

## Systemes répartis : *Remote Method Invocation*

Fabrice Rossi

<http://apiacoa.org/contact.html>.

Université Paris-IX Dauphine

1. Introduction
2. JRMP vs IIOP
3. *Serialization*
4. *Remote*
5. programmation RMI avec JRMP
6. programmation RMI avec IIOP

Systemes répartis : *Remote Method Invocation* – p.1/74

## *Remote Method Invocation*

En gros, RMI est l'implémentation en Java de RPC orientés objet. Principes de base :

- un programme Java peut obtenir une "référence" vers un objet qui n'existe pas dans la JVM qui exécute ce programme : c'est un **objet distant**
- un objet distant s'utilise exactement (ou presque) comme un objet local, grâce aux *stubs*

Comme tout système de RPC, RMI utilise :

- une technique de représentation portable des objets
- un protocole (couche transport et messages utilisés)
- un outil de publication et de résolution

Comme RMI est spécifique à Java, il n'utilise pas de langage de description des objets, mais se base directement sur le code.

Systemes répartis : *Remote Method Invocation* – p.3/74

Systemes répartis : *Remote Method Invocation* – p.2/74

## Rôle des *Stubs* (Souches)

Les souches :

- *stub*
- souvent appelé squelette (*skeleton*) du côté serveur
- sur le client :
  - paramètres → format portable (*marshalling*)
  - format portable → résultat (*unmarshalling*)
  - protocole côté client (identification de l'objet et de la méthode appelée)
- sur le serveur :
  - traductions inverses de celles du client
  - protocole côté serveur (appel de la bonne méthode sur le bon objet)
- en général engendrées automatiquement

Systemes répartis : *Remote Method Invocation* – p.4/74

## Publication et découverte

Un problème standard des systèmes distribués :

- pour le serveur, se faire connaître (publication)
- pour le client, trouver le serveur (découverte et résolution)

Plusieurs niveaux :

1. partie facile : nom de la machine du serveur + DNS → connexion possible (TCP ou UDP par exemple)
2. quelle couche transport ?
3. pour TCP/UDP, quel numéro de port ?

Solutions proposées par RMI :

- le *registry* en RMI-JRMP
- *COSNaming* en RMI-IIOP

## Deux versions

Il existe deux versions de RMI :

1. RMI-Java Remote Method Protocol (RMI-JRMP) :
  - la version d'origine (toujours utile et développée)
  - complètement spécifique à Java
  - la mieux intégrée au langage
  - la plus simple d'utilisation
2. RMI-Internet InterORB Protocol (RMI-IIOP)
  - la plus récente
  - compatible avec CORBA
  - moins intégrée au langage
  - un peu plus délicate à mettre en œuvre

## Principes communs

Les deux versions de RMI sont assez proches et partagent :

- description des objets :
  - directement en Java
  - deux catégories : objets distants et objets passés par valeur
  - objets distants : doivent implémenter l'interface `Remote` (`java.rmi`)
  - objets passés par valeur : doivent implémenter l'interface `Serializable` (`java.io`)
  - les types fondamentaux sont gérés de façon transparente
- la représentation des objets :
  - totalement automatique grâce au mécanisme de sérialisation (sauvegarde d'objets) de Java
  - les formats sont différents, mais cela n'induit aucune différence au niveau du programmeur

## Différences

Quelques différences entre les deux RMI qui induisent des codes différents :

- les services de publication et de résolution sont différents
- pour implémenter un objet distant, on doit définir différentes méthodes techniques qui sont déjà proposées dans :
  - `UnicastRemoteObject` pour RMI-JRMP
  - `PortableRemoteObject` pour RMI-IIOP
- RMI-JRMP propose un *Garbage Collecting* réparti (DGC), alors que la gestion de mémoire est explicite en avec CORBA (RMI-IIOP)
- beaucoup d'autres différences assez techniques et pointues :
  - paramétrage de la couche transport
  - objets activables
  - etc.

## La serialization

- mécanisme Java qui permet de transformer un objet en un flux d'octets (et *vice versa*)
- joue le rôle de XDR pour RMI
- cas simple vraiment élémentaire : pour pouvoir *sérialiser* un objet, il suffit que sa classe implémente `Serializable`
- `Serializable` est une interface vide
- en général, les objets de l'api standard sont `Serializable`
- exemple :

```
1 import java.io.Serializable;
2 public class Personne implements Serializable {
3     public String prénom;
4     public String nom;
5 }
```

- pas de sérialisation pour les objets locaux (e.g. `FileInputStream`)

## Définition d'un objet distant

Tout se passe au niveau interface. Pour définir un objet distant, on écrit une interface qui vérifie les contraintes suivantes :

- l'interface **doit** étendre l'interface `Remote`
- toutes les méthodes de l'interface **doivent** indiquer qu'elles peuvent renvoyer l'exception `RemoteException` (`java.rmi`) ou une de ses superclasses (cas intéressant, `IOException` dans `java.io`)
- les paramètres et résultats des méthodes de l'interface **doivent** être ou bien `Serializable` ou bien `Remote` (on peut mélanger les deux)

## La serialization (2)

Le mécanisme de base est très puissant :

- gestion automatique des références entre objets
- gestion automatique des types
- gestion de version
- facile à adapter aux besoins :
  - les variables d'instance déclarées `transient` ne sont pas sauvegardées (remarque : les variables `static` ne sont pas non plus sauvegardées)
  - on peut aussi donner explicitement la liste des variables à sauvegarder
  - on peut définir `readObject` et/ou `writeObject` pour effectuer des actions particulières lors de la lecture et/ou de l'écriture d'un objet

Pour des besoins spécifiques : `Externalizable`, mais il faut tout faire à la main !

## Exemple

```
1 import java.rmi.Remote;
2 import java.rmi.RemoteException;
3 public interface IHelloWorld extends Remote {
4     public String sayHelloTo(String firstname,
5                             String lastname)
6         throws RemoteException;
7 }
```

Pas de problème car `String` est `Serializable`

## Passage par valeur

**Enorme** différence entre une RMI et un appel classique, **les objets non Remote sont passés par valeur** :

- les objets sont représentés de façon binaire grâce au mécanisme de la *serialization*
- si une méthode de l'objet distant modifie un paramètre passé par valeur, le client ne voit pas la modification
- tout objet qui implémente *Remote* est distant ⇒ équivalent à un passage par référence
- attention, un paramètre ou un résultat est distant seulement s'il apparaît en tant qu'interface ⇒ source de bugs subtils

## Exemple

```
1 import java.rmi.Remote;
2 import java.rmi.RemoteException;
3 public interface ISalutation extends Remote {
4     public HelloWorld hello(String language)
5         throws RemoteException;
6 }
```

- HelloWorld implémente *IHelloWorld*
- passage par valeur → pas d'appel distant à partir de l'objet obtenu !
- subtilités liées à la *serialization*
- il faut être très attentif (cf la suite pour d'autres exemples) !

## JRMP

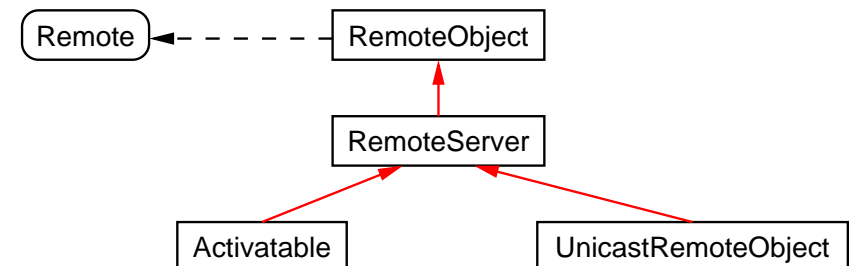
Plan :

1. Implémentation du serveur
2. Le *registry*
3. *Stubs* : *rmic*
4. Implémentation du client
5. Déploiement automatique et *Proxy*
6. Modes de passage
7. *DGC*
8. Gestion des erreurs

## Programmation de l'objet distant

On implémente l'interface pour fabriquer l'objet distant. Avec RMI-JRMP :

- solution simple : hériter de *UnicastRemoteObject*
- objets activables : hériter de *Activatable*
- solutions plus complexes : hériter de *RemoteObject* ou de *RemoteServer*, ou même de rien du tout !
- diagramme de classes :



## Fonctionnalités de UnicastRemoteObject

- paquet `java.rmi.server`
- s'occupe de tout !
- couche transport : TCP (avec *tunneling* HTTP)
- les constructeurs de `UnicastRemoteObject` exportent l'objet (le rendent visible depuis une autre JVM)
- attention, les constructeurs lancent `RemoteException`
- on peut préciser :
  - le port
  - des usines (*Factory*) pour les sockets utilisées (permet de coder les connexions par SSL par exemple)
- `UnicastRemoteObject` propose des méthodes de classe pour exporter un objet qui se contente d'implémenter l'interface `Remote` : permet d'éviter l'héritage

## Publication et résolution

- Deux solutions dans RMI :
  - un programme extérieur :
    - responsable de l'association nom (chaîne de caractères structurée) vers objet distant (de façon sous-jacente un port, etc.)
    - protocole RMI
    - le *registry* pour JRMP
    - le *Naming Service* (COSNaming) de CORBA pour IIOP
  - le passage par "référence" :
    - tout objet `Remote` est transmis comme objet distant
    - un objet distant peut donc donner accès à d'autres objets distants
- beaucoup plus distribué que les RPC classiques : un client peut envoyer un objet à un serveur pour échanger leurs rôles respectifs !

## Exemple

```
1 import java.rmi.server.UnicastRemoteObject;
2 import java.rmi.RemoteException;
3 public class HelloWorld extends UnicastRemoteObject
4     implements IHelloWorld {
5     public String sayHelloTo(String firstname,
6                             String lastname)
7         throws RemoteException {
8         return "Coucou "+firstname+" "+lastname+" !";
9     }
10    public void truc() {
11    }
12    public HelloWorld() throws RemoteException {
13        super();
14    }
15 }
```

On est obligé de redéfinir le constructeur (protected)

## Registry (RMI-JRMP)

Avec JRMP, on utilise un *registry* :

- le jdk contient `rmiregistry`, un programme qui implémente ce service
  - le service écoute par défaut sur le port 1099
  - le programme doit tourner en permanence
  - peut tourner sur une machine différente de celle de l'objet
- Pour enregistrer un objet, on utilise la classe `Naming` (`java.rmi`). Elle propose diverses méthodes de classe :
- `bind` et `rebind` pour associer un objet distant à un nom
  - `lookup` pour récupérer un objet distant à partir d'un nom
  - `list` pour obtenir la liste des noms connus
  - les noms acceptables sont de la forme  
`//host:port/name`
- `host` : machine sur laquelle `rmiregistry` fonctionne
  - `port` : port sur lequel `rmiregistry` écoute (optionnel)
  - `name` : nom de l'objet distant

## Exemple : démarrage du serveur

```
1 import java.rmi.Naming;
2 public class HelloWorldServer {
3     public static void main(String[] args) {
4         try {
5             HelloWorld server=new HelloWorld();
6             Naming.rebind("//localhost/hello/french",
7                 server);
8         } catch (Exception e) {
9             System.err.println("Erreur : "+e);
10        }
11    }
12 }
```

On associe ainsi hello/french à l'objet HelloWorld qui vient d'être créé.

## Mise en œuvre pratique (JRMP)

Pour développer et démarrer un serveur, on procède selon les étapes suivantes :

1. développement :
  - (a) écriture de l'interface de l'objet distant
  - (b) implémentation du serveur
  - (c) compilation des fichiers
  - (d) génération des *stubs* (programme *rmic*)
2. lancement
  - (a) démarrage de *rmiregistry*
  - (b) démarrage du serveur

Il est préférable que le CLASSPATH du *rmiregistry* ne contienne pas les *stubs*.

## Les Stubs

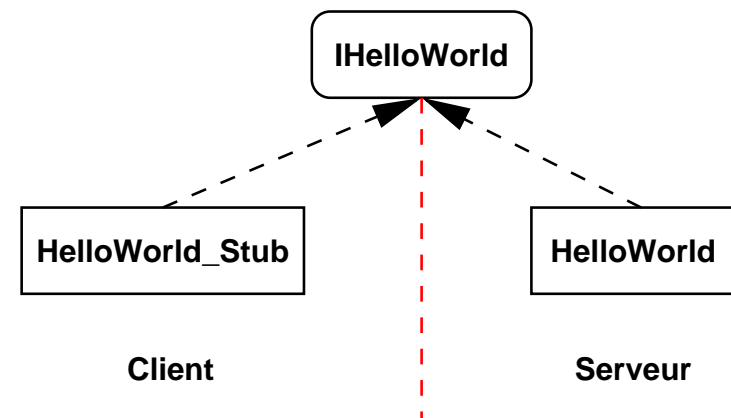
Vocabulaire Java (JRMP) :

- *stub* : correspond au *stub* client
- *skeleton* : correspond au *stub* serveur

Outil *rmic* :

- engendre le *stub* et le *skeleton*
- à partir du *bytecode* de l'objet distant
- directement sous forme de *bytecode*
- spécifique à l'objet, pas à son interface
- objet HelloWorld :
  - *stub* : HelloWorld\_Stub.class (implémente IHelloWorld)
  - *skeleton* : HelloWorld\_Skel.class
- si on ne travaille qu'avec des JVM 1.x avec  $x > 1$ , le *skeleton* est inutile

## Implémentation de IHelloWorld



## Développement du client

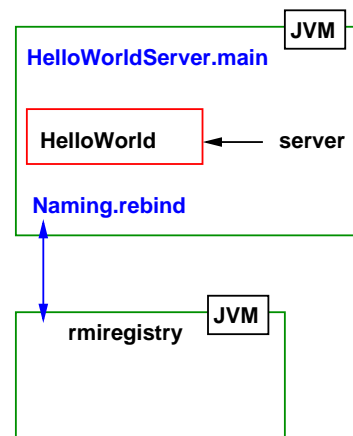
- pour **programmer**, le client, il suffit d'avoir accès aux **interfaces** des objets distants
- l'accès au "premier" objet se fait en général grâce au *registry* (méthode `lookup` de la classe `Naming`)
- pour **exécuter** le client, il faut que la JVM ait accès aux **stubs** des objets distants. Plusieurs solutions :
  - distribution "manuelle" des *stubs*
  - distribution automatique (par un serveur HTTP)
  - *Design Pattern Proxy* utilisé en conjonction avec l'une des approches précédentes
- modèle très supérieur à celui des RPC :
  - approche objet
  - déploiement évolué
  - publication/résolution très simple
  - etc.

## Exemple : le client

Stubs disponibles localement :

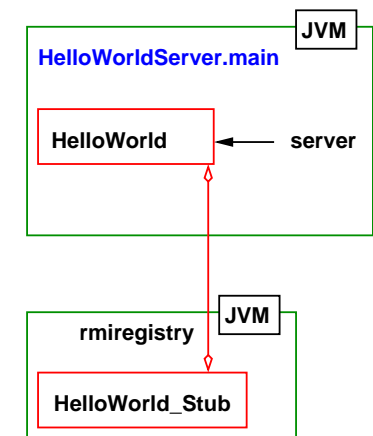
```
Client
1 import java.rmi.Naming;
2 public class Client {
3     public static void main(String[] args) {
4         try {
5             IHelloWorld hello=(IHelloWorld)
6                 Naming.lookup("//localhost/hello/french");
7             System.out.println(hello.sayHelloTo(args[0],
8                                     args[1]));
9         } catch (Exception e) {
10            System.err.println("Erreur : "+e);
11        }
12    }
13 }
```

## Principe de fonctionnement 1/5



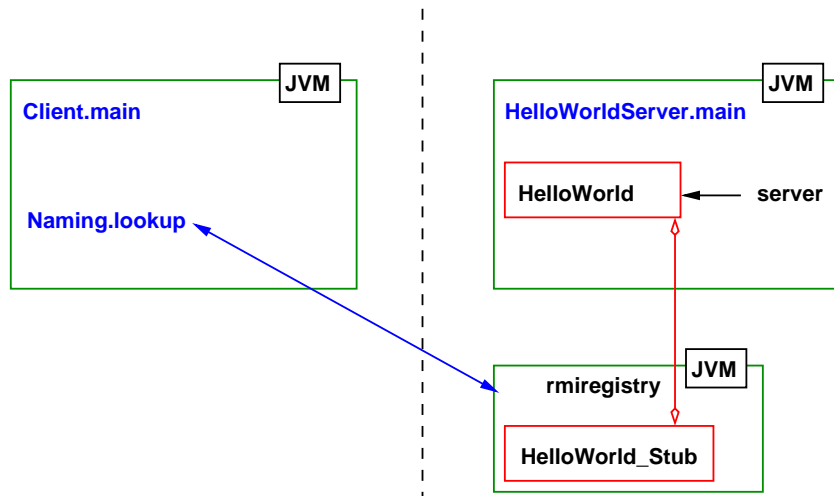
Enregistrement de l'objet

## Principe de fonctionnement 2/5



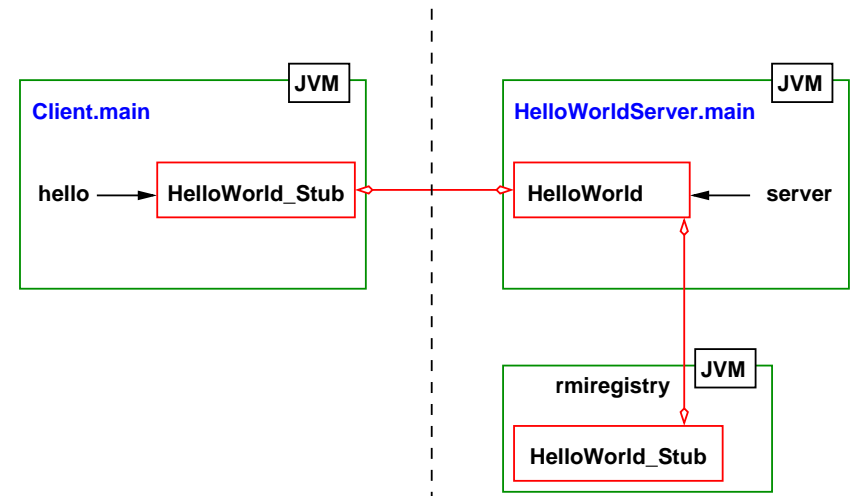
Stub dans le registry

## Principe de fonctionnement 3/5



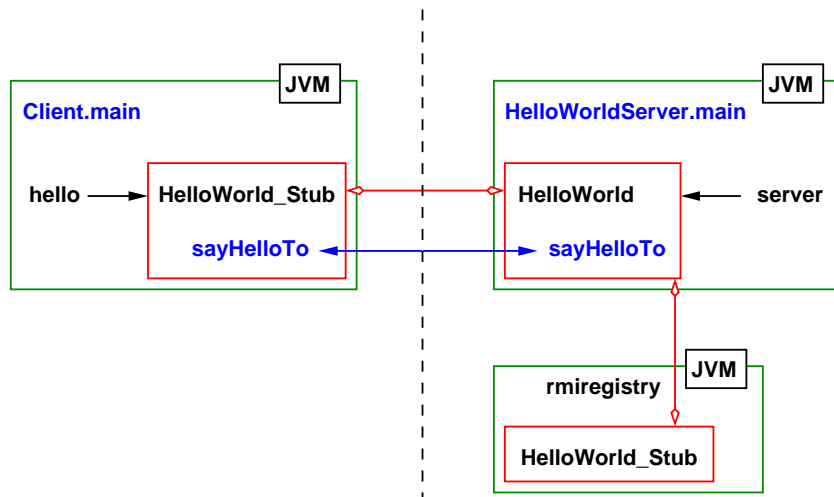
Résolution pour le client

## Principe de fonctionnement 4/5



Stub pour le client

## Principe de fonctionnement 5/5



Appel RMI

## Déploiement automatique

Très utile en production :

- principe : le client télécharge automatiquement les *stubs* au démarrage
- on doit disposer d'un serveur HTTP (ou FTP, même minimaliste)
- les *stubs* et les interfaces de objets distants doivent être accessible sur ce serveur (fichiers `.class`)
- modification du serveur : renseignement de la propriété `java.rmi.server.codebase` (URL des classes)
- modifications du client :
  - installation d'un gestionnaire de sécurité
  - renseignement de la propriété `java.security.policy` : fichier décrivant les règles de sécurités retenues

## Démarrage du serveur

- `java -Djava.rmi.server.codebase=URL HelloWorldServer`
- URL : emplacement des classes téléchargeables par les clients
- l'URL est générale : fichier, HTTP, FTP, etc.
- il est **impératif** que le *registry* n'ait pas accès aux *stubs*
- astuce de lancement (aucun rapport avec l'aspect dynamique !) : la propriété `java.rmi.server.hostname` permet de contourner certains problèmes (précise le nom du serveur)

## Nouveau code du client

```
Client
1 import java.rmi.Naming;
2 import java.rmi.RMISecurityManager;
3 public class Client {
4     public static void main(String[] args) {
5         if (System.getSecurityManager() == null) {
6             System.setSecurityManager(new RMISecurityManager());
7         }
8         try {
9             IHelloWorld hello=(IHelloWorld)
10                Naming.lookup("//localhost/hello/french");
11                System.out.println(hello.sayHelloTo(args[0],
12                                   args[1]));
13        } catch (Exception e) {
14            System.err.println("Erreur : "+e);
15        }
16    }
17 }
```

Ajout : lignes 2, 5, 6 et 7.

## Démarrage du client

- `java -Djava.security.policy=REGLES Client ...`
- REGLES : fichier contenant les règles de sécurité :
  - édition grâce à `policytool` (ou à la main)
  - précise exactement ce que le client peut faire. Au minimum, il doit pouvoir :
    - se connecter au *registry*
    - se connecter au serveur HTTP
    - définir dynamiquement une classe
- indispensable car par défaut, le client ne peut pas charger des classes en dehors du CLASSPATH
- on peut ajouter un gestionnaire de sécurité au serveur : dans ce cas il doit pouvoir se connecter au *registry* (au minimum)

## Exemple de fichier *policy*

Pour le client :

```
client-policy
1 /* AUTOMATICALLY GENERATED ON Mon May 20 08:22:05 CEST 2002*/
2 /* DO NOT EDIT */
3
4 grant {
5     permission java.net.SocketPermission "localhost:1024-",
6         "connect,resolve,accept";
7     permission java.lang.RuntimePermission "getClassLoader";
8     permission java.net.SocketPermission "localhost:80",
9         "connect";
10 };
```

## Fonctionnement

Tout passe par le *registry* :

- mécanisme sous-jacent : les *ClassLoaders* (utilisés au départ pour les applets)
- à l'enregistrement du serveur (*bind* ou *rebind*), l'information *codebase* est stockée par le *registry*
- le *registry* télécharge le *stub* correspondant à l'objet distant enregistré (seulement s'il ne peut pas y accéder directement !)
- quand un client demande l'accès à l'objet distant par le *registry*, celui-ci lui renvoie l'emplacement du *stub*
- si la JVM du client n'a pas accès directement (dans son *CLASSPATH*) au *stub*, elle utilise l'emplacement (le *codebase*) pour télécharger la classe
- le téléchargement n'est pas limité aux *stubs* : toute classe nécessaire au fonctionnement de l'objet distant

## Design Pattern Proxy et RMI

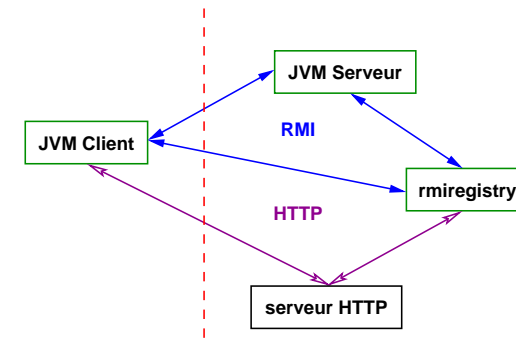
Problèmes :

- l'implémentation du serveur "doit" hériter de *UnicastRemoteObject*
- la distribution demande du travail : placer le code sur les clients ou sur un serveur web (chargement des *stubs* à chaque démarrage d'une JVM)

Une solution, le *Proxy* :

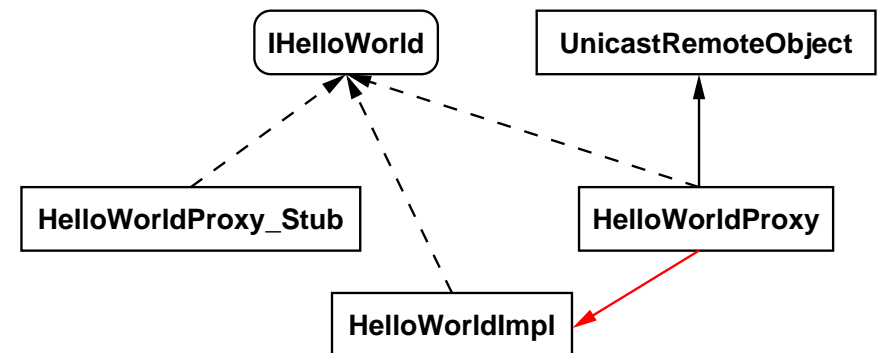
- on définit une interface d'objet distant classique (*IHelloWorld*)
- on implémente l'interface par un *proxy* (*HelloWorldProxy*) qui fonctionne de la façon suivante :
  - il hérite de *UnicastRemoteObject*
  - il transmet tous les appels à un objet sous-jacent
  - l'objet sous-jacent implémente l'interface distante
- on distribue une seule fois le *stub* du *proxy*

## Illustration du fonctionnement



1. appel RMI : serveur vers *rmiregistry*
2. appel HTTP : *rmiregistry* vers serveur de classes (*stub*)
3. appel RMI : client vers *rmiregistry*
4. appel HTTP : client vers serveur de classes (*stub*)
5. appel RMI : client vers serveur

## Diagramme de classes



Un objet *HelloWorldProxy* est construit à partir d'un objet *HelloWorldImpl*.

## Exemple

```
1 import java.rmi.server.UnicastRemoteObject;
2 import java.rmi.RemoteException;
3 public class HelloWorldProxy extends UnicastRemoteObject
4     implements IHelloWorld {
5     private IHelloWorld delegatedTo;
6     public String sayHelloTo(String firstname,
7         String lastname)
8         throws RemoteException {
9         return delegatedTo.sayHelloTo(firstname,lastname);
10    }
11    public HelloWorldProxy(IHelloWorld d) throws RemoteException {
12        super();
13        delegatedTo=d;
14    }
15 }
```

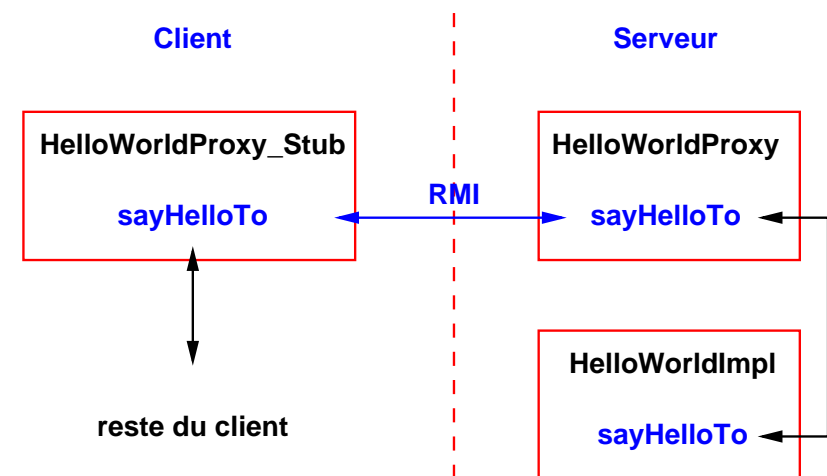
## Exemple (2)

```
1 import java.rmi.RemoteException;
2 public class HelloWorldImpl implements IHelloWorld {
3     public String sayHelloTo(String firstname,
4         String lastname)
5         throws RemoteException {
6         return "Bonjour "+firstname+" "+lastname+" !";
7     }
8 }
```

## Exemple (3)

```
1 import java.rmi.Naming;
2 public class HelloWorldServerProxy {
3     public static void main(String[] args) {
4         try {
5             HelloWorldProxy server=
6                 new HelloWorldProxy(new HelloWorldImpl());
7             Naming.rebind("//localhost/hello/french",
8                 server);
9         } catch (Exception e) {
10            System.err.println("Erreur : "+e);
11        }
12    }
13 }
```

## Fonctionnement



## Passage d'objets par référence

- si une méthode d'une interface Remote utilise des types qui étendent Remote, les paramètres et/ou résultat correspondant sont "passés" par référence distante
- permet de limiter l'utilisation du *registry* aux objets vraiment importants
- Dictionnaires distribués (un par utilisateur)

```
1  import java.rmi.Remote;
2  import java.rmi.RemoteException;
3  public interface IDictionnaire extends Remote {
4      public String définition(String mot)
5          throws RemoteException;
6      public void ajoute(String mot,String définition)
7          throws RemoteException;
8      public void supprime(String mot)
9          throws RemoteException;
10 }
```

Systèmes répartis : Remote Method Invocation – p.45/74

## Serveur de dictionnaires

```
1  import java.rmi.Remote;
2  import java.rmi.RemoteException;
3  public interface IServeur extends Remote {
4      public IDictionnaire connexion(Utilisateur p)
5          throws RemoteException;
6  }
```

```
1  import java.io.Serializable;
2  public class Utilisateur implements Serializable {
3      public String id;
4      public String pass;
5      public Utilisateur(String id,String pass) {
6          this.id=id;
7          this.pass=pass;
8      }
9  }
```

Systèmes répartis : Remote Method Invocation – p.46/74

## Modes de passage

- Utilisateur :
  - passage par **valeur** car n'implémente pas Remote
  - possible car Utilisateur implémente Serializable
  - même si connexion modifie (sur le serveur) le contenu d'un Utilisateur, le client ne verra pas cette modification
  - le fichier class doit être distribué sur le client
- IDictionnaire :
  - étend Remote : passage par **référence**
  - l'objet reste sur le serveur
  - référence distante sur le client

Systèmes répartis : Remote Method Invocation – p.47/74

## Dictionnaire

```
1  import java.util.Map;
2  import java.util.HashMap;
3  import java.rmi.server.UnicastRemoteObject;
4  import java.rmi.RemoteException;
5  public class Dictionnaire extends UnicastRemoteObject
6      implements IDictionnaire
7  {
8      private Map dico;
9      public Dictionnaire() throws RemoteException {
10         super();
11         dico=new HashMap();
12     }
13     public String définition(String mot)
14         throws RemoteException {
15         return (String)dico.get(mot);
16     }
17     public void ajoute(String mot,String définition)
18         throws RemoteException {
19         dico.put(mot,définition);
20     }
21     public void supprime(String mot)
22         throws RemoteException {
23         dico.remove(mot);
24     }
25 }
```

Systèmes répartis : Remote Method Invocation – p.48/74

## Serveur

```
1 import java.util.Map;
2 import java.util.HashMap;
3 import java.rmi.server.UnicastRemoteObject;
4 import java.rmi.RemoteException;
5 public class Serveur extends UnicastRemoteObject implements IServeur {
6     private Map dicos;
7     private Map pass;
8     public Serveur() throws RemoteException {
9         super();
10        dicos=new HashMap();
11        pass=new HashMap();
12    }
13    public IDictionnaire connexion(Utilisateur p) throws RemoteException {
14        if(pass.containsKey(p.id)) {
15            if(pass.get(p.id).equals(p.pass)) {
16                return (IDictionnaire)dicos.get(p.id);
17            } else {
18                System.err.println("Tentative d'accès non autorisé à "+p.id);
19                return null;
20            }
21        } else {
```

## Serveur

```
22        System.err.println("Création de "+p.id);
23        pass.put(p.id,p.pass);
24        Dictionnaire d=new Dictionnaire();
25        dicos.put(p.id,d);
26        return d;
27    }
28 }
29 }
```

## LanceurServeur

```
1 import java.rmi.Naming;
2 public class LanceurServeur {
3     public static void main(String[] args) {
4         try {
5             Serveur server=new Serveur();
6             Naming.rebind("//localhost/dico",
7                 server);
8         } catch (Exception e) {
9             System.err.println("Erreur : "+e);
10        }
11    }
12 }
13 }
```

- Il ne faut surtout pas créer de Dictionnaire!
- On doit engendrer et distribuer les *stubs* pour Dictionnaire et Serveur (éventuellement de façon automatique)

## Le client

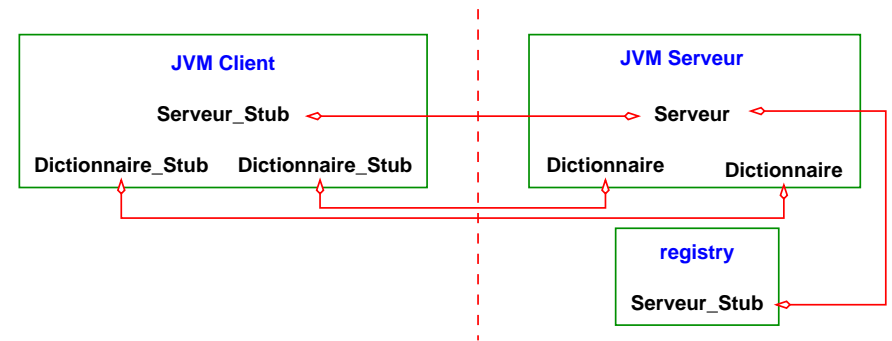
```
1 import java.rmi.Naming;
2 public class Client {
3     public static void main(String[] args) {
4         if(args.length<4) {
5             System.err.println("Utilisation : java Client id pass opération "
6                 +"mot [définition]");
7             System.exit(1);
8         }
9         try {
10            IServeur serveur=(IServeur)
11                Naming.lookup("//localhost/dico");
12            Utilisateur user=new Utilisateur(args[0],args[1]);
13            IDictionnaire dico=serveur.connexion(user);
14            if(dico==null) {
15                System.err.println("Problème de connexion : pas de dictionnaire");
16                System.exit(1);
17            }
18            if(args[2].equals("définition")) {
19                String s=dico.définition(args[3]);
20                if(s==null) {
```

## Le client

```
Client
21     System.err.println("Mot non défini : "+args[3]);
22     } else {
23         System.out.println(args[3]+"=\""+s+"\"");
24     }
25     } else if(args[2].equals("ajout")) {
26         if(args.length!=5) {
27             System.err.println("Argument manquant pour l'ajout");
28             System.exit(1);
29         }
30         dico ajoute(args[3],args[4]);
31     } else if(args[2].equals("suppression")) {
32         dico.supprime(args[3]);
33     } else {
34         System.err.println("Commande inconnue : "+args[2]);
35     }
36     } catch (Exception e) {
37         System.err.println("Erreur RMI : "+e);
38     }
39 }
40 }
```

Systèmes répartis : Remote Method Invocation – p.53/74

## Fonctionnement



Les Dictionnaires n'apparaissent pas dans le *registry*.

Systèmes répartis : Remote Method Invocation – p.54/74

## Distributed Garbage Collection

Un mécanisme très évolué de RMI-JRMP :

- extension répartie du *Garbage Collection*
- basé sur un comptage de référence :
  - le serveur compte les objets qu'il exporte
  - chaque client possède un bail (*lease*) pour chaque objet distant
  - la durée par défaut du bail est de 10 minutes
  - quand le bail est à moitié expiré, le client le renouvelle
  - si un bail n'est pas renouvelé, le serveur peut considérer la référence comme morte
  - quand un objet n'est plus référencé, le serveur peut le détruire
- pratique, mais pose des problèmes de montée en charge
- n'existe pas en RMI-IIOP

Systèmes répartis : Remote Method Invocation – p.55/74

## Interface avec le DGC

- un objet distant peut implémenter l'interface *Unreferenced*
- il doit définir une méthode `void unreferenced()`
- la méthode est appelée quand l'objet n'est plus référencé par aucun client
- attention, le *registry* maintient une référence vers chacun des objets enregistrés
- on peut régler la durée de l'ensemble des baux par une propriété : `java.rmi.dgc.leaseValue` (en millisecondes)
- détecte la disparition propre (déclarée par le client) ou consécutive à un plantage (plus lent...)

Systèmes répartis : Remote Method Invocation – p.56/74

## Exemple

ICConnexion

```
1 import java.rmi.Remote;
2 import java.rmi.RemoteException;
3 public interface IConnexion extends Remote {
4     public int number() throws RemoteException;
5 }
```

IServeur

```
1 import java.rmi.Remote;
2 import java.rmi.RemoteException;
3 public interface IServeur extends Remote {
4     public IConnexion register() throws RemoteException;
5 }
```

## Exemple (2)

Serveur

```
1 import java.rmi.server.UnicastRemoteObject;
2 import java.rmi.RemoteException;
3 import java.rmi.Naming;
4 public class Serveur extends UnicastRemoteObject implements IServeur {
5     public Serveur() throws RemoteException {
6         super();
7     }
8     public IConnexion register() throws RemoteException {
9         return new Connexion();
10    }
11    public static void main(String[] args) {
12        try {
13            Serveur server=new Serveur();
14            Naming.rebind("//localhost/dgcExp",server);
15        } catch (Exception e) {
16            System.err.println("Erreur : "+e);
17        }
18    }
19 }
```

## Exemple (3)

Client

```
1 import java.rmi.Naming;
2 public class Client {
3     public static void main(String[] args) {
4         try {
5             IServeur serveur=(IServeur)Naming.lookup("//localhost/dgcExp");
6             int n=0;
7             while(true) {
8                 n++;
9                 IConnexion cnx=serveur.register();
10                System.out.println(cnx.number());
11                // pour illustrer le mécanisme
12                if(n>10) {
13                    System.exit(1);
14                }
15                Thread.sleep(2000);
16                cnx=null;
17                System.gc();
18                Thread.sleep(2000);
19            }
20        } catch (Exception e) {
21            System.err.println("Erreur : "+e);
22        }
23    }
24 }
```

## Exemple (4)

Connexion

```
1 import java.rmi.server.UnicastRemoteObject;
2 import java.rmi.RemoteException;
3 import java.rmi.server.Unreferenced;
4 public class Connexion extends UnicastRemoteObject
5     implements IConnexion, Unreferenced {
6     private static int NB=0;
7     private int nb;
8     public Connexion() throws RemoteException{
9         super();
10        nb=NB++;
11    }
12    public int number() throws RemoteException {
13        return nb;
14    }
15    public void unreferenced() {
16        System.out.println("Libération de "+nb);
17    }
18 }
```

## Gestion des erreurs

Pour faire une application correcte, il faut dépasser le simple try catch. Erreurs possibles pour le client :

- pendant la résolution : accès au *rmiregistry* ou objet non trouvé
- pendant un appel RMI : problèmes de connexion, de disponibilité des objets, etc.
- les exceptions sont propagées depuis le serveur vers le client

Erreurs possibles pour le serveur :

- pendant la publication : accès au *rmiregistry* ou nom déjà utilisé
- pendant le fonctionnement : problème de création d'objets Remote, d'accès au réseau, etc.

## Exemple : le serveur

```
1 import java.rmi.Remote;
2 import java.rmi.RemoteException;
3 public interface IRandomSelect extends Remote {
4     public int random(int[] tab,int max)
5         throws RemoteException;
6 }
```

```
1 import java.rmi.server.*;
2 import java.rmi.RemoteException;
3 import java.util.Random;
4 public class RandomSelect extends UnicastRemoteObject
5     implements IRandomSelect {
6     private Random rng;
7     public RandomSelect() throws RemoteException {
8         super();
9         rng=new Random();
10    }
11    public int random(int[] tab,int max) throws RemoteException {
12        int n=rng.nextInt(max);
13        return tab[n];
14    }
15 }
```

## Exemple : démarrage du serveur

```
1 import java.rmi.*;
2 import java.rmi.server.*;
3 public class RandomSelectServer {
4     public static final int MAX_RETRY=10;
5     public static void main(String[] args) {
6         RandomSelect server=null;
7         int essai=0;
8         do {
9             essai++;
10            try {
11                server=new RandomSelect();
12            } catch (ExportException e) {
13                System.err.println("Impossible d'exporter le serveur : "+e);
14            } catch (RemoteException e) {
15                System.err.println("Exception RMI : "+e);
16            }
17            if(essai<MAX_RETRY && server==null) {
18                System.err.println("Pause de 5 secondes avant une nouvelle tentative");
19                try {
20                    Thread.sleep(5000);
21                } catch (InterruptedException e) {
22                    System.err.println("Uh : "+e);
23                }
24            }
25        } while(essai<MAX_RETRY && server==null);
}
```

## Exemple : démarrage du serveur (2)

```
26 boolean retry=false;
27 essai=0;
28 do {
29     essai++;
30     try {
31         Naming.rebind("//localhost/random",server);
32         retry=false;
33     } catch (RemoteException e) {
34         System.err.println("Connexion impossible au registry "+e.getMessage());
35         retry=true;
36     } catch (Exception e) {
37         // vraie erreur
38         System.err.println("Erreur : "+e);
39         System.exit(1);
40     }
41     if(retry) {
42         System.err.println("Pause de 5 secondes avant une nouvelle tentative");
43         try {
44             Thread.sleep(5000);
45         } catch (InterruptedException e) {
46             System.err.println("Uh : "+e);
47         }
48     }
49     } while(essai<MAX_RETRY && retry);
50 }
51 }
```

## Exemple : le client (1)

```
Client
1 import java.rmi.*;
2 public class Client {
3     public static final int MAX_RETRY=10;
4     public static void main(String[] args) {
5         boolean getObjectAgain;
6         do {
7             getObjectAgain=false;
8             IRandomSelect rng=null;
9             int essai=0;
10            do {
11                boolean sleep=false;
12                essai++;
13                try {
14                    rng=(IRandomSelect)Naming.lookup("//localhost/random");
15                } catch (RemoteException e) {
16                    System.err.println("Connexion impossible au registry "+e.getMessage());
17                    sleep=true;
18                } catch (NotBoundException e) {
19                    System.err.println("random n'est pas associé à un objet RMI");
20                    sleep=true;
21                } catch (Exception e) {
22                    // vraie erreur
23                    System.err.println("Erreur : "+e);
24                    System.exit(1);
25                }

```

Systèmes répartis : *Remote Method Invocation* – p.65/74

## Exemple : le client (2)

```
Client
26
27     if(essai<MAX_RETRY && sleep) {
28         System.err.println("Pause de 5 secondes avant une nouvelle tentative");
29         try {
30             Thread.sleep(5000);
31         } catch (InterruptedException e) {
32             System.err.println("Uh : "+e);
33         }
34     } while(essai<MAX_RETRY && rng==null);
35     if(rng==null) {
36         System.err.println("Impossible de se connecter au serveur");
37         System.exit(1);
38     }
39     System.out.println("Object distant récupéré");

```

Première phase : récupération de l'objet !

Systèmes répartis : *Remote Method Invocation* – p.66/74

## Exemple : le client (3)

```
Client
40     essai=0;
41     boolean retry=true;
42     do {
43         boolean sleep=false;
44         essai++;
45         try {
46             System.out.println(rng.random(new int[] {1,2,3},5));
47             retry=false;
48         } catch (ConnectException e) {
49             System.err.println("Connexion refusée (!)");
50             sleep=true;
51             getObjectAgain=true;
52             retry=false;
53         } catch (RemoteException e) {
54             System.err.println("Exception RMI : "+e);
55             sleep=true;
56         }

```

Systèmes répartis : *Remote Method Invocation* – p.67/74

## Exemple : le client (4)

```
Client
57
58     if(essai<MAX_RETRY && sleep) {
59         System.err.println("Pause de 5 secondes avant une nouvelle tentative");
60         try {
61             Thread.sleep(5000);
62         } catch (InterruptedException e) {
63             System.err.println("Uh : "+e);
64         }
65     } while(retry && essai<MAX_RETRY);
66     if(retry) {
67         System.err.println("Impossible d'utiliser l'objet distant");
68         System.exit(1);
69     }
70     } while(getObjectAgain);
71 }
72 }

```

Deuxième phase : utilisation de l'objet !

Systèmes répartis : *Remote Method Invocation* – p.68/74

## RMI-IIOP

Modifications sur le serveur :

- Le serveur ne peut plus hériter de `UnicastRemoteObject`. Deux solutions (même principe que RMI-JRMP) :
  - hériter de `PortableRemoteObject` (`javax.rmi`)
  - utiliser la méthode `exportObject` de cette classe
- le *rmiregistry* est remplacé par le service équivalent de CORBA (COSNaming) :
  - on doit passer par JNDI (Java Naming and Directory Interface), une API qui permet d'accéder à divers services de nom (DNS, COSNaming, LDAP, etc.)
  - paquet `javax.naming`

## Nouveau serveur

```
1 import java.rmi.Remote;
2 import java.rmi.RemoteException;
3 public interface IHelloWorld extends Remote {
4     public String sayHelloTo(String firstname,String lastname)
5         throws RemoteException;
6 }

1 import javax.rmi.PortableRemoteObject;
2 import java.rmi.RemoteException;
3 public class HelloWorld extends PortableRemoteObject
4     implements IHelloWorld {
5     public String sayHelloTo(String firstname,String lastname)
6         throws RemoteException {
7         return "Coucou "+firstname+" "+lastname+" !";
8     }
9     public HelloWorld() throws RemoteException {
10        super();
11    }
12 }
```

## Nouveau lanceur

```
1 import javax.naming.*;
2 public class HelloWorldServer {
3     public static void main(String[] args) {
4         try {
5             Context initialNamingContext = new InitialContext();
6             System.out.println("Naming context ok");
7             HelloWorld server=new HelloWorld();
8             initialNamingContext.rebind("hello",server);
9             System.out.println("Server bound");
10        } catch (Exception e) {
11            System.err.println("Erreur : "+e);
12        }
13    }
14 }
```

## Démarrage du serveur

Un peu plus lourd que RMI-JRMP :

1. fabrication des *stubs* :
  - outil `rmic`, option `-iiop`
  - engendre un *stub* pour le client (`_<interface>_Stub.class`)
  - engendre un *tie* pour le serveur (`_<serveur>_Tie.class`)
2. démarrage de COSNaming :  
`tnameserv -ORBInitialPort <port>`
3. port par défaut 900 : privilégié !
4. lancement du serveur :

```
java -Djava.naming.factory.initial=com.sun.jndi.cosnaming.CNCTXFactory
-Djava.naming.provider.url=iiop://<host>:<port> HelloWorldServer
```

## Modifications pour le client

Même esprit que pour le serveur :

- on utilise COSNaming
- la conversion ne se fait plus par *cast* :
  - on obtient en général un `org.omg.CORBA.Object`
  - la mécanique IIOP doit être utilisée pour vérifier que le *cast* est possible
  - méthode `narrow` de `PortableRemoteObject`
- même principe pour le démarrage

```
java -Djava.naming.factory.initial=com.sun.jndi.cosnaming.CNCtxFactory  
-Djava.naming.provider.url=iiop://<host>:<port> Client
```

## Nouveau client

```
----- HelloWorldServer -----  
1 import javax.rmi.PortableRemoteObject;  
2 import javax.naming.*;  
3 public class Client {  
4     public static void main(String[] args) {  
5         try {  
6             Context initialNamingContext = new InitialContext();  
7             System.out.println("Naming context ok");  
8             Object o=initialNamingContext.lookup("hello");  
9             IHelloWorld hello=  
10                (IHelloWorld)PortableRemoteObject.narrow(o, IHelloWorld.class);  
11             System.out.println(hello.sayHelloTo(args[0],  
12                                args[1]));  
13  
14             } catch (Exception e) {  
15                 System.err.println("Erreur : "+e);  
16             }  
17         }  
18     }
```