RMI
(Remote Method Invocation)

Université Française d'Egypte
Richard Grin
Version 0.6 – 10/10/12

Présentation de RMI

Client – serveur, situation traditionnelle

- Sur la machine A un client a besoin d'informations qui sont sur une autre machine B (par exemple des données d'une base qui n'est pas accessible à distance)
- Le client construit la requête dans un format convenu avec le serveur
- Il envoie la requête formatée sur la machine B (en utilisant un socket par exemple)

Client – serveur, situation traditionnelle

- Le serveur reçoit la requête ; il l'analyse (*parse*) et cherche ou calcule les informations demandées
- Il met en forme la réponse dans un message et envoie le message (par socket par exemple) au client
- Le client reçoit la réponse et l'analyse pour obtenir ce qu'il demandait
- Tout ça est un peu compliqué !

Client – serveur, RMI

- Avec RMI, un objet client sur la machine C appelle tout simplement une méthode d'un objet placé sur la machine S et reçoit les informations qu'il cherchait en valeur de retour de la méthode
- Grâce à RMI un objet placé sur une machine distante peut recevoir des messages (presque) aussi simplement que s'il était sur la JVM locale
- Un tel objet est appelé un **objet distant** (*remote object*)

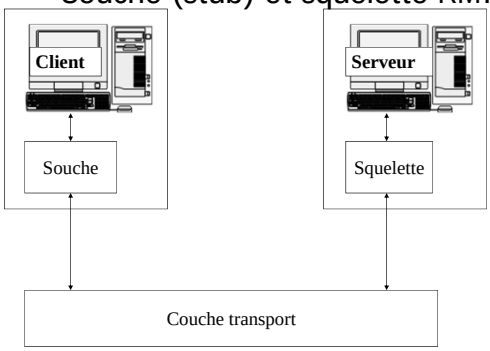
Objet distant RMI

- Sa classe doit implémenter une interface distante (*remote*) qui contient les méthodes qui pourront être appelées à distance
- Une interface distante
  - hérite de l'interface **`java.rmi.Remote`** (marqueur, ne contient aucune méthode)
  - a toutes ses méthodes qui déclarent pouvoir lever (**`throws`**) une **`java.rmi.RemoteException`**

Objets proxy

- L'appel de méthode d'un objet distant utilise en fait 2 objets proxy générés automatiquement
  - la souche (*stub*) qui représente l'objet distant sur la machine locale ; le message envoyé à l'objet distant passe par le stub
  - le squelette qui représente l'objet local sur la machine distante ; la valeur de retour de la méthode passe par le squelette
- Ils cachent la complexité : ils se chargent du formatage et de l'analyse de la requête et de la réponse, de la communication entre les 2 machines

Souche (stub) et squelette RMI



Vocabulaire RMI

- Serveur : classe Java qui fournit des méthodes qui peuvent être appelées à distance (depuis une autre JVM)
- Client : classe Java qui appelle une méthode d'un serveur (d'une classe qui tourne sur une autre JVM)
- Le plus souvent, les serveurs sont répartis sur les différentes JVM et une classe peut être à la fois client et serveur, selon les méthodes qui sont exécutées

Principes de fonctionnement des communications dans RMI

Souche (ou *stub*) (1/2)

- Une souche par objet distant
- La classe de la souche implémente la même interface distante que l'objet distant et implémente donc toutes les méthodes de cette l'interface
- Quand du code fait un appel à une méthode distante, c'est en fait la méthode correspondante de la souche locale qui est appelée

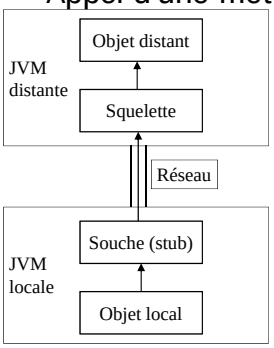
Souche (ou *stub*) (2/2)

- Cette méthode de la souche
  - prépare l'appel à la méthode correspondante de l'objet distant en mettant en forme les données à envoyer, en particulier les paramètres de la méthode (« *marshalisation* », emballage)
  - utilise le protocole de communication de RMI pour envoyer les données au squelette (en utilisant un socket)

Squelette

- Situé sur la machine distante (la même que l'objet distant)
- Il « *démarshalise* » les données envoyées par la souche
- Il fait un appel à la méthode de l'objet distant
- Il « *marshalise* » les données renvoyés par la méthode et les envoie à la souche

Appel d'une méthode distante



Exécution de la méthode

Envoi d'un message à l'objet distant

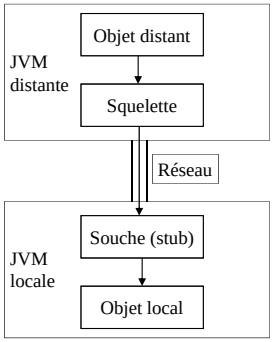
Déballage des paramètres

Communication avec le squelette

Emballage des paramètres

Envoi d'un message à la souche

Renvoi de la valeur de retour



Renvoi de la valeur de retour au squelette

Emballage de la valeur de retour

Communication avec la souche

Déballage de la valeur de retour

Envoi de la valeur de retour à l'objet local

Protocoles utilisés par RMI

- RMI est un une API (*Application Programming Interface*) qui s'appuie sur
  - le protocole de communication JRMP (*Java Remote Method Protocol*)
  - la sérialisation des objets
- Pour améliorer l'interopérabilité, il est possible d'utiliser le protocole IIOP (Internet Inter-ORB Protocol) au lieu de JRMP ; on facilite ainsi l'interface avec CORBA

Fonctionnalités de RMI

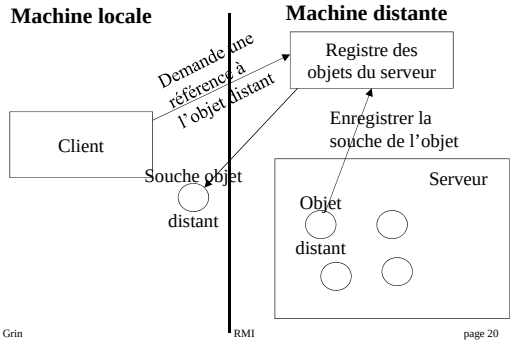
- RMI offre les fonctionnalités requises pour permettre d'envoyer des messages aux objets distants créés par le serveur :
  - permettre aux clients de localiser les objets distants
  - passer les paramètres des méthodes et récupérer les valeurs de retour à travers le réseau
  - chargement dynamique des classes distantes (classes liées aux paramètres et aux valeurs de retour)
    - classes de la machine locale pour la machine distante
    - classes de la machine distante pour la machine locale

Enregistrement et localisation des objets distants

Enregistrement des objets distants

- Les objets distants sont enregistrés auprès d'un **registre** (situé sur la même machine que ces objets distants pour des raisons de sécurité) que les clients peuvent interroger (pour obtenir une référence à ces objets)
- Ce service d'enregistrement est à l'écoute sur un port (au sens TCP/IP) connu des clients (1099 par défaut)
- Ce service est fourni par un objet distant dont l'accès est connu

Enregistrement des objets distants



Classe de la souche

- La souche est passée au registre et au client en utilisant la sérialisation
- Pour la désérialiser, le registre et le client ont besoin de la classe de la souche

Annotation des objets passés à distance avec l'adresse de leur classe

- Au moment de l'enregistrement d'un objet distant dans le registre, on peut annoter l'objet avec l'URL d'un emplacement où se trouve la classe de l'objet
- Il suffit de donner une valeur correcte à la propriété **`java.rmi.server.codebase`** de la JVM d'où vient l'objet
- Avec cette adresse, la JVM qui reçoit l'objet pourra charger la classe de l'objet (et ses classes ancêtres) si elle ne la trouve pas dans son *classpath*

Attention !

- Le registre ne doit pas avoir les classes des souches dans son *classpath* au moment où les objets distants sont enregistrés
- Sinon, l'objet distant ne sera pas annotée avec la valeur du *codebase* lors des transferts vers des JVM distantes
- Le plus sûr est de démarrer le registre avec un *classpath* vide

Propriété **codebase**

- La propriété **`java.rmi.server.codebase`** contient une suite d'URL
  - de répertoires (terminés par un « **/** »)
  - ou de fichiers **`.jar`**

  séparés par des espaces
- Comme toutes les propriétés, le plus simple pour donner la valeur de **`codebase`** est l'option **`-D`** de la commande *java*

rmiregistry

- Le programme ***rmiregistry*** lance l'exécution d'un registre qui peut être utilisé par l'intermédiaire de la classe **`java.rmi.Naming`**
- Il écoute sur un certain port (1099 par défaut)
  - les demandes d'enregistrement d'un objet distant situé sur la même machine que lui,
  - les clients qui souhaitent récupérer une référence à un objet distant à partir d'un URL

URL des objets distants

- Les objets distants enregistrés sont identifiés par un **URL**
- Format de l 'URL :
  `//machine:port/nomObjet`
- Exemples :
  - **`//clio.unice.fr:2585/compte`**
  - **`//clio.unice.fr/compte`** (port TCP 1099 par défaut pour le registre)
  - **`compte`** (machine locale par défaut)

Enregistrement des objets distants

- **Enregistrement** d'un objet distant :
  `Naming.rebind(url,objetDistant);`
- **Obtenir la référence** à un objet distant (et charger la classe de la souche si nécessaire) :
  `od = Naming.lookup(url);`
- **Liste** des objets enregistrés :
  `String[] liste = Naming.list(url);`
  (**`url`** indique le registre)

Code pour l'enregistrement des objets distants

```
try {
  Naming.rebind("//clio.unice.fr/unNom", this);
}
catch(RemoteException) {
  System.err.println("Impossible joindre registre");
}
catch(MalformedURLExceptionException e) {
  System.err.println("Mauvais format d'adresse");
}
```


RemoteException

- L'exception **`java.rmi.RemoteException`** est une exception qui peut être lancée par toute méthode qui est appelée à distance (et par les constructeurs des objets distants)
- C'est une exception contrôlée
- Elle est provoquée par une anomalie de fonctionnement du réseau (ou d'une machine distante)

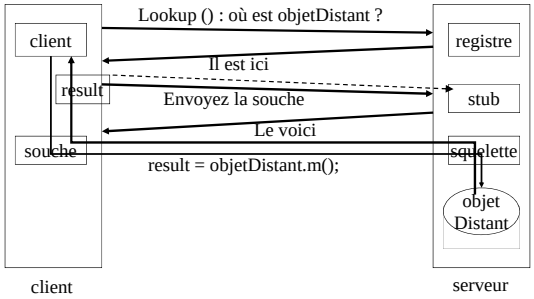
Code pour l'accès aux objets distants

```
try {
  Bourse bourse = (Bourse)
     Naming.lookup("//clio.unice.fr/unNom");
}
catch(NotBoundException e) {
  System.err.println("Pas d'objet de ce nom");
}
catch(MalformedURLException e) { . . . }
catch(RemoteException e) { . . . }
```

Obtenir une référence à un objet distant

- En fait, le plus souvent les objets distants sont obtenus non pas par un lookup mais sont plutôt renvoyés par un appel distant précédent

Interaction client-enregistreur



Passage des paramètres et de la valeur de retour d'une méthode distante

Passer à distance les paramètres et les valeurs de retour (1/2)

- La syntaxe du passage des paramètres à une méthode d'un objet distant, et la récupération de la valeur de retour, est la même que pour l'appel d'une méthode « locale »
- Mais la sémantique est différente pour les objets passés d'une JVM à une autre (les paramètres de la méthode et la valeur de retour)

Passer à distance les paramètres et les valeurs de retour (2/2)

- Comme pour les méthodes « locales »
  - les paramètres de type primitif (**`double`**, **`int`**,…) sont passés par copie de leur valeur
  - les objets distants (*remote*) sont passés par copie de leur référence (une référence distante)
- Mais les objets non *remote* sont passés par copie de leur valeur ; ils doivent être sérialisables

Copie des objets non *remote*

- L'objet est sérialisé au départ, et désérialisé à l'arrivée
- Ces sérialisation et désérialisation, et les opérations annexes, sont appelées marshalisation et démarshalisation (ou emballage et déballage)
- Elles sont effectuées par la souche et le squelette, et donc transparentes pour le programmeur

Mécanisme pour le passage en paramètre d'objets distants

- Soit un appel distant « **`od1.m(od2)`** » où od1 et od2 sont des objets distants (*remote*)
- **od2** est passé par copie de sa référence (comme pour les paramètres non primitifs des appels à des méthodes locales)
- En fait, la souche (*stub*) de **od2** est sérialisée et passée à la méthode distante (par la souche de **od1**)
- Ainsi, la méthode pourra modifier od2 comme pour un appel local : l'appel d'une méthode à od2 provoquera un appel distant, en passant par le *stub*

Chargement dynamique de classes (1/3)

- Les classes suivantes peuvent être chargées dynamiquement lors de l'appel d'une méthode d'un objet distant :
  - classe de la souche (chargée par un client RMI)
  - classes des types réels des paramètres de la méthode (chargée par un serveur RMI, si les types réels sont des sous-types des types déclarés)
  - classes des types réels de la valeur de retour de la méthode (chargée par un client RMI, si le type réel est un sous-type du type déclaré)
  - classes utilisées par les classes ci-dessus (y compris les classes « ancêtres »)

Chargement dynamique de classes (2/3)

- Un chargeur de classes spécial à RMI, ***RMIClassLoader***, est utilisé pour charger la souche et les classes des arguments et des valeurs de retour des méthodes distantes
- Le chargeur de classe fourni avec le JDK essaiera d'utiliser un serveur HTTP pour charger les classes
- Mais on peut écrire son propre chargeur de classe pour utiliser un autre protocole (comme FTP)

Chargement dynamique de classes (3/3)

- Si ces classes ne sont pas accessibles par un serveur HTTP (ou autre), elles doivent être installées « à la main » dans les chemins de recherche des classes (*classpath*) des machines qui en auront besoin

Sécurité

Sécurité

- Quand une classe distante est chargée par le chargeur spécial à RMI, elle est vérifiée par un **gestionnaire de sécurité**
- Si aucun gestionnaire de sécurité n'a été installé (par **`System.setSecurity()`** ou par une option au lancement de la JVM)**,** seules les classes placées dans le *classpath* peuvent être récupérées

Sécurité

- Le gestionnaire de sécurité par défaut pour RMI est **`RMISecurityManager`**
- Il est très simple :
  - il vérifie la définition des classes et autorise seulement les passages des paramètres et des valeurs retour des méthodes distantes
  - il ne prend pas en compte les signatures éventuelles des classes pour décider des autorisations (il pourrait accepter seulement l'appel de certaines méthodes)

Police de sécurité

- Si on a besoin de charger des classes distantes, il est indispensable de fournir un fichier de sécurité pour préciser quelles autorisations on permet (voir cours sur la sécurité)

Rappels sur la police de sécurité

- Le nom du fichier qui contient la politique de sécurité est donné par la propriété **`java.security.policy`** (indiquée par exemple au moment du lancement de java par l'option **`-D`**)
- Par défaut, la politique de sécurité est donnée dans les fichiers de sécurité dont les emplacements sont dans le fichier **`lib/security/java.security`** du répertoire où est installé jre ; ces fichiers sont **`java.policy`** (du même répertoire que **`java.security`**) et **`.java.policy`** du répertoire HOME de l'utilisateur

Exemple de fichier de sécurité

- Le fichier suivant autorise les connexions avec des *sockets* sur le port 80 ou un port de numéro compris entre 1024 et 65535, et autorise les demandes de connexions sur ces derniers ports

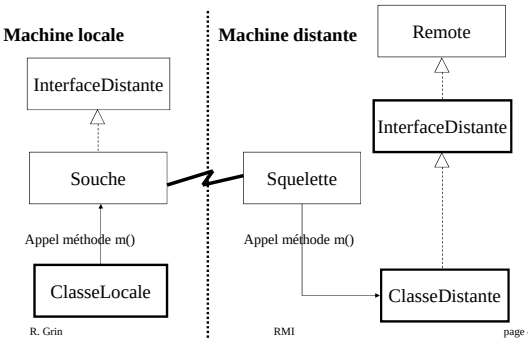
```
grant {
  permission java.net.SocketPermission "*:1024-
65535", "connect,accept";
  permission java.net.SocketPermission "*:80",
"connect";
};
```


Autre exemple

```
grant {
    permission java.net.SocketPermission
  "*:1024-65535", "connect,accept";
    permission java.io.FilePermission
        "/home/toto/public_html/classes/-",
  "read";
    permission java.io.FilePermission
        "/home/bibi/public_htmlclasses/-",
  "read";
};
```


Écriture du code des classes des objets distants et des objets locaux qui font des appels aux méthodes distantes

Classes et interfaces



Classes et interfaces à écrire

1. Définir les **interfaces** pour les objets **distants**
   Ces interfaces comportent les méthodes qui pourront être appelées à distance (les méthodes « exportées »)
2. Ecrire les **classes distantes** qui implémentent les interfaces
   Les classes peuvent comporter d'autres méthodes que celles des interfaces qu'elles implémentent
3. Ecrire les programmes **clients**
   (En fait, il arrive souvent que les classes soient à la fois des clientes et des classes distantes)

Interface « Remote » pour les objets distants

- Le programmeur indique les méthodes de l'objet distant qui pourront être appelées depuis une JVM distante en donnant une interface « Remote » qui contient ces méthodes
- La classe de l'objet distant devra implémenter cette interface
- Les autres méthodes ne pourront pas être appelées à distance

Exemple d'interface Remote

```
import java.rmi.*;

public interface CoursBourse extends Remote {
  double getValeur(String nomValeur)
    throws RemoteException;
}
```


Écriture du serveur

- On doit mettre l'objet distant à disposition des clients :
  (souvent fait par la méthode main() de la classe distante)
  - lancer un gestionnaire de sécurité (une fois, pour tous les objets),
  - créer une instance de la classe,
  - l'associer à un nom pour l'enregistreur d'objets distants.

```
System.setSecurityManager(
                  new RMISecurityManager());
ClasseDistante c = new ClasseDistante();
Naming.rebind("compte", c);
```


Classe `UnicastRemoteObject`

- Pour écrire une classe distante le plus simple est d'hériter de la classe **`UnicastRemoteObject`**
- Elle hérite de **`RemoteServer`**
- Cette classe offre toutes les fonctionnalités des classes distantes : (dé)marshalisation en particulier
- Elle utilise TCP/IP pour la couche transport

Exporter un objet distant

- Si on ne souhaite pas hériter de la classe **`UnicastRemoteObject`** on doit exporter l'objet distant en appelant explicitement la méthode **`static exportObject`** de la classe **`UnicastRemoteObject`** :

```
ClasseDistante c = (ClasseDistante)
  UnicastRemoteObject.export(objDistant, 0);
Naming.rebind("compte", c);
```

- Cette exportation est faite automatiquement par le constructeur de la classe **`UnicastRemoteObject`**

Méthodes distantes

- On doit indiquer que les méthodes qui peuvent être appelées à distance peuvent lancer l'exception **`RemoteException`**

Constructeurs d'objets distants

- Même les constructeurs des classes des objets distants doivent lancer cette exception (à cause du mécanisme d'exportation des objets distants)
- Ce qui implique que l'on ne doit plus utiliser de constructeur par défaut ; si on veut un constructeur sans paramètre, on doit l'écrire explicitement :

```
public ClDistante() throws RemoteException {}
```


Paramètres et valeurs de retour des méthodes distantes

- Tous les types non primitifs qui correspondent à des objets locaux des paramètres des méthodes des objets distants, doivent être **sérialisables**
- Il en est de même des valeurs de retour de ces méthodes
- Mais les objets distants peuvent ne pas être sérialisables

Écriture du client

- Le client récupère l'instance de l'objet distant par son nom (en fait, il récupère une souche)
- Cette instance doit être une instance de **l'interface distante** et pas de la classe qui l'implémente

```
InterfaceDistante objetDistant =
    (InterfaceDistante)Naming.lookup(
                  "//clio.unice.fr/compte");
```

- Tous les appels aux méthodes distantes doivent être dans des blocs **`try`** attrapant les **`RemoteException`**, ou être dans des méthodes avec un **`throws RemoteException`**

Principe important pour l'utilisation des objets distants

- **Les objets locaux ne peuvent utiliser que les méthodes déclarées dans l'interface distante**
- En effet, le client travaille avec la souche et pas avec l'objet distant
- La souche n'implémente que les méthodes de l'interface distante
- Le client ne peut donc pas appeler les méthodes de la classe de l'objet distant qui ne sont pas dans l'interface distante, même si elles sont **`public`**

Mise en place de l'application

Étapes pour la mise en place

1. Compiler les sources Java (interfaces distantes, classes serveurs et clientes)
2. Déployer les classes (dont les instances seront passées par sérialisation) sur les différentes machines, ou à disposition des serveurs Web (en vue d'un téléchargement)

Étapes pour la mise en place

3. Écrire des fichiers de police de sécurité qui seront placés sur les machines des serveurs (objets distants) Le serveur pourra ainsi charger dynamiquement des classes (de paramètres de méthodes distantes) depuis le *codebase*

```
grant {
   permission java.net.SocketPermission "*:1024-65535",
"connect,accept";
   permission java.io.FilePermission
"/home/toto/public_html/classes/-", "read";
   permission java.io.FilePermission
"/home/bibi/public_html/classes/-", "read";
};
```


Étapes pour la mise en place

4. Démarrer le registre,
   sur la machine distante on lance (sous Unix)
   `rmiregistry &`
   `(start rmiregistry` sous Windows**`)`**

Étapes pour la mise en place

5. Démarrer les serveurs ; on doit lancer chaque serveur en indiquant (par des propriétés)
   - un *codebase* qui indique où les clients pourront éventuellement récupérer les classes (**`java.rmi.server.codebase`**)
   - un fichier de police de sécurité (**`java.security.policy`**) qui indique quelles sont les classes qui pourront être chargées dynamiquement
   - et il est plus sûr de donner aussi le nom de l'ordinateur sur lequel tourne le serveur (**`java.rmi.server.hostname`**)

Étapes pour la mise en place

Exemple de lancement d'un serveur (sur une seule ligne) :

```
java -Djava.rmi.server.codebase=
http://clio.unice.fr/~toto/classesPubliques/
-Djava.rmi.server.hostname=clio.unice.fr
-Djava.security.policy=java.policy Serveur
```

Étapes pour la mise en place

6. Démarrer les clients, avec un codebase et un fichier de sécurité

   Exemple (sur une seule ligne):

```
java -Djava.rmi.server.codebase=
http://asterix.unice.fr/~bibi/classesPubliques/
-Djava.security.policy=java.policy Client
```


Avec les anciennes versions du JDK

- Il fallait générer les souches (stubs) et squelettes (pas de squelettes avec l'option **`-v1.2`**) et éventuellement les déployer sur les différentes machines

  sur la machine distante on lance

  **`rmic [-v1.2] [`*`package`*`.]`*`ClasseDistante`*

  (nom complet de la classe, avec son paquetage)

Les paquetages

- **`java.rmi`** : pour accéder à des objets distants
- **`java.rmi.server`** : pour créer des objets distants
- **`java.rmi.registry`** : lié à la localisation et au nommage d'objets distants
- **`java.rmi.dgc`** : ramasse-miettes pour les objets distants (très rarement utilisé)

Principales exceptions

- **`java.rmi.RemoteException`**
- **`java.rmi.NotBoundException`**
- **`java.rmi.StubNotFoundException`**

Compléments

RMI et les *threads*

- Chaque appel de méthode distante est pris en charge par un *thread*
  - qui exécute le code de la méthode
  - en partageant les données avec les autres threads qui traitent d'autres appels
- Il faut donc prévoir des synchronisations

Ramasse-miette (*garbage collector*)

- Fonctionne aussi avec RMI
- Utilise le comptage de lien et donc ne repère pas les cycles isolés (par exemple 2 classes non référencées qui ont des références l'une vers l'autre)
- La machine virtuelle Java locale envoie un message au serveur si un objet distant n'est plus référencé

Communications spéciales

- Si on veut un transport spécial des objets sérialisés, par exemple, encryptage ou compression, on peut utiliser une facilité offerte par le JDK 1.2 qui permet de faire effectuer le transport par un type de socket particulier (*Custom RMI Socket Factory*)

Objets activables

- Avant le JDK 1.2, tout objet distant devait exister avant de pouvoir être contacté par un autre objet
- Depuis le JDK 1.2, un objet distant peut être activé (créé) sur demande (interface **`activatable`**)

Objets distants et tables de hachage

- On ne peut redéfinir les méthodes **`hashcode`** et **`equals`** pour les classes souches (*stub*) engendrées automatiquement par *rmic*
- Donc, dans une table de hachage locale, 2 objets distants différents ne seront jamais égaux au sens de **`equals`**, même si on a redéfini **`equals`** dans la classe de ces objets car la classe de la souche implémente seulement l'interface distante (des méthodes exportées) et aura donc la méthode **`equals`** de **`RemoteObject`**

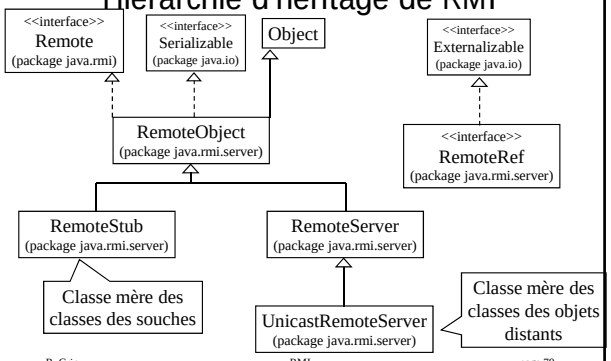
Objets distants et tables de hachage (2)

- Donc, si on veut mettre des objets avec comme clé des objets distants dans une table de hachage du côté du client (en local), 2 clés égales au sens de **`equals`** de la classe des objets pourront se retrouver dans la table
- Ce problème n'arrivera pas si la table de hachage est du côté du serveur (car la méthode **`equals`** de la classe d'implémentation sera utilisée)

Clonage des objets distants

- La méthode **`clone`** ne peut être redéfinie dans la classe souche (mêmes raisons que pour **`hashcode`** et **`equals`**)
- Si on veut pouvoir cloner un objet distant, on doit définir une nouvelle méthode (par exemple, **`remoteClone`**) dans l'interface **`Remote`** pour faire le clonage

Hiérarchie d'héritage de RMI



Interface RemoteRef

- Cette interface correspond à une référence à un objet distant (pas placé sur la même JVM)
- Sa méthode **`invoke`** permet d'appeler une méthode d'un objet distant ; **`invoke`** prépare l'appel à la méthode distante (*marshalise*,…)
- Elle est utilisée par la classe **`RemoteObject`**, et en particulier par sa sous-classe **`RemoteStub`**, mère des classes souches (*stub*)
- Elle permet ainsi à une souche de garder une référence à l'objet distant correspondant

Bibliographie

- Tutoriel Oracle : http://docs.oracle.com/javase/tutorial/rmi/index.html