
Université Lille 1
Master mention Informatique – M1
Construction d'applications réparties

V. Java RMI

Lionel.Seinturier@univ-lille1.fr

Java RMI

1

Lionel Seinturier

1. Caractéristiques

Java RMI

Un mécanisme d'invocation de méthodes distantes sur des objets Java

- inclus par défaut dans le JDK depuis 1.1 (évolutions dans JDK 1.2 puis 5)
- package `java.rmi`
- + outils : générateur de souches, serveur de noms, démon d'activation

Passage de paramètres

- | | |
|--|---|
| • types simples (<code>int</code> , <code>float</code> , ...) | transmission par copie |
| • objet local | transmission par copie (sérialisation) |
| • objet RMI distant | transmission par référence |

- implantations alternatives
 - NinjaRMI, Jeremie, NRMI (copie-restauration)

Java RMI

3

Lionel Seinturier

Plan

1. Caractéristiques
2. Modèle de programmation
3. Services
4. Fonctionnalités additionnelles
5. Protocole

Java RMI

2

Lionel Seinturier

2. Modèle de programmation

Principes

Chaque classe d'objets serveur doit être associée à une interface

⇒ seules les méthodes de l'interface pourront être invoquées à distance

1. Ecriture d'une interface
2. Ecriture d'une classe implantant l'interface
3. Ecriture du programme serveur
4. Ecriture du programme client

≡

1. déclaration des services accessibles à distance
2. définition du code des services
3. instanciation et enregistrement de l'objet serveur
4. interactions avec le serveur

Java RMI

4

Lionel Seinturier

2. Modèle de programmation

Ecriture d'une interface

- interface Java normale
- doit étendre `java.rmi.Remote`
- toutes les méthodes doivent lever `java.rmi.RemoteException`

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

interface CompteInterf extends Remote {
    public String getTitulaire() throws RemoteException;
    public float solde() throws RemoteException;
    public void déposer( float montant ) throws RemoteException;
    public void retirer( float montant ) throws RemoteException;
    public List historique() throws RemoteException;
}
```

2. Modèle de programmation

Ecriture d'une classe implantant l'interface

- classe Java normale implantant l'interface
- doit étendre `java.rmi.server.UnicastRemoteObject`
- constructeurs doivent lever `java.rmi.RemoteException`
- si pas de constructeur, en déclarer un vide qui lève `java.rmi.RemoteException`

```
import java.rmi.server.UnicastRemoteObject;
import java.rmi.RemoteException;

public class CompteImpl extends UnicastRemoteObject
    implements CompteInterf {
    private String nom;
    private float solde;

    public CompteImpl(String nom) throws RemoteException {
        super();
        this.nom = nom;
    }

    public String getTitulaire() { return nom; }
    ... }
}
```

2. Modèle de programmation

Ecriture d'une classe implantant l'interface

1. Compilation de l'interface et de la classe avec `javac`
2. Génération de la souche cliente avec `rmic`

```
javac CompteInterf.java CompteImpl.java
rmic CompteImpl
```

Fichier généré

`CompteImpl_Stub.class` : souche cliente

Options ligne de commande `rmic`

`-d path` répertoire pour les fichiers générés

2. Modèle de programmation

Ecriture du programme serveur

1. Instanciation de la classe serveur
2. Enregistrement de l'instance dans le serveur de noms RMI

```
public class Serveur {
    public static void main(String[] args) throws Exception {
        CompteInterf compte = new CompteImpl("Bob");
        Naming.bind("Bob", compte);
    }
}
```

- `compte` prêt à recevoir des invocations de méthodes
- programme "tourne" en permanence tant que `compte` reste enregistré dans le runtime RMI

2. Modèle de programmation

Ecriture du programme client

1. Recherche de l'instance dans le serveur de noms
2. Invocation des méthodes

```
public class Client {  
    public static void main(String[] args) throws Exception {  
        CompteInterf compte = (CompteInterf) Naming.lookup("Bob");  
        compte.deposer(10);  
        System.out.println( compte.solde() );  
    }  
}
```

2. Modèle de programmation

Exécution des programmes

1. Lancer le serveur de noms (`rmiregistry`)
 - une seule fois
 - doit avoir accès au *bytecode* de la souche cliente (→ `CLASSPATH`)
2. Lancer le programme serveur
3. Lancer le programme client

2. Modèle de programmation

Compléments

Passage de paramètres avec RMI

- types de base (`int`, `float`, ...) par copie
- objets implémentant `java.io.Serializable` passés par copie (sérialisés)
- objets implémentant `java.rmi.Remote` passés par référence
 - la référence RMI de l'objet est transmise
- dans les autres cas une exception `java.rmi.MarshalException` est levée



2. Modèle de programmation

Compléments

Ecriture d'une classe d'objet serveurs

- héritage `UnicastRemoteObject`
- pas toujours possible (si classe ∈ à une hiérarchie d'héritage)

→ méthode

```
static UnicastRemoteObject.exportObject(Remote)  
enregistrement côté serveur
```

→ méthode de désenregistrement

```
static UnicastRemoteObject.unexportObject(Remote, boolean)  
false : attente fin les requêtes en cours / true : immédiat
```

2. Modèle de programmation

Compléments

Objets serveur prévus pour fonctionner avec ≠ types de liaisons

1. objet distant joignable en point à point
2. objets distants répliqués
3. objets distants joignables par diffusion

En pratique seul 1. mis en oeuvre (classe `UnicastRemoteObject`)

Références d'objets distants RMI avec `UnicastRemoteObject`

- adresse IP
- n° port TCP
- identifiant local d'objet (entier)

3. Services RMI

Services RMI

- service de nommage
- service d'activation d'objets
- ramasse-miettes réparti

2. Modèle de programmation

Compléments

Invocations concurrentes de méthodes RMI

Un objet serveur RMI est susceptible d'être accédé par plusieurs clients simultanément

⇒ toujours concevoir des méthodes RMI *thread-safe*
i.e. exécutable concurremment de façon cohérente

⇒ la création de *thread* est faite automatiquement par le *runtime* RMI

3. Services RMI

Service de nommage

Permet d'enregistrer des liaisons entre un objet serveur et un nom symbolique

- par défaut port 1099 (autre port : `rmiregistry` 12345)
- noms "plats" (pas de noms composés, pas de hiérarchies)

URL RMI `rmi://machine.com:1099/nomSymbolique`

`rmi://` et `:1099` facultatifs
`machine.com` par défaut `localhost`

Serveur de noms demarrable

- de façon autonome dans un *shell* avec l'outil `rmiregistry`
- dans un programme par appel de la méthode `static`
`java.rmi.registry.LocateRegistry.createRegistry(int port)`

3. Services RMI

Service de nommage

Classe `java.rmi.Naming` pour l'accès local

Toutes les méthodes sont `static`

- | | |
|---|-------------------------------------|
| 1. <code>void bind(String, Remote)</code> | enregistrement d'un objet |
| 2. <code>void rebind(String, Remote)</code> | ré enregistrement d'un objet |
| 3. <code>void unbind(String)</code> | désenregistrement d'un objet |
| 4. <code>String[] list(String)</code> | liste des noms d'objets enregistrés |
| 5. <code>Remote lookup(String)</code> | recherche d'un objet |

Les paramètres `String` correspondent à une URL d'objet RMI

- | | |
|-------|--|
| 1 2 3 | accessibles localement uniquement |
| 1 | lève une exception si le nom est déjà enregistré |
| 4 | URL du <code>rmiregistry</code> |

3. Services RMI

Service de nommage pour l'accès distant

Classe `java.rmi.registry.LocateRegistry`
`static Registry getRegistry(String host, int port)`

`java.rmi.registry.Registry` : même méthodes que `Naming`

3. Services RMI

Service d'activation d'objets

Objets serveurs activables à la demande (depuis JDK 1.2)
en fonction demande clients

- démon `rmid`
- package `java.rmi.activation`

Avantages

- évite d'avoir des objets serveurs actifs en permanence
(ie "tournant" dans une JVM)
 - trop coûteux si beaucoup d'objets serveurs
- permet d'avoir des références d'objets persistantes
 - en cas de *crash* d'objet serveur
le démon peut le relancer avec la même référence
 - les clients continuent à utiliser la même référence

3. Services RMI

Service d'activation d'objets

```
public class Serveur {  
    public static void main(String[] args) throws Exception {  
        ActivationGroupDesc gDesc = new ActivationGroupDesc(null, null);  
        ActivationGroupID gID =  
            ActivationGroup.getSystem().registerGroup(gDesc);  
        ActivationGroup.createGroup(gID, gDesc, 0);  
  
        MarshalledObject mo = new MarshalledObject("Bob");  
        ActivationDesc desc = new ActivationDesc("CompteImpl", "", mo);  
        CompteInterf compte = (CompteInterf) Activatable.register(desc);  
        Naming.bind("Bob", compte);  
  
    }  
}
```

Programme client inchangé

Lancer `rmiregistry` et **démon d'activation** (`rmid`)

3. Services RMI

Extension du principe : *pool* d'objets

Avoir un ensemble d'objets prêts à traiter les requêtes

Même problématique que le *pool* de *threads*

- ⇒ gestion de la taille du *pool* (fixe, variable)
- ⇒ à programmer

3. Services RMI

Ramasse-miettes réparti

Récupérer les ressources (mémoire, ...)
occupées par un objet que personne ne référence
→ i.e. que l'on ne pourra plus jamais accéder

Difficulté : environnement distribué donc référencement à distance

Dans une JVM : mécanisme *mark-and-sweep*

1. parcours du graphe de référencement des objets
2. destruction des objets non visités

Avec RMI : double mécanisme géré par le Remote Reference Manager

- comptage de référence : # de clients référençant l'objet
- bail (*lease*) : mémoire "louée" à l'objet pour un temps fini

3. Services RMI

Ramasse-miettes réparti

Comptage

- chaque transmission de référence +1
- chaque fin de référencement -1

Bail

- par défaut 10 min (réglable par la propriété `java.rmi.dgc.leaseValue`)
- but : se prémunir
 - partitions de réseaux
 - pertes de message de déréférencement

Si le compteur tombe à 0 ou si le bail expire,
l'objet devient candidat pour le ramassage local (*mark-and-sweep*)

3. Services RMI

Ramasse-miettes réparti

Attention : un objet serveur "normal"
instancié par un programme serveur qui "tourne" en permanence
est toujours référencé par ce programme

⇒ il n'est pas ramassé (même au delà des 10 min)

⇒ le ramassage concerne des objets créés dont on "perd" la référence

Client

```
BarRemote b = foo();  
...  
b = null;
```

Serveur

```
BarRemote foo() {  
    ...  
    return new BarRemote();  
}
```

4. Fonctionnalités additionnelles

- chargement dynamique de classes (`RMIClassLoader`)
- personnalisation des communications
- génération dynamique des souches

4. Fonctionnalités additionnelles

Chargement dynamique de classes

- Java charge les `.class` à la demande à partir du disque (`CLASSPATH`)
- RMI introduit en + un mécanisme de chargement des classes à distance par HTTP ou FTP

Avantage : classes déployées sur 1 seul site (+ rapide + simple à gérer)

Inconvénient : *single point of failure*

Utilisation

- propriété `java.server.rmi.codebase` : URL du serveur de téléchargement
- classe `RMISecurityManager`

4. Fonctionnalités additionnelles

Personnalisation des communications

RMI utilise des *sockets* TCP

- côté client et serveur
- attribuées automatiquement par défaut

Possibilité de personnaliser ces sockets pour

- forcer l'utilisation de *sockets* précises
- tracer les communications
- crypter et/ou signer les données
- introduire des traitements "à l'insu" des objets clients/serveurs RMI

⇒ redéfinir le constructeur

```
protected UnicastRemoteObject( int port,  
                               RMIClientSocketFactory csf, RMIServerSocketFactory ssf )
```

- ou utiliser `static UnicastRemoteObject.export (Remote, int, ...)`

4. Fonctionnalités additionnelles

Personnalisation des communications

```
interface RMIClientSocketFactory {  
    java.net.Socket createSocket( String host, int port );  
}
```

```
interface RMIServerSocketFactory {  
    java.net.ServerSocket createServerSocket( int port );  
}
```

⇒ déf. 2 classes qui implantent ces interfaces

⇒ déf. des sous-classes de `Socket` et `ServerSocket`

pour personnaliser le fonctionnement des sockets

```
ex : javax.net.ssl.SSLSocket, javax.net.ssl.SSLServerSocket
```

4. Fonctionnalités additionnelles

Génération dynamique des souches

- rmic génère les souches de communication
- A partir JDK 5, possibilité de générer les souches dynamiquement
 - génération de *bytecode* à la volée
 - chargement dynamique de la classe générée

Exemple de code généré pour la souche client

```
public final class CompteImpl_Stub
extends java.rmi.server.RemoteStub implements CompteInterf, java.rmi.Remote {
    private static final long serialVersionUID = 2;
    private static java.lang.reflect.Method $method_getTitulaire_0;
    static {
        try {
            $method_getTitulaire_0 =
                CompteInterf.class.getMethod(
                    "getTitulaire", new java.lang.Class[] {});
        } catch (java.lang.NoSuchMethodException e) {
            throw new java.lang.NoSuchMethodError(
                "stub class initialization failed"); } }
}
```

Java RMI

29

Lionel Seinturier

4. Fonctionnalités additionnelles

Génération dynamique des souches

Exemple de souche générée

```
public CompteImpl_Stub(java.rmi.server.RemoteRef ref) {
    super(ref);
}

public java.lang.String getTitulaire() throws java.rmi.RemoteException {
    try {
        Object $result = ref.invoke(
            this, $method_getTitulaire_0, null, 3068049948190996319L);
        return ((java.lang.String) $result);
    } catch (java.lang.RuntimeException e) {
        throw e;
    } catch (java.rmi.RemoteException e) {
        throw e;
    } catch (java.lang.Exception e) {
        throw new java.rmi.UnexpectedException("undeclared checked exception", e);
    }
}
```

Java RMI

30

Lionel Seinturier

5. Protocole

Protocole JRMP

2 possibilités pour acheminer les invocations de méthodes distantes

- JRMP : protocole Sun (*UnicastRemoteObject*) utilisé par défaut
- IIOP : protocole OMG pour CORBA

Messages JRMP sortants

- *Call* : véhicule une invoc. de méth. (+ *callData*)
- *Ping* : teste le bon fonctionnement d'un serveur

Messages JRMP entrants

- *Return* : véhicule le retour de l'invoc. (+ *returnValue*)
- *PingAck* : acquittement d'un message *Ping*

Java RMI

31

Lionel Seinturier

5. Protocole

Structure des paquets échangés par JRMP

4 octets	2 octets	1 octet	n octets
En-tête	Version	Protocole	Message(s)

En-tête : *magic number* (JRMI)

Version : numéro de version du protocole

Protocole : 3 possibilités

- SingleOpProtocol : 1 seule invoc. par paquet
- StreamProtocol : +sieurs invoc. **vers un même obj.** les unes à la suite
- MultiplexProtocol : +sieurs invoc. **vers une même machine** multiplexées sur la même connexion

Java RMI

32

Lionel Seinturier

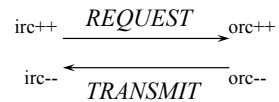
5. Protocole

Mécanisme de contrôle de flux JRMP

But : éviter qu'un buffer plein pour 1 connexion bloque toutes les autres
éviter qu'1 connexion qui ne se termine pas bloque toutes les autres
(par exemple en cas d'appels récursifs)

2 compteurs (en nombre d'octets) pour chaque cx RMI multiplexée

- *input request count* (irc)
- *output request count* (orc)



- irc et orc ne doivent jamais être négatifs
- irc ne doit pas dépasser une valeur (en nb d'octets) qui le bloquerait

5. Protocole

RMI/IIOP

Modèle de programmation

⇒ permet interopérabilité RMI - CORBA

- les classes doivent étendre `javax.rmi.PortableRemoteObject`
- les souches doivent être générées avec `rmic -iiop`
2 fichiers générés pour chaque classe d'objet serveur
`_nomClasse_Stub.java` : souche cliente
`_nomClasse_Tie.java` : souche serveur
- service de nommage JNDI (Java Naming and Directory Interface)
- outil `tnameserv`
- les conversions de type sur les objets distants doivent utiliser la méthode `static PortableRemoteObject.narrow()`

Bibliographie

- **Java RMI**. W. Grosso. O'Reilly.
- **jGuru RMI Tutorial**
<http://developer.java.sun.com/developer/onlineTraining/rmi/RMI.html>
- **Guide RMI**. Sun.
<http://java.sun.com/products/jdk1.4/docs/guide/rmi/>
- **RMI-IIOP**. IBM.
<http://www.ibm.com/java/jdk/rmi-iiop/>