

Java RMI

Julien Sopena

Julien.Sopena@lip6.fr

(basé sur un cours de **Gaël Thomas** et de **Lionel Seinturier**)

Université Pierre et Marie Curie

Master Informatique

M1 – Spécialité SAR

1. Caractéristiques
2. Construction d'une application RMI
3. Passage de paramètres
4. Architecture interne de RMI
5. Intégration RMI et CORBA

1. Caractéristiques

Java RMI = Solution de Sun pour l'invocation de méthodes Java à distance

- Inclus par défaut depuis le JDK 1.1
- Nouveau modèle de souches dans JDK 1.2
- Génération dynamique des souches dans JDK 5

- Implantation alternatives (open-source)
 - NinjaRMI (Berkley)
 - Jeremie (ObjectWeb)

- Suite de développement
 - Package java.rmi
 - + outils
 - ✓ Générateur de souches (avant jdk 5)
 - ✓ Serveur de noms (rmiregistry)
 - ✓ Démon d'activation

1. Caractéristiques

Principe : chaque classe d'objet serveur doit être associé à une interface Java

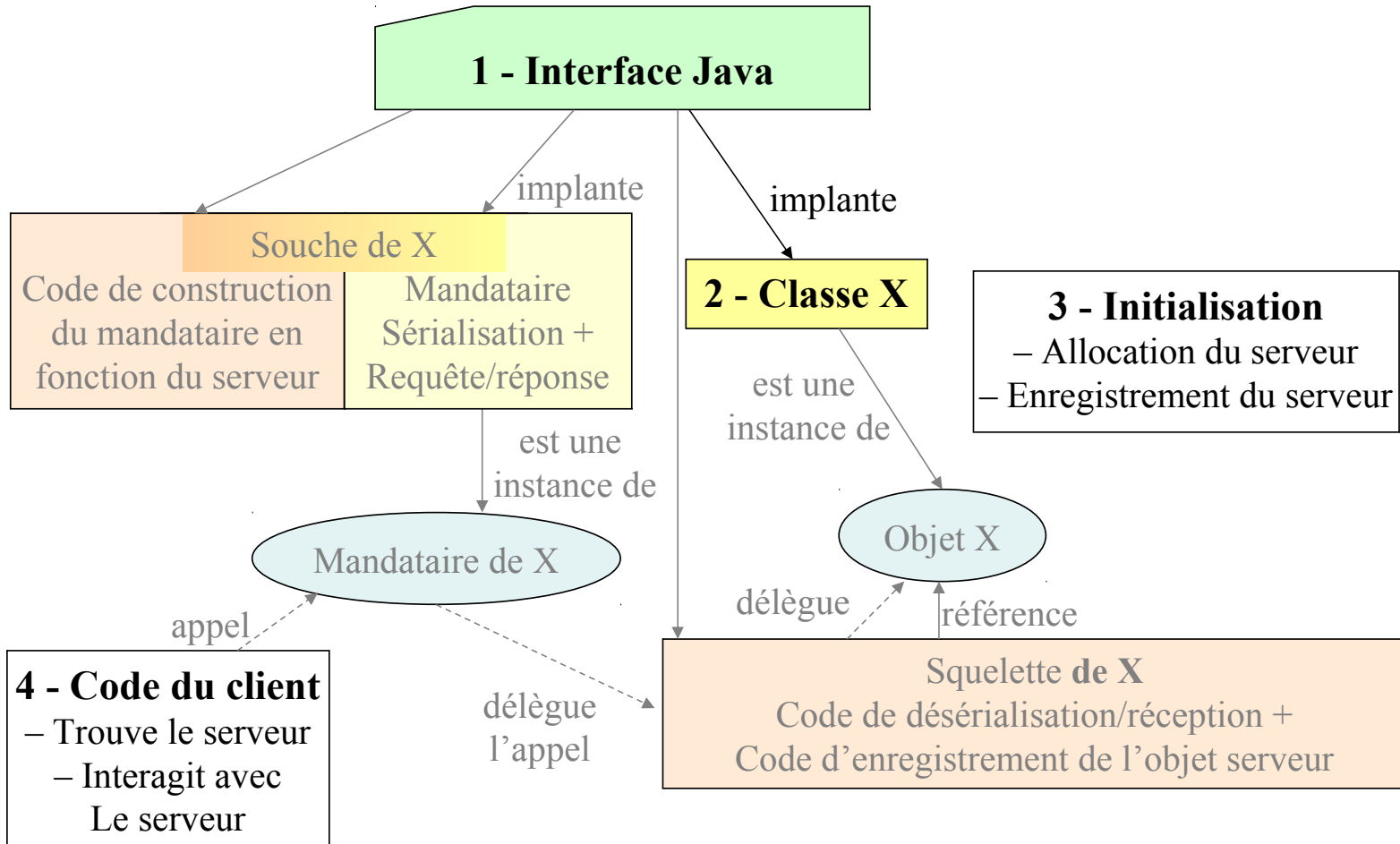
❖ Seules les méthodes de l'interface peuvent être invoquées

1. Écriture d'une interface Java
2. Écriture d'une class implantant l'interface
3. Écriture d'un programme serveur
4. Écriture du programme client



1. Déclaration des services accessibles à distance
2. Définition du code du service
3. Instanciation et enregistrement du serveur
4. Recherche et interaction du serveur

1. Caractéristiques



2. Construction d'une application RMI

1 - Définition du serveur = écriture d'une interface de service

- Interface Java normale
- Doit étendre `java.rmi.Remote`
- Toutes les méthodes doivent lever `java.rmi.RemoteException`

```
import java.rmi.Remote;
```

```
Import java.rmi.RemoteException;
```

```
public interface Compte extends Remote {  
    public String getTitulaire() throws RemoteException;  
    public float getSolde() throws RemoteException;  
}
```

2. Construction d'une application RMI

2 - Implantation du serveur = écrire une classe implantant l'interface

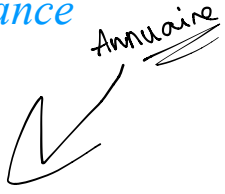
- Class Java normale
- Constructeurs doivent lever java.rmi.RemoteException;
- Si pas de constructeur, en déclarer un vide qui lève RemoteException

```
public class CompteImpl implements Compte {  
    private String propriétaire;  
    private double solde;  
  
    public CompteImpl(String propriétaire) throws RemoteException {  
        this.propriétaire = propriétaire; this.solde = 0; }  
  
    public String getTitulaire() { return propriétaire; }  
    public float getSolde(){ return solde; }  
}
```

2. Construction d'une application RMI

3 - Écriture du serveur = instantiation + enregistrement

```
public static void main(String args[]) {  
    Compte compte = new CompteImpl("Bob"); // création de l'objet serveur  
    UnicastRemoteObject.exportObject(compte, 0);  
    // enregistre compte dans RMI : on peut déjà l'appeler à distance  
  
    // trouve le service de résolution de nom de RMI  
    Registry registry = LocateRegistry.getRegistry(hostName);  
    // enregistre la référence distante dans le service de résolution  
    registry.bind("Bob", compte);  
}
```



Le programme ne s'arrête pas tant que `compte` est enregistré dans RMI
Pour désenregistrer : `UnicastRemoteObject.unexportObject(compte, false);`
`false` : attend la fin du traitement des requête, `true` : immédiat

2. Construction d'une application RMI

4 - Écriture du client : trouver compte + interagir avec lui

```
public class Client {  
    public static void main(String[] args) {  
        // trouve le service de résolution de nom de RMI qui se trouve sur hostName  
        Registry registry = LocateRegistry.getRegistry(hostName);  
        // demande un mandataire vers le compte de Bob  
        Compte compte = (Compte)registry.lookup("Bob");  
        // utilise le compte  
        System.out.println("Bob possède " + compte.getSolde() + " euros");  
    }  
}
```

↑ retourne un objet en données pures → il faut casten.

`hostName` est un nom de machine, celui sur lequel se trouve le serveur de résolution de noms

2. Construction d'une application RMI

Compilation : compiler les quatre fichiers normalement

Exécution :

- Lancer le service de résolution de nom "**rmiregistry**" sur **hostName**
Doit avoir accès au fichier classe de l'interface (Compte.class)
- Lancer le **serveur** sur n'importe quelle machine
Doit avoir accès à Compte.class, CompteImpl.class et Server.class
- Lancer le **client** sur n'importe quelle machine
Doit avoir accès à Compte.class et Client.class

Mais quand les souches et squelettes sont-ils générés?

Elles sont générées dynamiquement par la JVM depuis la version 5

Utilisation de l'API de réflexion `java.lang.reflect` côté serveur

Utilisation de la classe Proxy de `java.lang.reflect` côté client

Compatibilité ascendante : utiliser `rmic` pour générer squelettes et souches

3. Passage de paramètres

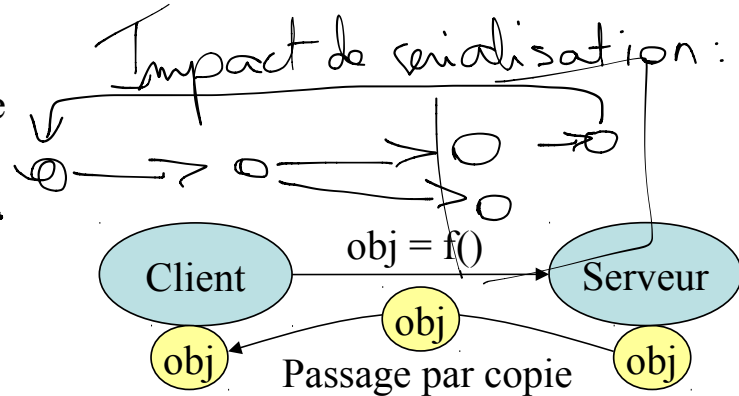
Objets Remote et objets Serializable

Java choisit automatiquement si un paramètre est passé par copie ou par référence.

Duplication \rightarrow ~~const. par copie~~ : Serialisation \Rightarrow Désérialisation

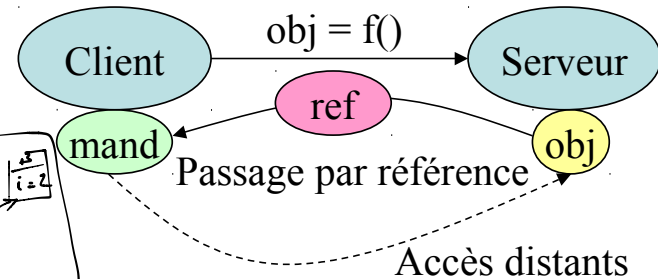
➤ Passage par copie si

- ✓ Types simples (float, int, double...)
- ✓ Objet implante `java.lang.Serializable`



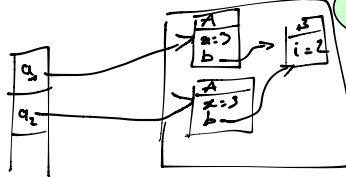
➤ Par référence si

- ✓ Objet implante `java.rmi.Remote`



Constructeur par copie : $a_2 = \text{new } A(a_1)$:

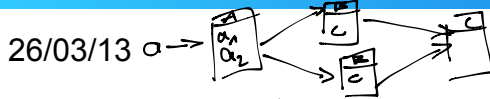
$A(A\ a)$ $\left\{ \begin{array}{l} \text{this.x} = a.x \\ \text{this.b} = a.b \rightarrow \text{new } B(a.b) \end{array} \right.$



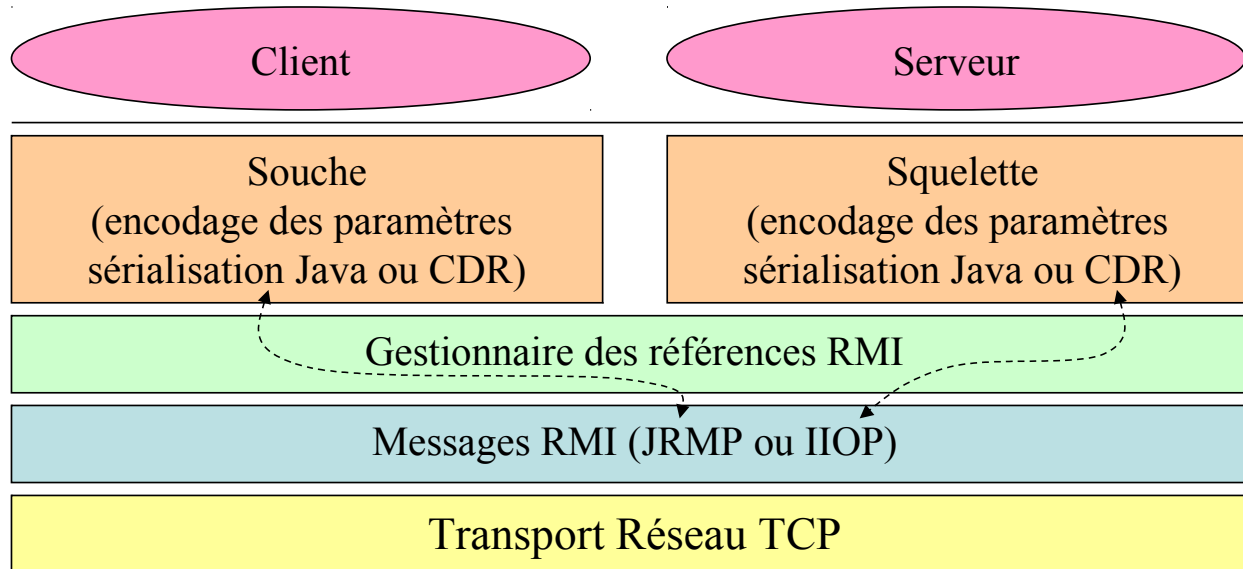
Remarque : Si un objet n'est ni Remote, ni Serializable, ni primitif une exception

MarshalException est levée

\sim "Pliage"



4. Architecture interne de RMI



- Souche/squelette : encode/décodes les paramètres des méthodes
- Gestionnaire de références : associe les mandataires (Remote) aux références distantes + ramasse-miettes réparti
- Message : définit la structure des messages, leurs identifiants et l'interaction requête/réponse
- Transport : transporte un message entre deux machines machines virtuelles Java

4. Architecture interne de RMI

Architecture prévue pour supporter différents types de liaison

- Objet distant joignable en point à point (UnicastRemoteObject)
- Objets distants joignables par diffusion

En pratique seul UnicastRemoteObject est mis en œuvre

Définition d'une référence distante RMI un Unicast

- Adresse IP de la machine hébergeant la JVM
- N° de port TCP
- Identifiant de l'objet dans le serveur (entier)

5. Services RMI

Service de résolution de noms

- Permet d'enregistrer un Remote sous un nom symbolique
 - Par défaut sur le port 1099
 - Noms "plats" (pas de hiérarchie) → Pas d'espace de nom, obligé d'utiliser des string
- Deux manières de démarrer le service
 - De façon autonome dans un shell avec l'outil `rmiregistry`
 - Dans un programme avec `static LocateRegistry.createRegistry(int port)`
- Trouver le service de résolution de noms
 - `static LocateRegistry(String host, int port)` : à distance
 - `static LocateRegistry(String host)` : à distance sur le port 1099
 - `static LocateRegistry(int port)` : localement
 - `static LocateRegistry()` : localement sur le port 1099

5. Services RMI

Remarque :

Le service de résolution de noms de RMI (rmiregistry) ne manipule que des Remote : **il enregistre des mandataires et non des copies**

L'interface du serveur de nom est

```
public interface Registry extends Remote { // accessible à distance
    void bind(String name, Remote obj); // envoie la référence distante
    Remote lookup(String name); // renvoie une référence distante
    void rebind(String name, Remote obj); // écrase l'ancien enregistrement
    void unbind(String name); // supprime l'enregistrement
    String[] list(); // liste les objets présents
}
```

↑
accessible depuis client.

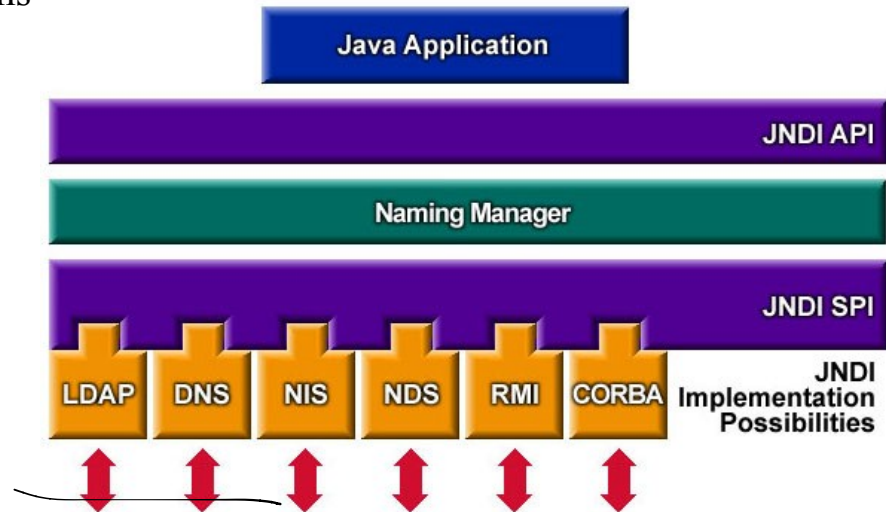
Tout serveur qui utilise le Registry est client du Registry...

5. Services RMI

Service de résolution de noms

Implantation d'un fournisseur de service de résolution de nom RMI
au dessous de **JNDI**

- JNDI : Java Naming and Directory Interface
Spécification Java masquant les différentes implantations de service de résolution de nom
- URL RMI :
`rmi://localhost:1099/obj`



5. Services RMI

Service d'activation d'objets = Thread \rightarrow mtho quelq'un pour répondre

- Permet de n'activer des objets que quand ils sont utilisés
 - Évite d'avoir des objets serveurs actifs en permanence
 - Trop coûteux si beaucoup d'objets dans une JVM
 - Rend les objets persistants
 - Enregistrés dans le système de fichier lorsqu'ils sont désactivés
- Un démon (rmid) s'occupe d'activer les objets quand ils reçoivent des requêtes
 - Les références distantes restent constantes d'une activation sur l'autre
 - ❖ Transparent pour le client
- Mise en œuvre
 - Objets doivent étendre `java.rmi.activation.Activable`
 - Un programme doit installer cette classe dans rmid (`ActivationDesc`)

5. Services RMI

```
public class Setup {  
    public void main(String args[]) {  
        // création d'un groupe d'activables (regroupés surtout pour de l'observation)  
        ActivationGroupDesc group = new ActivationGroupDesc(null, null);  
  
        // enregistre le groupe et obtient un identifiant de ce groupe  
        ActivationGroupID gid = ActivationGroup.getSystem().registerGroup(group);  
  
        // création d'une description de l'objet activable  
        // (groupe de l'objet, nom de la classe, classpath, données initiales)  
        ActivationDesc desc = new ActivationDesc(gid, "Compte", "a classpath", null);  
  
        // enregistre cet objet activable dans RMI  
        // ⇔ UnicastRemoteObject.exportObject(...)  
        Compte compte = Activable.register(desc);  
  
        // enregistre cet objet dans le rmiregistry  
        Registry registry = LocateRegistry.getRegistry(hostName);  
        registry.bind("Bob", compte); } }
```

5. Services RMI

Ramasse-miettes réparti : récupération des ressources mémoires inutilisées

- Qui ne peuvent plus être accédées localement
- Qui ne sont plus référencées à distance

Exemple :

```
public class Client {  
    Service r;  
    public void f() {  
        Remote o = r.call();  
        ...  
        ...  
    }  
    o = null;  
}
```

**f ne doit pas être détruit
sur le serveur**

```
public class Service extends Remote {  
    public Remote call();  
}  
  
public class Impl implements Service {  
    public Remote call() {  
        return new Service();  
    }  
}
```

5. Services RMI

Difficulté : environnement distribué

⇒ **référencement à distance**

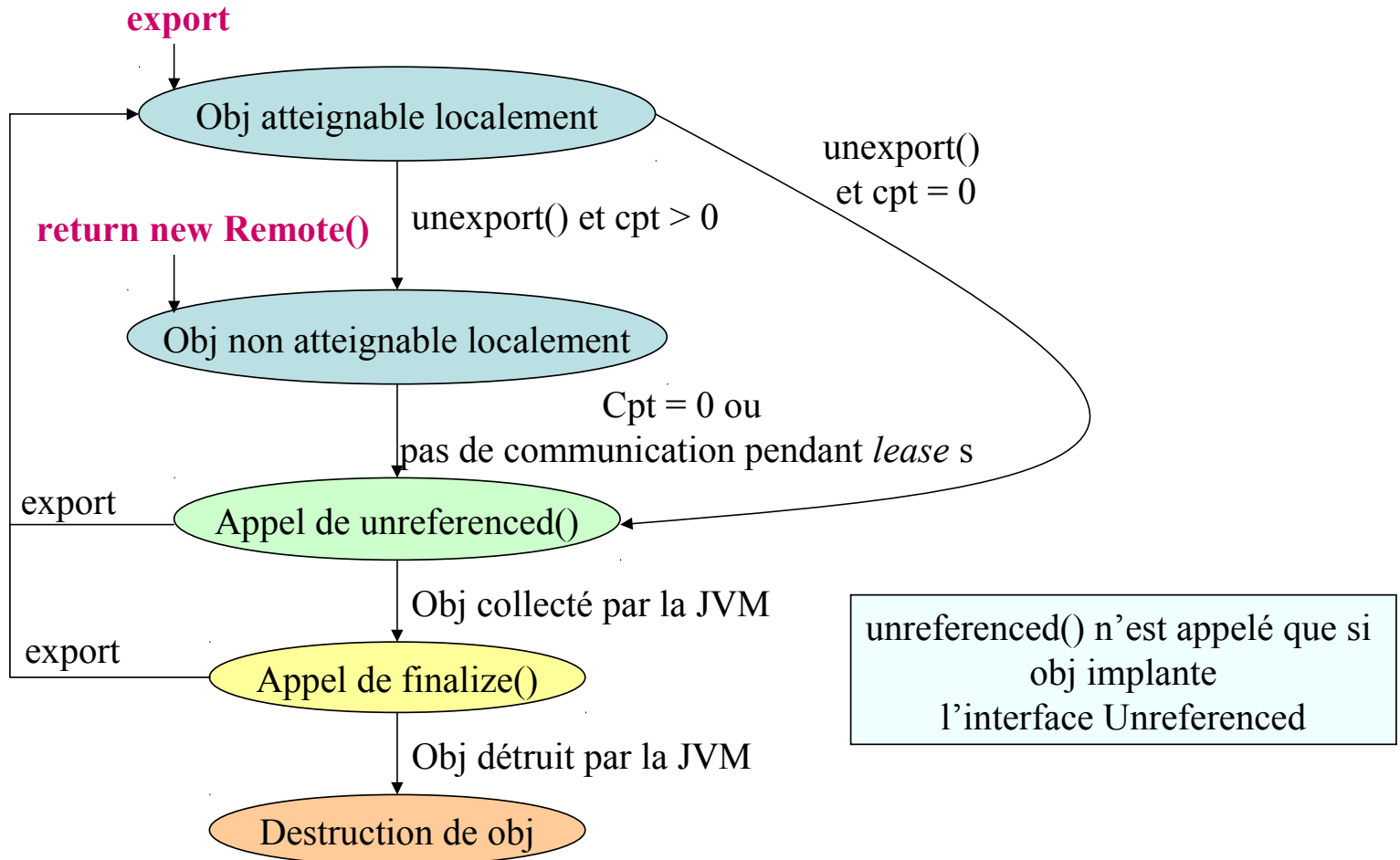
Ramasse-miettes dans une JVM : *mark-and-sweep*

- Parcourt du graphe des objets vivants
- Destruction des objets non atteints
- Insuffisant avec RMI (lorsque l'objet f n'est plus référencé localement)

Ramasse-miettes RMI : gestionnaire de référence RMI

- Compteur de référence
 - ✓ Chaque référencement distant incrémente le compteur de 1
 - ✓ Si compteur tombe à zéro, utilisation du RM de la JVM
 - Impossible de gérer les pannes, les références circulaires
A->B, B->A, mais ni A, ni B ne sont référencés ailleurs sur le serveur ou le client
- Location d'espace mémoire (bail ou *lease*, 10s par défaut)
 - ✓ Un objet serveur non référencé localement est supprimé si il n'y a pas de communication avec lui pendant la durée du bail

5. Services RMI



6. Intégration RMI et CORBA

Protocole RMI : utilisable uniquement avec des Objets Java

Ne gère pas l'hétérogénéité des langages

RMI-IIOP : RMI over IIOP

Utilisation du protocole de Corba pour communiquer

- Remplace JRMP par IIOP pour les messages
- Remplace Serialization Java par CDR pour l'encodage

Utilisation

- Objet exporté via `PortableRemoteObject.export()`
- Utilisation du CosNaming de Corba au lieu du rmiregistry
- Obligation de générer les Stub/Squelette (`rmic -iiop`)
- Pas de ramasse-miettes répartis
- Pas de surcharge de méthodes (limitation de CORBA)
- Impossible d'utiliser les SocketFactory