

La technologie RMI

Table des matières

1	Présentation de RMI	1
2	Déployer des objets serveurs	2
2.1	Spécification de la couche métier	2
2.2	Fourniture d'une implantation	3
2.3	La classe <code>UnicastRemoteObject</code>	3
2.4	Architecture client/serveur	4
2.5	Génération des classes d'interface	4
2.6	Registre d'inscription RMI	5
2.6.1	Nommage des objets distribués	5
2.6.2	La classe <code>java.rmi.Naming</code>	6
3	Accès aux objets distribués	6
3.1	Architecture des serveurs RMI	6
3.2	Passage de paramètres	7
3.3	Création de nouveaux objets	7
4	Le ramasse miettes distribué	7
5	Chargement dynamique de classes	8
6	Implanter une politique de sécurité	8
7	Autres fonctions des RMI	9

1 Présentation de RMI

RMI = *Remote Method Invocation* (Appel de procédures à distance).

RMI est basé sur

- IP,
- TCP,
- JRMP (pour Java Sun),
- RMI-IIOP (interopérabilité CORBA),

Objectifs :

- Simple (les aspects réseau sont transparents),
- Développement d'une couche métier,
- Architecture *n*-tiers,
- Mise en place de bus logiciels,

Note : La technologie RMI est une solution simple pour implanter un bus logiciel. Il existe néanmoins deux inconvénients :

- Le premier est lié au fait que cette technologie n'est supportée que par Java. Elle ne peut donc pas être utilisée pour réaliser un serveur métier multi-plateforme. Les différents ponts logiciels vers CORBA ou DCOM réduisent un peu cet inconvénient.
- Le second est lié à la lenteur relative de cette technologie due notamment à la sérialisation des paramètres.

2 Déployer des objets serveurs

Le déploiement d'un objet serveur est réalisé en cinq étapes :

- Spécification de l'interface du service distant.
- Rédaction d'un implantation.
- Préparation d'un annuaire RMI.
- Création d'une instance de l'objet serveur.
- Inscription de cette instance dans l'annuaire RMI.

2.1 Spécification de la couche métier

Spécification du *javabean* permettant de représenter une voiture :

```

package rmisample;

import java.io.Serializable;

public class Car implements Serializable {

    private static final long serialVersionUID = 1L;
    private int number;
    private String model;

    // getters and setters for properties number and model
    ...
}

```

et la spécification d'une usine à voiture :

```

package rmisample;

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

public interface ICarFactory extends Remote {

    Car newCar() throws RemoteException;

    int carCounter() throws RemoteException;

}

```

Note : La définition d'un objet serveur se fait en deux temps. On commence par définir l'interface accessible au client puis on passe à la réalisation. Les interfaces RMI ont une seule contrainte à respecter : les méthodes doivent pouvoir générer une erreur réseau. En effet, personne ne peut garantir l'absence d'erreur quand on passe par une couche réseau.

2.2 Fourniture d'une implantation

Contraintes :

- étendre la classe `UnicastRemoteObject`,
- traiter les erreurs d'exécution,

```
package rmisample;

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

public class CarFactoryImpl extends UnicastRemoteObject implements ICarFactory {

    private static final long serialVersionUID = 1L;
    private String model;
    private int counter = 0;

    public CarFactoryImpl(String model) throws RemoteException {
        super();
        this.model = model;
    }

    public Car newCar() {
        Car c = new Car();
        c.setModel(model);
        c.setNumber(++counter);
        return c;
    }

    public int carCounter() {
        return counter;
    }

}
```

2.3 La classe `UnicastRemoteObject`

Quelques méthodes de `java.rmi.server.UnicastRemoteObject`¹ :

`UnicastRemoteObject()`

Exportation de l'objet `this`.

`exportObject(Remote obj)`

Exporter un objet distant.

`exportObject(Remote obj, int port)`

Exporter un objet distant sur un port spécifié.

`unexportObject(Remote obj, boolean force)`

défaire une exportation.

`getClientHost()`

pour obtenir le nom du client.

`getLog()/setLog(OutputStream out)`

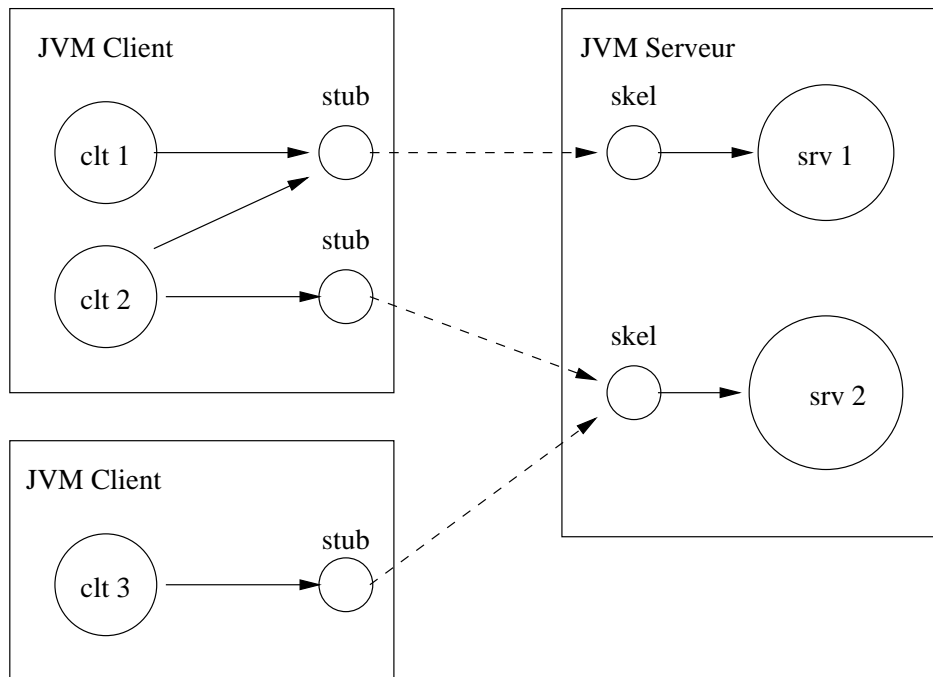
générer une trace de l'utilisation de l'objet.

Note : Il est fortement conseillé de lire attentivement la JavaDoc de cette classe.

¹<http://java.sun.com/javase/7/docs/api/java/rmi/server/UnicastRemoteObject.html>

2.4 Architecture client/serveur

Organisation des JVM clients et serveur :



Note : Dans une application RMI les objets clients accèdent aux méthodes des objets serveurs de manière tout à fait naturelle. Les objets clients travaillent comme si les objets serveurs étaient dans la même JVM. De la même manière, les objets serveur ont l'impression d'être utilisés par des clients situés dans la même JVM.

Dans la pratique, les objets clients dialoguent avec des objets « stub » qui se chargent des transmissions réseau. De l'autre côté les « skel » reçoivent les requêtes et les retransmettent vers les objets serveur. **Moralité** : les classes serveurs et clients sont débarrassés de la tâche réseau. Leur travail est donc plus simple.

2.5 Génération des classes d'interface

Génération automatique des « stub » et « skel » avec

```
$ rmic nom_de_la_classe_d'implantation
```

exemple :

```
$ rmic -classpath classes/ -d classes/ rmisample.CarFactoryImpl
```

Production des classes :

- rmisample/CarFactoryImpl_Stub.class
- rmisample/CarFactoryImpl_Skel.class

Note : Le compilateur RMI (rmic) se charge de générer automatiquement les classes stub et skel à partir de la définition des classes serveur. N'oubliez pas de lire le manuel de rmic (<http://java.sun.com/javase/7/docs/tooldocs/solaris/rmic.html>).

Important : depuis la version 1.4 la création des classes skel n'est plus nécessaire. Depuis la version 1.5 la création des classes skel et stub n'est plus nécessaire. Des Proxies assurent automatiquement l'envoi et la réception des requêtes.

2.6 Registre d'inscription RMI

Le lancement des serveurs doit être précédé par la mise en place d'un registre RMI. Le but d'un registre RMI est d'offrir un annuaire des objets serveurs disponibles en associant, à chaque objet serveur, un nom.

```
$ rmiregistry &
```

Autre solution, le code de lancement des serveurs assure la mise en place d'un registre :

```
java.rmi.registry.LocateRegistry.createRegistry(1099);  
... code de lancement des serveurs ...
```

Un registre est un objet distant qui peut être récupéré via la classe `LocateRegistry` et interrogé :

```
java.rmi.registry.Registry r =  
    LocateRegistry.getRegistry("monserveur", 1099) ;  
String[] name = r.list();
```

Note : N'oubliez pas de lire le manuel d'utilisation de la commande `rmiregistry`.
(<http://java.sun.com/javase/7/docs/tooldocs/solaris/rmiregistry.html>)

2.6.1 Nommage des objets distribués

Code du lancement du serveur RMI :

```
package rmisample;  
  
import java.rmi.registry.LocateRegistry;  
import java.rmi.registry.Registry;  
  
public class CarFactoryServer {  
  
    public static void main(String[] args) {  
        try {  
            // create a RMI registry  
            Registry r = LocateRegistry.createRegistry(1099);  
  
            // create and publish car factory server object  
            CarFactoryImpl meganeFactory = new CarFactoryImpl("Megane");  
            r.rebind("carFactory", meganeFactory);  
        } catch (Exception e) {  
            e.printStackTrace();  
        }  
        System.err.println("CarFactoryServer is running.");  
    }  
}
```

Pour publier un service RMI sur un annuaire indépendant situé sur une autre machine vous pouvez utiliser :

```
java.rmi.Naming.bind("rmi://nom-de-l-annuaire-rmi/carFactory", meganeFactory);
```

L'URL est de la forme (le port est optionnel) :

```
rmi://nom_du_serveur_RMI:numéro_de_port/nom_d'objet
```

Le port par défaut est 1099.

Note : Pour que les objets serveur soient utilisables, nous devons les créer et les publier en leur donnant un nom. Les clients pourront ensuite utiliser ce nom pour retrouver ces objets. Ce nom va être déposé dans un annuaire.

2.6.2 La classe `java.rmi.Naming`

Quelques méthodes de `java.rmi.Naming`² :

`bind(String name, Remote obj)`

Nommage de l'objet distant et introduction dans le registre RMI.

`rebind(String name, Remote obj)`

Remplacement d'objet dans le registre RMI.

`Remote lookup(String name)`

Recherche un objet distant dans le registre RMI.

`unbind(String name)`

Supprimer l'association.

3 Accès aux objets distribués

```
package rmisample;

import java.rmi.Naming;
import java.rmi.RemoteException;
import java.rmi.registry.LocateRegistry;
import java.rmi.registry.Registry;

public class Client {

    public static void main(String[] args) throws Exception {
        String url = "rmi://annuaire-RMI/carFactory";
        ICarFactory carFactory = (ICarFactory) Naming.lookup(url);

        Car c = carFactory.newCar();
        System.out.println("car model : " + c.getModel());
        System.out.println("car number : " + c.getNumber());
        System.out.println("cars counter : " + carFactory.carCounter());
    }
}
```

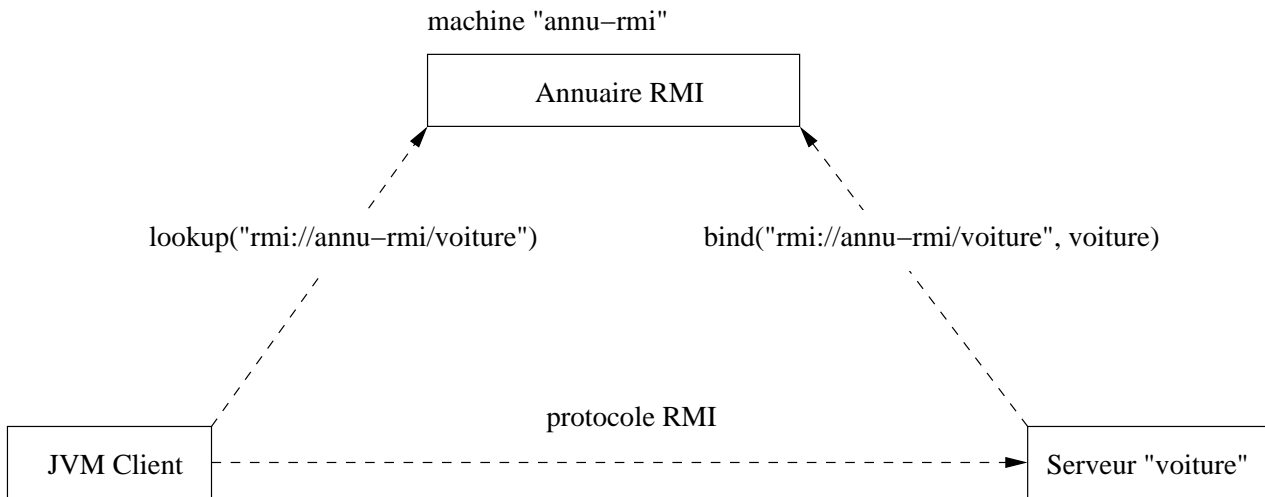
Le code est indépendant de la nature et de la localisation des objets.

3.1 Architecture des serveurs RMI

Pour résumer, il existe trois types de machines :

- les registres RMI : ils sont connus des serveurs et des clients,
- les serveurs RMI : ils ne sont connus de personne et ils se déclarent auprès du registre (méthode `bind`),
- les clients RMI : ils recherchent les objets distants auprès du registre (méthode `lookup`).

²<http://java.sun.com/javase/7/docs/api/java/rmi/Naming.html>



3.2 Passage de paramètres

Étude de cas :

- Les types primitifs sont passés par valeur.
- Les objets non distants sont passés par valeur. Ils doivent donc être sérialisables (implanter l'interface `java.io.Serializable`).
- Pour les objets distants, une référence est passée vers le « skel » distant (`java.rmi.server.RemoteRef`). Le code utilisateur récupère une référence vers la classe « stub » qui implante l'interface initiale.

Note : Bien entendu, la JavaDoc³ reste indispensable.

3.3 Création de nouveaux objets

Un client ne peut créer un nouvel objet distant. Par contre un serveur peut créer un nouvel objet et renvoyer sa **référence distante** au client :

```

...

public class CarFactoryImpl
    extends UnicastRemoteObject
    implements ICarFactory {

    ....

    public ICarFactory newFactory(String model) throws RemoteException {
        CarFactoryImpl v = new CarFactoryImpl(model);
        return v;
    }

}
  
```

Note : Il faut remarquer que cette nouvelle usine n'a pas d'URL. Ou, en d'autres termes, elle n'est pas inscrite dans un annuaire RMI. En fait, cette usine est uniquement accessible via la référence distante renvoyée par la méthode `newFactory`.

4 Le ramasse miettes distribué

Les objets distants non référencés sont automatiquement supprimés du serveur. Ils peuvent en être informé par :

³<http://java.sun.com/javase/7/docs/api/java/rmi/server/RemoteRef.html>

```

package rmisample;

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

public class CarFactoryImpl extends UnicastRemoteObject
    implements ICarFactory, Unreferenced {

    ...

    public void unreferenced() {
        System.out.println("plus de référence pour l'usine de " + model);
    }

}

```

Les objets placés dans un registre RMI sont toujours référencés. La fréquence de nettoyage du DGC est réglable via une propriété :

```
java -Djava.rmi.dgc.leaseValue=10000 fr.monServeur
```

Note : Comme dans le cas de l'usine précédente, il arrive qu'un objet serveur devienne inutile. C'est le cas quand aucun client ne possède de référence distante vers ce serveur.

5 Chargement dynamique de classes

Partons du principe que le code Java est rangé sur un serveur WEB à l'adresse

```
http://monserveur.fr/test/
```

Le lancement du serveur devient :

```
java -Djava.rmi.server.codebase=http://monserveur.fr/test/ fr.monServeur
```

Attention : le « / » à la fin est important.

Note : Ce mécanisme est très important car, dans les exemples précédents, nous sommes partis du principe que le client a accès aux classes Car et ICarFactory. Si tel n'est pas le cas, alors le client peut automatiquement télécharger le code Java de ces classes.

6 Implanter une politique de sécurité

Pour empêcher les chargements de code dangereux en provenance de machine inconnue :

```

System.setSecurityManager(new java.rmi.RMISecurityManager());

java.rmi.Naming.bind("rmi://localhost/carFactory",
    new CarFactoryImpl());

...

```

ou bien :

```
java -Djava.security.policy=./java.policy fr.monServeur
```

Note : Vous devez lire la Javadoc⁴ de la classe `RMISecurityManager` qui implante la stratégie de sécurité.

Note : Pour rédiger le fichier `java.policy`, vous devez lire le document de description des permissions⁵.

7 Autres fonctions des RMI

Quelques possibilités :

- Les « callbacks » distants.
- L'utilisation d'annuaire JNDI pour associer des noms à des objets.
- L'interopérabilité avec CORBA, c'est à dire la mise en place d'une passerelle pour
 - appeler des objets CORBA à partir de client Java ;
 - appeler des objets RMI à partir de client CORBA.
- L'encapsulation du protocole RMI dans des connexions HTTP et/ou SSL.
- L'activation dynamique d'objets.

⁴<http://java.sun.com/javase/7/docs/api/java/rmi/RMISecurityManager.html>

⁵<http://java.sun.com/javase/6/docs/technotes/guides/security/permissions.html>