

Java : RMI

B. Dupouy

Plan

- Présentation
- Exemple de base
- Exemple 2 : Agent mobile

..... Présentation

- Exemple de base
- Exemple 2 : Agent mobile

Objectif

- Définir une architecture distribuée qui s'intègre dans le langage de façon transparente, donc qui permette :
 - l'invocation de méthodes sur des objets situés sur des machines virtuelles différentes (cette invocation doit être similaire à une invocation locale)
 - l'utilisation de l'environnement de sécurité Java classique lors de l'exécution d'applications distribuées
 - l'affranchissement du programmeur de la gestion de la mémoire, en définissant une extension distribuée au *garbage collecting*

Modèle objet de RMI

- Un objet distant (*Remote Object*) est un objet dont les méthodes peuvent être invoquées par des clients situés sur des machines virtuelles distantes,

• Cet objet distant est manipulé au travers d'une ou plusieurs **interfaces**, appelées *Remote Interfaces* qui contiennent les méthodes que les clients pourront invoquer sur ces objets distants,

- Une référence distante (*Remote Reference*) est une référence qui permet d'adresser un objet situé sur une machine virtuelle différente,

- Une invocation de méthode distante (*Remote Method Invocation*) invoque une méthode définie dans une interface d'objet distant. Elle a la même syntaxe qu'une invocation de méthode locale.

Différences appel local/distant

- Rappel : les clients manipulent les objets qui implémentent les *remote interfaces*, mais pas directement les objets qui implémentent ces interfaces,
- Les paramètres de type simple (`int`, ...) sont passés par copie
- Les objets locaux sont sérialisés (`serializable`) et passés par copie: différence avec le modèle de Java, dans lequel les objets sont passés par référence,
- Les objets distants sont passés par référence : les méthodes sont bien exécutées sur l'objet lui-même, quelle que soit la machine virtuelle dans laquelle il réside
- Les méthodes distantes provoquent des exceptions supplémentaires : les clients doivent traiter des exceptions liées au caractère **distribué** des méthodes qu'ils invoquent,
- Le bytecode n'est transmis à la JVM qu'au moment de son utilisation

rmic

- Une fois qu'on a écrit une classe qui **implémente** des interfaces, on crée le stub et le squelette correspondants grâce à `rmic` :
 - ajout de `_Stub` au nom de la classe pour le stub, et `_Skel` pour le squelette (skeleton).
 - association de numéros aux méthodes,

- Notes :

l'option `keepgenerated` conserve les sources des stub et squelette

`Object` contient une méthode `getClass()` qui donne la classe de l'objet : en ajoutant les suffixes adéquats, on obtient les noms du stub et du squelette,

rmiregistry : *protocole RMI*

- RMI assure les fonctions de localisation, communication et chargement de bytecode
- **Enregistrement** pour la **Localisation** des objets distants :
 - le serveur enregistre les méthodes qu'il exporte auprès de `rmiregistry` en utilisant `bind`,
 - le client trouve des références sur ces méthodes exportées et récupère les stubs correspondants en donnant le nom des objets correspondants lors de l'appel à `lookup`,
- Communication transparente grâce au protocole RMI
- Téléchargement de code transparente grâce au protocole RMI et à l'utilisation de serveurs web

- Présentation
- Exemple de base
- Exemple 2 : Agent mobile

Exemple de base : mise en place

- Soit les fichiers : `Hello.java`, `HelloServeur.java` du côté serveur, et `HelloClient.java` du côté client.

Contenu de `Hello.java` : l'interface (ce que voit le client):

```
public interface Hello extends java.rmi.Remote
{
    String lireMessage() throws java.rmi.RemoteException;
}
```

Remote signifie que les méthodes de cet objet `Hello` peuvent être appelées depuis une JVM autre que la JVM locale.

`HelloServeur.java` implémente cette interface :

```
public class HelloServeur extends UnicastRemoteObject
    implements Hello
...

```

Exemple de base: Le serveur

```

...
// Creation de l'objet qui va etre invoque par les clients
HelloServeur MonServeur = new HelloServeur();

// On publie le service au serveur de noms : rmiregistry.
// La structure d'un nom est : machine:port/nom
// "machine" est le nom de la machine ou tourne le serveur
// et "nom" correspond au nom du service.
// "port" est le numero de port utilise par
// rmiregistry pour attendre les requetes destinees au
// service de noms.

myHostname machine = new myHostname();

String nomService = "/" + machine.QualifiedHost()
                  + ":" + args[0] + "/HelloServeur";

Naming.rebind(nomService, MonServeur);

```

...

- Après compilation du serveur, pour avoir le stub (HelloServeur_Stub.class) destiné aux clients et le squelette (HelloServeur_Skel.class), il faut faire:
rmic HelloServeur

Exemple de base: Le client

- Voici comment invoquer une méthode sur l'objet distant :

```

String nomService = "/" + Machine + ":"
                  + portRmiregistry + "/HelloServeur";
Hello obj = (Hello) Naming.lookup (nomService);

// On peut maintenant invoquer la methode de l'objet
// qui est heberge par le serveur

message = obj.lireMessage ();

```

- On rappelle l'interface :

```

public interface Hello extends java.rmi.Remote
{
    String lireMessage() throws java.rmi.RemoteException;
}

```

Exemple de base : exécution

- En étant toujours dans le même répertoire :
 - lancer `rmiregistry` en arrière plan,
 - lancer le serveur sur la même machine (LaMachine) que celle où tourne `rmiregistry`,
 - lancer le client sur une machine quelconque, mais en étant dans le **même répertoire** pour retrouver les stubs :

```
java HelloClient LaMachine
```
- Le partage de code (pour les stubs) est assuré par `nfs`, puisque le client et le serveur, même s'ils ne sont pas sur la même machine, partagent **le même disque**... Dans le cas général, il faut faire transiter les stubs par un serveur `http`.

Ecriture d'une application

COTE SERVEUR :

- Ecrire :
 - les interfaces "prototypant" les méthodes distantes (`extends java.rmi.Remote`)
 - les objets qui les implémentent
- Lancer :
 - `rmic` : création du stub et du squelette à partir des fichiers `.class` des implémentations,
 - `rmiregistry` : l'enregistrement des méthodes distantes sera fait lors de l'appel à `bind`
- Rendre accessibles par un **serveur web** les objets à télécharger (stubs, bytecode),

COTE CLIENT :

- l'appel à `lookup` passera un stub au client

**Exemple :
lancement d'un service**

- Voici la commande pour lancer un serveur :

```
java -
  Djava.rmi.server.codebase=http://www.enst.fr/~$USER/tp\
-Djava.rmi.server.hostname=$HOST\
-Djava.security.policy    =java.policy Hote_implem $*
```

-server.codebase donne le nom du serveur web d'où les stubs seront téléchargés,

-server.hostname donne le nom de la machine où se trouve le serveur

-java.policy donne les droits d'accès qui seront vérifiés par le security manager, voici le contenu de ce fichier :

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

- Présentation
- Exemple de base
- Exemple 2 : Agent mobile

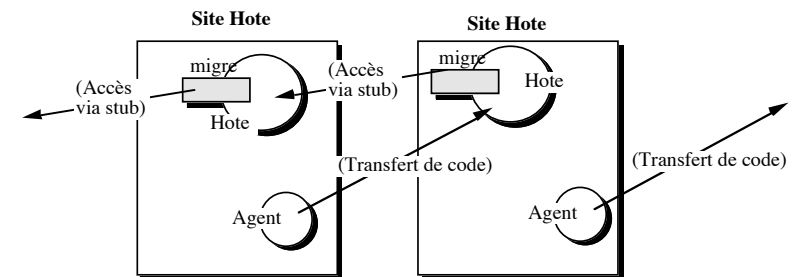
Exemple 2: agent mobile

- Un "client" voulant acheter un produit va créer un agent au lieu d'interroger lui-même tous les sites détenteurs de magasins vendant le produit.
- L'agent donnera au "client" le nom du site qui propose le meilleur prix.
- Mise en oeuvre :

Le client va lancer sa demande depuis un site appelé "initiateur". Il initialise un tableau de sites à parcourir, puis **invoque à distance** sur le premier site la méthode **migre**, à laquelle il a passé l'agent en argument.

Exemple 2 : agent mobile

- Un "client" va envoyer un agent interroger successivement plusieurs serveurs (Hotes)
- On donne ci-dessous le circuit parcouru par l'agent :

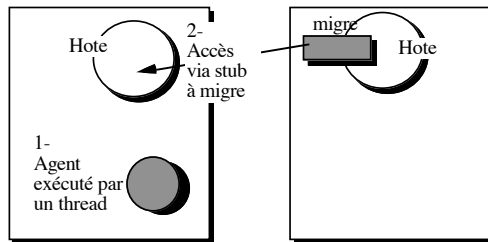


Remarquer les différents objets téléchargés :

- l'agent, simple bytecode,
- le stub qui permet l'exécution distante de la méthode migre

Le serveur

- Le serveur exécute la fonction `migre` pour le client : il récupère alors le code de l'agent. Il le fait exécuter par un thread et appelle lui-même `migre` sur le site suivant :



- Voici l'interface fournie par le serveur (`Hote.java`):

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

```
public interface Hote extends Remote {
    void migre(Agent a) throws RemoteException;
}
```

Remarque : le code d'Agent n'est **pas présent** lors de l'écriture de ce serveur

Le serveur

- Implémentation de l'interface (`Hote_imlem.java`):

La méthode `migre` invoquée à distance crée un thread : elle rend donc la main sans attendre la fin du traitement de l'agent (exécution puis migration).

```
public class Hote_imlem
    extends UnicastRemoteObject    implements Hote
public Hote_imlem(String fichier) throws RemoteException
{
    super();
    ...
}

public void migre(Agent a)
{
    threadAgent monThread =
        new threadAgent(a, monMagasin);
    monThread.start();
}
...

```

L'agent

- `Agent.java` est l'interface pour les agents. L'objet sera implémenté par le "**client**" (initiateur).

...

```
import java.io.Serializable;  
public interface Agent extends Serializable {  
    // Traitement effectue par l'agent sur chaque hote  
    void traitement(...);  
  
    // Affiche le resultat des traitements : effectue par  
    // l'agent lorsqu'il revient sur le site initiateur  
    void afficheResultat();  
  
    // Renvoie le nom de l'hote suivant a visiter par l'agent  
    String hoteSuivant();  
}
```

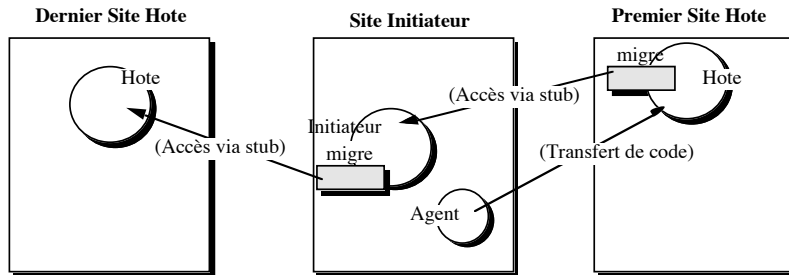
- `Serializable` indique que les paramètres utilisés seront sérialisés et normalisés lors de l'appel distant (marshalling).

Le thread qui gère l'agent

- `threadAgent` est le support système offert par un `Hote` pour traiter un agent : cette classe définit le thread qui va être lancé chaque fois qu'un agent arrive sur un `Hote`.

```
...  
class threadAgent extends Thread  
...  
public void run()  
...  
    // On effectue ici le traitement demande par l'agent  
    ...  
    // On fait ensuite migrer l'agent vers l'hote suivant  
    ...  
    Hote hote = (Hote) Naming.lookup(leSuivant);  
    hote.migre(monAgent);  
    ...
```

L'initiateur



- Implémentation de l'initiateur, c'est à dire du site "client" qui **créé** et lance l'objet vers les différents serveurs. La méthode **migre** est **particulière** : elle ne déclenche pas l'exécution de l'agent mais lui demande d'afficher les résultats obtenus.

```
public class Initiateur
    extends UnicastRemoteObject implements Hote
    ...
public void migre(Agent a)
{
    a.afficheResultat();
    ...
public static void main(String args[])
{
    ...
    agentIngredient agent = new agentIngredient(...);
    hote.migre(agent);
    ...
}
```

L'initiateur

- Voici la commande pour lancer l'initiateur :

```
java
-Djava.rmi.server.codebase=http://www.enst.fr/~$USER/tpmi
-Djava.rmi.server.hostname=$HOST
-Djava.security.policy=java.policy  Initiateur $*
```

- server.codebase donne le nom du serveur web d'où l'agent sera téléchargé,
- server.hostname donne le nom de la machine où se trouve l'initiateur
- le fichier java.policy donne les droits d'accès qui seront vérifiés par le security manager

Annexe 1

- Extraits de la fonction main d'un Hote

```
public class Hote_implem extends UnicastRemoteObject
    implements Hote
...
public static void main(String[] args)
...
System.setSecurityManager(new mySecurity());
...
// Creation de l'objet reparti et publication
Hote monHote = new Hote_implem(args[0]);
Naming.rebind("Hote", monHote);
...
```

- Extraits de la fonction main de l'initiateur

```
public class Initiateur extends UnicastRemoteObject
    implements Hote
...
System.setSecurityManager(new mySecurity());
...
// Creation de l'objet reparti et publication Hote
monInitiateur = new Initiateur();
Naming.rebind("Initiateur", monInitiateur);

// Recherche du nom de service du premier site
// a visiter et obtention d'un stub
String name = "/" + args[1] + "/Hote";
Hote hote = (Hote) Naming.lookup(name);
...
// Creation de l'agentIngredient
agentIngredient agent = new agentIngredient(...);
// Migration de l'agent
hote.migre(agent);
...
```

Annexe 2

- Extraits de la fonction **run** de l'Agent

```
public void run()
{
...
// On effectue le traitement demandé par l'agent
monAgent.traitement(...);

// On questionne l'agent pour savoir quel est
// l'hote suivant a visiter
String leSuivant = monAgent.hoteSuivant();

// On fait migrer l'agent vers l'hote suivant
Hote hote = (Hote) Naming.lookup(leSuivant);
hote.migre(monAgent);
...
}
```