Java Remote Method Invocation

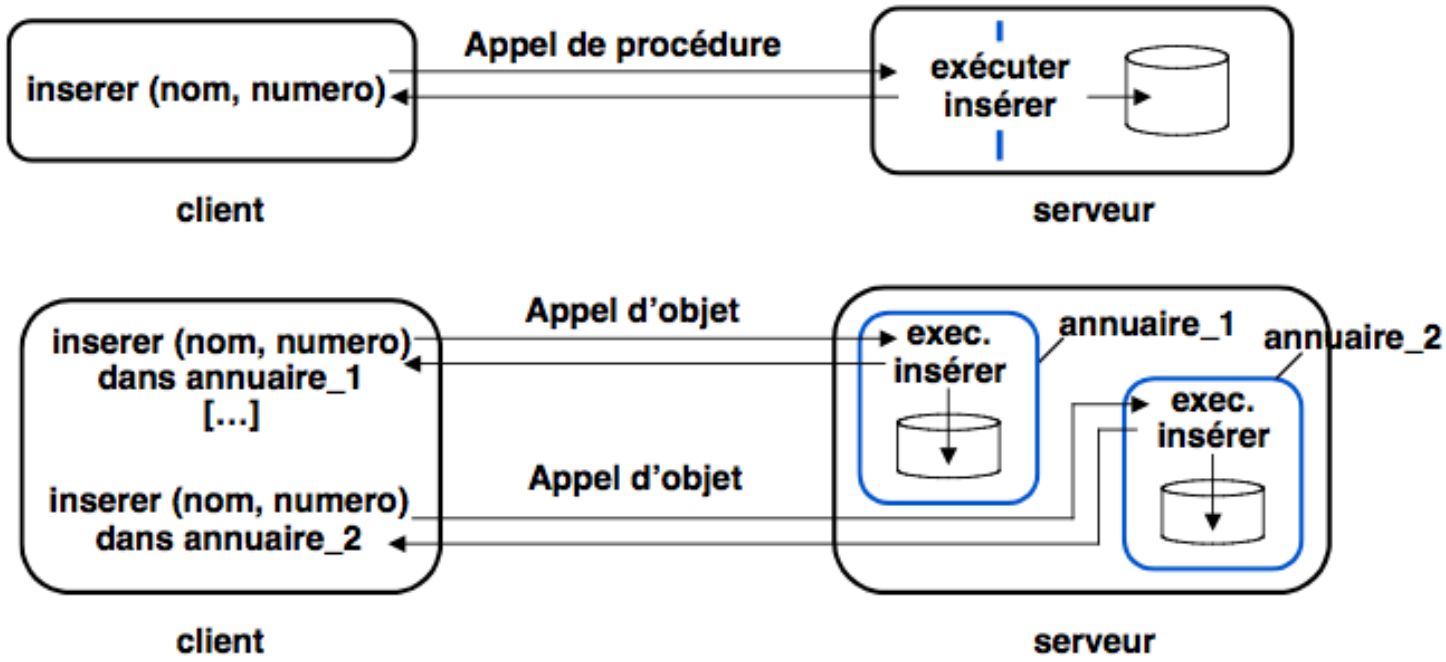
Ce support est très largement inspiré du livre et du cours de S. Krakowiak

Intérêt des objets pour la construction d'applications réparties

- Encapsulation
  - L'interface (méthodes + attributs) est la seule voie d'accès à l'état interne, non directement accessible
- Classes et instances
  - Mécanismes de génération d'exemplaires conformes à un même modèle
- Héritage
  - Mécanisme de spécialisation : facilite récupération et réutilisation de l'existant
- Polymorphisme
  - Mises en oeuvre diverses des fonctions d'une interface
  - Remplacement d'un objet par un autre si interfaces "compatibles"
  - Facilite l'évolution et l'adaptation des applications

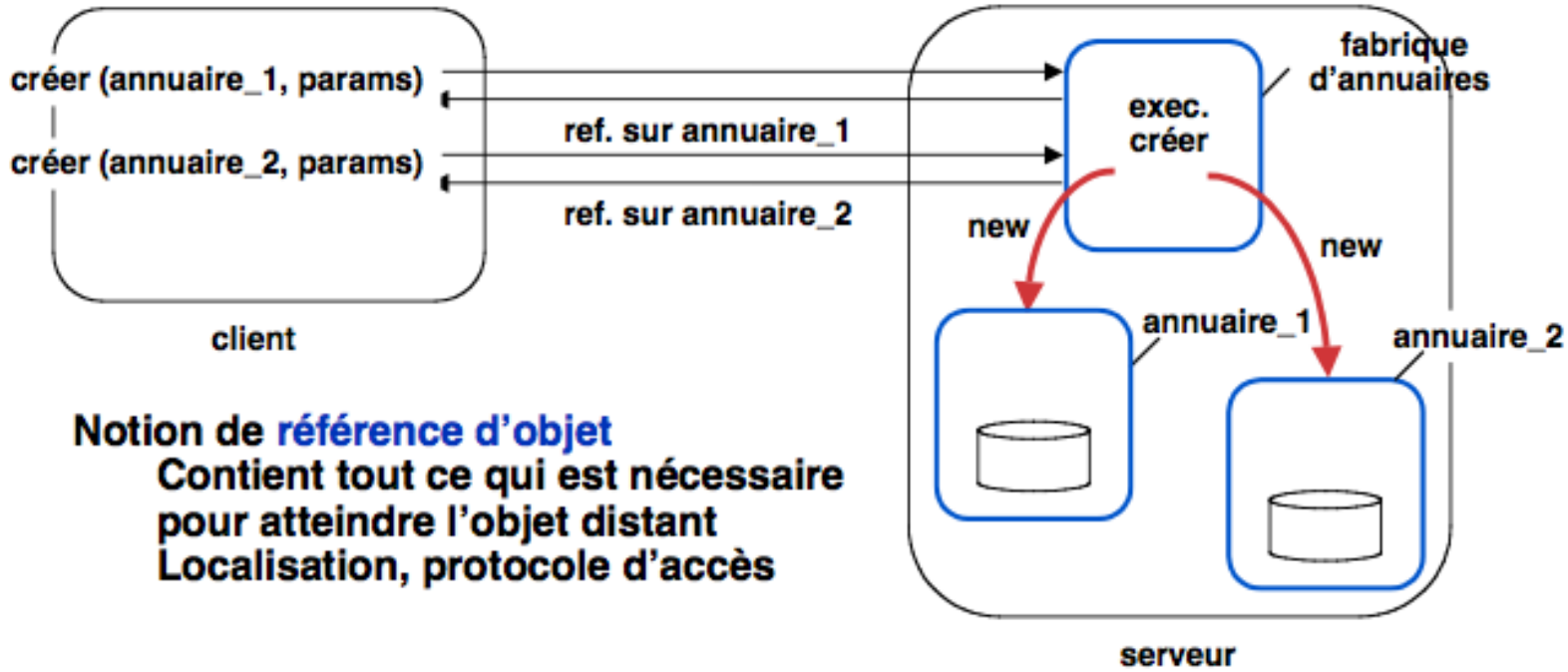
Extension du RPC aux objets

- Appel de procédure vs appel de méthode sur un objet
  - Exemple : insérer une entrée dans un annuaire



Extension du RPC aux objets

- Phase préalable : création d'instances d'une classe d'objects
  - Notion de fabrique (factory)



Notion de référence d'objet
Contient tout ce qui est nécessaire pour atteindre l'objet distant
Localisation, protocole d'accès

Java RMI (Remote Method Invocation)

- Motivation : construction d'applications réparties avec Java
  - Appel de méthode au lieu d'appel de procédure
- Principe
  - Même schéma que RPC
- Le programmeur fournit
  - Une (ou plusieurs) description(s) d'interface
    - Ici pas d'IDL séparé : Java sert d'IDL
  - Le programme du serveur
    - Objets réalisant les interfaces
    - Serveur
  - Le programme du client
- L'environnement Java fournit
  - Un générateur de talons (rmic)
  - Un service de noms (Object Registry)

Règles d'usage

- Interface
  - L'interface d'un objet distant (Remote) est celle d'un objet Java, avec quelques règles d'usage :
  - L'interface distante doit être publique
  - L'interface distante doit étendre l'interface java.rmi.Remote
  - Chaque méthode doit déclarer au moins l'exception java.rmi.RemoteException
- Passage d'objets en paramètre
  - Les objets locaux sont passés par valeur (copie) et doivent être sérialisables (étendent l'interface java.io.Serializable)
  - Les objets distants sont passés par référence et sont désignés par leur interface
- Réalisation des classes distantes (Remote)
  - Une classe distante doit implémenter une interface elle-même distante (Remote)
  - Une classe distante doit étendre la classe java.rmi.server.UnicastRemoteObject (d'autres possibilités existent)
  - Une classe distante peut aussi avoir des méthodes appelables seulement localement (ne font pas partie de son interface Remote)

Règles d'écriture du serveur

- Un serveur est une classe qui implémente l'interface de l'objet distant
  - Spécifier les références distantes qui doivent être implémentées (objets passés en paramètres)
  - Définir le constructeur de l'objet distant
  - Fournir la réalisation des méthodes appelables à distance
  - Créer et installer le gestionnaire de sécurité
  - Créer au moins une instance de la classe serveur
  - Enregistrer au moins une instance dans le serveur de noms

Exemple "Hello world" : interface et classe de l'objet distant

```java
// HelloInterface.java
// Définition d'interface
import java.rmi.Remote;
import java.rmi.RemoteException;

public interface HelloInterface extends Remote {
    /* methode qui imprime un message
    predefini dans l'objet appele */
    public String sayHello () throws RemoteException;
}
```

```java
// Hello.java
// Classe réalisant l'interface
import java.rmi.RemoteException;
import java.rmi.server.UnicastRemoteObject;

public class Hello extends UnicastRemoteObject implements HelloInterface {
        private String message;

        public Hello(String message) throws RemoteException {
                this.message = message;
        }

        /* l'implementation de la methode */
        public String sayHello () throws RemoteException
        {
                                System.out.println("On me demande de dire : " + message);
                return message;
        };

}
```

Exemple "Hello world" : client

```java
// HelloClient.java
// Programme du client
import java.rmi.*;

public class HelloClient {
        public static void main (String [ ] argv) {
                /* lancer SecurityManager */
                System.setSecurityManager (new RMISecurityManager()) ;
                try {
                        /* trouver une reference vers l'objet distant */
                        HelloInterface hello = (HelloInterface) Naming.lookup("rmi://localhost/Hello1") ;
                        /* appel de methode a distance */
                        System.out.println (hello.sayHello()) ;

                } catch (Exception e) {
                        System.out.println ("Erreur client : " + e) ;
                }
        }
}
```

Exemple "Hello world" : serveur

```java
// HelloServer.java
// Programme du serveur
import java.rmi.RMISecurityManager;
import java.rmi.Naming;

public class HelloServer {
        public static void main (String [ ] argv) {
                /* lancer SecurityManager */
                System.setSecurityManager (new RMISecurityManager()) ;
                try {
                        /* creer une instance de la classe Hello et l'enregistrer dans le serveur de noms */
                        Naming.rebind("Hello1", new Hello("Hello")) ;
                        System.out.println ("Serveur pret.") ;
                } catch (Exception e) {
                        System.out.println("Erreur serveur : " + e) ;
                }
        }
}
```

Exemple "Hello world" : compilation

- Sur la machine serveur : compiler les interfaces et les programmes du serveur
  - javac HelloInterface.java Hello.java HelloServer.java
- Sur la machine serveur : créer les talons client et serveur pour les objets appelés à distance (à partir de leurs interfaces) - ici une seule classe, Hello
  - rmic -keep Hello
  - N.B. cette commande construit et compile les talons. L'option -keep permet de garder les sources de ces talons. Depuis Java Platform, Standard Edition 5.0, cette étape n'est plus nécessaire.
- Sur la machine client : compiler les interfaces et le programme client
  - javac HelloInterface.java HelloClient.java
  - N.B. il est préférable de regrouper dans un fichier .jar les interfaces des objets appelés à distance, ce qui permet de les réutiliser pour le serveur et le client

Exemple "Hello world" : exécution

- Lancer le serveur de noms (sur la machine serveur)
  - rmiregistry
  - N.B. Par défaut, le registry écoute sur le port 1099. Si on veut le placer sur un autre port, il suffit de l'indiquer, mais il faut aussi modifier les URL en conséquence : rmi ://<serveur> :<port>/<répertoire>
- Lancer le serveur
  - java -Djava.rmi.server.codebase=file :///mnt/disk/res-rmi/ -Djava.security.policy=Hello.policy HelloServer
    - java -Djava.rmi.server.codebase=file :Q :\res-rmi -Djava.security.policy=Hello.policy HelloServer
  - N.B. Signification des propriétés (option -D) :
    - Le contenu du fichier Hello.policy spécifie la politique de sécurité, cf plus loin.
    - L'URL donnée par codebase sert au chargement de classes par le client
- Lancer le client
  - java -Djava.security.policy=Hello.policy HelloClient
  - N.B. Le talon client sera chargé par le client depuis le site du serveur, spécifié dans l'option codebase lors du lancement du serveur

Exemple "Hello world" : sécurité

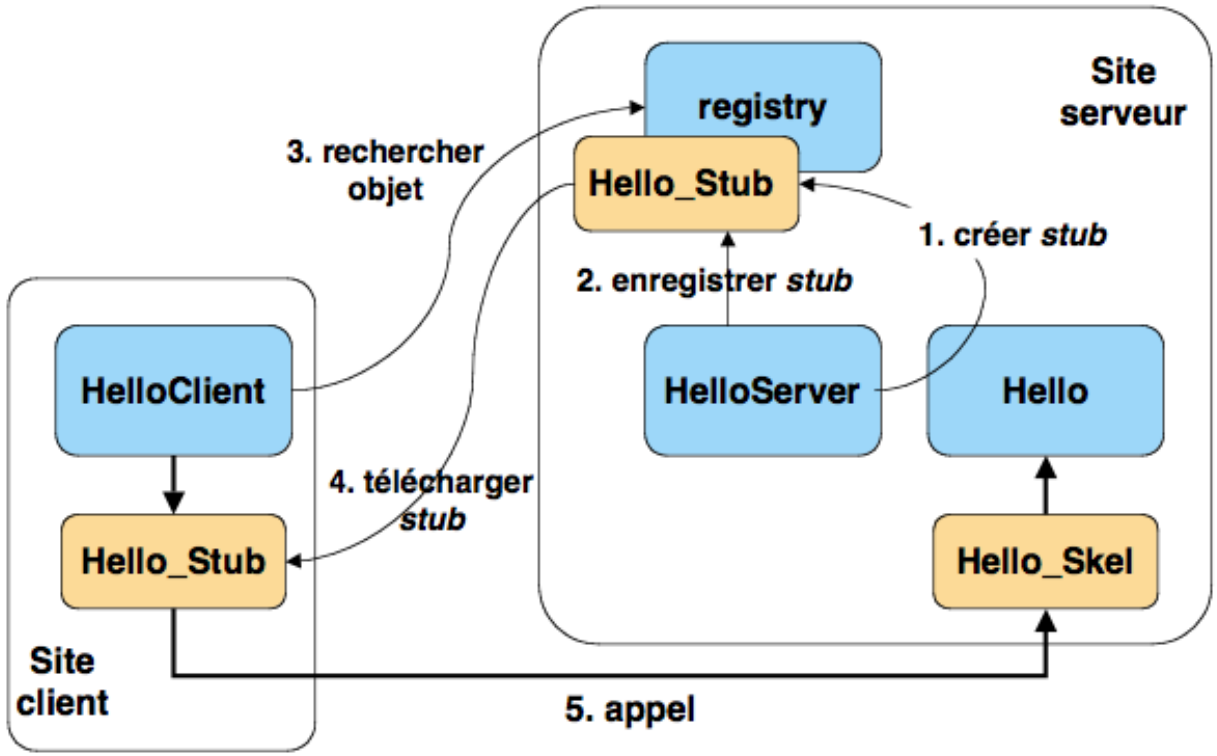
- Motivations
  - La sécurité est importante lorsqu'il y a téléchargement de code (il peut être dangereux d'exécuter le code chargé depuis un site distant)
- Mise en oeuvre
  - La politique de sécurité spécifie les actions autorisées, en particulier, sur les sockets
  - Exemples de contenu du fichier Hello.policy

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

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

  - Permet d'utiliser les sockets comme indiqué. Toute autre utilisation est interdite

Fonctionnement d'ensemble de Java RMI



Fabrique d'objets (Factory)

- Motivation
  - Permettre au client de construire des instances multiples d'une classe C sur le site serveur
  - Le new n'est pas utilisable tel quel (car il ne gère que la mémoire locale, celle du client)
  - Solution : appel d'un objet FabriqueC, qui crée localement (sur le serveur) les instances de C (en utilisant new C)

Exemple "Hello world Factory"

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

public interface HelloFactoryInterface extends Remote {
        /* methode qui cree un objet Hello */
        public HelloInterface createHello(String message) throws RemoteException;
}
```

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

public class HelloFactory extends UnicastRemoteObject implements HelloFactoryInterface {
        public HelloFactory() throws RemoteException {
        }

        /* l'implementation de la methode */
        public HelloInterface createHello(String message) throws RemoteException       {
                return new Hello(message);
        };

}
```

Exemple "Hello world Factory"

```
1  // HelloFactoryClient.java
2  import java.rmi.*;
3
4  public class HelloFactoryClient {
5          public static void main (String [ ] argv) {
6                  /* lancer SecurityManager */
7                  System.setSecurityManager (new RMISecurityManager()) ;
8                  try {
9                          /* trouver une reference vers l'objet distant */
10                         // HelloInterface hello = (HelloInterface) Naming.lookup("rmi://localhost/Hello1") ;
11                         /* appel de methode a distance */
12                         // System.out.println (hello.sayHello()) ;
13                         HelloFactoryInterface helloFactory = (HelloFactoryInterface) Naming.lookup("rmi://localhost
       /Factory") ;
14                         HelloInterface hello = helloFactory.createHello("Bonjour ");
15                         System.out.println (hello.sayHello());
16
17                 } catch (Exception e) {
18                         System.out.println ("Erreur client : " + e) ;
19                 }
20         }
21 }
```

Exemple "Hello world Factory"

```java
// HelloFactoryServer.java
import java.rmi.RMISecurityManager;
import java.rmi.Naming;

public class HelloFactoryServer {
        public static void main (String [ ] argv) {
                /* lancer SecurityManager */
                System.setSecurityManager (new RMISecurityManager()) ;
                try {
                        /* creer une instance de la classe Hello et l'enregistrer dans le serveur de noms */
                        // Naming.rebind("Hello1", new Hello("Hello")) ;
                        Naming.rebind("Factory", new HelloFactory()) ;
                        System.out.println ("Serveur pret.") ;
                } catch (Exception e) {
                        System.out.println("Erreur serveur : " + e) ;
                }
        }
}
```

Passage d'objets en paramètre

- Deux cas possibles
  - Passage en paramètre d'un objet local (sur la JVM de l'objet appelant)
    - Passage par valeur : on transmet une copie de l'objet (plus précisément : une copie de l'ensemble de ses variables d'état). Pour cela l'objet doit être sérialisable (i.e. implémenter l'interface java.io.Serializable)
  - Passage en paramètre d'un objet non-local (hors de la JVM de l'objet appelant, par ex. sur un site distant)
    - Passage par référence : on transmet une référence sur l'objet (plus précisément : un stub de l'objet). Le destinataire utilisera ce stub pour appeler les méthodes de l'objet.

Passage d'objets en paramètre

- Notions sur les objets sérialisables
  - Un objet sérialisable (transmissible par valeur hors de sa JVM) doit implémenter l'interface java.io.Serializable. Celle-ci est réduite à un marqueur (pas de variables ni d'interface)
  - Les objets référencés dans un objet sérialisable doivent aussi être sérialisables
  - Comment rendre effectivement un objet sérialisable ?
    - Pour les variables de types primitifs (int, boolean, ...), rien à faire
    - Pour les objets dont les champs sont constitués de telles variables : rien à faire
    - On peut éliminer une variable de la représentation sérialisée en la déclarant transient
    - Pour un champ non immédiatement sérialisable (ex. : Array),il faut fournir des méthodes readObject() et writeObject()
  - Exemples de sérialisation : passage en paramètres, écriture sur un fichier. Le support de sérialisation est un stream (flot) : classes java.io.ObjectOutputStream et java.io.ObjectInputStream.
  - Pour des détails techniques : voir javadoc de l'interface Serializable et des classes ObjectOutputStream et ObjectInputStream

Notions sur le fonctionnement interne de Java RMI

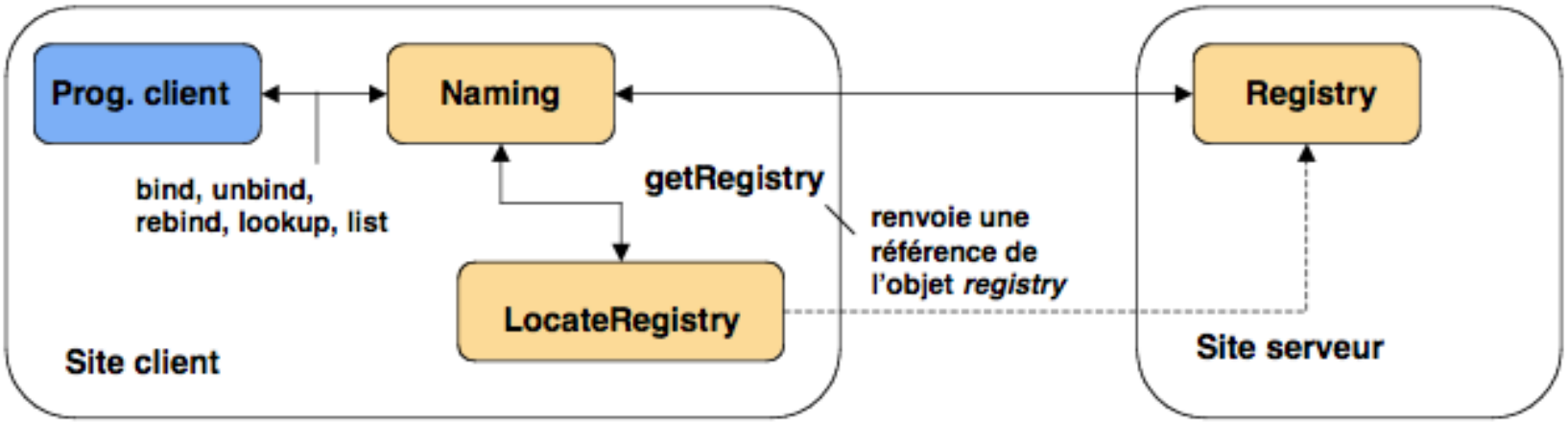
- La classe UnicastRemoteObject
  - Rappel de la règle d'usage : une classe d'objets accessibles à distance étend la classe java.rmi.UnicastRemoteObject.
    - Le principale fonction de cette classe se manifeste lors de la création d'une instance de l'objet.

```
1 public class AnnuaireImpl extends UnicastRemoteObject {...}
2 AnnuaireImpl (...) { // le constructeur de C appelle automatiquement
3         // le constructeur de la superclasse UnicastRemoteObject
4         ...
5         monAnnuaire = new AnnuaireImpl (...) ;
6 }
```

  - Le constructeur crée une instance de stub pour l'objet (en utilisant la classe Annuaire_Stub engendrée par rmic, et retourne ce stub comme résultat de la création
- Le contenu d'un stub
  - Un stub contient essentiellement une variable ref de type RemoteRef qui contient la localisation de l'objet (adresse IP, port)
  - Un appel de méthode se fait par appel de ref.invoke(...) qui utilise les sockets pour la communication

Notions sur le fonctionnement interne de Java RMI

- Le serveur de noms (registry)
  - Classes utiles (fournies par java.rmi)
    - Naming : sert de représentant local du serveur de noms. Permet d'utiliser les méthodes bind(), rebind(), lookup(), unbind(), list()
    - LocateRegistry : permet de localiser un serveur de noms (rmiregistry) et éventuellement d'en créer un. En général invisible au client (appelé en interne par Naming)



Conclusion sur Java RMI

- Extension du RPC aux objets
  - Permet l'accès à des objets distants
  - Permet d'étendre l'environnement local par chargement dynamique de code
  - Pas de langage séparé de description d'interfaces (IDL fourni par Java)
- Limitations
  - Environnement restreint à un langage unique (Java)
    - Mais passerelles possibles, en particulier RMI/IIOP
  - Services réduits au minimum
    - Service élémentaire de noms (sans attributs)
    - Pas de services additionnels : Duplication d'objets, Transactions, ...

Compléments

- Tout n'a pas été traité ici. Il reste des points complémentaires, dont
  - Parallélisme sur le serveur
    - Un serveur RMI peut servir plusieurs clients. Dans ce cas, un thread séparé est créé pour chaque client sur le serveur. C'est au développeur du programme serveur d'assurer leur bonne synchronisation (exemple : méthodes synchronized) pour garantir la cohérence des données)
  - Activation automatique de serveurs
    - Un veilleur (demon) spécialisé, rmid, peut assurer le lancement de plusieurs classes serveur sur un site distant (cette opération est invisible au client)
  - Téléchargement de code
    - Un chargeur spécial (ClassLoader) peut être développé pour assurer le téléchargement des classes nécessaires

References

- Cours de S. Krakowiak
- http ://docs.oracle.com/javase/tutorial/rmi/
- "Distributed Systems : Principles and Paradigms", A. S. Tanenbaum
- "Distributed Systems", P. Krzyzanowski