Intergiciels

Java Remote Method Invocation

© 2010, Laurent Vercouter

Ecole Nationale Supérieure des Mines SAINT-ETIENNE

Remote Method Invocation

Solution proposée par SUN pour adapter le principe des RPC à la programmation orientée objet.

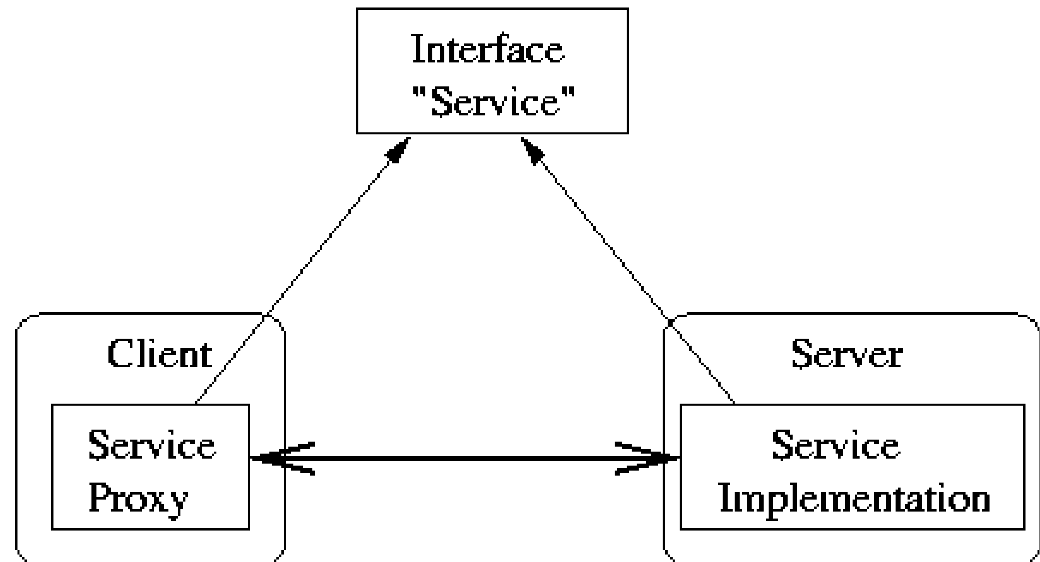
- Technologie fondée sur le langage Java (à partir du JDK 1.1)
- Localisation d'objets distants
- Invocation de méthodes sur des objets distants
- Transfert d'objets en paramètres ou en retour (*serialization*)

© 2010, Laurent Vercouter

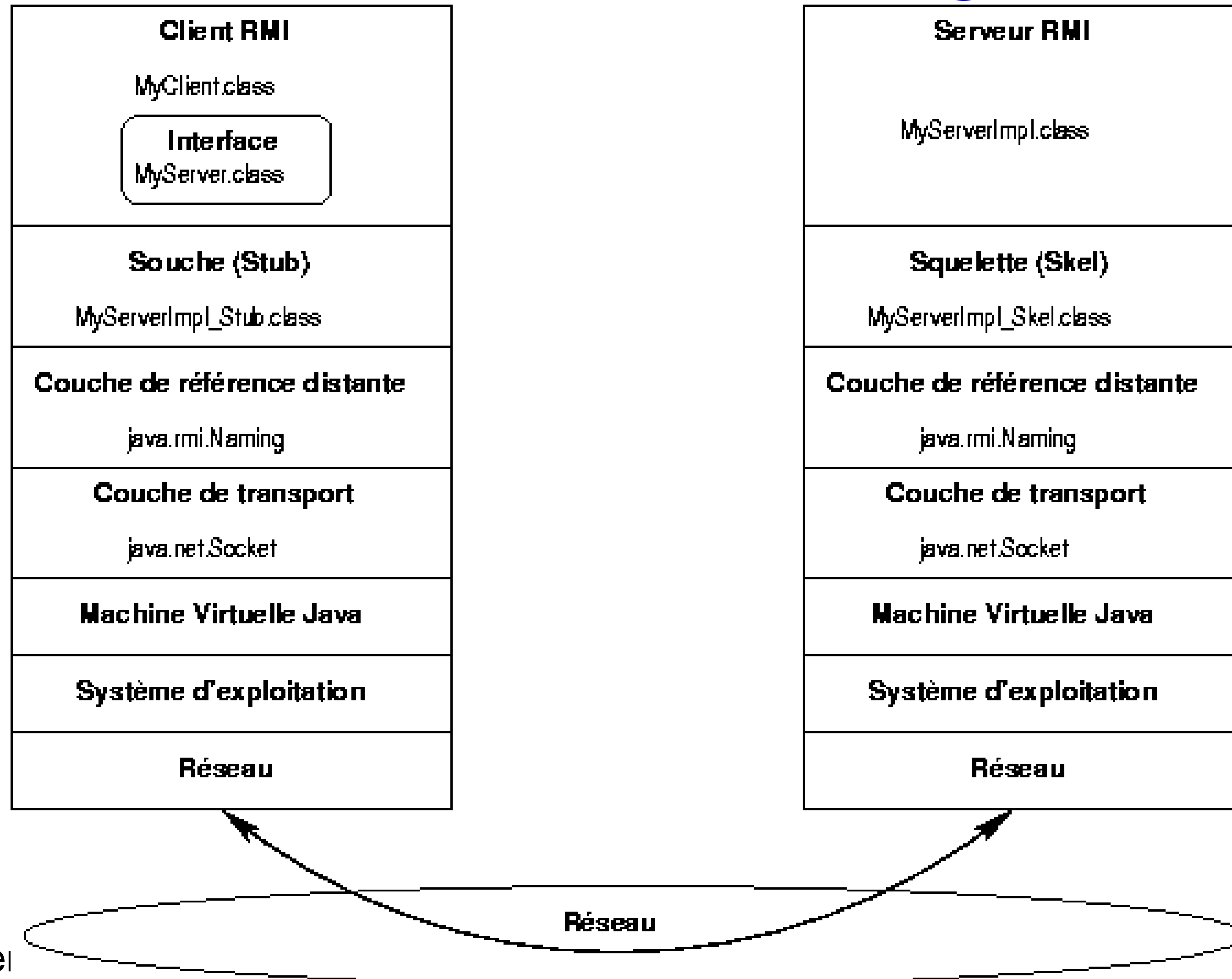
RMI : principe général

Séparation entre la définition d'un service et son implémentation

- Interface du service
- Implémentation du service
- Proxy vers le service



RMI : architecture logique



Couche Application

du côté serveur et client : l'interface *MyServer.class*

- hérite de l'interface *java.rmi.Remote*
- décrit les méthodes invocables à distance
- chacune de ces méthodes peut lever une exception *java.rmi.RemoteException*

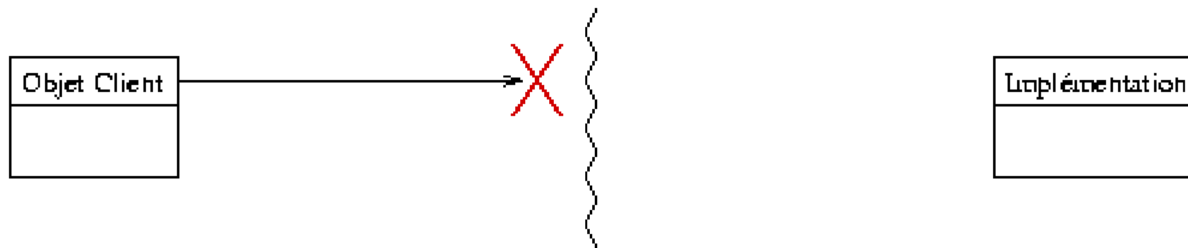
du côté serveur : la classe *MyServerImpl.class*

- hérite de la classe *java.rmi.UnicastRemoteObject*
- implémente l'interface *MyServer.class*
- doit s'enregistrer auprès d'un serveur de références (cf. couche référence distante)

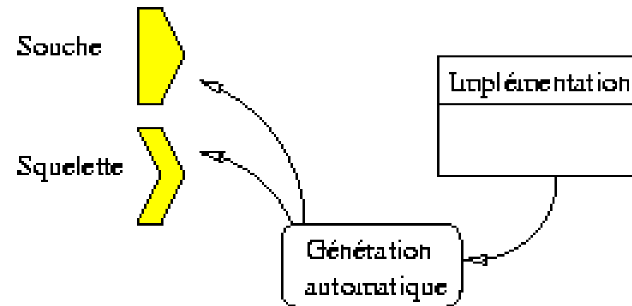
© 2010, Laurent Vercouter

Couche Souche et Squelette

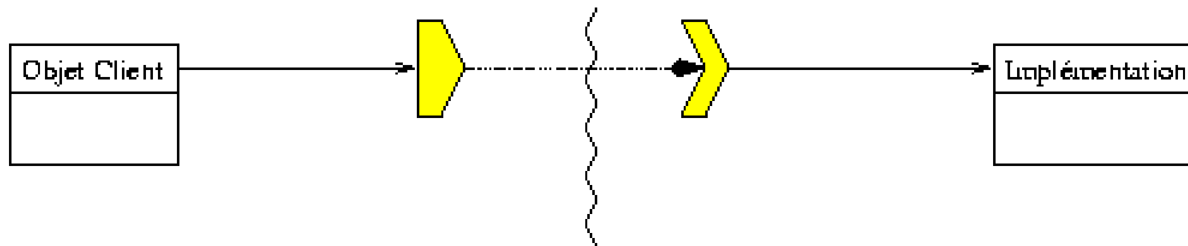
1. Le client ne peut pas invoquer directement une méthode de l'implémentation par le réseau



2. Une souche et un squelette capables de communiquer par le réseau sont automatiquement générés

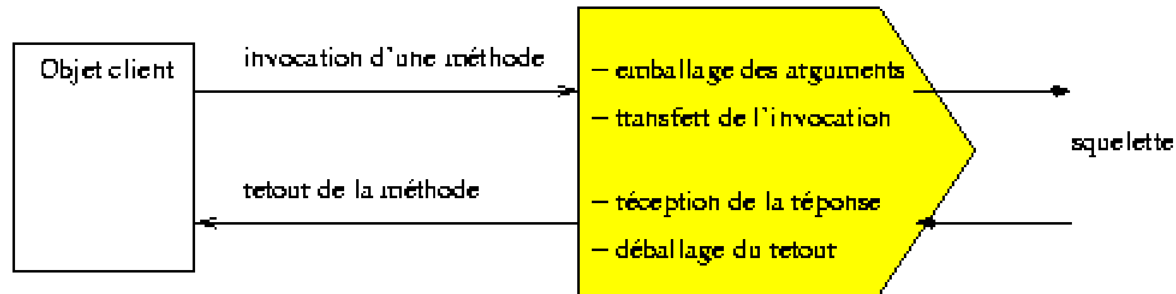


3. La souche est déployée chez le client et le squelette chez le serveur



© 2010, Laurent Vercouter

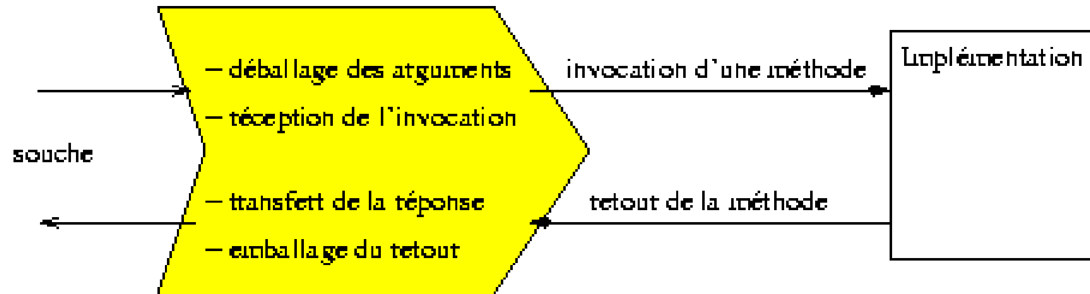
La souche



- Présente du côté client
- Représente un objet distant
- Convertit les arguments en un format transmissible via le réseau (*marshalling*)
- Reconstruit les valeurs de retour à partir de données reçues par le réseau (*unmarshalling*)
- **Il est généré automatiquement depuis le JDK 1.5**

© 2010, Laurent Vercouter

Le squelette



- Présent du côté serveur
- Invoque des méthodes sur l'objet local référencé pour le compte d'une souche
- Convertit les valeurs de retour en un format transmissible via le réseau (*unmarshalling*)
- Reconstruit les argments à partir de données reçues par le réseau (*marshalling*)
- **Il est généré automatiquement depuis le JDK 1.2**

© 2010, Laurent Vercouter

Couche de référence distante et de transport

Couche de **référence distante**

- gère les invocations de méthode distante
- repose sur un processus *rmiregistry* qui centralise les références à des serveurs RMI

Couche de **transport**

- assure la connexion avec un hôte distant
- écoute les connexions entrantes

© 2010, Laurent Vercouter

Enregistrement d'un service

Hote du client

MyObjectClient

Hote du serveur

rmiregistry

MyObject

...

...

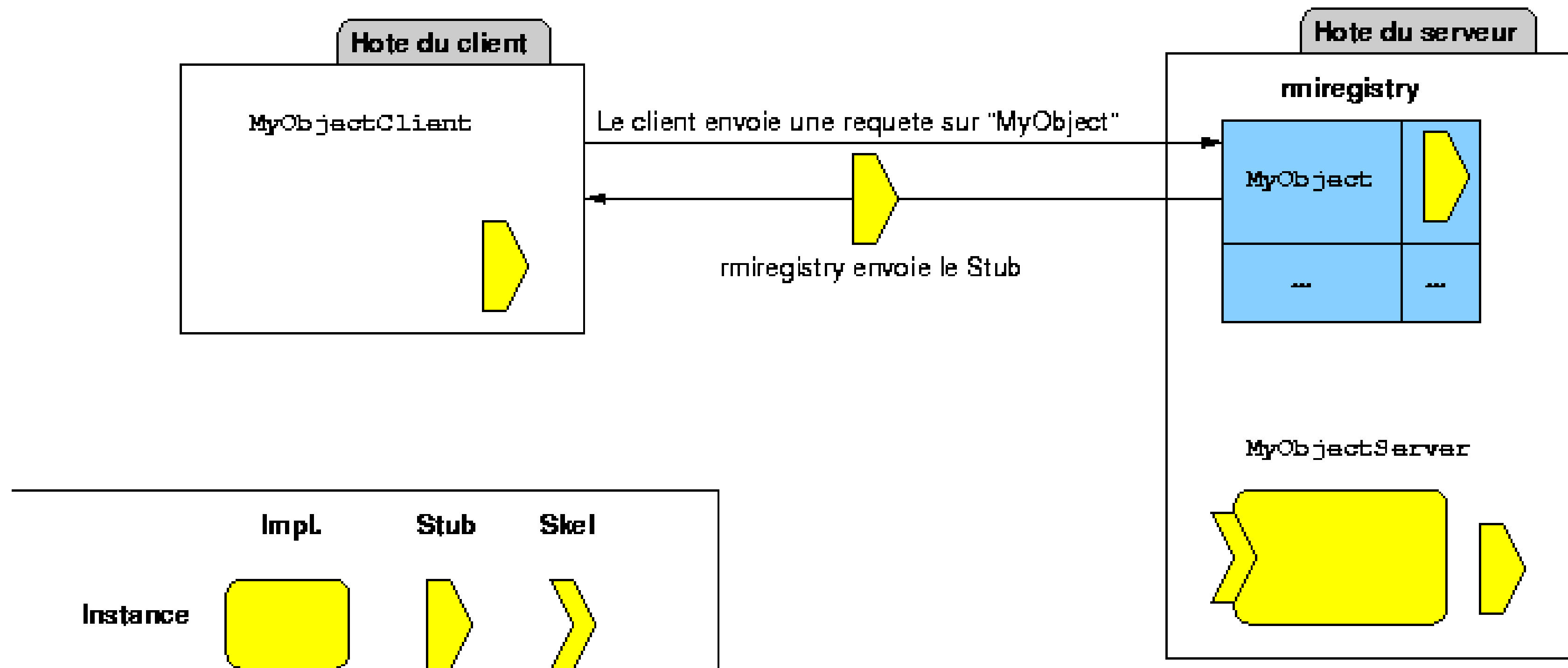
enregistrement de l'objet

MyObjectServer

	Impl.	Stub	Skel
Instance			

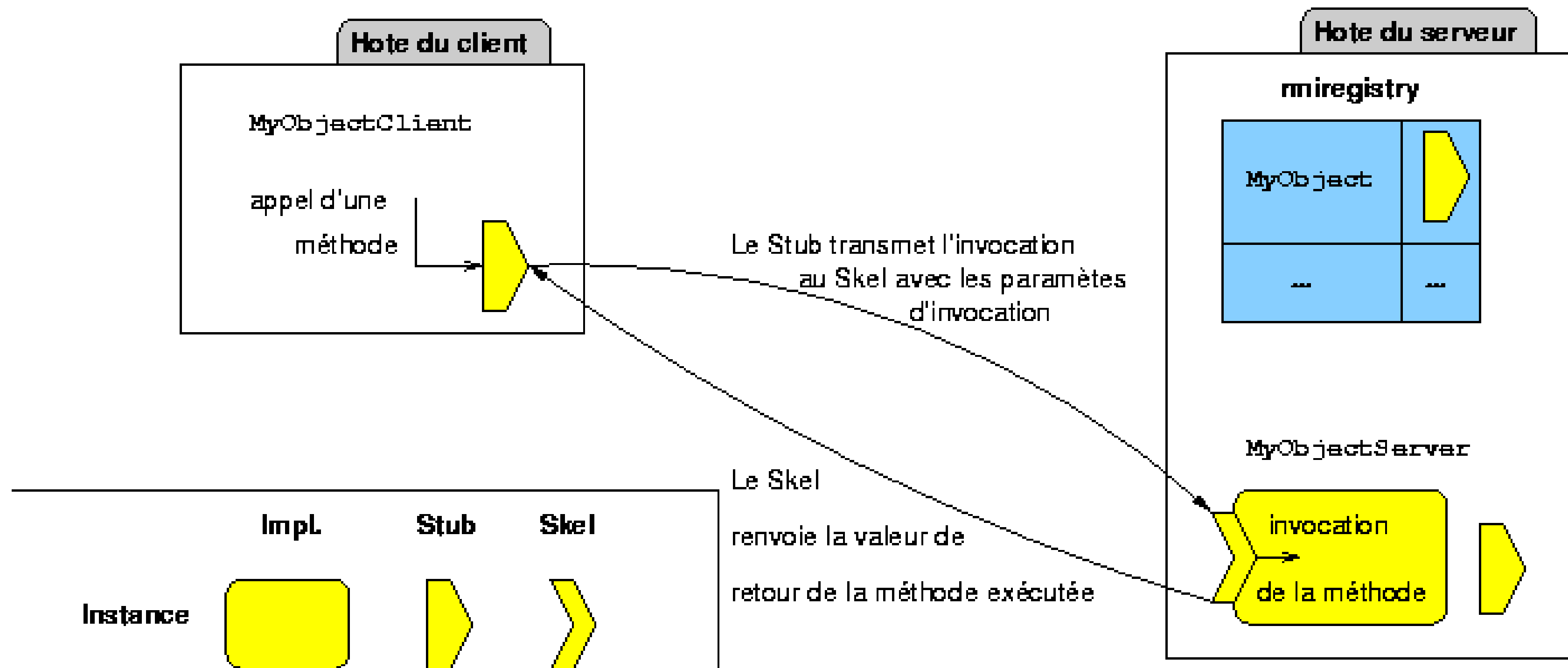
© 2010, Laurent Vercouter

Accès à une référence distante

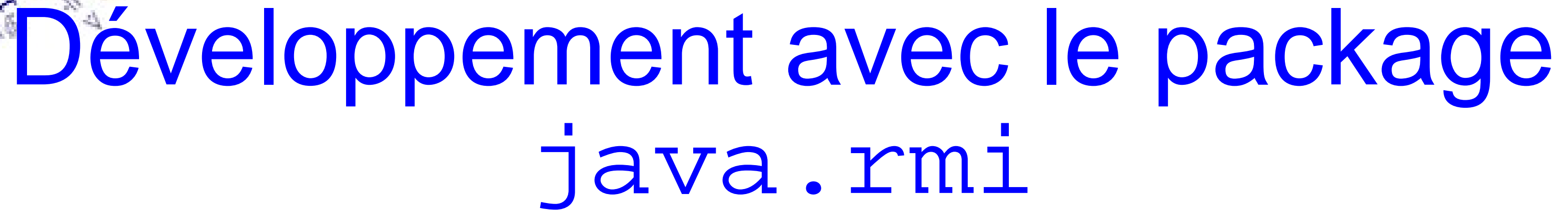


© 2010, Laurent Vercouter

Invocation d'une méthode



© 2010, Laurent Vercouter

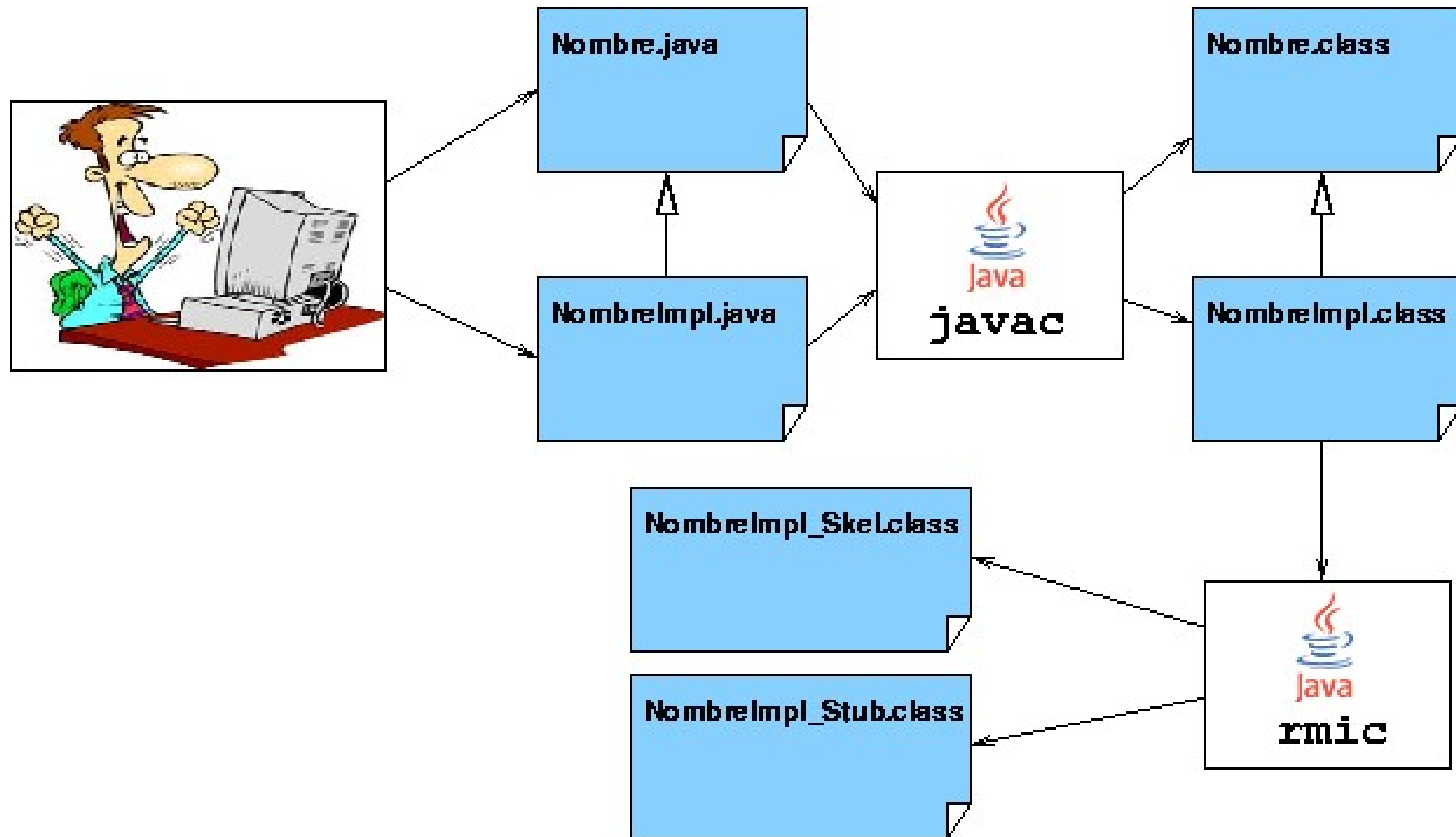


Développement avec le package `java.rmi`

1) L'interface du service (hérite de *java.rmi.Remote*)
2) L'implémentation du service (hérite de *java.rmi.server.RemoteObject*, par exemple *java.rmi.server.UnicastRemoteObject*)
3) La création des Stub et Skel anciennement par *rmic*, maintenant transparente
4) L'enregistrement du service dans un *rmiregistry*
5) L'accès à une référence distante chez le client (par *java.rmi.registry.Registry*)

© 2010, Laurent Vercouter

Exemple : Implémentation et génération du service Nombre



© 2010, Laurent Vercouter

Interface d'un service Nombre

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


public interface Nombre extends Remote {
   public int hasard() throws RemoteException;
   public boolean estPair(int a) throws
     RemoteException;
}
```

© 2010, Laurent Vercouter

Implémentation d'un service Nombre

```
import java.rmi.RemoteException;
import java.rmi.server.UnicastRemoteObject;
public class NombreImpl extends UnicastRemoteObject
            implements Nombre {
   java.util.Random random = new java.util.Random();
   public NombreImpl() throws RemoteException
       {super();}
   public int hasard() throws RemoteException
       {return random.nextInt();}
   public boolean estPair(int a) throws RemoteException
       {return (a%2 == 0);}
}
```

© 2010, Laurent Vercouter

Génération de la souche et du squelette

La génération des classes *NombreImpl_Stub* (et *NombreImpl_Skel)* se fait par l'outil **rmic** à partir d'une classe implémentant un service.

```
rmic NombreImpl
```

Les fichiers *NombreImpl_Stub.class* et *NombreImpl_Skel.class* doivent être présents ou téléchargeables sur le serveur et *NombreImpl_Stub.class* chez le client.

Les fichiers .java générés sont effacés sauf si l'option `-keepgenerated` est spécifiée.

© 2010, Laurent Vercouter

Enregistrement du service

Pour enregistrer une instance de la classe *NombreImpl*, il faut qu'un processus **rmiregistry** s'exécute sur l'hôte.

L'instance s'enregistre en inscrivant dans le rmiregistry sa classe *NombreImpl_Stub* associée à un nom. Une URL est formée pour l'enregistrement puis l'accès sous la forme :

```
rmi://<host_name>[:port]/[service_name]
```

Par exemple :

```
rmi://localhost/ObjetNombre
```

© 2010, Laurent Vercouter

Enregistrement du service (2)

```
public static void main(String[] args) {
    try {
        NombreImpl obj = new NombreImpl();
        if (args.length > 0)
            Naming.rebind("rmi://"+args[0]+"/ObjetNombre"
              , obj);
        else
            Naming.rebind("rmi://localhost/ObjetNombre",
              obj);
        System.out.println("ObjetNombre bound in
          registry");
    } catch (Exception ex) {ex.printStackTrace();}
}
```

© 2010, Laurent Vercouter

Enregistrement du service (2)

```
public static void main(String[] args) {
    try {
        NombreImpl obj = new NombreImpl();
        if (args.length > 0)
            Registry registry =
              LocateRegistry.getRegistry(args[0]);
            registry.rebind(ObjetNombre, obj);
        else
            Registry registry =
              LocateRegistry.getRegistry();
            registry.rebind(ObjectNombre, obj);
        System.out.println("ObjetNombre bound in
          registry");
    } catch (Exception ex) {ex.printStackTrace();}
}
```

© 2010, Laurent Vercouter

Accès à une référence distante

```
import java.rmi.Naming;
public class ClientNombre {
    public static void main(String[] args) {
        try {
            Nombre objDistant;
            if (args.length > 0) objDistant = (Nombre)Naming.lookup(
                    "rmi://" + args[0] + "/ObjetNombre");
            else objDistant = (Nombre)Naming.lookup(
                    "rmi://localhost/ObjetNombre");
            int tmp = objDistant.hasard();
            if (objDistant.estPair(tmp))
                System.out.println(tmp + " est pair");
            else System.out.println(tmp + " est impair");
        } catch (Exception ex) {
            ex.printStackTrace();
        }
    }
}
```

© 2010, Laurent Vercouter

Accès à une référence distante

```
public class ClientNombre {
    public static void main(String[] args) {
        try {
            Nombre objDistant;
            if (args.length > 0) objDistant = (Nombre) LocateRegistry
                .getRegistry(args[0])
                .lookup("ObjetNombre");
            else objDistant = (Nombre)LocateRegistry.getRegistry()
                .lookup("ObjetNombre");
            int tmp = objDistant.hasard();
            if (objDistant.estPair(tmp))
                System.out.println(tmp + " est pair");
            else System.out.println(tmp + " est impair");
        } catch (Exception ex) {
            ex.printStackTrace();
        }
    }
}
```

© 2010, Laurent Vercouter

Exécution du serveur

L'exécution d'un serveur RMI nécessite de fixer quelques propriétés:

- L'hote et le chemin d'accès des classes (de l'interface, éventuellement des implémentations) du serveur

```
-Djava.rmi.server.codebase=<repertoire>
-Djava.rmi.server.hostname=<hote>
```

- Le fichier définissant la politique de sécurité

```
-Djava.security.policy=<fichier_policy>
```

Exemple:

```
java
  -Djava.rmi.server.codebase=''http://www.emse.fr/~vercouter/rmi/''
  -Djava.rmi.server.hostname=pc-vercouter.emse.fr
  -Djava.security.policy=./server.policy
  NombreImpl
```

Exercice : place de Marché

Définir une interface Java décrivant les méthodes accessibles à distance d'un serveur RMI gérant une place d'enchères.

- Un utilisateur s'identifie par login/mot de passe et se voit attribuer un numero de session (utilisé pour toutes les transactions).
- Les utilisateurs peuvent être à la fois vendeurs et acheteurs.
- S'il joue le rôle de vendeur, il place un produit (décrit par un simple nom) à vendre avec un prix de base et une date de fin d'enchères. Il peut ensuite connaître le login de l'acheteur ayant remporté l'enchère.
- S'il joue le rôle d'acheteur, il peut consulter la liste des produits en vente. Il peut enchérir sur un produit puis à la fin d'une enchère savoir s'il l'a emportée ou non.
- N'importe quel utilisateur identifié peut consulter les informations relatives à un produit (prix de base, plus haute mise en cours, date de fin d'enchère)

© 2010, Laurent Vercouter

Sécurité et RMI (1)

Un niveau spécifique de permissions est accordé à un serveur pour renforcer la sécurité du système.

Il faut d'abord ajouter un *SecurityManager*, il en existe un dédié aux applications RMI qui sera installé par défaut :

```
public void main(String[] args) {
    try {
        System.setSecurityManager(new RMISecurityManager());
        NombreImpl obj = new NombreImpl();
        Registry registry = LocateRegistry.getRegistry();
        registry.rebind("rmi://localhost/ObjetNombre", obj);
        System.out.println("ObjetNombre bound in registry");
    } catch (Exception ex) {
        ex.printStackTrace();
    }
}
```

© 2010, Laurent Vercouter

Sécurité et RMI (2)

L'ensemble des permissions accordées par ce *SecurityManager* sont définies soit :

- en surchargeant certaines méthodes de l'instance (cf. la classe *java.lang.SecurityManager*)
- en écrivant un fichier externe indiqué dans l'option d'exécution `-Djava.security.policy=<fichier_policy>`

Syntaxe simplifiée d'un fichier policy :

```
grant {
    permission <permission_class_name> [target_name] [,action]
    permission <permission_class_name> [target_name] [,action]
    ...
};
```

© 2010, Laurent Vercouter

Exemple de fichiers `policy`

```
grant {
    permission java.security.AllPermission;
};

grant {
    permission java.io.FilePermission "/tmp/*",
      "read,write";
    permission java.util.PropertyPermission
      "user.country", "read";
    permission java.net.SocketPermission
      "localhost:1099", "connect,accept,resolve";
};
```

© 2010, Laurent Vercouter

Problèmes fréquents

- Assurez-vous qu'un processus *rmiregistry* tourne lors de vos tests d'exécution
- Vérifier bien que la machine virtuelle de votre serveur est lancée avec 3 paramètres
  - -Djava.security.policy=./my_policy
  - -Djava.rmi.server.codebase="http://www.emse.fr/~vercouter/TPrmi/"
  - -Djava.rmi.server.hostname=pc-vercouter.emse.fr
- Si un *SecurityManager* est utilisé, il faut préciser un fichier de type *Policy* au lancement de la JVM

© 2010, Laurent Vercouter

Passage d'arguments

Il y a 3 possibilités de passage d'arguments lors de l'invocation d'une méthode distante

- Le paramètre est d'un type primitif : passage par valeur
- Le paramètre est un objet
  - Il est sérialisé et envoyé au serveur
  - Une référence distante est envoyée

Idem pour la valeur de retour d'une méthode distante

© 2010, Laurent Vercouter

Sérialisation d'objets (1)

La sérialisation est une opération (on parle aussi de *marshalling* et d'*unmarshalling*) qui consiste à transformer un objet dans un format transférable par un flux de données (cf. les classes *ObjectInputStream* et *ObjectOutputStream* du package *java.io*). On s'en sert principalement dans 2 cas :

- Sauvegarder un objet dans un fichier
- Déplacer un objet d'une machine virtuelle vers une autre

C'est donc une **copie** de l'objet qui est envoyée au serveur (ou au client dans le cas d'une valeur de retour).

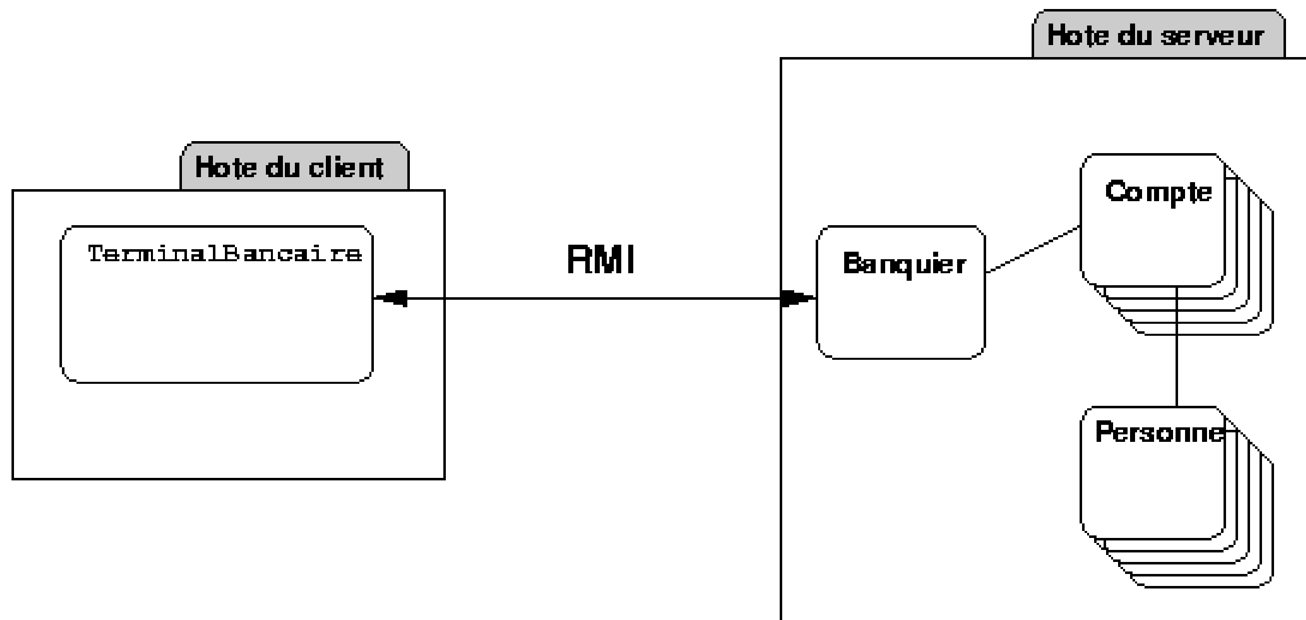
© 2010, Laurent Vercouter

Sérialisation d'objets (2)

- Pour être sérialisable, un objet doit implémenter l'interface *java.io.Serializable*;
- Tous les objets contenus dans l'objet seront aussi sérialisés (ils doivent donc être aussi sérialisables);
- La sérialisation est effectuée automatiquement.

© 2010, Laurent Vercouter

Exemple : application banque (1)



© 2010, Laurent Vercouter

Exemple : application banque (2)

```
public class TerminalBancaire {
    public static void main(String[] args) {
        try {
            Banquier bank = (Banquier)LocateRegistry
              .getRegistry(args[0]).lookup("Banquier")
            Personne pers = new Personne("Laurent");
            Compte co = bank.creeCompte(pers);
            co.verse(12);
        } catch (Exception ex) {
            ex.printStackTrace();
        }
    }
}
```

Exemple : application banque (3)

```
public class BanquierImpl extends UnicastRemoteObject implements
   Banquier {

    Hashtable liste = new Hashtable();

    public BanquierImpl() throws RemoteException {
        super();
    }
    public Compte creeCompte(Personne p) throws RemoteException
      {
        Compte c = new Compte(p);
        liste.put(p.getNom(),c);
        return c;
    }
}
```

© 2010, Laurent Vercouter

Exemple : application banque (4)

```
public class Personne implements java.io.Serializable {
    String nom;
    public Personne(String n) {
        nom = n;
    }
    public String getNom() {
        return nom;
    }
}
```

La sérialisation est-elle aussi adaptée pour les objets `Compte` ?

© 2010, Laurent Vercouter

Référence distante

De la même manière qu'on accède à un service RMI grâce à une référence distante (*Stub*), on peut transférer uniquement cette référence pour un objet passé en paramètre ou comme valeur de retour.

Pour cela 3 conditions doivent être respectées :

- L'objet en question hérite de la classe *UnicastRemoteObject*;
- Il implémente une interface partagée avec l'autre partie (client ou serveur);
- Sa référence distante (*Stub*) est accessible par l'autre partie.

© 2010, Laurent Vercouter

Exemple : application banque (5)

```
public class CompteImpl extends UnicastRemoteObject
               implements Compte {
    Personne proprio;
    float solde = 0;
    public CompteImpl(Personne p) throws RemoteException {
        proprio = p;
    }
    public void verse(float montant) throws RemoteException {
        solde += montant;
    }
    public float getSolde() throws RemoteException {
        return solde;
    }
}
```

© 2010, Laurent Vercouter

Exemple : application banque (6)

```
public class BanquierImpl extends UnicastRemoteObject
                implements Banquier {
    Hashtable liste = new Hashtable();
    public BanquierImpl() throws RemoteException {
        super();
    }
    public Compte creeCompte(Personne p) throws
      RemoteException {
        Compte c = new CompteImpl(p);
        liste.put(p.getNom(),c);
        return c;
    }
}
```

© 2010, Laurent Vercouter

Liens utiles

- Tutoriaux SUN sur Java RMI
  - http://java.sun.com/docs/books/tutorial/rmi/index.html
- Quelques autres cours
  - http://etna.int-evry.fr/cours/middleware/
  - http://www2.lifl.fr/~seinturi/middleware/index.html
  - http://rangiroa.essi.fr/cours/langage/01-java-rmi.pdf

© 2010, Laurent Vercouter