

3) RMI avancé

Répartition des classes

- ▶ RMI distingue deux types d'objets
 - ▶ Ceux accessibles à distance
 - ▶ Les autres
- ▶ Ils sont souvent sur des machines différentes (un client et un serveur)
- ▶ Comment classes sont (initialement) réparties (+-placées "en dur")
 - ▶ Côté Client:
 - ▶ Implémentation du client
 - ▶ Interface distante avec toutes les classes nécessaires pour décrire les méthodes de cette interface
 - ▶ Classe du Stub (objet stub, lui, est téléchargé auprès du registry par exemple, ou suite retour de méthode) **si pas générée à la volée**
 - ▶ Côté Serveur:
 - ▶ Interface Distante avec toutes les classes nécessaires pour décrire les méthodes de cette interface,
 - ▶ Implémentation du serveur avec les éventuelles sous-classes nécessaires (sous classes pour les paramètres des méthodes remote)

Téléchargement de classes

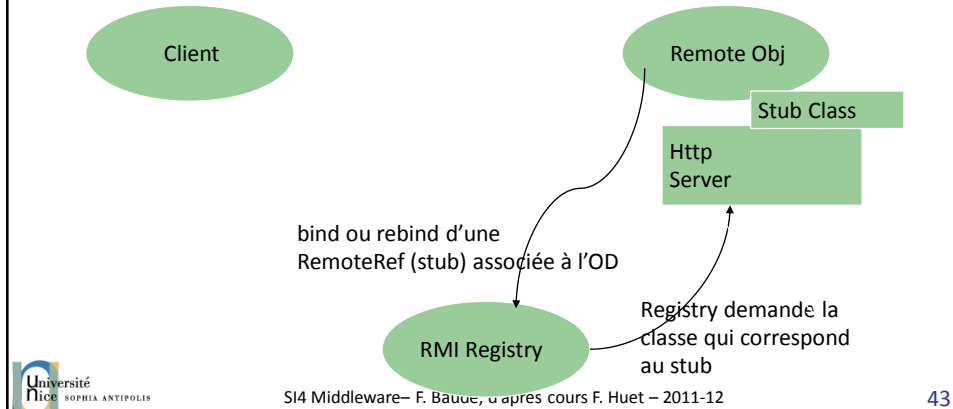
- ▶ Dans un monde parfait
 - ▶ Les classes sont distribuées
 - ▶ Rien ne change, tout est connu
- ▶ En pratique
 - ▶ Les classes sont *plus ou moins bien* distribuées là où on pense qu'on en aura besoin
 - ▶ Certains ordinateurs n'ont pas les classes nécessaires
 - ▶ Ex: Appel d'une méthode distante avec en paramètre une sous classe de la classe déclarée dans la signature distante
 - ▶ Le serveur doit télécharger le .class dispo côté machine client pour connaître cette sous-classe
- ▶ Solution: pouvoir télécharger les classes manquantes

Téléchargement de classes

- ▶ Mécanisme fourni par RMI
- ▶ Utilise HTTP
 - ▶ Permet de passer tous les firewalls
 - ▶ Mais nécessite un serveur HTTP, localisé où on veut sur le réseau ☺
- ▶ Principe:
 - ▶ Les flots de sérialisation sont annotés avec des *codebase*
 - ▶ Ils indiquent où peut être téléchargée une classe si nécessaire
 - ▶ Lors de la désérialisation, si une classe manque, le serveur HTTP indiqué par le codebase est contacté
 - ▶ Si la classe est disponible, le programme continue, sinon, `ClassNotFoundException`

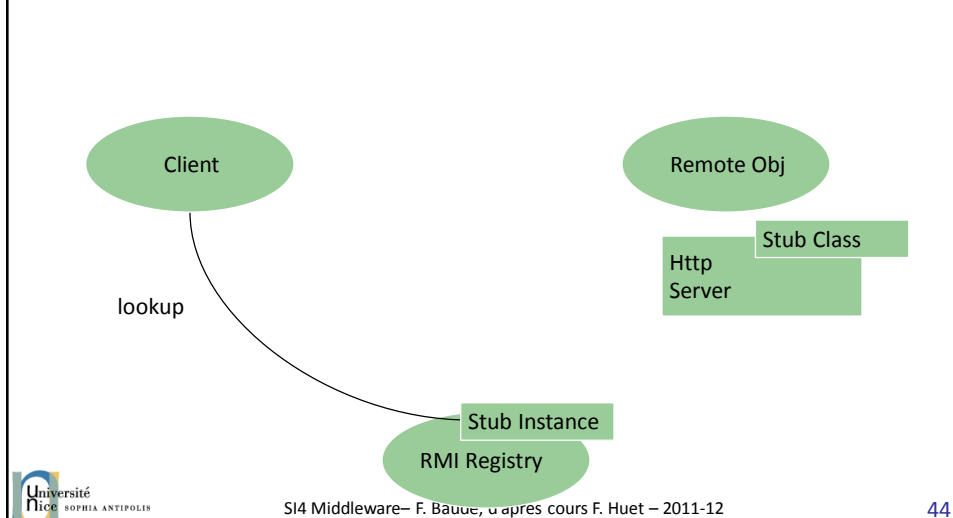
Exemple : principe

- ▶ Déploiement minimal : dans ce cas (et sous Hyp. stub créé par rmic et pas de manière automatique)
 - ▶ Le stub n'est pas disponible pour le registry ou pour le client
 - ▶ Seul le remote object possède la classe de l'objet stub dans son classpath

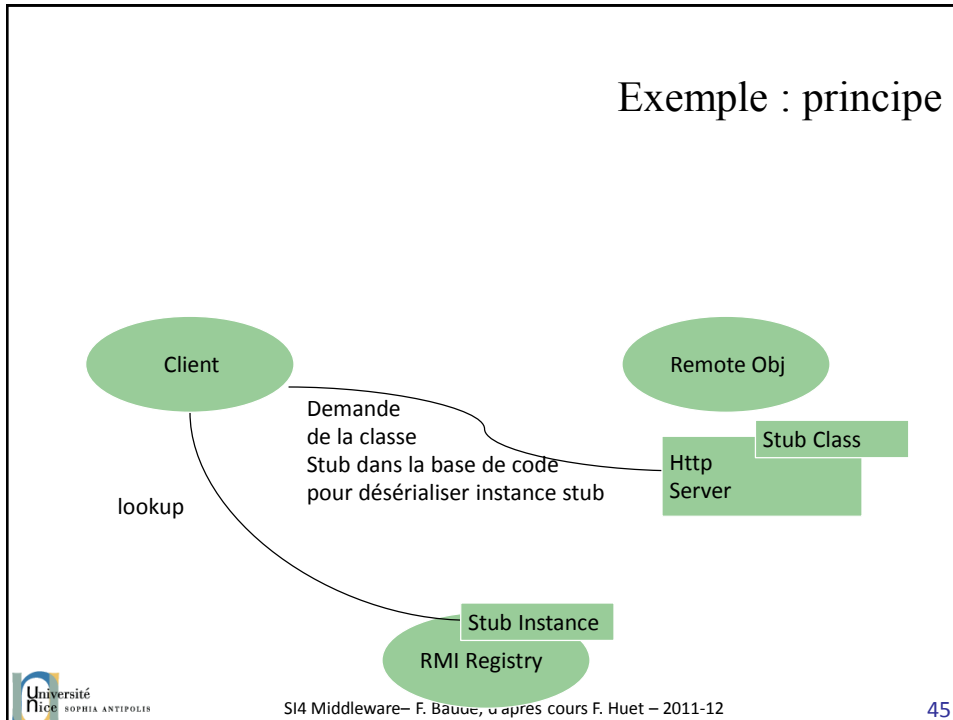


Exemple : principe

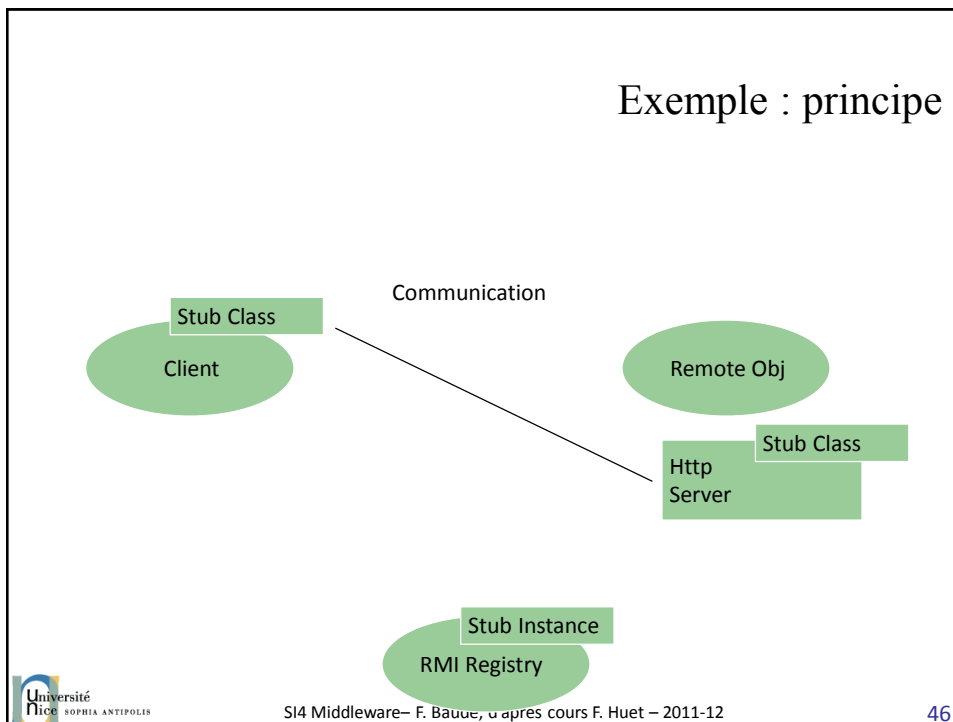
- ▶ Quand il contacte le registry, le client doit en plus télécharger la classe correspondant au stub : où est-elle ?



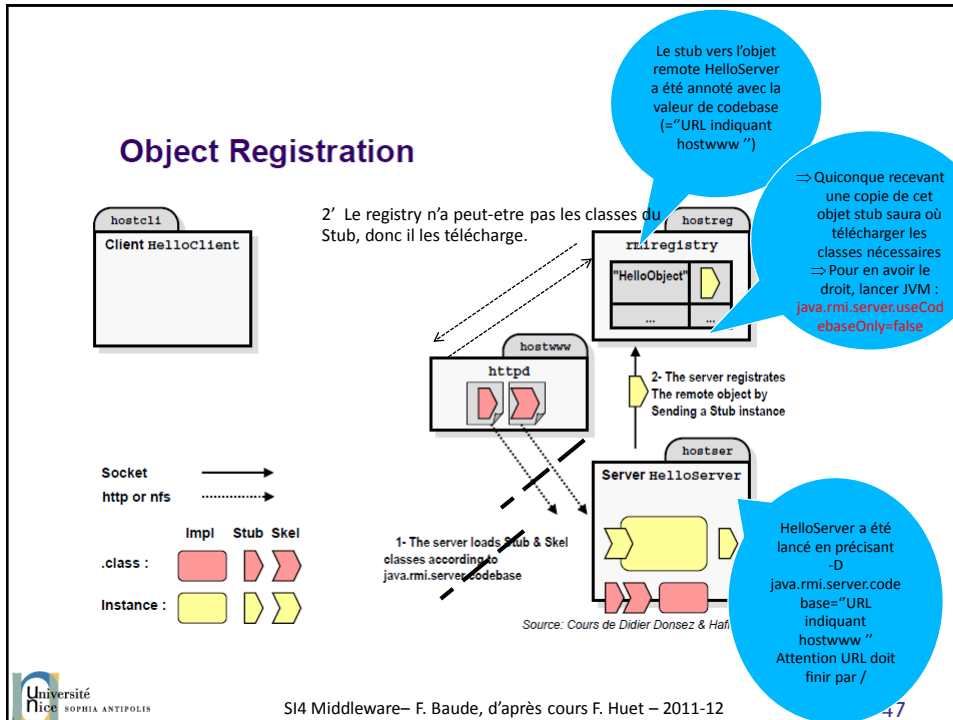
Exemple : principe



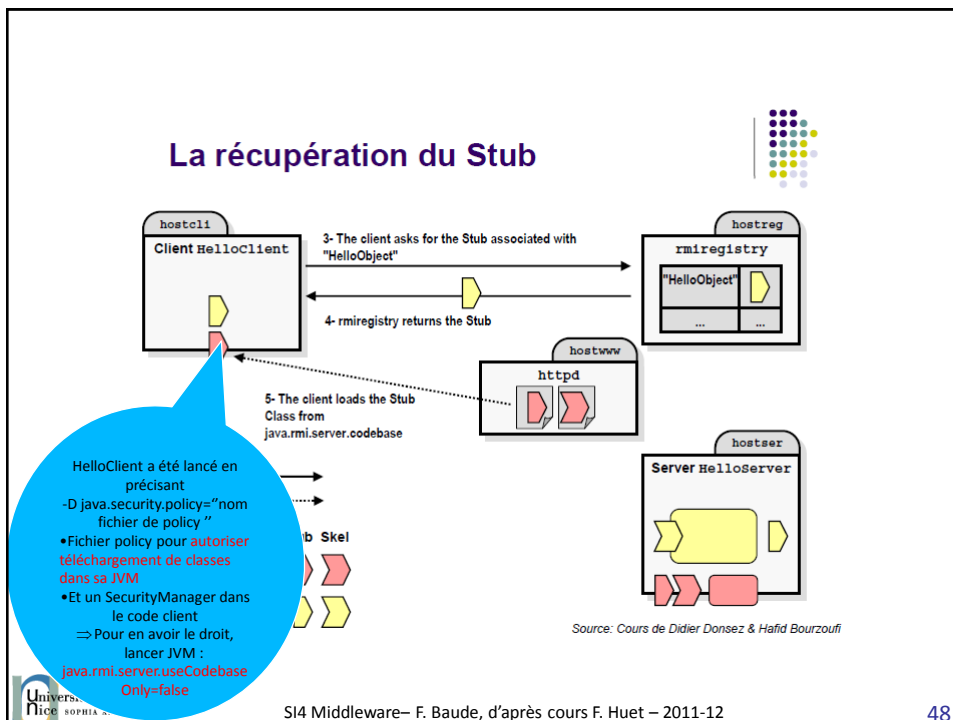
Exemple : principe



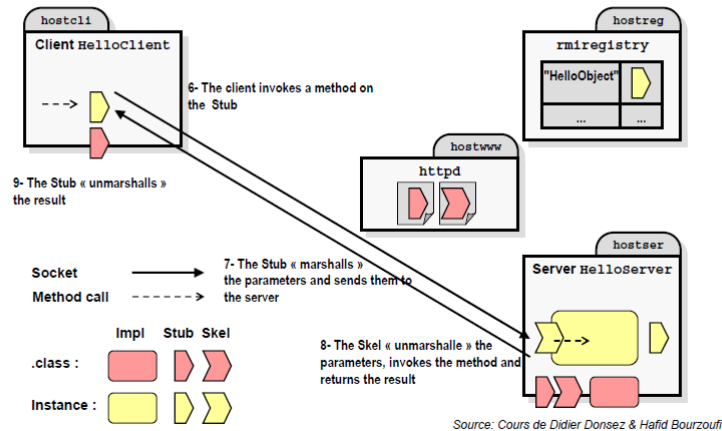
Object Registration



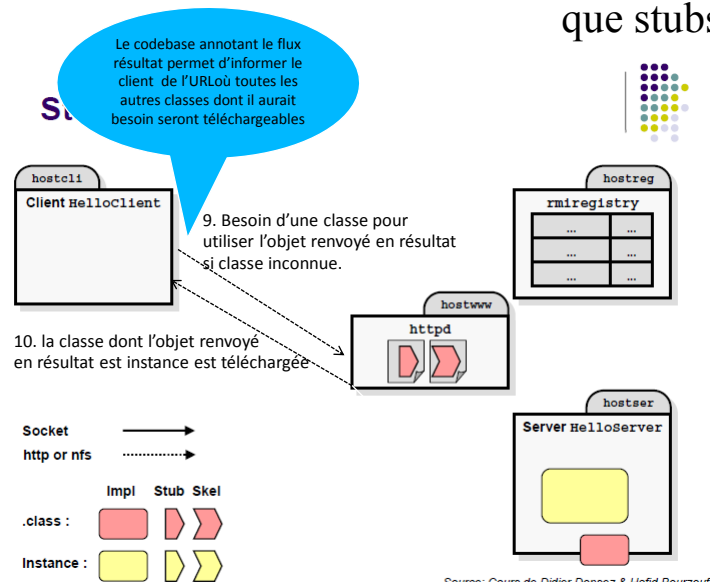
La r cup ration du Stub



Remote Method Invocation



...ayant besoin du téléchargement d'autres classes que stubs RMI



Bilan téléchargement dynamique de classes auprès de serveur web

- ▶ Lancement d'une JVM avec l'option de JVM
`java.rmi.server.codebase=""http://hostWWW:portWWW/"`
 - ▶ Tout flux qu'elle sérialise est annoté avec <http://hostWWW:portWWW/>
- ▶ Tout flux reçu pour lequel une JVM n'a pas connaissance des .class pour réussir la désérialisation
 - ▶ Déclenche une communication avec <http://hostWWW:portWWW/>
 - ▶ pour télécharger les .class, d'une autre base de code que CLASSPATH
 - ▶ Autorisation de le faire : la JVM doit avoir été lancée avec
 - ▶ Option `java.rmi.server.useCodebaseOnly=false`
 - ▶ un Security Manager, contrôlé par un fichier de policy, donnant permission d'ouvrir connection HTTP vers hostWWW:portWWW
- ▶ Le rmiregistry peut être lancé sans aucun CLASSPATH
 - ▶ N'importe quel stub sérialisé reçu de n'importe quelle JVM serveur est reconstruit via téléchargement HTTP auprès du bon serveur web indiqué
 - ▶ Idem pour la JVM cliente: peut interagir avec plusieurs JVM serveur, donc ne pas la lancer avec une URL de codebase prédéfinie

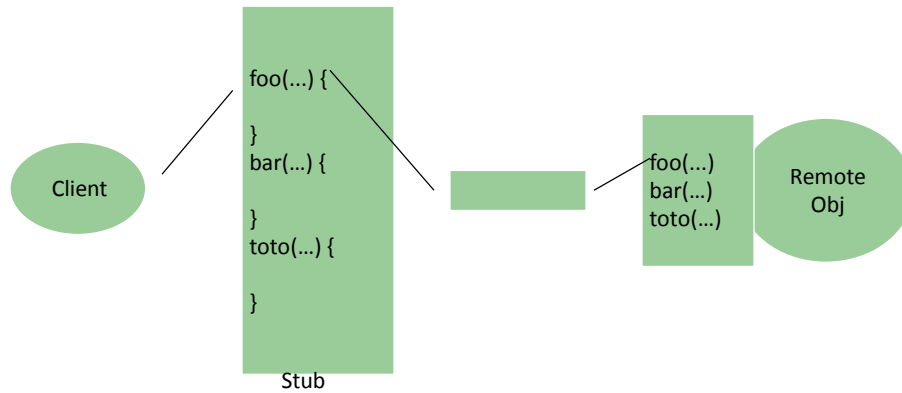


Le Stub

- ▶ Le rôle du stub est de se faire passer pour l'objet distant
 - ▶ Il implémente l'interface distante
 - ▶ Exemple de ce qu'il affiche si on invoque toString():
`Proxy[HelloWorld,RemoteObjectInvocationHandler[UnicastRef [liveRef: [endpoint:[193.51.208.206:10003](remote),objID:[75300dd9:145187ca75d:-7fff,2812291573724520304]]]]]`
- ▶ Il doit aussi convertir l'appel de méthode en flot
 - ▶ Facile pour les paramètres
 - ▶ Pour la méthode, il suffit de la coder sur quelques octets, convention avec le skeleton
- ▶ Et attendre le résultat en retour
 - ▶ Lecture sur une socket et désérialisation
- ▶ Donc un stub, c'est très simple!
 - ▶ On peut générer son bytecode à la réception, pour désérialisation
 - ▶ A condition d'avoir sur le client le fichier .class de l'interface distante ...



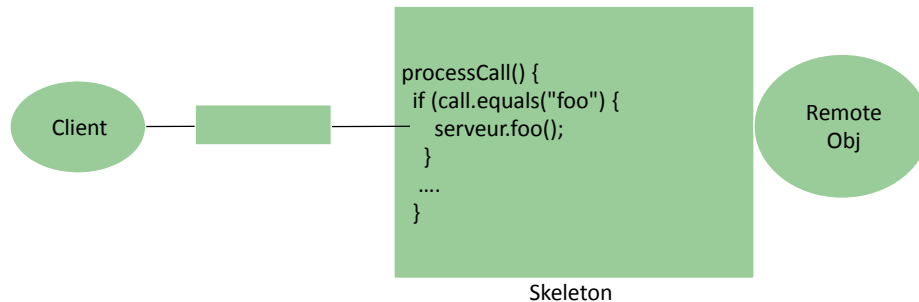
Le Stub



Le Skeleton

- ▶ Le skeleton appelle les méthodes sur l'objet distant
- ▶ Et retourne le résultat
- ▶ Est-il dépendant de l'objet distant?
 - ▶ Oui si implémentation statique (naïf)

Le Skeleton (version naïve)



Le Skeleton

- ▶ Y'a-t-il moyen de séparer le skeleton de l'objet appelé?
 - ▶ Oui, si on a un moyen de dire "je veux appeler la méthode dont le nom est foo" sans l'écrire explicitement
- ▶ Reflection
 - ▶ Capacité qu'a un programme à observer ou modifier ses structures internes de haut niveau
 - ▶ Concrètement, le langage permet de manipuler des objets qui représentent des appels de méthodes, des champs...
 - ▶ On fabrique un objet qui représente une méthode, et on demande son exécution
 - ▶ Partie intégrante de Java. Vital pour RMI, la sérialization...

Exemple de reflexion

```
String firstWord = "blih";
String secondWord = "blah";
String result = null;
Class c = String.class;
Class[] parameterTypes = new Class[]
{String.class};
Method concatMethod;
Object[] arguments = new Object[] {secondWord};

concatMethod =
c.getMethod("concat",parameterTypes);
result = (String)
concatMethod.invoke(firstWord,arguments);
```

Réflexion : utilisation par RMI

Côté Stub: encoder la méthode à invoquer côté serveur en tant qu'un objet

Method m = ... getMethod("sayHello"); // serveur offre méthode sayHello
qu'on appelle en faisant **invoke de m sur la référence distante (notée ref.)**

Aspect générique
du skeleton

```
// Stub class generated by rmic, do not edit.
public final class HelloWorldImpl_Stub extends
java.rmi.server.RemoteStub implements HelloWorld, java.rmi.Remote
{
    $method sayHello_0 = HelloWorld.class.getMethod("sayHello", new
    java.lang.Class[] {});
    public java.lang.String sayHello() throws java.rmi.RemoteException
    {
        try {
            Object $result = ref.invoke(this, $method_sayHello_0, null,
6043973830760146143L);
            return ((java.lang.String) $result); // appel méthode remote est
            donc synchrone
        } catch (java.lang.RuntimeException e) {
            throw e;
        } catch (java.rmi.RemoteException e) {
            throw e;
        }
    }
}
```

RMI et les threads

- ▶ Un appel RMI est initié par un thread côté appelant
- ▶ Mais exécuté par un autre thread côté appelé
- ▶ Le thread de l'appelant est bloqué jusqu'à ce que le thread du côté appelé ait fini l'exécution
- ▶ Si multiples appelants, multiples threads côté appelé
 - ▶ Un objet distant est par essence multithread!
 - ▶ Il faut gérer la synchronisation des threads (synchronized, wait, notify)
- ▶ Pas de lien entre le thread appelant et le thread côté appelé

RMI et les threads

- ▶ L'implémentation n'est pas spécifiée
- ▶ En pratique
 - ▶ Lorsqu'un appel arrive, RMI crée un thread pour l'exécuter
 - ▶ Une fois l'appel fini, le thread peut resservir pour un nouvel appel
 - ▶ Si multiples appels simultanés, de nouveaux threads sont créés
- ▶ Technique du thread pool
- ▶ Problème des appels ré-entrants
 - ▶ A fait un appel distant sur B, qui fait un appel distant sur A
 - ▶ Très courant: cycle dans le graphe d'objets
 - ▶ Pas de problèmes dans la plupart des cas (si ce n'est la latence)
 - ▶ Gros problèmes si les méthodes sont synchronized: lesquels ?

Interblocage (Deadlock) distribué

Même moniteur

```

CLIENT (et serveur)
public synchronized foo() {
    serveur.bar()
}

public synchronized toto() {
    .....
}
        
```

1

2

```


SERVEUR
public bar() {
    client.toto()
}
        
```

2 objets distants communiquent
 Cycle dans le graphe d'appels
 Le thread (step 2) ne peut pas entrer dans la méthode toto() tant que l'autre thread n'a pas fini d'exécuter foo, puisque le moniteur associé à l'objet CLIENT est occupé
 => Deadlock!

Parfois difficile à identifier

- Pas de vue globale de l'application car elle est répartie
- Pas d'information de la part de la JVM

Cours 5ème année (option/Ubinet) : Algorithmique pour les systèmes répartis!




SI4 Middleware– F. Baude, d'après cours F. Huet – 2011-12

61

Principe de fonctionnement du GC distribué (DGC)

- ▶ Se base sur le GC de chaque JVM concernée
 - ▶ Fondé sur un comptage des références : quand un objet n'est plus référencé depuis un objet "racine", alors, il peut être désalloué
- ▶ Lorsqu'un stub est envoyé dans une JVM, l'objet distant est donc référencé
 - ▶ La JVM qui reçoit ce stub incrémente le compteur de références distantes de l'objet distant (en lui envoyant un message particulier)
- ▶ Lorsque le stub n'est plus utilisé, cela a pour effet de décrémenter le compteur de références distantes de l'objet distant
 - ▶ [en théorie] Se produit lorsque l'objet dans lequel l'attribut pour stocker le stub est lui-même désalloué, ou bien, l'attribut est modifié
 - ▶ [en pratique] Se produit si le client n'utilise pas le stub pendant une certaine durée de temps ('lease'=bail, que l'on peut paramétrer via la propriété `java.rmi.dgc.leaseValue=tms`).
 - ▶ Lorsque le bail pour une référence arrive à 0 (maintenu côté serveur), le GC côté serveur décrémente de 1 le compteur de références
 - ▶ Pour éviter ceci, le seul moyen côté client est d'invoquer régulièrement les méthodes!
- ▶ Lorsque le compteur de références distantes est parvenu à 0 (donc sur serveur):
 - ▶ L'objet distant est considéré comme "garbageable"
 - ▶ Il le sera véritablement seulement lorsque plus de références locales vers cet objet

Cours 5ème année (option/Ubinet) : Algorithmique pour les systèmes répartis!



SI4 Middleware– F. Baude, d'après cours F. Huet – 2011-12

62

Et les stubs enregistrés dans le rmiregistry ?

- ▶ RMI est lui-même un objet distant, qui stocke des stubs
- ▶ Tant qu'un stub existe au niveau du registry, cela bloque le GC concernant l'objet distant qu'il référence
 - ▶ D'ailleurs, le RMI registry re-initialise son bail régulièrement (pour pas que l'objet serveur soit garbagé !)
- ▶ Donc... tant qu'on a un objet distant pour lequel un *bind* a été effectué, celui-ci est vivant, et occupe des ressources dans sa JVM.
 - ▶ Ceci peut être inutile
 - ▶ Notion d'**objet Activable**: est instancié seulement si un client tente d'en invoquer des méthodes à distance.
 - ▶ Un stub particulier est stocké dans le RMI registry, même si l'objet n'est pas instancié
 - ▶ L'usage d'un tel stub déclenche l'instantiation par le démon rmi



▶ <http://download.oracle.com/javase/1.4.2/docs/guide/rmi/activation.html>

SI4 Middleware– F. Baude, d'après cours F. Huet – 2011-12

63