

Java et les objets distribués

R.M.I.

Patrick Itey

INRIA - Sophia

itey@sophia.inria.fr

<http://www.inria.fr/acacia/personnel/itey>

Le rêve de tout système distribué

- L'idéal serait d'avoir un système distribué utilisant la technologie objet et permettant :
 - 1) d'invoquer une méthode d'un objet se trouvant sur une autre machine exactement de la même manière que s'il se trouvait au sein de la même machine :

```
objetDistant.methode( ) ;
```

Le rêve de tout système distribué (suite)

- 2) d'utiliser un objet distant (OD), sans savoir où il se trouve, en demandant à un service « dédié » de renvoyer son adresse :

```
objetDistant =  
ServiceDeNoms.recherche("monObjet");
```

8 janvier, 2001

© P. Itéy - Inria

Java - RMI - p 3

Le rêve de tout système distribué (suite)

- 3) de pouvoir passer un OD en paramètre d'appel à une méthode locale ou distante :

```
resultat =  
objetLocal.methode(objetDistant);
```

```
resultat =  
objetDistant.methode(autreObjetDista  
nt);
```

8 janvier, 2001

© P. Itéy - Inria

Java - RMI - p 4

Le rêve de tout système distribué (suite)

- 4) de pouvoir récupérer le résultat d'un appel distant sous forme d'un nouvel objet qui aurait été créé sur la machine distante :

```
ObjetDistant = ObjetDistant.methode() ;
```

8 janvier, 2001

© P. Itey - Inria

Java - RMI - p 5

Le rêve devient réalité

- ... avec Java qui va répondre à tous ces besoins par 2 offres :
 - **JOE (Java Object Environment)** : offre compatible CORBA) est destiné à des applications distribuées faisant interagir des objets Java avec d'autres objets non Java
 - **RMI (Remote Method Invocation)** est un système d'objets distribués performant destiné au développement d'applications distribuées entièrement en Java

8 janvier, 2001

© P. Itey - Inria

Java - RMI - p 6

Remote Method Invocation

- ❑ RMI est une *core* API (intégré au JDK 1.1)
 - gratuit (différent de CORBA)
 - 100 % Java
- ❑ Développé par JavaSoft
 - et non SunSoft comme JOE (Java IDL)
- ❑ RMI propose un système d'objets distribués plus simple que CORBA
- ❑ RMI est uniquement réservé aux objets Java

8 janvier, 2001

© P. Itey - Inria

Java - RMI - p 7

Présentation

- ❑ Mécanisme permettant l'appel de méthodes entre objets Java s'exécutant sur des machines virtuelles différentes (espaces d'adressage distincts), sur le même ordinateur ou sur des ordinateurs distants reliés par un réseau
 - utilise directement les sockets
 - code ses échanges avec un protocole propriétaire : R.M.P. (**R**emote **M**ethod **P**rotocol)

8 janvier, 2001

© P. Itey - Inria

Java - RMI - p 8

Objectifs

- ❑ Rendre transparent l'accès à des objets distribués sur un réseau
- ❑ Faciliter la mise en œuvre et l'utilisation d'objets distants Java
- ❑ Préserver la sécurité (inhérent à l'environnement Java)
 - RMISecurityManager
 - Distributed Garbage Collector (DGC)

8 janvier, 2001

© P. Itey - Inria

Java - RMI - p 9

Principes (1)

- ❑ Un objet distant (objet serveur) : ses méthodes sont invoquées depuis une autre JVM
 - dans un processus différent (même machine)
 - ou dans une machine distante (via réseau)
- ❑ Un OD (sur un serveur) est décrit par une interface (ou plus) distante Java
 - déclare les méthodes distantes utilisables par le client

8 janvier, 2001

© P. Itey - Inria

Java - RMI - p 10

Principes (2)

- ❑ Une invocation distante (RMI) est l'action d'invoquer une méthode d'une interface distante d'un objet distant.
- ❑ Une invocation de méthode sur un objet distant a la même syntaxe qu'une invocation sur un objet local.

Un OD se manipule comme un objet local

8 janvier, 2001

© P. Itey - Inria

Java - RMI - p 11

La distribution d'objets (1)

- ❑ Une référence à un OD peut être passée en argument ou retournée en résultat d'un appel dans toutes les invocations (locales ou distantes)
- ❑ Un OD peut être transformé (*cast*) en n'importe quelles interfaces distantes supportées par l'implémentation de l'objet

8 janvier, 2001

© P. Itey - Inria

Java - RMI - p 12

La distribution d'objets (2)

- ❑ Les clients (objets clients) des OD (objets serveurs) interagissent avec des interfaces distantes, jamais directement avec leurs implémentations.
- ❑ Les arguments locaux et les résultats d'une invocation distante sont toujours passés par copie et non par référence
 - leurs classes doivent implémentées
`java.io.Serializable` (sérialisation d'objets)

8 janvier, 2001

© P. Itey - Inria

Java - RMI - p 13

Le passage de paramètres (1)

- ❑ Pour résumer :
 - l'argument ou la valeur de retour d'une méthode distante peut être de n'importe quel type Java y compris les types simples, les objets locaux ou les objets distants.
 - Si jamais le type n'est pas disponible localement, il est chargé dynamiquement (RMIClassLoader)

8 janvier, 2001

© P. Itey - Inria

Java - RMI - p 14

Le passage de paramètres (2)

- pour les paramètres locaux (objets ou types primitifs), le transfert se fait nécessairement par copie. S'il s'agit d'un objet, l'ensemble de ses variables est copié par sérialisation
 - il doit donc implémenter `java.io.Serializable`
- pour les paramètres qui seraient déjà des références d'OD, l'objet amorce (Stub) est passé :
 - impossible de passer en paramètre un objet uniquement visible par les classes serveurs

8 janvier, 2001

© P. Itey - Inria

Java - RMI - p 15

La notion d'interface

- L'interface constitue le contrat - abstrait - liant objets serveurs et objets clients
 - elle est destinée à être implémentée par l'OD et constitue la base d'appel pour les objets clients
 - elle définit les signatures (noms, types de retours, paramètres) d'un ensemble de méthodes et seules ces méthodes seront accessibles par un objet client

8 janvier, 2001

© P. Itey - Inria

Java - RMI - p 16

La notion d'interface (suite)

- ❑ Pour RMI, c'est une interface Java traditionnelle ... mais dérivant de la classe `java.rmi.Remote`

8 janvier, 2001

© P. Itey - Inria

Java - RMI - p 17

L'exception `RemoteException`

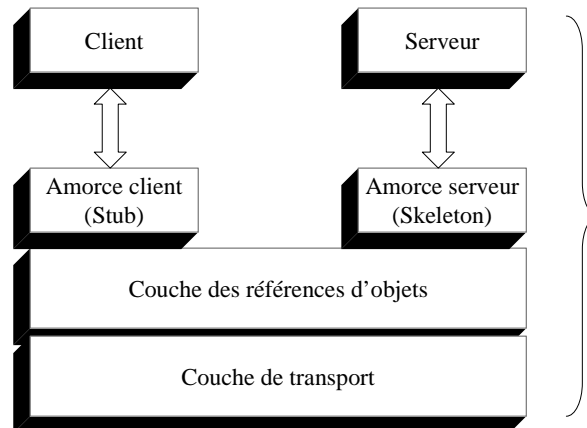
- ❑ Les objets clients doivent traiter des exceptions supplémentaires comme `RemoteException`
- ❑ Toute invocation distante de méthode doit lever ou traiter cette exception
- ❑ Peut se produire si la connexion a été interrompue ou si le serveur distant ne peut être trouvé

8 janvier, 2001

© P. Itey - Inria

Java - RMI - p 18

Architecture



8 janvier, 2001

© P. Itey - Inria

Java - RMI - p 19

Les amorces (Stub/Skeleton)

- ❑ Programmes jouant le rôle d'adaptateurs pour le transport des appels distants
 - réalisent les appels sur la couche réseau
 - pliage / dépliage des paramètres
- ❑ A une référence d'OD manipulée par un client correspond une référence d'amorce
- ❑ Les amorces sont générées par le compilateur d'amorces : **rmic**

8 janvier, 2001

© P. Itey - Inria

Java - RMI - p 20

L'amorce client (Stub) (1)

- ❑ Représentant local de l'OD qui implémente ses méthodes « *exportées* »
- ❑ Transmet l'invocation distante à la couche inférieure *Remote Reference Layer*
- ❑ Il réalise le pliage (« *marshalling* ») des arguments des méthodes distantes
- ❑ Dans l'autre sens, il réalise le dépliage (« *demarshalling* ») des valeurs de retour

8 janvier, 2001

© P. Itey - Inria

Java - RMI - p 21

L'amorce client (Stub) (2)

- ❑ Il utilise pour cela la sérialisation des objets
 - il les transforme en un flot de données (flux de pliage) transmissible sur le réseau
 - les objets doivent implémenter l'interface **java.io.Serializable** **OU** **java.io.Externalizable**

8 janvier, 2001

© P. Itey - Inria

Java - RMI - p 22

L'amorce serveur (Skeleton)

- ❑ Réalise le dépliage des arguments reçus par le flux de pliage
- ❑ Fait un appel à la méthode de l'objet distant
- ❑ Réalise le pliage de la valeur de retour

8 janvier, 2001

© P. Itéy - Inria

Java - RMI - p 23

La couche des références distantes

- ❑ Permet l'obtention d'une référence d'objet distant à partir de la référence locale au Stub
- ❑ Ce service est assuré par le lancement du programme **rmiregister**
 - à ne lancer qu'une seule fois par JVM, pour tous les objets à distribuer
 - une sorte de service d'annuaire pour les objets distants enregistrés

8 janvier, 2001

© P. Itéy - Inria

Java - RMI - p 24

La couche de transport

- ❑ Connecte les 2 espaces d'adressage (JVM)
- ❑ Suit les connexions en cours
- ❑ Ecoute et répond aux invocations
- ❑ Construit une table des OD disponibles
- ❑ Réalise l'aiguillage des invocations

8 janvier, 2001

© P. Itey - Inria

Java - RMI - p 25

Processus de développement d'une application RMI

- 1) définir une interface Java pour un OD
- 2) créer et compiler une classe implémentant cette interface
- 3) créer et compiler une application serveur RMI
- 4) créer les classes Stub et Skeleton (**rmic**)
- 5) démarrer **rmiregister** et lancer l'application serveur RMI
- 6) créer, compiler et lancer un programme client accédant à des OD du serveur

8 janvier, 2001

© P. Itey - Inria

Java - RMI - p 26

Vite, un exemple !

- Un « echo » distribué :
 - invocation distante de la méthode `echo()` d'un objet distribué avec le mécanisme RMI.
 - 4 programmes sont nécessaires :
 - `Echo` : l'interface décrivant l'objet distant (OD)
 - `EchoImpl` : l'implémentation de l'OD
 - `EchoAppliServer` : une application serveur RMI
 - `EchoClient` : l'application cliente utilisant l'OD

8 janvier, 2001

© P. Itey - Inria

Java - RMI - p 27

1) définir l'interface pour la classe distante

- Rappel : les classes placées à distance sont spécifiées par des interfaces qui doivent dériver de `java.rmi.Remote` et dont les méthodes lèvent une `java.rmi.RemoteException`

```
package demo.rmi;
import java.rmi.*;

public interface Echo extends Remote {
    public String echo(String str)
        throws RemoteException;
}
```

8 janvier, 2001

© P. Itey - Inria

Java - RMI - p 28

2) définir l'implémentation de l'OD

```
package demo.rmi;
import java.rmi.*;
import java.rmi.server.*;

public class EchoImpl
    extends UnicastRemoteObject implements Echo {

    public EchoImpl() throws RemoteException {super();}

    public String echo(String str) throws RemoteException {
        return "Echo : " + str;
    }
}
```

8 janvier, 2001

© P. Itey - Inria

Java - RMI - p 29

3) définir une application serveur

- C'est le programme qui sera à l'écoute des demandes des clients.
 - il lui faut un SecurityManager spécifique :
RMISecurityManager
 - pour rendre l'objet disponible, il faut l'enregistrer dans le « RMIregistry » par la méthode statique :
 - **Naming.rebind("echo", od);**

8 janvier, 2001

© P. Itey - Inria

Java - RMI - p 30

3) définir une application serveur (suite)

- Par la suite, cet objet distant sera accessible par les autres machines en indiquant son nom et la machine sur laquelle est exécuté ce serveur par l'url :

```
String url = "rmi://nomServeurRMI:port/nomOD"  
String url = "rmi://leo.inria.fr/echo »
```

- Cet " url RMI " sera utilisé par les clients pour interroger le serveur grâce à l'appel :

```
Naming.lookup(url);
```

8 janvier, 2001

© P. Itey - Inria

Java - RMI - p 31

3) définir une application serveur (suite)

```
package demo.rmi;  
  
public class EchoAppliServer {  
    public static void main(String args[]) {  
        // Création et installation du manager de sécurité  
        System.setSecurityManager(new RMISecurityManager());  
  
        try { // Création de l'OD  
            EchoImpl od = new EchoImpl();  
            // Enregistrement de l'OD dans RMI  
            Naming.rebind("echo", od);  
        }  
        catch(Exception e) {...}  
    }  
}
```

8 janvier, 2001

© P. Itey - Inria

Java - RMI - p 32

Compiler les classes

```
> javac -d $HOME/classes Echo.java
> javac -d $HOME/classes EchoImpl.java
> javac -d $HOME/classes EchoAppliServer.java
```

Ces commandes créent les fichiers **.class** correspondants et les déposent dans **\$HOME/classes/demo/rmi/**

8 janvier, 2001

© P. Itey - Inria

Java - RMI - p 33

4) créer les amorces

- Il s'agit d'invoquer le compilateur d'amorces **rmic** sur la classe compilée de l'OD :

```
> rmic -d $HOME/classes demo.rmi.EchoImpl
```

- 2 classes sont alors créées représentant la souche (ou Stub) et le Skeleton :

EchoImpl_**stub**.class

EchoImpl_**skel**.class

8 janvier, 2001

© P. Itey - Inria

Java - RMI - p 34

5) Lancer `rmiregister` et l'application serveur

- Il faut d'abord lancer le `rmiregister` puis le serveur :

```
> rmiregister&  
> java demo.rmi.EchoAppliServer
```

- Ce qui rend maintenant le serveur disponible pour de futurs clients ...

8 janvier, 2001

© P. Itey - Inria

Java - RMI - p 35

6) définir l'application cliente utilisant l'OD

```
package demo.rmi;  
import java.rmi.*;  
import java.rmi.registry.*;  
  
public class EchoClient {  
    public static void main(String args[]) {  
        // Création et installation du manager de sécurité  
        System.setSecurityManager(new RMISecurityManager());  
        // Recherche de l'OD  
        String url = "rmi://" + args[0] + "/echo";  
        Echo od = (Echo)Naming.lookup(url);  
        System.out.println(od.echo(args[1]));  
    }  
}
```

8 janvier, 2001

© P. Itey - Inria

Java - RMI - p 36

6) définir l'application cliente utilisant l'OD (suite)

□ Remarque :

- ce code manipule l'OD comme s'il était local.
- Après avoir installé un RMISecurityManager, le client recherche l'objet distant en interrogeant le service « d'annuaire » RMI par :
 - `Naming.lookup(url)` ou url est un « url RMI » de la forme :
 - `String url="rmi://nomServeurRMI:port/nomOD"`
 - si une seule machine : `nomServeurRMI= " localhost "`

8 janvier, 2001

© P. Itey - Inria

Java - RMI - p 37

6) compiler et lancer l'application cliente

```
% javac -d $HOME/classes EchoClient.java
% java demo.rmi.EchoClient leo.inria.fr coucou

% Echo : coucou
```

8 janvier, 2001

© P. Itey - Inria

Java - RMI - p 38

Chargement dynamique des amorces

- ❑ Rappel : un objet client ne peut utiliser un objet distant qu'au travers des amorces
- ❑ RMI permet l'utilisation des OD dont les amorces ne sont pas disponibles au moment de la compilation
- ❑ A l'exécution, RMI réclamera au serveur l'amorce cliente manquante et la téléchargera dynamiquement (byte code)

8 janvier, 2001

© P. Itey - Inria

Java - RMI - p 39

Chargement dynamique de classes

- ❑ Plus généralement, le système RMI permet le chargement dynamique de classes comme les amorces, les interfaces distantes et les classes des arguments et valeurs de retour des appels distants
- ❑ C'est un chargeur de classes spécial RMI qui s'en charge :
`java.rmi.server.RMIClassLoader`

8 janvier, 2001

© P. Itey - Inria

Java - RMI - p 40

Sécurité (1)

- ❑ RMI n'autorise pas le téléchargement dynamique de classes (avec *RMIClassLoader*) si l'application (ou l'applet) cliente n'utilise pas de *Security Manager* pour les vérifier.
- ❑ Dans ce cas, seules les classes situées dans le `CLASSPATH` peuvent être récupérées

8 janvier, 2001

© P. Itey - Inria

Java - RMI - p 41

Sécurité (2)

- ❑ Le gestionnaire de sécurité par défaut pour RMI est `java.rmi.RMISecurityManager`
- ❑ Il doit être absolument utilisé (`System.setSecurity()`) pour les applications *standalone*
- ❑ Pas de problème pour les applets, c'est l'*AppletSecurityManager* (par défaut) qui s'en charge

8 janvier, 2001

© P. Itey - Inria

Java - RMI - p 42

RMISecurityManager

- ❑ Il est très simple :
 - vérifie la définition des classes et autorise seulement les passages des arguments et des valeurs de retour des méthodes distantes

 - ne prend pas en compte les signatures éventuelles des classes

8 janvier, 2001

© P. Itey - Inria

Java - RMI - p 43

Les packages

- ❑ `java.rmi` : pour les classes côté client (accéder à des OD)
- ❑ `java.rmi.server` : pour les classes côté serveur (création des OD)
- ❑ `java.rmi.registry` : lié à l'enregistrement et à la localisation des OD
- ❑ `java.rmi.dgc` : pour le *Garbage Collector* des OD

8 janvier, 2001

© P. Itey - Inria

Java - RMI - p 44

Vite, un autre exemple !

- Un gestionnaire de comptes bancaires :
 - il doit être capable de gérer plusieurs comptes
 - il sera placé sur une machine distante, interrogé et manipulé par des clients.

8 janvier, 2001

© P. Itey - Inria

Java - RMI - p 45

Exemple : définir les interfaces pour les classes distantes

- Rappel : les classes placées à distance sont spécifiées par des interfaces implémentant `java.rmi.Remote` et dont les méthodes peuvent lever une `java.rmi.RemoteException`

- Création de 2 OD :
 - **CreditCard** et **CreditManager**

8 janvier, 2001

© P. Itey - Inria

Java - RMI - p 46

Exemple : définir les interfaces pour les classes distantes (suite)

```
package credit;
import java.rmi.*;

public interface CreditCard extends Remote {
    public float getCreditLine()
        throws RemoteException;

    public void makePurchase(float amount, int signature)
        throws RemoteException,
            InvalidSignatureException,
            CreditLineExceededException;
}
```

8 janvier, 2001

© P. Itey - Inria

Java - RMI - p 47

Exemple : définir les interfaces pour les classes distantes (suite)

```
package credit;
import java.rmi.*;

public interface CreditManager extends Remote {
    /** recherche le compte bancaire d'une personne (le
    cree s'il n'existe pas */
    public CreditCard findCreditAccount(String customer)
        throws RemoteException,
            DuplicateAccountException;

    public CreditCard newCreditAccount(String
    newCustomer) throws RemoteException;
}
```

8 janvier, 2001

© P. Itey - Inria

Java - RMI - p 48