocations de méthodes.
- Mais l'objet serveur n'est **visible** par des clients distants **que si il est aussi enregistrée dans le serveur de noms RMI** → appel à *Naming.bin()/rebind().*
- Tant que *compte* reste enregistré dans le runtime RMI (différent du registry !):
  - Les services sont disponibles.
  - La thread principale du programme serveur est maintenue en vie par le runtime RMI (même si sa méthode *main* s'est terminée).
  - Pour procéder à son désenregistrement du runtime il faut exécuter l'instruction suivante coté serveur: `UnicastRemoteObject.unexportObject(compte, force);`

Modèle de programmation
Ecriture du programme client

- Tous les clients RMI vont suivre ce modèle de programmation:

1. Recherche de l'objet serveur dans le serveur de noms RMI

2. Invocation des méthodes de l'objet serveur (via son stub, mais cela reste transparent pour le développeur coté client).

```
import java.rmi.Naming;

public class ClientCompte {

    public static void main(String [] args ) throws Exception {
        RemoteCompte compte = (RemoteCompte) Naming.lookup("Bob");
        compte.deposer(100);
        System.out.println(compte.solde());
    }
}
```

Modèle de programmation
Concurrence

- Un objet serveur RMI est susceptible d'être accédé par plusieurs clients simultanément.
- A plusieurs appels simultanés d'une même méthode peuvent correspondre plusieurs threads concurrentes sur le même objet serveur.
- Il faut donc toujours concevoir des méthodes RMI "thread-safe", c'est à dire exécutables concurremment de façon cohérente.
  - Faire particulièrement attention aux méthodes qui changent ou accèdent à l'état de l'objet serveur (les variables internes).
  - Les problématiques sont les mêmes que dans un environnement multithreadé classique (non RMI).

Services de l'intergiciel RMI

- RMI est considéré comme un intergiciel (middleware) relativement basique dans la mesure où les services qu'il fournit aux objets répartis sont assez primitifs, mais néanmoins nécessaires.

  Outre sa fonctionnalité principale qui est de permettre la répartition transparente sur un réseau des objets java (mécanisme de communication), On détaille ici 3 services complémentaires de RMI:

  - **Service de nommage** (rmiregistry)
  - **Service d'activation d'objets à la demande**
  - **Service de Garbage Collector** (gestion de la mémoire)

Services de l'intergiciel RMI

Nommage

- **Problématique**: comment faire en sorte que les objets serveurs soient connus de tous les clients ?
- **Solution**: utilisation du service de nommage de RMI: le "RMI Registry".
  - doit être lancé, une seule fois, avant les programmes clients et serveurs
  - disponible par défaut sur le port 1099
- Il permet d'enregistrer les liaisons entre un objet serveur et un nom symbolique.
  - Il s'agit de noms simples (pas composés, pas de hiérarchie)
  - URL d'un objet dans le registry: [rmi://]machine[:1099]/nomSymbolique
    - rmi:// et et :1099 sont facultatifs
    - machine correspond à localhost par défaut.

Services de l'intergiciel RMI

Nommage

- Le registry peut être lancé:
  - de façon autonome dans un shell avec l'outil Java rmiregistry
  - à partir d'un programme par l'appel de le méthode statique: *java.rmi.registry.LocateRegistry.createRegistry(int port)*
- Le registry peut être accédé:
  - **directement** via les méthodes statiques de la classe *java.RMI.Naming* qui prennent en argument l'url complète du registry (url+port): *bind(String, Remote)*, *rebind(String, Remote)*, *unbind(String), list(String) et lookup(String)*
  - **en 2 étapes:**
    1. il faut récupérer un pointeur sur le registry java.rmi.registry.LocateRegistry.getRegistry(String host, int port)
    2. utiliser les mêmes méthodes que Naming sur l'objet retourné (sans l'url du serveur)
- Les méthodes bind/rebind/unbind ne sont disponibles que dans la JVM locale au registry par mesure de sécurité.

Services de l'intergiciel RMI

Activation d'objets

- Les objets serveur peuvent être activés à la demande (des clients) depuis le JDK 1.2.
- Les avantages sont multiples:
  - Cela permet d'éviter d'avoir des objets serveurs actifs en permanence (qui "tournent" dans une JVM), car cela est trop coûteux (en mémoire) si il y a beaucoup d'objets serveur.
  - Cela permet d'avoir des références d'objet persistantes:
    - En cas d'arrêt inopiné d'un objet serveur, le démon rmid peut le relancer à partir de la même référence.
    - Les clients peuvent continuer à utiliser la même référence.
- Pour utiliser ce service il faut lancer le démon *rmid* et les objets serveurs doivent étendre la classe java.rmi.activation.Activable.

Services de l'intergiciel RMI

Garbage Collector réparti

- Afin de conserver une des caractéristiques essentielles du langage Java, à savoir sa capacité à gérer automatiquement la mémoire, il était nécessaire au mécanisme RMI d'implémenter un Garbage Collector (GC, ramasse-miettes) qui sache se séparer des objets non utilisés, même par des clients distants.

- Le Garbage Collector réparti de RMI utilise l'algorithme implémenté dans le langage modula-3 avec ses Network Objects. Il passe par la tenue permanente d'**une liste de toutes les références à des objets distant**s dans chaque machine virtuelle impliquée. L'algorithme est complexe est s'appuie sur l'établissement d'un protocole entre les références d'objets dans les machines clientes et les objets sur le serveur. Ce protocole permet l'**incrémentation et la décrémentation de compteurs de référencement**: lorsqu'un compteur atteint 0, l'objet réparti est candidat pour le ramassage dans sa JVM locale (GC RMI passe la main au GC local).

- A cela s'ajoute la notion de **bail** (lease). Ainsi la mémoire est allouée à un objet pour un temps fini (par défaut 10 min) de manière à se prémunir contre les partitions de réseau ou les pertes de message de déréférencement.

Services de l'intergiciel RMI

Garbage Collector réparti

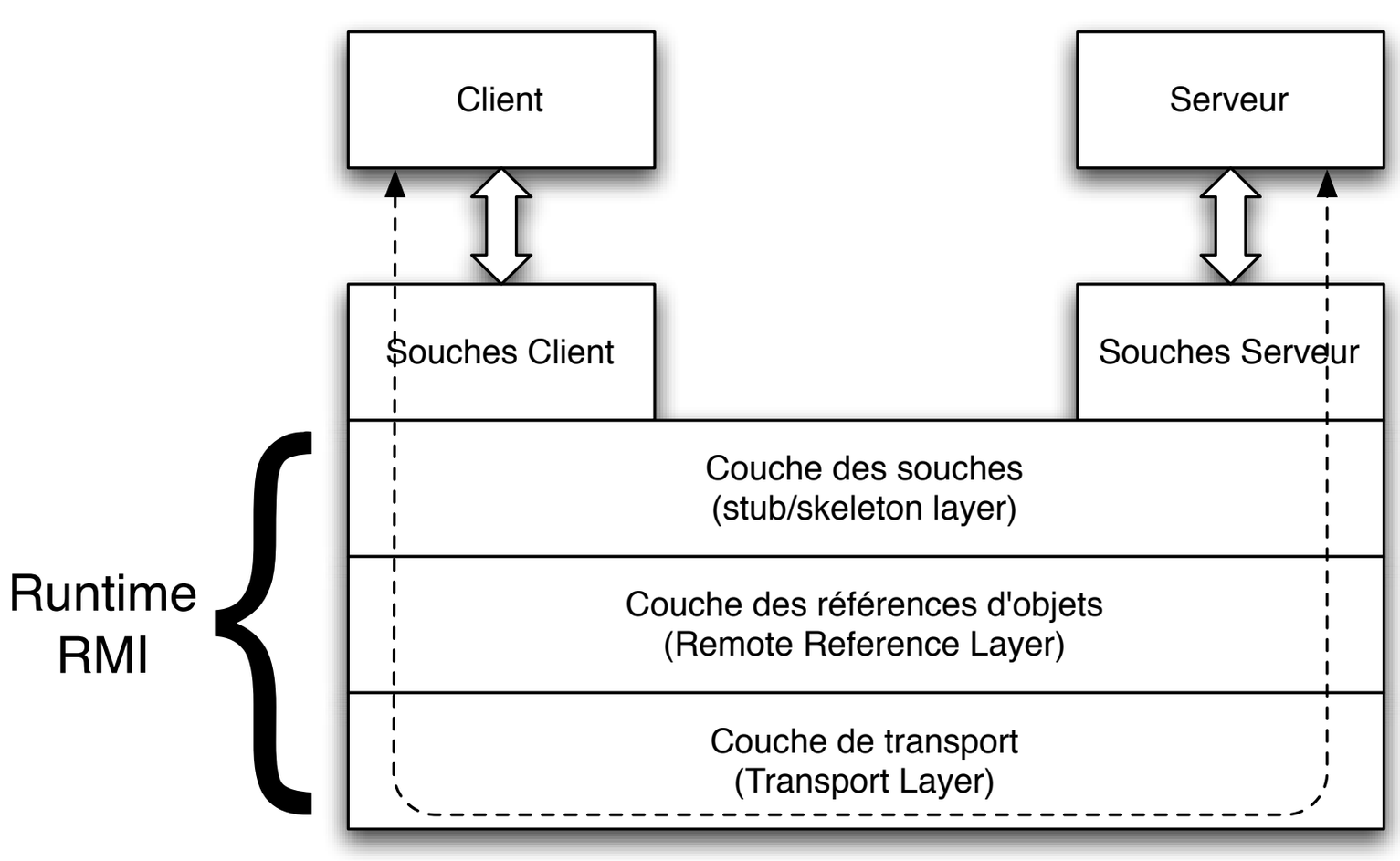
- Il est également possible de demander au GC réparti d'alerter un objet donné lorsqu'il n'est plus référencé à distance (alors qu'il peut l'être encore en local). Il suffit pour cela que la classe de l'objet implémente l'interface *java.rmi.server.Unreferenced.*
    - Dans ce cas le message de déréferencement du GC réparti correspondra à un appel de la méthode *unreferenced()*.
- Attention: tant qu'il existe une référence à un objet serveur dans une JVM locale ou distante, l'objet ne sera pas collecté. Lorsqu'un programme serveur enregistre un objet serveur dans le registry RMI, **ce dernier reste accessible par une JVM: celle du registry !**

  L'objet serveur reste ainsi accessible tant qu'il n'a pas été **explicitement retiré du registry** ET que que des clients maintiennent une référence sur cet objet.

Architecture du runtime RMI

- Il nous faut détailler le fonctionnement de RMI pour mieux comprendre ses possibilités et ses limites.

- Comme d'habitude, c'est une affaire de couches. Elles sont au nombre de trois:



Architecture du runtime RMI

Couche des souches

- Interface entre l'application et le système RMI
- Les stubs sont des implémentations des interfaces décrivant les objets distants, en vue d'effectuer les invocations. Pour chaque objet distant manipulé par un client correspond un stub.
- Une fois le stub obtenu, le client peut effectuer une invocation distante. Il assure alors les fonctionnalités suivantes:

1. Obtention d'une référence d'objet distant qui par appel à la couche inférieure de gestion des références d'objets (Remote Reference Layer).
2. Sérialisation des arguments de la méthode invoquée.
3. Notification à la couche inférieure (Remote Reference Layer) que l'appel peut être effectué.
4. Désérialisation de la valeur de retour, s'il n'y a pas eu d'exceptions levées.
5. Notification à la couche inférieure que l'appel est terminé.

Architecture du runtime RMI

Couche des souches

- Coté serveur, le skeleton va réagir ainsi lors de la réception d'un appel distant :

  1. Il désérialise les arguments reçus par le stub.
  2. Il effectue l'appel de méthode demandée sur l'objet serveur.
  3. Il sérialise la valeur de retour et la transmet au stub.

Architecture du runtime RMI

Couche des références d'objet

- Cette couche assure l'aspect fonctionnel de l'invocation. Elle doit notamment déterminer si l'objet invoqué est un objet simple ou un objet répliqué sur plusieurs serveurs.
- Elle remplit les mêmes fonctions que l'adaptateur objet CORBA.
- C'est elle qui démarre une nouvelle thread si l'objet serveur le nécessite ou qui fait appel à une thread existante si l'objet est déjà actif dans cette thread (la thread du programme serveur qui est maintenue en vie par le runtime).
- Plus généralement, il est possible à un objet serveur d'utiliser **différents ~~protocoles~~ types de liaisons** selon l'architecture logicielle mise en place (cf. transparent suivant).
- Cette couche, comme la précédente, dispose d'une partie cliente et d'une partie serveur distinctes, qui assurent notamment la gestion des objets répliqués.

Architecture du runtime RMI

~~Protocoles~~ Types de liaisons

- RMI à été conçue de manière à ce que les les objets serveurs puissent fonctionner avec différents types de liaisons :
  - invocation d'objet Java distant joignable en point à point
  - invocation de groupes d'objets Java distants répliqués
  - invocation d'objets Java distant joignables par diffusion
- **En pratique, seul le premier cas de figure (point à point) est mis en oeuvre pour les objets Java dans la version actuelle de RMI: avec la classe *UnicastRemoteObject*.**

Architecture du runtime RMI

Couche de transport

- Cette couche assure la connexion entre le serveur et le client et la gestion de la localisation de tous les objets distants répertoriés.

- Elle présente un niveau d'abstraction par flux, ce qui permet ensuite d'utiliser différences possibilités de transmission des appels. Il serait possible d'implémenter une couche de transport de type datagramme (qui serait plus performante, mais n'assurerait pas automatiquement une transmission fiable).

- Cette couche de transport utilise un certain nombre d'abstractions (ou d'objets) qui représentent:

  - les machines virtuelles qui sont en communication

  - les canaux de communication établi entre clients et serveurs

  - les connexions effectives

Architecture du runtime RMI

Couche de transport

- La couche de transport assure les fonctions suivantes:
  - Connexion entre les deux espaces d'adressage (les deux machines virtuelles client/serveur).
  - Suivi des connexions en cours.
  - Ecoute et réponse aux invocations.
  - Constitution d'une table de tous les objets distants disponibles.
  - Aiguilage des invocations.

Notions de protocole

- Nous avons vu précédemment que différents ~~protocoles~~ types de liaison entre objets peuvent être utilisés avec RMI.

- Le développeur RMI n'a pas conscience du ~~protocole~~ type de liaison utilisé.

- Le protocole de transport utilisé par défaut est **JRMP** (Java Remote Method Protocol) **construit au dessus de TCP, il fonctionne en point à point**.

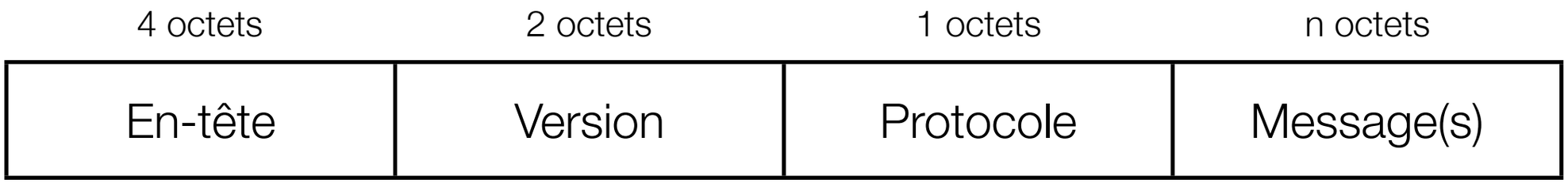
- Mais ce protocole ne permet la communication qu'entre objets répartis RMI implémentés en Java.

- Il existe un autre protocole de transport nommé **RMI/IIOP**, supporté par le JDK, qui permet d'obtenir un certain niveau d'intéropérabilité entre les objets répartis RMI et les objets répartis CORBA (cf. cours #4.2).

Notions de protocole

JRMP

- Protocole de transport des invocations de méthodes distantes pour Java RMI, aussi connu sous le nom de “RMI Wire Protocol”.
- Structure des paquets échangés par le protocole JRMP:
  - En-tête: magic number (JRMI)
  - Version: numéro de version du protocole
  - Protocole:
    - SingleOpProtocol: une seule invocation par paquet (eg. RMI over HTTP)
    - StreamProtocol: plusieurs invocations à la suite vers un même objet
    - MultiplexProtocol: plusieurs invocations vers une même machine multiplexées sur la même connection
  - Message(s): cf. transparent suivant



Notions de protocole

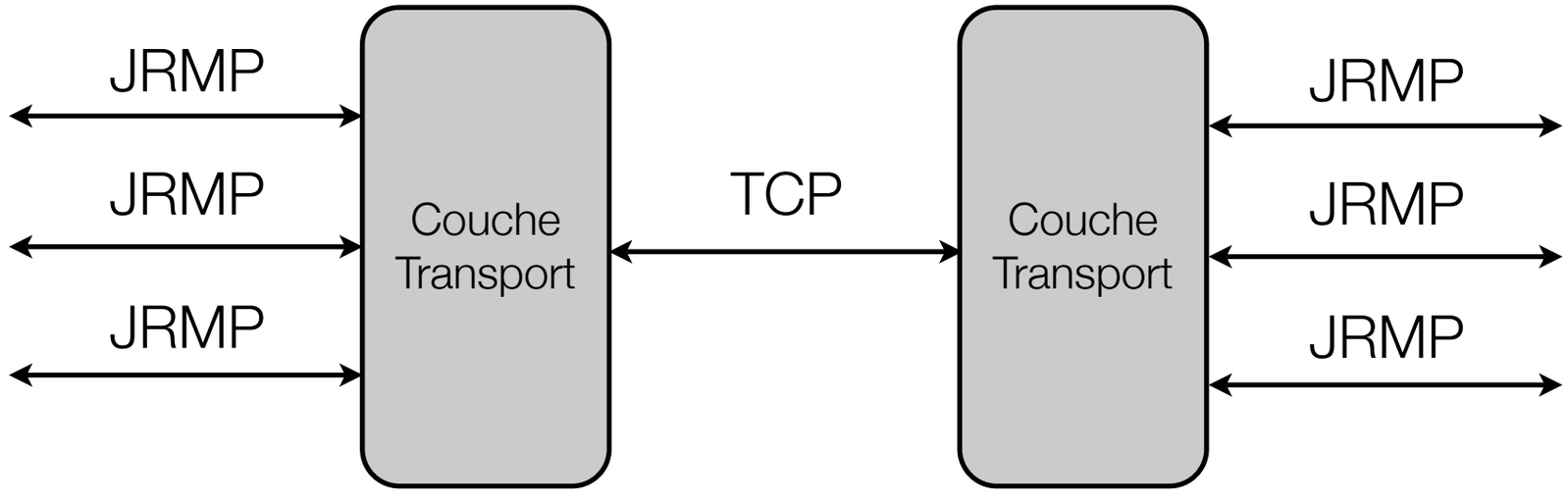
JRMP

- **Messages sortants:**
  - Call: véhicule une invocation de méthode (+ callData)
  - Ping: teste le bon fonctionnement d'un serveur
  - DcgAck: utilisé par l'algorithme de garbage-collecting pour signaler que les objets répartis retournés par le serveur ont été reçus par le client.
- **Messages entrants:**
  - Return: véhicule le retour de l'invocation (+returnValue)
  - HttpReturn: idem mais encapsulé dans une requête HTTP (+returnValue)
  - PingAck: acquittement d'un message Ping

Notions de protocole

JRMP

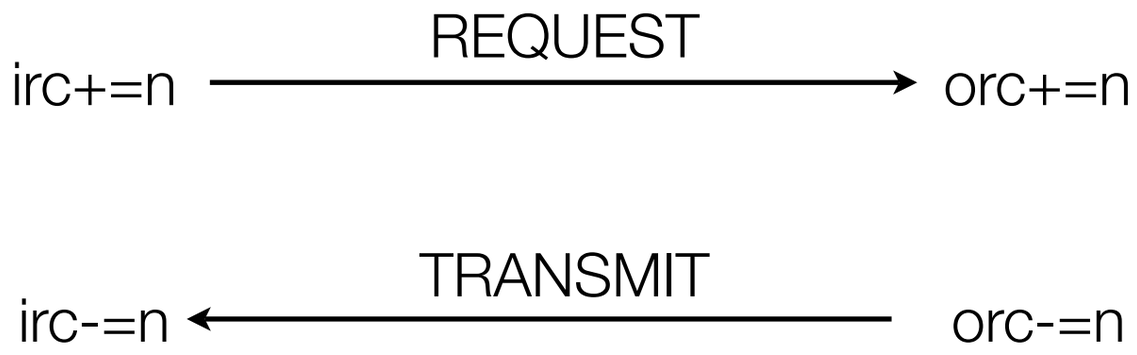
- **Multiplexage de connexion:**
  - **But**: transmettre plusieurs invocations RMI sur une même connexion TCP.
  - **Protocole de multiplexage dans JRMP**:
    - OPEN, CLOSE, CLOSEACK: pour gérer une connexion RMI multiplexée.
    - REQUEST, TRANSMIT: pour assurer le transport de l'information.



Notions de protocole

JRMP

- **Mécanisme de contrôle de flux:**
  - **But:** éviter qu'un buffer plein pour une connexion bloque toutes les autres, et aussi éviter qu'une connexion qui ne se termine pas bloque toutes les autres (par exemple en cas d'appels récursifs).
  - **Solution:** 2 compteurs (en nombre d'octets échangés) pour chaque connexion RMI multiplexée:
    - **irc**: input request count
    - **orc**: output request count
    - irc et orc ne doivent jamais être négatifs.
    - irc ne doit pas dépasser une certaine valeur (en nombre d'octets) qui le bloquerait.



Notions de protocole
RMI/IIOP

- CORBA correspond à la spécification d'un autre type de middleware à objets répartis qui n'est pas spécifiquement lié au monde Java.

- IIOP (Internet Inter-ORB Protocol) est un protocole OMG pour la communication inter et intra-middlewares CORBA.

- RMI/IIOP correspond à la fusion du modèle simple de programmation RMI avec le protocole robuste IIOP.

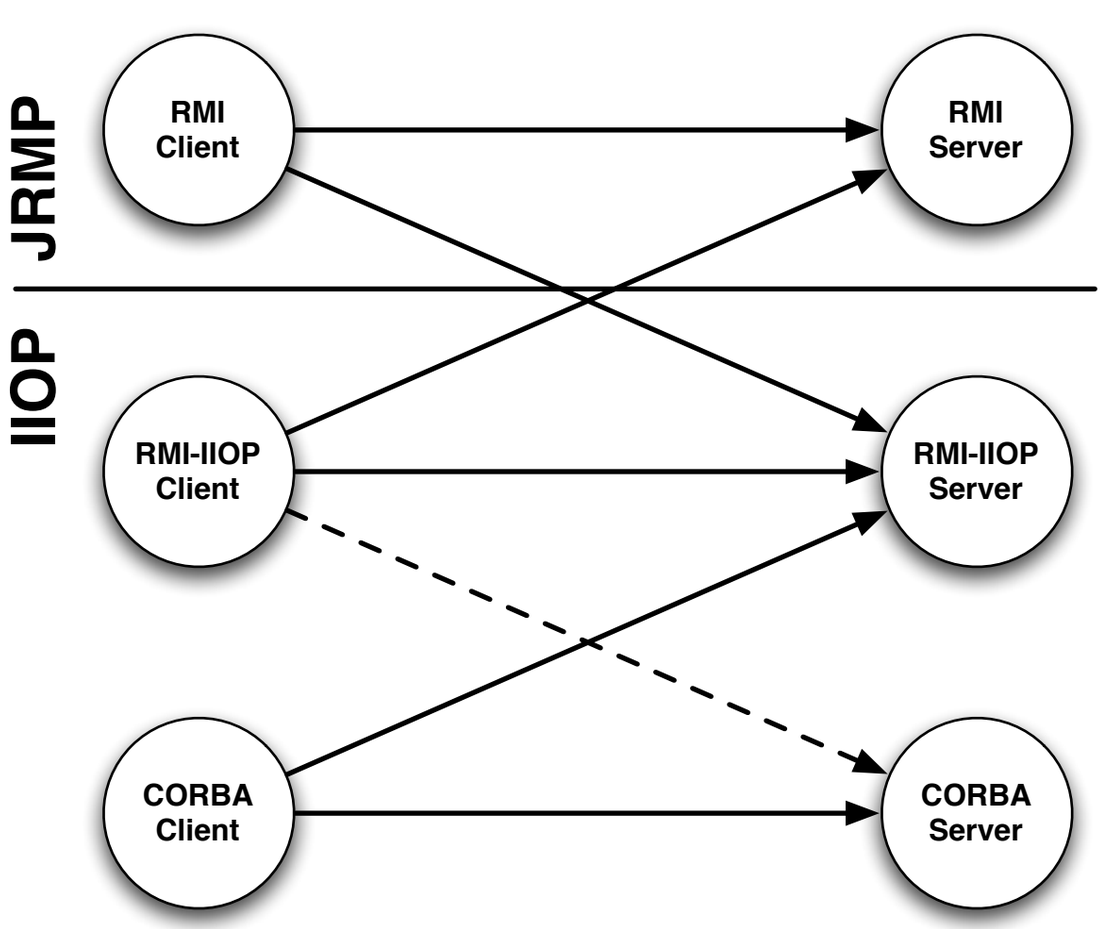
- L'utilisation de l'implémentation d'IIOP avec RMI en lieu et place de JRMP permet d'obtenir un certain niveau d'**intéropérabilité entre RMI et CORBA** :

  - Les clients CORBA peuvent alors "parler" à des objets serveurs RMI-IIOP.

  - Et vice-versa (à supposer que vous ayez un ORB CORBA >= v2.3)

Notions de protocole
RMI/IIOP

- **Limitation:** RMI-IIOP permet à un client RMI d'interagir avec des objets serveur CORBA implémentés dans d'autres langages de programmation, **mais uniquement si il existe une interface distance RMI de ces objets.**

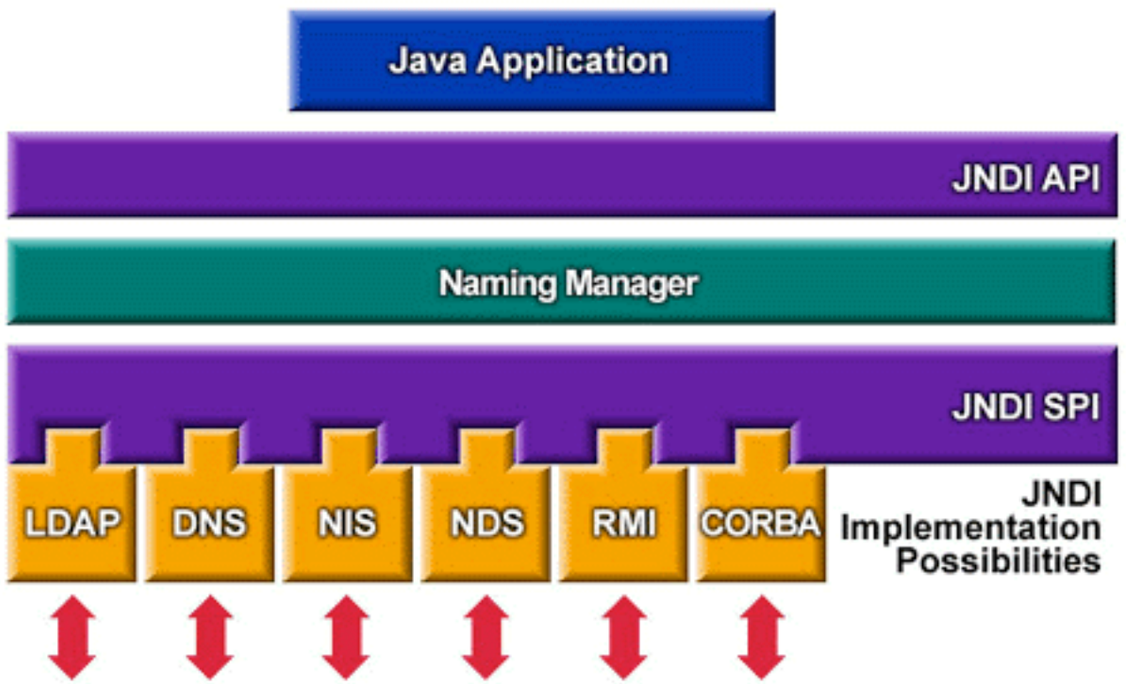


Notions de protocole
RMI/IIOP

- **Modèle de programmation:**
  - Les classes d'objets répartis doivent étendre *javax.rmi.PortableRemoteObject*
  - **Les souches doivent êtres générées** avec *rmic -iiop*. Deux fichiers sont alors générés pour chaque classe d'objet serveur:
    - *_nomClasse_stub.java* : souche cliente
    - *_nomClasse_Tie.java* : souche serveur
- Service de nommage JNDI (Java Naming and Directory Interface)
  - outil *tnameserv*
- Les conversions de type sur les objets distants doivent utiliser la méthode *static PortableRemoteObject.narrow()*

Notions de protocole
RMI/IIOP

- Pour accéder à tous types d'objets répartis compatibles IIOP on n'utilise plus directement le service de nommage de RMI mais on passe par **JNDI (Java Naming and Directory Interface)**:
    - Ce n'est pas un service mais un ensemble d'interfaces (une API Java).
    - Cette API permet d'unifier l'accès à différents **serveurs de noms** (dont le RMI registry et le registry CORBA).



Notions de protocole
RMI/IIOP

- Exemple d'instanciation d'un objet compatible RMI/IIOP et enregistrement dans un serveur de nom **local**.
- Cet objet sera accessible simultanément depuis les mondes CORBA et RMI.

```
import javax.rmi.PortableRemoteObject;

public class RemoteCompteImpl extends PortableRemoteObject implements RemoteCompte {
(...)
}



public class ServeurCompte {

    public static void main(String [] args) throws Exception {
        RemoteCompte compte = new RemoteCompteImpl("Bob");
        Context ic = new InitialContext();
        ic.rebind("Bob", compte);
    }
}
```

Notions de protocole
RMI/IIOP

- Exemple d'instanciation d'un objet compatible RMI/IIOP et enregistrement dans un serveur de nom **distant**.
- Cet objet sera accessible simultanément depuis les mondes CORBA et RMI.

```
public class ServeurCompte {

    public static void main(String [] args) throws Exception {
        RemoteCompte compte = new RemoteCompteImpl("Bob");

        Hashtable env = new Hashtable();
        //The CORBA Common Object Services (COS) name server is used to store CORBA
        //object references.
        env.put(Context.INITIAL_CONTEXT_FACTORY, "com.sun.jdni.cosnaming.CNCtxFactory");
        //adresse du serveur de noms
        env.put("java.naming.provider.url", "iiop://localhost:1704");

        Context ic = new InitialContext(env);

        ic.rebind("Bob", compte);
    }
}
```

Notions de protocole
RMI/IIOP

- **Restrictions liées à l'utilisation de RMI/IIOP**
  - Les constantes définies dans les interfaces Remote doivent être de type primitif ou de type String et être évaluables à la compilation.
  - Pas de surcharge pour les méthodes des interfaces Remote, c'est à dire pas deux méthodes avec le même nom et des profils différents.
  - Pas de mécanisme de Garbage Collector réparti
  - Les fonctionnalités fournies par les interfaces suivantes ne sont pas utilisables: *UnicastRemoteObjet, RMISocketFactory, Unreferenced, java.rmi.dgc.**

Chargement de classes à distance

Problématique #1

- En utilisant l'API RMI, des applications peuvent créer des objets serveur qui acceptent des appels de méthode de clients situées dans d'autres JVM au travers du réseau.
- Le client utilise un "proxy" de l'objet serveur pour effectuer les appels de méthodes à distance: le stub.
- **Mais si la classe du stub à été générée coté serveur, elle n'est pas directement disponible sur le client**. Comment effectuer l'appel de méthode dans ces conditions ?
- **Nota :** la problématique se pose uniquement pour Java < 1.5

Chargement de classes à distance

Problématique #2

- Le client dispose d'une interface distante pour chaque type d'objets serveur qu'il est susceptible d'appeler. Cette interface indique les types des paramètres et les types de retour des méthodes définies sur l'objet serveur.
- Le client et le serveur disposent dans leur CLASSPATH du "bytecode" (les .class) des classes ou des interfaces **déclarées** dans chacune de ces interfaces distantes.
- **Mais que ce passe-t-il si l'objet serveur renvoi dans une méthode une instance d'un sous-type de celui déclaré dans l'interface distante ?**
- **Ou si le client passe en argument d'une méthode distante une instance d'un sous-type de celui d'un paramètre déclaré** ?
- C'est tout à fait permis par le langage Java, mais le récipiendaire ne dispose pas forcément du bytecode de cette sous-classe !
- Même problématique que #1 → **transfert de .class entre les clients et serveurs.**

Chargement de classes à distance

Solution

- Java charge les .class à la demande à partir du disque (via la variable CLASSPATH).
- **RMI introduit en plus un mécanisme de chargement des classes à distance via HTTP ou FTP (uniquement avec JRMP).**
  - Avantage: les classes sont déployées sur un seul site (plus rapide et plus simple à gérer, surtout en cas de mise à jours).
  - Inconvénient: le serveur de classe est un point centralisé sensible qui peut mettre en jeux le bon fonctionnement de l'application répartie.
- La propriété ***java.rmi.server.codebase***, positionnée par le **programme serveur (celui qui effectue la publication des objets serveurs) ou client** permet d'indiquer l'emplacement d'un serveur de fichier HTTP ou FTP disposant des .class nécessaire au bon déroulement des appels de méthodes.
- Clients et serveur doivent instancier un RMISecurityManager pour permettre le téléchargement: `System.setSecurityManager(new java.rmi.RMISecurityManager());`

Chargement de classes à distance

Téléchargement de stub coté client

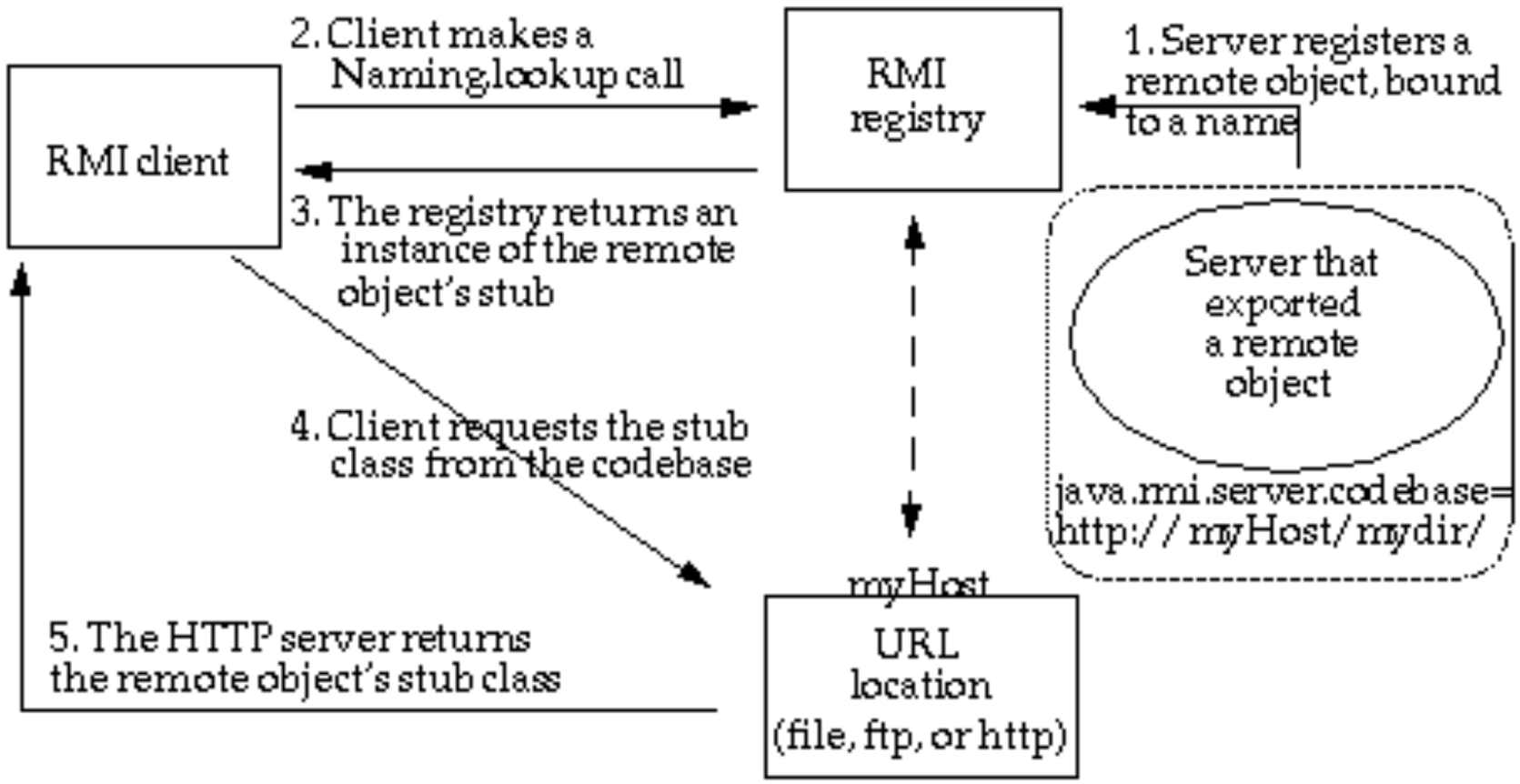


Image © Sun

Chargement de classes à distance

Passage d'une sous classe en param.

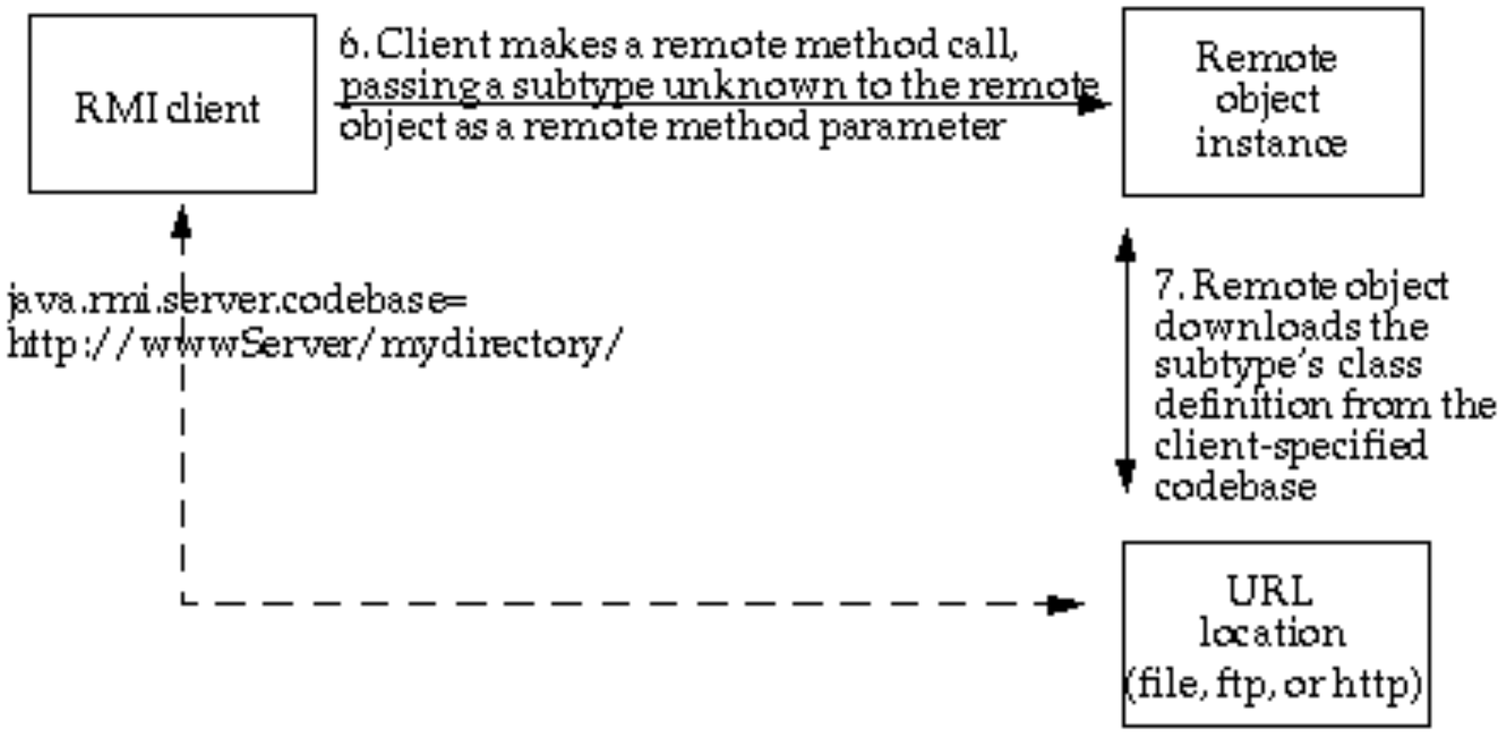


Image © Sun

Chargement de classes à distance

Comment l'éviter ?

- Le principal inconvénient pour le développeur d'une application RMI qui utiliserait le chargement de classe à distance se situe dans la lourdeur de la mise en place du système.
- Dans la pratique, pour de petites architectures, il est souvent préférable d'éviter son utilisation qui peut souvent apporter plus d'inconvénients que d'avantages.
- Il suffit que clients et serveurs possèdent dans leur classpaths respectifs l'ensemble des classes pour tous les types d'instances qu'ils sont susceptibles d'échanger :
  - Les clients et serveurs sont souvent développés en commun.
  - On connaît souvent au moment du développement l'ensemble des types concrets des données échangées.
  - On choisit de "packager" les bonnes classes dans l'archive du client et de celle du serveur au moment de la distribution.

Bibliographie

- Java RMI. W. Grosso. O'Reilly
- **RMI**: http://java.sun.com/j2se/1.5.0/docs/guide/rmi/
- **RMI/IIOP**: http://java.sun.com/products/rmi-iiop/
- **JNDI**: http://java.sun.com/j2se/1.5.0/docs/guide/jndi/index.html
- **Chargement de classes à distance:**
  http://java.sun.com/j2se/1.5.0/docs/guide/rmi/codebase.html

Fin du cours #6