Objet distants et Java RMI

Jonathan Lejeune

Sorbonne Université/LIP6-INRIA

SRCS – Master 1 SAR 2019/2020

sources :

Cours précédents de Julien Sopena et Gaël Thomas

Rappel : Appel distant

Objectif des RPC

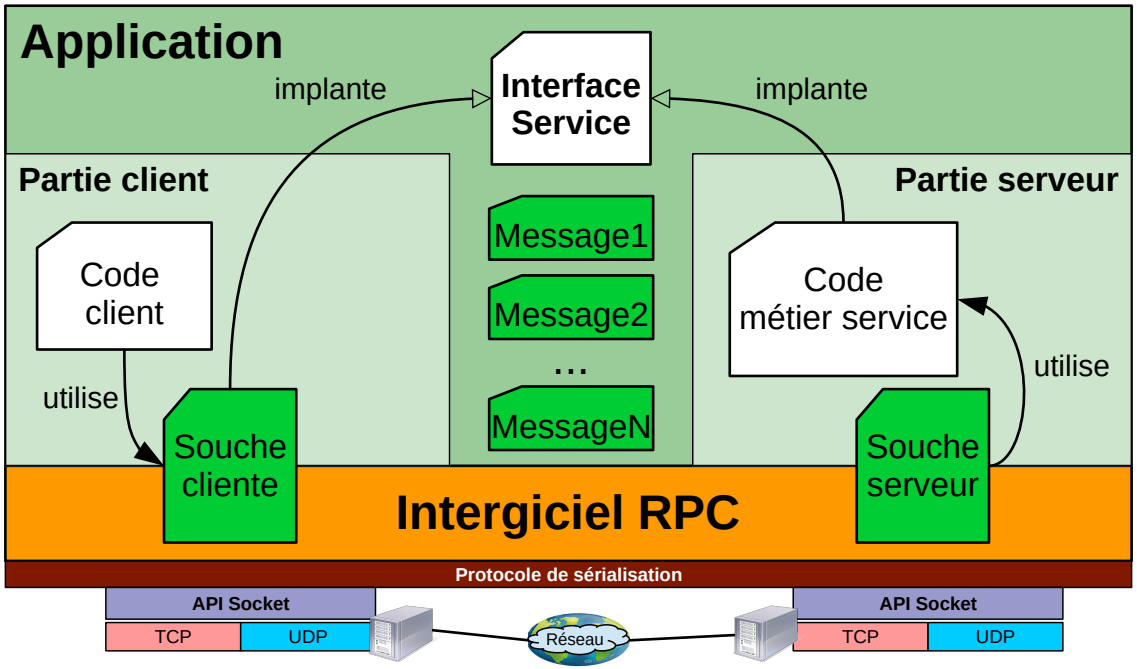
- Diminuer le travail de développement des serveurs et des clients
- Conversion d'appel de méthode en protocole requête/réponse
- Sérialisation/désérialisation des arguments/retour
- Format pivot pour les données (Protobuf, sérialisation Java, . . .)
- Plus facile pour un serveur d'offrir plusieurs services (méthodes)
- Le client appelle une méthode locale qui est déléguée au serveur
- La méthode est exécutée par le serveur
- Masquer une partie de la répartition et de l'hétérogénéité des machines

Comment ?

Génération de code de la délégation de l'appel

- Une souche pour le client
- Une souche pour le serveur (on parle aussi de squelette)

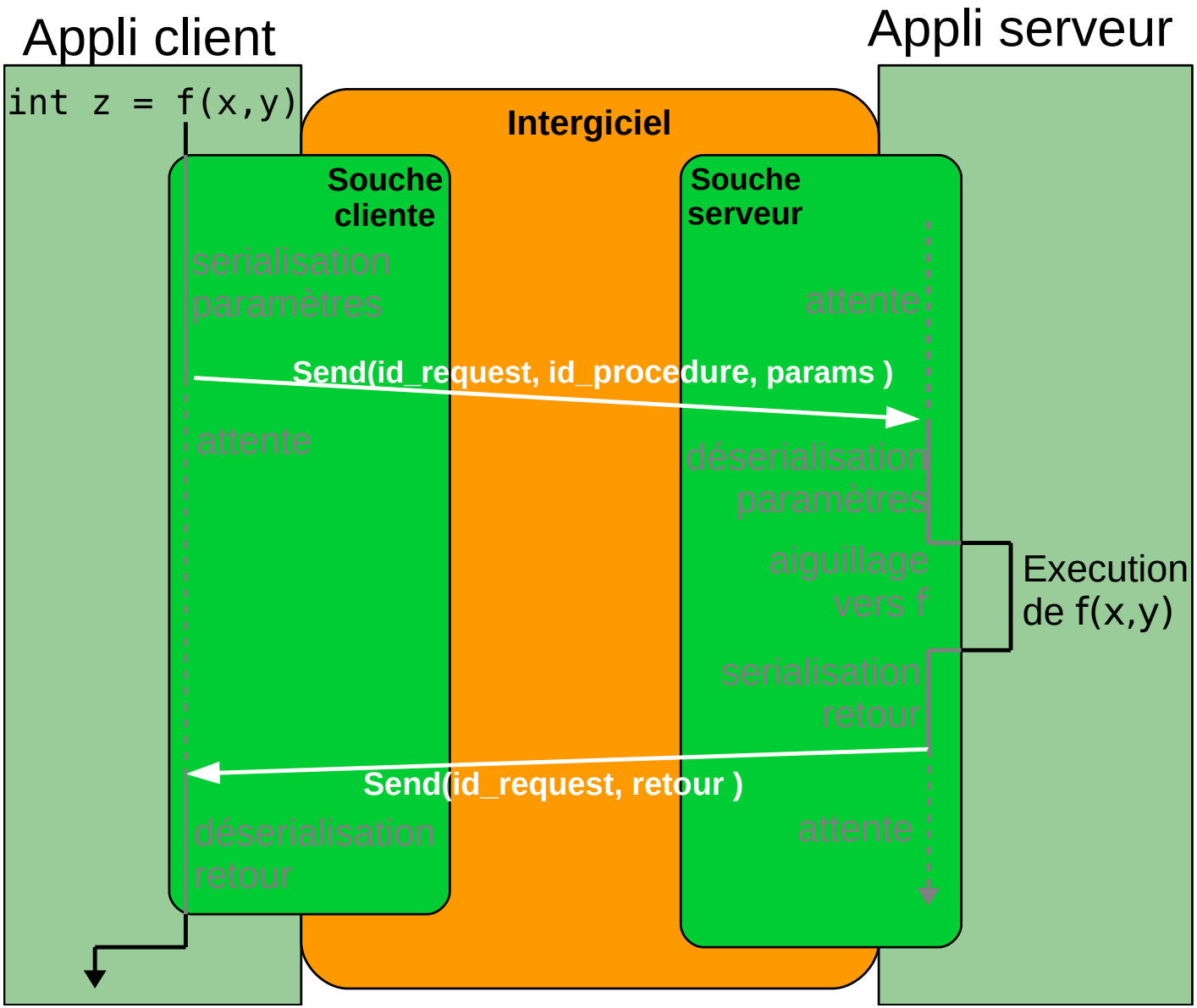
Rappel : Développement d'une application RPC



Étapes

1) Définir l'interface du service
2) Générer les souches et messages
3) Définir le code serveur
4) Définir le code client

Rappel : Déroulement d'un appel RPC synchrone



Vers une évolution du modèle des RPC

Les RPC sont inadéquats pour un langage objet

- La quasi-totalité des applications sont développées en suivant le paradigme objet
- L'appel de procédure distante ne prend pas en compte l'aspect objet
- Le serveur et les clients sont développés avec des objets mais doivent changer de paradigme si ils doivent communiquer

  $\Rightarrow$ Perte de transparence pour le développeur

Besoin de faire évoluer le paradigme vers l'objet

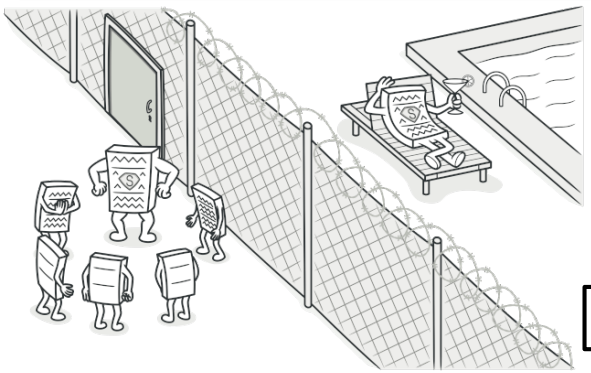
- Le serveur fournit un objet distant avec des méthodes et des champs
- Le client utilise un objet représentant l'objet serveur localement

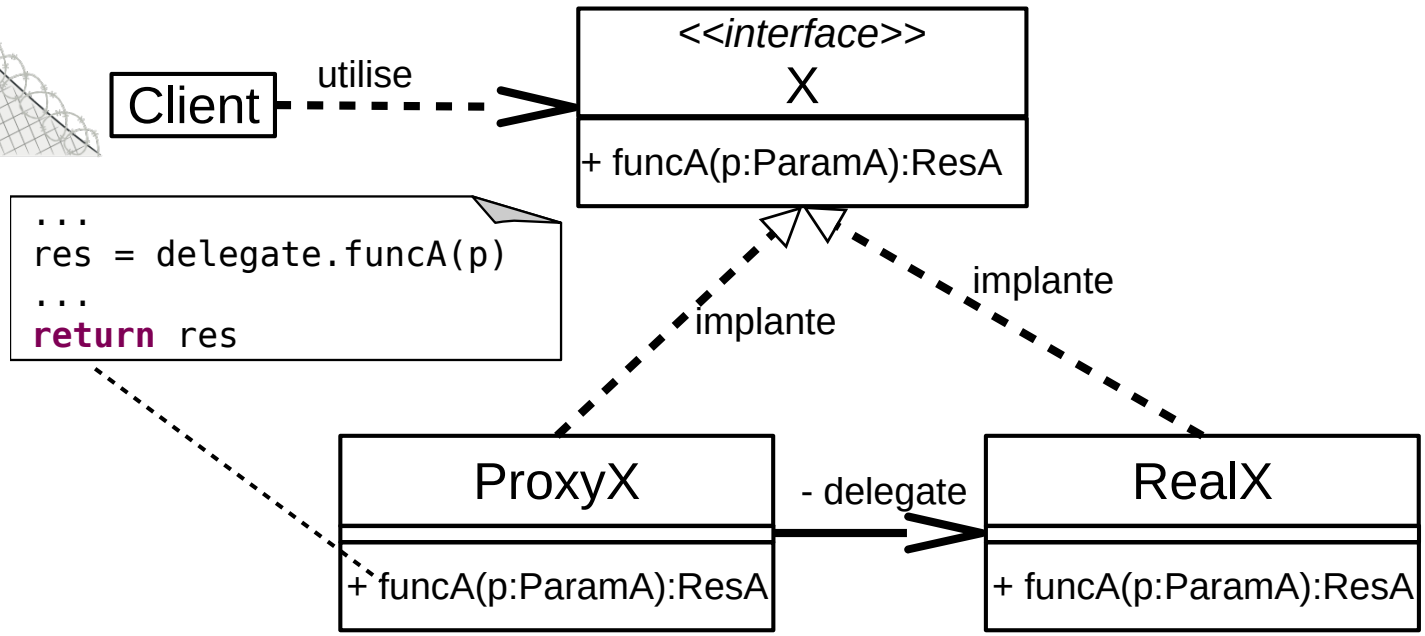
  $\Rightarrow$ Notion de proxy

Le Design Pattern Proxy

Définition

Utilisation d'un objet mandataire dialoguant avec un objet principal et qui offrent tous les deux la même interface.





Le Design Pattern Proxy

Intérêt pour la programmation répartie

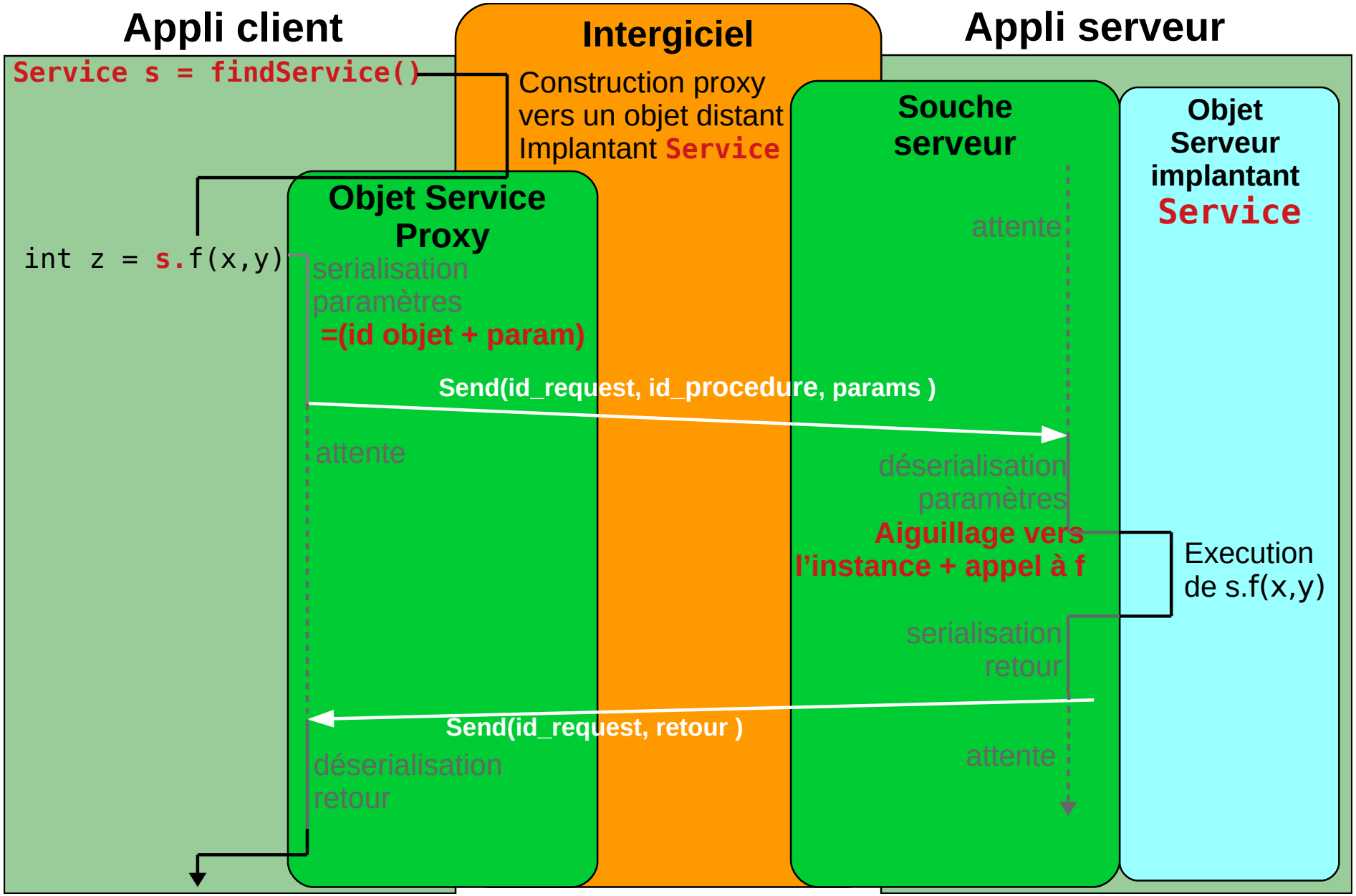
- Adapter le code de l'objet principal de manière transparente pour l'utilisateur
- Le client utilise l'objet proxy à la place de l'objet principal

  Transparence pour le client

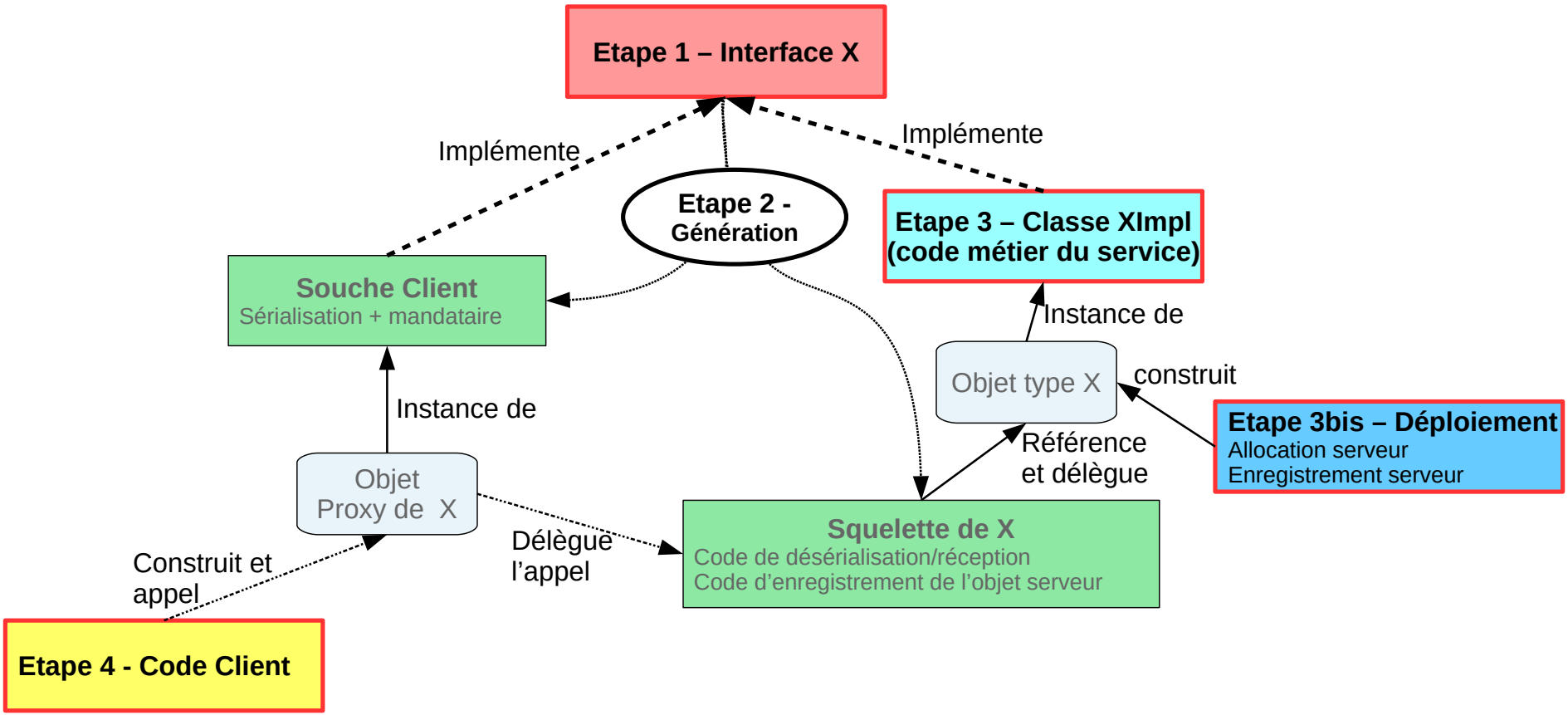
Application aux objets distants

- Le proxy implante la communication réseau pour dialoguer avec l'objet principal
- L'objet proxy = souche cliente

Appel de méthode synchrone sur objet distant



Synthèse des objets distants

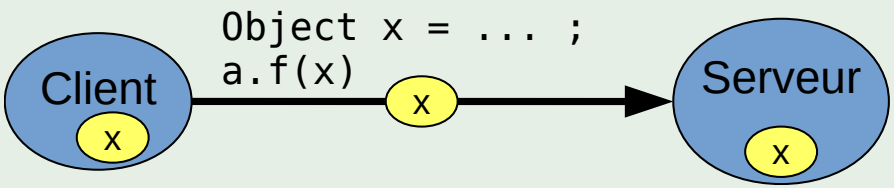


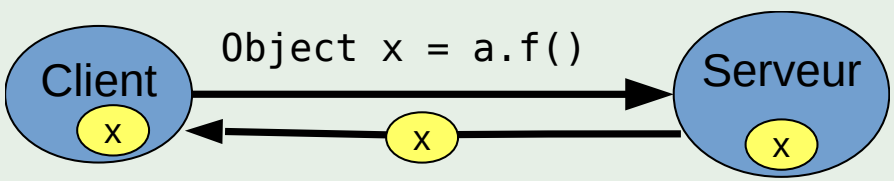
Transmission d'informations par copie

Principes

- L'objet est copié lors de l'appel ou lors du retour de l'appel
- La copie est différente de l'original

Passage de paramètres par copie	Passage du retour par copie





Avantages/inconvénients

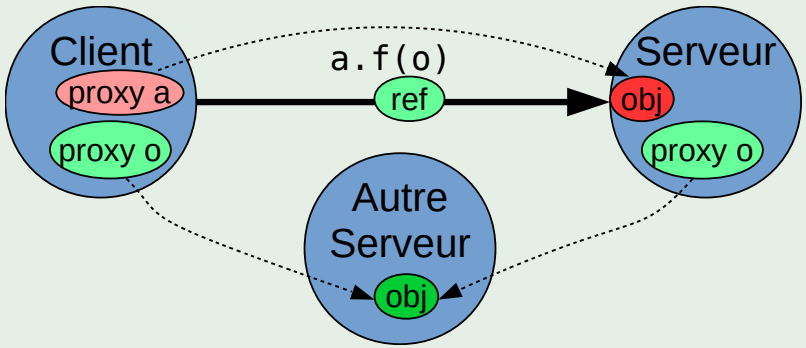
- ✓ Une copie locale évite les accès distants qui sont coûteux
- ✗ Les modifications sur la copie ne sont pas répercutées sur l'original
- ✗ Les copies ne sont pas forcément à jour si l'original est modifié

Transmission d'informations par référence

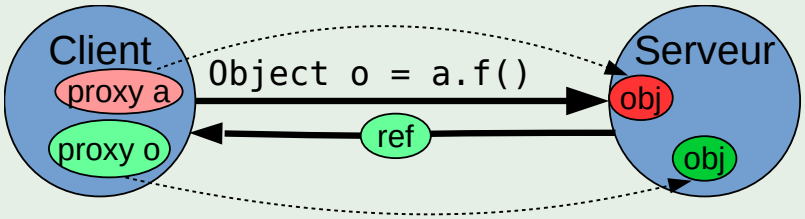
Principes

- Une référence distante est envoyée et un mandataire est construit lors de l'appel **ou** lors du retour
- **Un objet passé par référence = objet réparti**

Passage de paramètres par référence



Passage du retour par référence



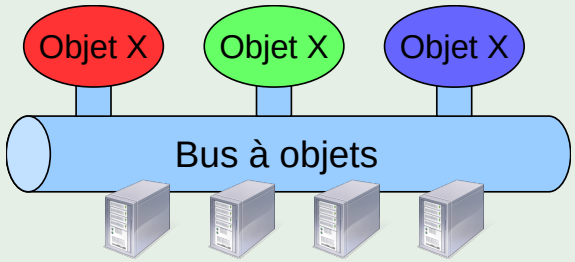
Avantages/inconvénients

- ✓ Toujours accès à la dernière version
- ✓ Modification partagée par tous les utilisateurs du serveur
- ✗ Accès distant à chaque accès à l'objet

Bus à objets

Définition

- Bus logiciel qui permet un vue abstraite d'un système à objets répartis
- Constitué de l'ensemble des souches des objets du système
- Composant de l'intergiciel qui propose des services systèmes



Exemple de services systèmes proposés

- Résolution de nom : permet d'associer un nom à un objet réparti
- Persistance : permet de sauvegarder l'état d'un objet
- Transaction
- Ramasse-miette réparti

La notion de référence distante

Définition

Une référence distante identifie de manière unique un objet sur un bus à objets

Contenu d'une référence distante

- Adresse de la machine hébergeant l'objet
- Port d'écoute du processus du bus à objets
- Identifiant de l'objet au sein du serveur

Remarque

Un objet proxy peut être vu comme la référence distante :

- Communication par référence = copie du proxy
- **Attention** : référence locale du mandataire $\neq$ référence distante

Trouver une référence distante d'un objet

Une mauvaise solution : via un fichier commun

- Noter la référence distante dans le fichier
  ⇒ Dépendant de la localisation physique du serveur
  ⇒ L'identifiant de l'objet peut changer à chaque exécution
- Le client charge le fichier
  ⇒ impossible car le serveur et le client doivent s'échanger des info avant de se connaître

Solution utilisée : **annuaire** (ou serveur de noms)

- L'annuaire = service qui associe un nom et une réf distante
- Le serveur enregistre son/ses objets dans l'annuaire
- Le client interroge l'annuaire et récupère la référence distante
- Peut être accédé :
  - soit via une référence distante connue a priori (exemple : rmiregistry)
  - soit via une API standard avec une URL (exemple : JNDI)

Java **R**emote **M**ethod **I**nvocation

Qu'est-ce que c'est ?

Un intergiciel objet Java d'invocations distantes :

- Appelant et appelé peuvent être dans des JVM différentes
- Les JVMs peuvent être sur des machines différentes si elles peuvent communiquer par le réseau.

Java Remote Method Invocation

Avantages/inconvénients

- ✓ Déploiement et installation très simples
  - ⇒ tout est fourni avec le JDK et déjà implanté dans la JVM
- ✓ Gestion dynamique des souches (depuis le JDK 5)
  - ⇒ pas de phase de génération manuelle
- ✓ Protocole de communication binaire
  - ⇒ utilisation de la sérialisation Java
- ✗ Pas d'hétérogénéité des langages
  - ⇒ Utilisable uniquement avec des objets Java
- ✗ Les appels sont forcément synchrones
  - ⇒ Les appels asynchrones doivent être gérés manuellement

Développement en Java RMI : étape 1

Déclaration de l'interface du service

- Interface Java normale qui étend `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;
}
```

Développement en Java RMI : étape 2

Génération des souches

- Commande `rmic` à partir du fichier *.class* de l'implantation du service coté serveur pour générer une souche cliente

  ⇒**Déprécié**

- Depuis le JDK 5, les souches sont générées dynamiquement :
  - Utilisation de l'API de réflexion Java côté serveur
  - Utilisation d'un sous-type de `java.lang.reflect.Proxy`

Génération des messages

Tout objet héritant de l'interface `java.io.Serializable` est un message.

Pas d'étape de génération manuelle en Java RMI

Développement en Java RMI : étape 3

Implanter le code métier du service

- Classe Java normale implantant l'interface définie en étape 1
- Les constructeurs doivent lever `java.rmi.RemoteException`
- Si pas de constructeur, en déclarer un vide qui lève RemoteException

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

Développement en Java RMI : étape 3 bis

Implanter le code de déploiement du service

- Instancier l'objet normalement
- Déclarer l'objet comme étant un objet distant et produire sa référence avec la méthode

  `UnicastRemoteObject.exportObject(Remote obj, int port)`

  $\Rightarrow$ **Une JVM ne s'arrête pas tant qu'il existe des objets distants déployés**
- Pour désenregistrer un objet distant, appeler la méthode

  `UnicastRemoteObject.unexportObject(Remote r, boolean force);`

```
public class Serveur{
 public static void main(String args[]){
   Compte compte = new CompteImpl("Bob");
   UnicastRemoteObject.exportObject(compte, 0);
   //0 -> n'importe quel port libre sur la machine hote
   //le programme continue tant que pas unexportObject
 }
}
```

Développement en Java RMI : Annuaire

Interface `java.rmi.registry.Registry`

- Étend `Remote` ⇒ l'annuaire est un service RMI distant
- Port d'écoute par défaut = 1099
- Noms "plats" (pas de hiérarchie)

```
public interface Registry extends Remote {
//rechercher un enregistrement
  public Remote lookup(String name) throws ...
//ajouter un nouvel enregistrement
  public void bind(String name, Remote obj) throws ...
//annuler un enregistrement
  public void unbind(String name) throws ...
//remplacer un enregistrement existant
  public void rebind(String name, Remote obj) throws ...
//obtenir la liste des enregistrements
  public String[] list() throws ...
}
```

Développement en Java RMI : Annuaire

Démarrer le Registry

- Via un shell avec la commande `rmiregistry`
- Dans un programme java : `LocateRegistry.createRegistry(int port)`
  $\Rightarrow$ la JVM devient le serveur hébergeant le registry

Attention

Le classpath de la JVM hébergeant le registry doit référencer les fichiers *.class* des interfaces que l'annuaire est censé référencer (pour les proxy)

sinon `ClassNotFoundException`

Obtenir une référence sur le Registry

- `LocateRegistry.getRegistry(String host, int port)` : à distance
- `LocateRegistry.getRegistry(String host)` : à distance sur le port 1099
- `LocateRegistry.getRegistry(int port)` : localement
- `LocateRegistry.getRegistry()` : localement sur le port 1099

Développement en Java RMI : Annuaire et étape 3 bis

```
public class Serveur{
 public static void main(String args[]){
   Compte compte = new CompteImpl("Bob");
   UnicastRemoteObject.exportObject(compte, 0);

   Registry registry = LocateRegistry.getRegistry(
       hostregistry);
   registry.rebind("LeCompte", compte);
 }
}
```

Attention

L'annuaire stocke en général des objets bootstrap

$\Rightarrow$ **son utilisation ne doit pas être systématique pour tous les objets distants que l'on déploie**

Développement en Java RMI : étape 4

Récupérer une référence distante sur un objet

- Soit via une référence distante sur le registry
- Soit via JNDI
- Soit à partir d'une méthode d'un objet distant qui renvoie une référence distante

Via une référence distante sur le registry

```
Registry registry = LocateRegistry.getRegistry(host, port);
Compte cpt = (Compte) registry.lookup("LeCompte");
```

Via JNDI

```
Context ctx = new InitialContext();
String url = "rmi://"+host+":"+port+"/LeCompte";
Compte cpt = (Compte) ctx.lookup(url);
```

Transmission d'informations

Par copie ou par référence ?

- Java choisit automatiquement le mode de transmission :
  - **par référence** : tout objet implantant `java.rmi.Remote` **et** exporté
  - **par copie** : tout objet `Serializable` et type primitif

Si aucun des deux cas

Une exception `MarshalException` est levée

Cas des objets à la fois `Remote` et `Serializable`

- Si l'objet est exporté alors transmission par référence
- Si l'objet n'est pas exporté alors transmission par copie

Java RMI et Thread

Thread et objet distant

- Dans la spécification : aucune garantie sur la correspondance entre thread et invocation
- En pratique dans openJDK :
  - **un thread dédié par invocation distante**
  - **pas de création de thread si l'invocation se fait dans la même JVM**

Conséquence

Les appels de méthode d'un objet réparti ne sont pas thread-safe :

$\Rightarrow$ c'est au programmeur de gérer les accès concurrents

Synchronisation indispensable

Appel asynchrone et semi-synchrone

Rappel

Toutes les invocations RMI se font de manière synchrone

Solutions pour les appels asynchrone et semi-synchrone

- Utilisation d'un thread dédié à l'appel qui recevra le retour
- Cas du semi-synchrone :
  - faire un `join` sur le thread
  - stocker le résultat dans une variable globale (attribut)

Asynchrone

```
Compte cpt = ...;
new Thread(()->{
   cpt.setTItulaire("Jean");
}).start();
```

Semi-synchrone

```
String tmp;
public void f(){
   Compte cpt = ...;
   Thread t = new Thread(()->{
      tmp = cpt.getTitulaire();
   })
   t.start();
   ...
   t.join();
}
```

Piège de l'imbrication d'appels distants

```
public class Aimpl implements A {//A extends Remote
 public synchronized void f(B b) throws RemoteException{
    b.h(this);
 }
 public synchronized void g()throws RemoteException{
    System.out.println("Bonjour");
 }
}

public class Bimpl implements B{//B extends Remote
  public synchronized void h(A a)throws RemoteException{
    a.g();
  }
}
```

Que se passe-t-il lors de l'appel distant `a.f(b)` ?

- Si `a` et `b` se trouvent dans la même JVM : tout va bien

  $\Rightarrow$ Même thread et réentrance des verrous java
- Si `a` et `b` se trouvent dans deux JVM différentes : deadlock

  $\Rightarrow$ 1 thread pour `f`, 1 thread pour `h`, 1 thread pour `g` qui reste bloqué à cause du thread de `f` qui a gardé le verrou

Service RMI d'activation d'objets

Qu'est ce que c'est ?

Service qui 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

Le 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.Activatable`
- Un programme doit installer cette classe dans rmid (ActivationDesc)

Service RMI d'activation d'objets : exemple

```
public class Setup {
  public void main(String args[]) {
   //création d'un groupe d'activables
   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 activatable dans RMI
   // équivalent à UnicastRemoteObject.exportObject(...)
   Compte compte = Activatable.register(desc);
   // enregistre cet objet dans le rmiregistry
   Registry registry = LocateRegistry.getRegistry(hostName );
   registry.bind("Bob", compte);
  }
}
```

Service RMI de Garbage Collection

Garbage Collector (Rappel)

- Tache de fond (démon) de la JVM chargé de libérer la mémoire des objets qui ne sont plus atteignables
- Algorithme *mark-and-sweep* : parcours du graphe d'objets et suppression des objets non atteints
- Le GC appel la méthode `finalize()` de la classe `Object` lorsque l'instance est en phase de se faire collectée

GC classique inadapté pour les objets distants

Un objet peut ne plus être référencé localement mais toujours l'être à distance.

Comment gérer le fait qu'un objet puisse être référencé à distance dans une autre JVM ?

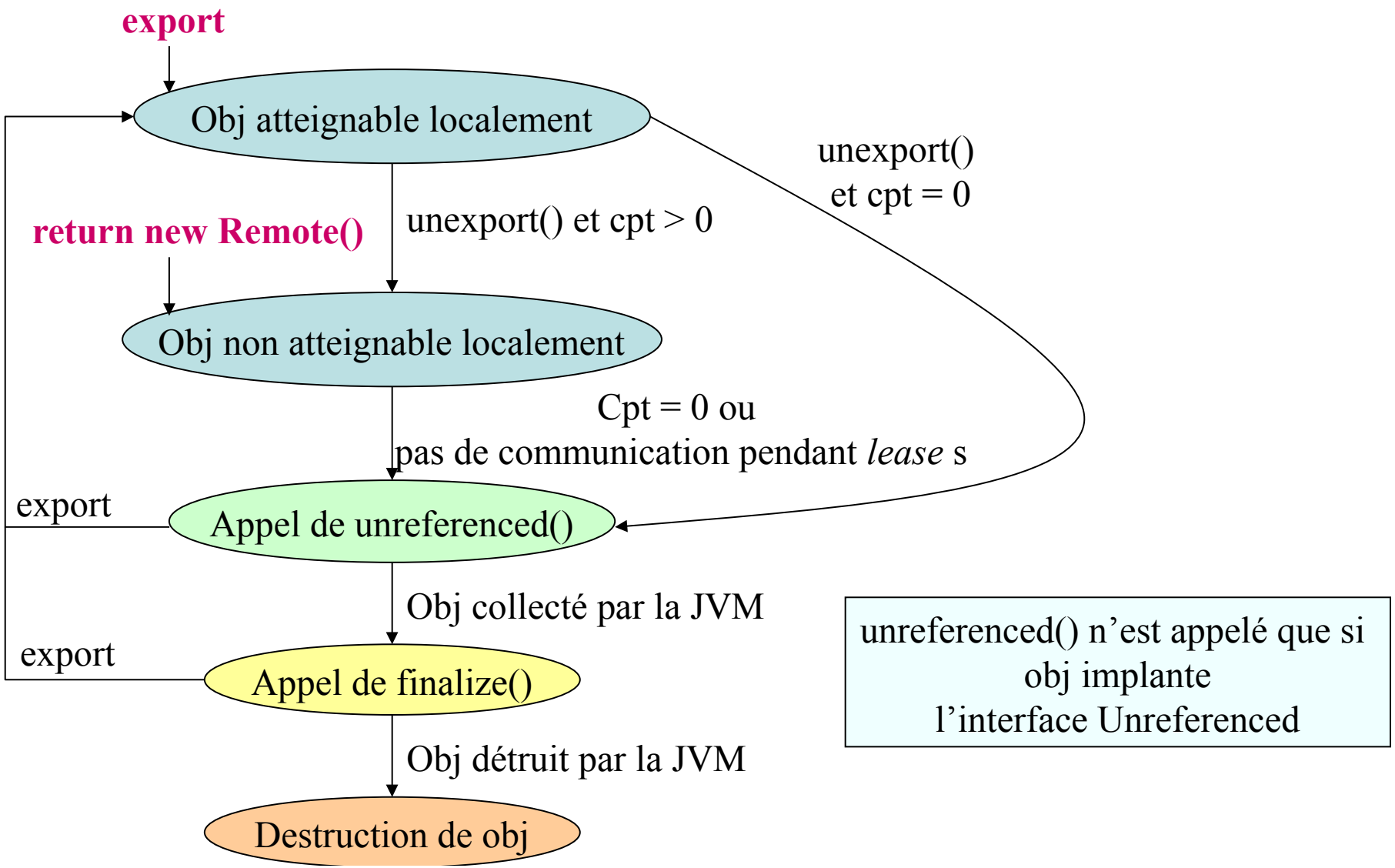
Service RMI de Garbage Collection

Fonctionnement du GC de RMI

Chaque objet maintient un compteur de références distantes

- incrémentation dès la création d'un nouveau proxy
- décrémentation dès qu'un proxy a été "garbage collecté" sur sa JVM
- si l'objet n'est plus exporté :
  - si compteur à zéro ou si pas de communication pendant `lease` secondes par un proxy
    $\Rightarrow$ objet collecté et supprimé par le gc de la JVM serveur

Service RMI de Garbage Collection



unreferenced() n'est appelé que si obj implante l'interface Unreferenced