



# Architecture client/serveur Programmation middleware

Dr. Dimassi J.

*Jamildimassi@topnet.tn*



# Plan du semestre


- Architecture Client / Serveur
- Appel de méthodes distantes dans les langages OO (Remote Method Invocation) RMI
- Modèle d'intergiciel orienté objets répartis (Common Object Request Broker Architecture) CORBA
- Web service
- Programmation Middleware : EJB



# Déroulement du cours

- Cours : 1heure et demi par semaine
- TP : 3 heures par quinzaine (Projet)
- Pré requis pour le cours :

☐ **Java**, UML, Design patterns

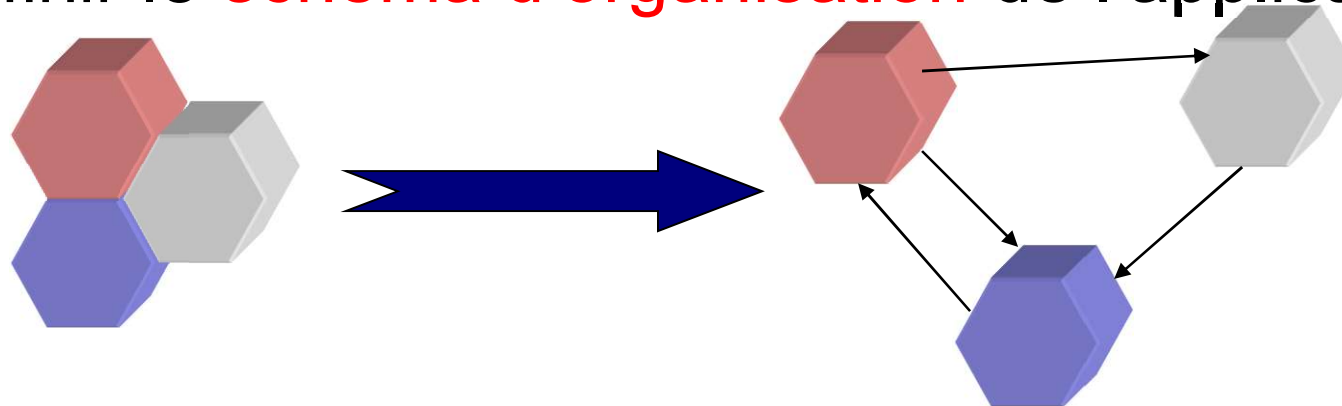


# Qu'est ce qu'une application répartie ?

- Il s'agit d'une application découpée en plusieurs unités
  - Chaque unité peut être placée sur **une machine différente**
  - Chaque unité peut s'exécuter sur **un système différent**
  - Chaque unité peut être programmée dans **un langage différent**

# Construction d'une application répartie

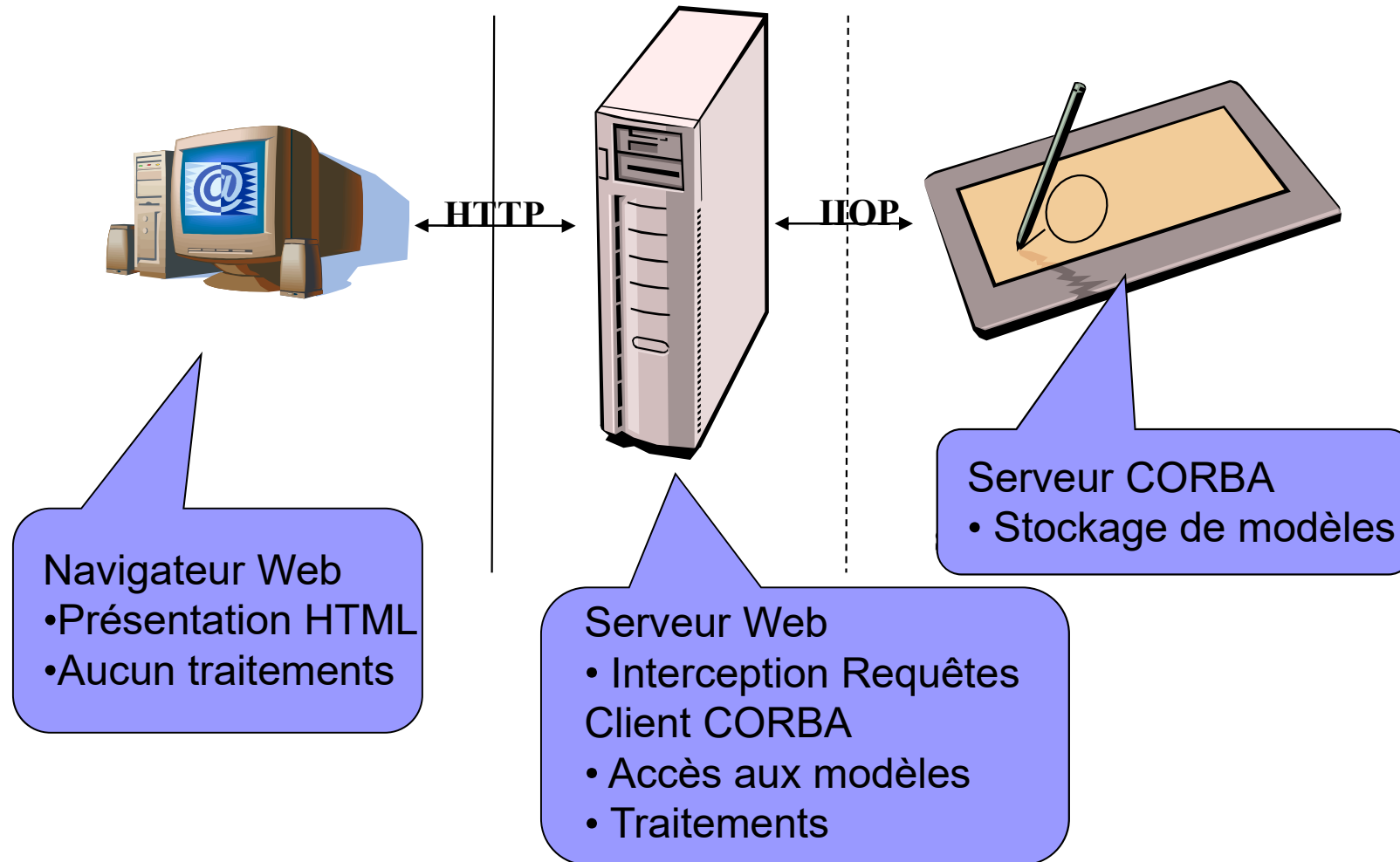
- **Identifier** les éléments fonctionnels de l'application pour les **regrouper** au sein d'unités
- **Estimer les interactions** entre unités
- Définir le **schéma d'organisation** de l'application



*Application monolithique*

*Application répartie*

# Exemple d'application répartie





# Avantages du réparti

- Organisationnel

- ☐ Décentraliser les responsabilités
- ☐ Découpage en unité

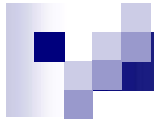
- Fiabilité et disponibilité

- ☐ Individualisation des défaillances
- ☐ Duplication des constituants de l'application

- Performance

- ☐ Partage de la charge

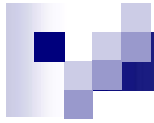
- Maintenance et évolution



# Inconvénients du réparti

- Une mise en œuvre plus délicate
  - ☐ Gestion des erreurs
  - ☐ Suivi des exécutions
- Pas de vision globale instantanée
  - ☐ Délais des transmissions
- Administration plus lourde
  - ☐ Installation
  - ☐ Configuration
  - ☐ Surveillance
- Coût
  - ☐ Formation
  - ☐ Achat des environnements





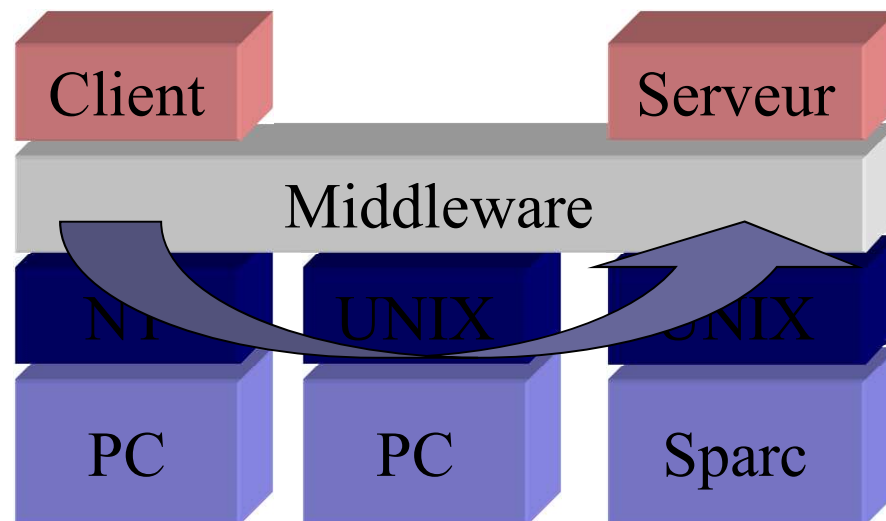
# Middleware : Rôles de base

- Résoudre l'Interopérabilité : Unifier l'accès à des machines distantes
- Résoudre l'Hétérogénéité : Etre indépendant des systèmes d'exploitation et du langage de programmation des applications

# Middleware : Mécanismes de base

## 1.(C/S)

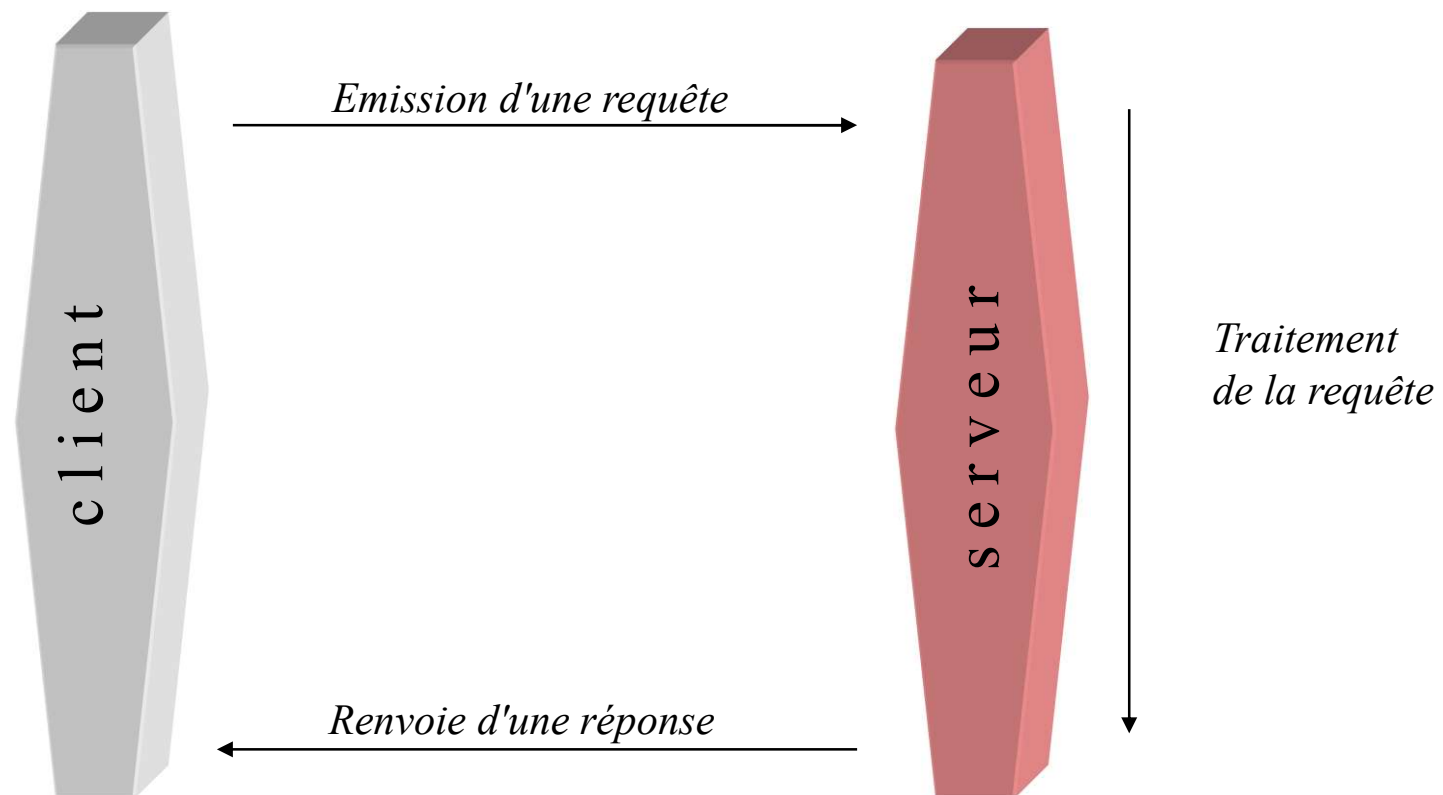
Les environnements répartis sont basés ( pour la plupart ) sur **un mécanisme RPC ( Remote Procedure Call )**.



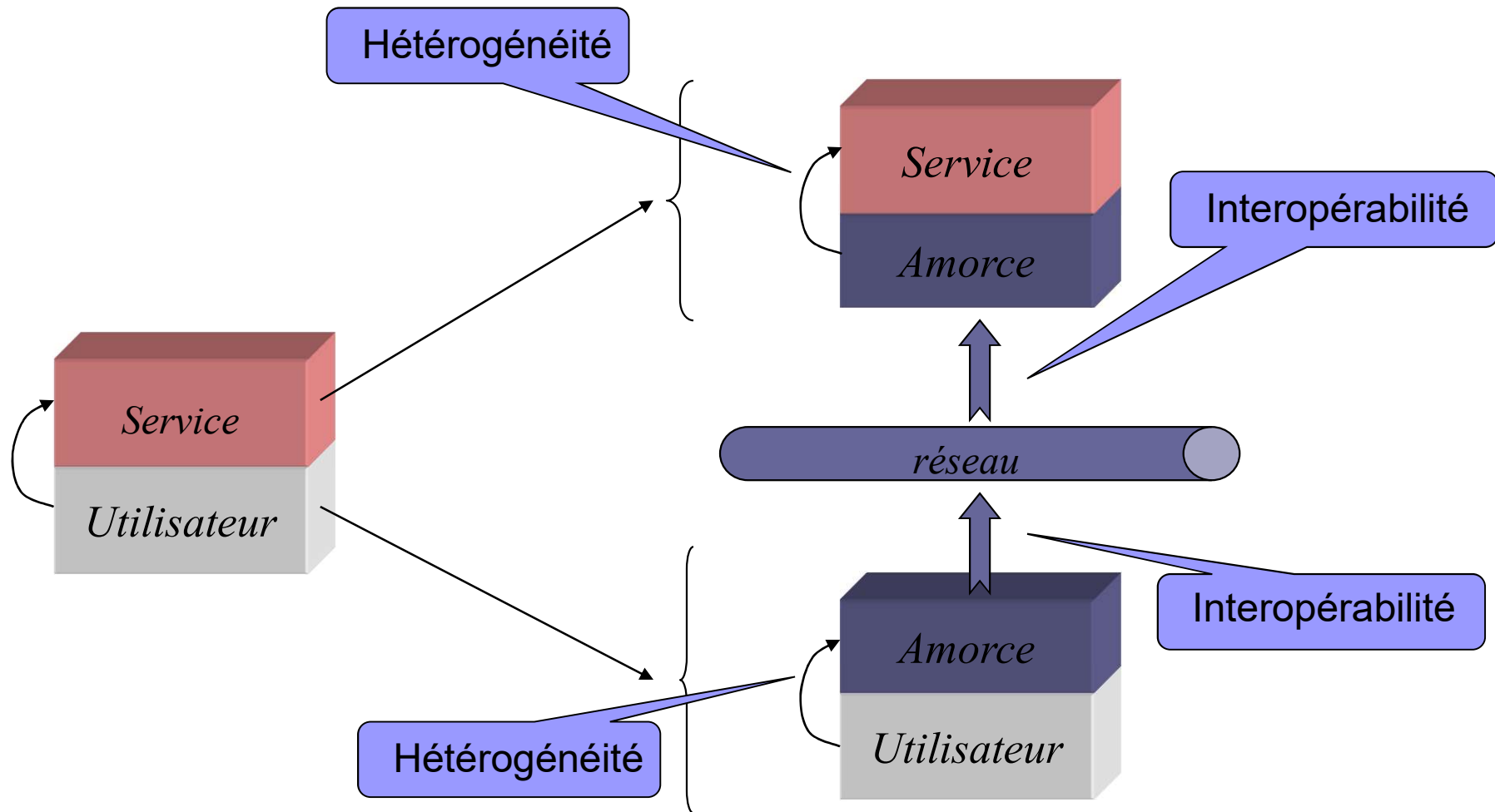
Ce mécanisme fonctionne en mode **requête / réponse**.

- Le client effectue une requête ( demande un service ),
- Le serveur traite la demande puis retourne une réponse au client

# Illustration du RPC



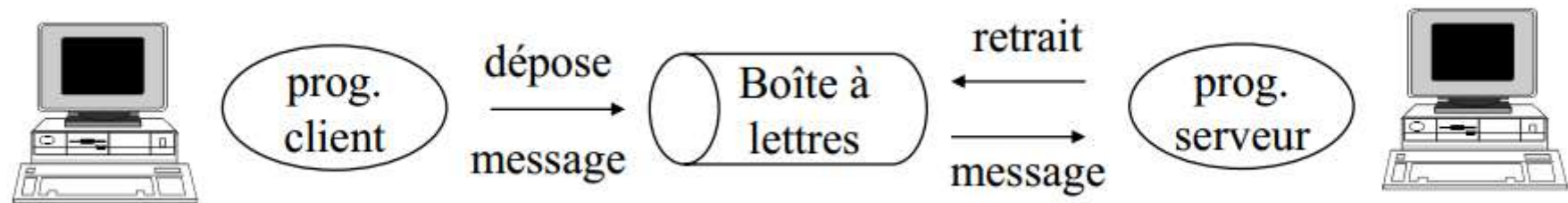
# Les amorces



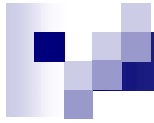
# Middleware : Mécanismes de base

## 2.(MOM)

- interaction par messagerie (MOM : Message-Oriented Middleware)

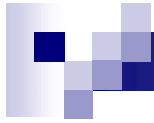


- MOM : comm. asynchrone (fonctionnement client et serveur découplés)
- Interaction client/serveur comm. synchrone



# Middleware : Rôles Avancés

- Nommage
  - Identification logique (DNS)
- Persistance
  - Liens vers SGBD
- Sécurité
  - Authentification, Autorisation, ...
- Transaction
  - ACID (Atomic , Coherent, Isolation, durability)
- Événement
  - Message Oriented Middleware (MOM)



# Evolution des Middlewares

- Objets

- ☐ CORBA (ORBIX, VisiBroker, OpenORB, ...)
- ☐ DCOM

- Composant

- ☐ J2EE (Websphere, Weblogic, JBOSS)
- ☐ .Net

- Web-Service

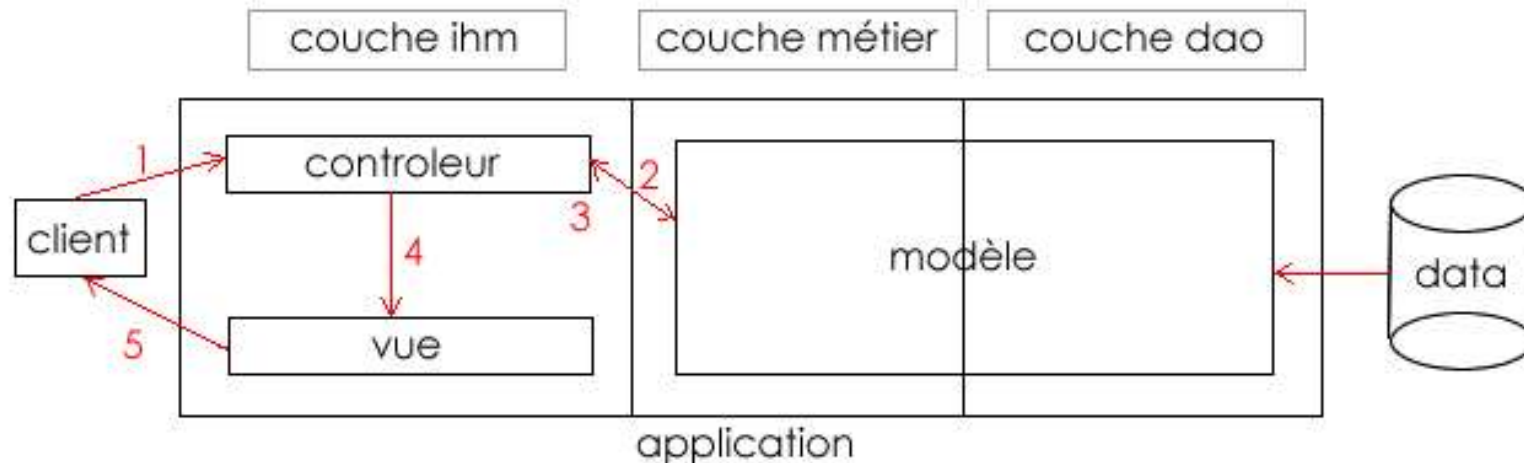


# Vue globale de MVC

“Dans le paradigme MVC l’entrée utilisateur, la modélisation du monde extérieur, l’aspect visuel présenté à l’utilisateur sont explicitement séparés et gérés par trois types d’objet, chacun spécialisé dans sa tâche.” [Burbeck 92]



# MVC en action



1. le client fait une demande au contrôleur. Ce contrôleur voit passer toutes les demandes des clients
2. le contrôleur doit traiter la demande. Pour ce faire, il peut avoir besoin de la couche métier, cette dernière peut éventuellement accéder aux données (via la couche dao)
3. le contrôleur effectue les traitements nécessaires sur / avec les objets renvoyés par la couche métier
4. le contrôleur sélectionne et nourrit la (les) vue(s) pour présenter les résultats du traitement qui vient d'être effectuée
5. la vue est enfin envoyée au client par le controleur



## Partie II

# Remote Method Invocation RMI



# RMI : Origine et Objectifs

- Solution (SUN) pour adapter le principe des RPC à la POO (à partir du JDK 1.1).
- Rendre transparent la manipulation d'objets situés dans un autre espace d'adressage
- Les appels doivent être transparents que l'objet soit local ou distant

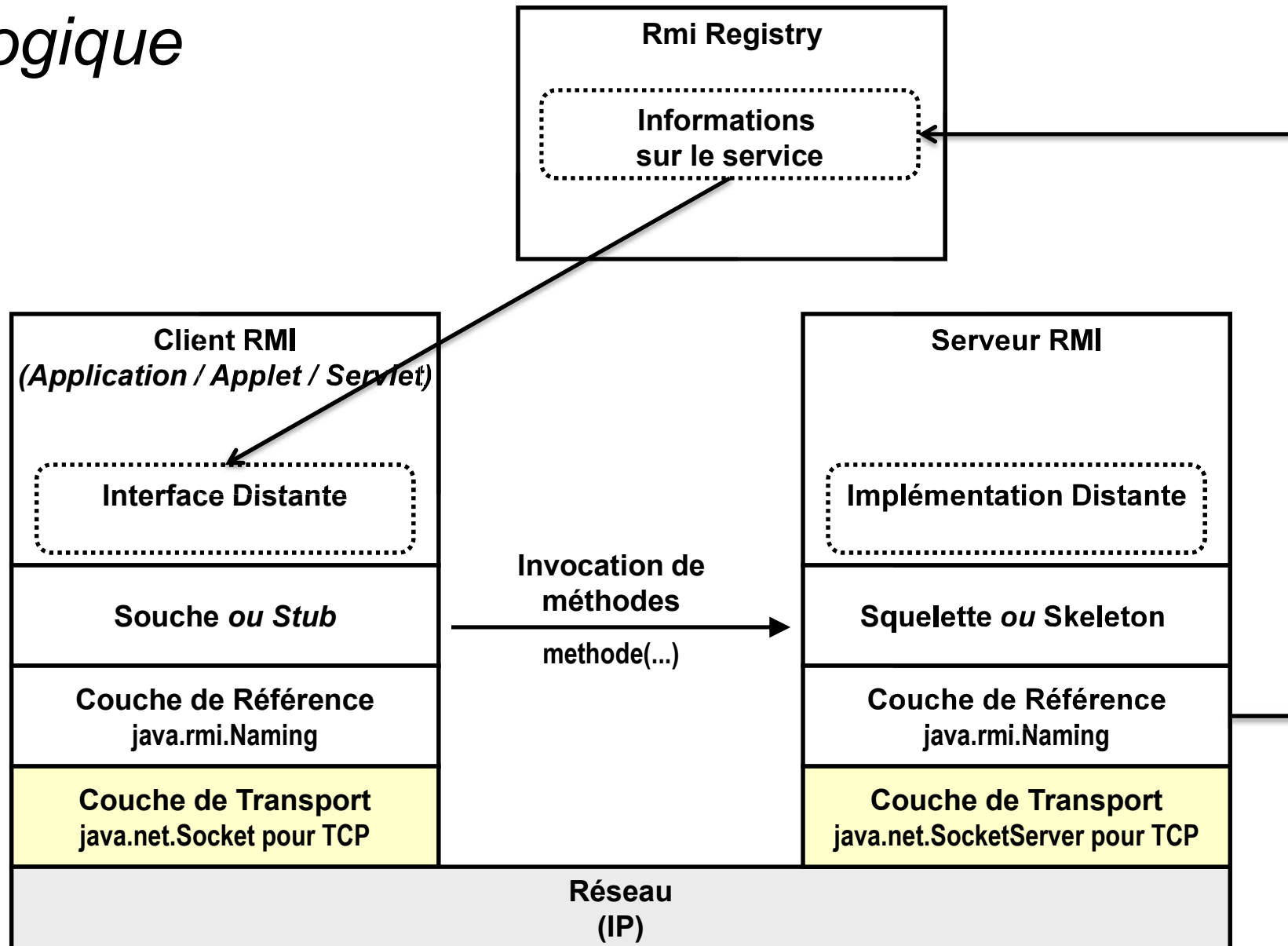
```
Personne Opersonne = new Personne();  
Int qi = Opersonne.calculerQi() ;
```



# RMI : principes

- Outils pour :
  - la génération des stub/skeleton,
  - l'enregistrement par le nom,
  - l'activation
- Mono-langage et Multiplateforme: de JVM à JVM (*les données et objets ont la même représentation qqs la JVM*)
- Orienté Objet : Les RMIs utilisent le mécanisme standard de sérialisation de JAVA pour l'envoi d'objets.
- Dynamique : Les classes des Stubs et des paramètres peuvent être chargées dynamiquement via HTTP (<http://>) ou NFS (<file://>)
- Sécurité : un SecurityManager vérifie si certaines opérations sont autorisés par le serveur

# Structure des couches RMI (i) : l'architecture logique





# Structure des couches RMI (ii)

## ■ Souche *ou Stub* (sur le client)

- représentant local de l'objet distant qui implémente les méthodes "exportées" de l'objet distant
- "marshalise" les arguments de la méthode distante et les envoie en un flot de données au serveur
- "démarshalise" la valeur ou l'objet retournés par la méthode distante
- la classe `xx_Stub` peut être chargée dynamiquement par le client

## ■ Squelette *ou Skeleton* (sur le serveur)

- "démarshalise" les paramètres des méthodes
- fait un appel à la méthode de l'objet local au serveur
- "marshalise" la valeur ou l'objet renvoyé par la méthode



# Structure des couches RMI (ii)

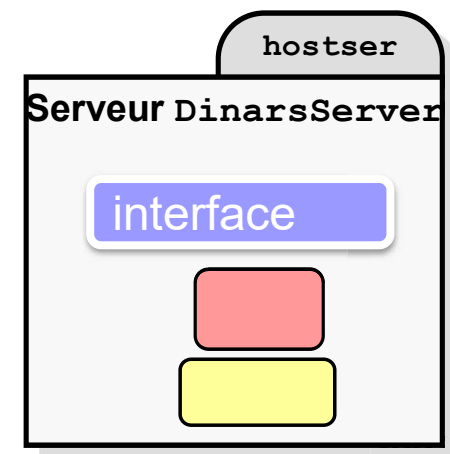
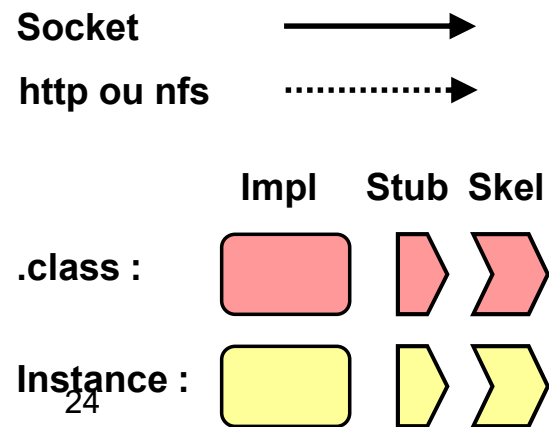
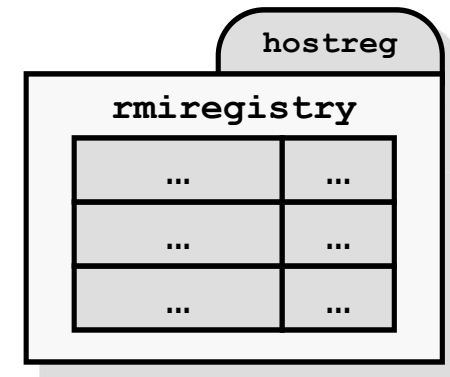
## ■ Couche des références distantes

- traduit la référence locale au stub en une référence à l'objet distant
- elle est servie par un processus tier : rmiregistry

## ■ Couche de transport

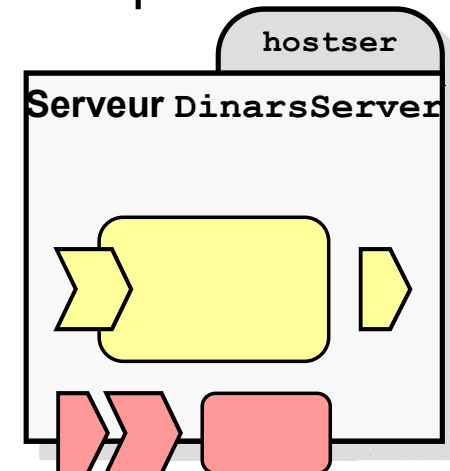
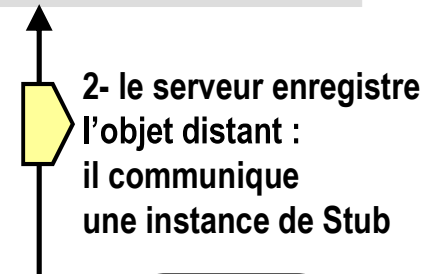
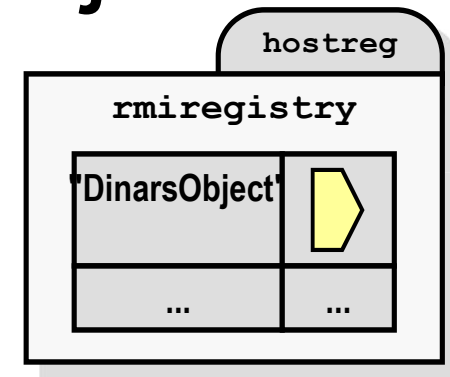
- écoute les appels entrants
- établit et gère les connexions avec les sites distants

# La configuration



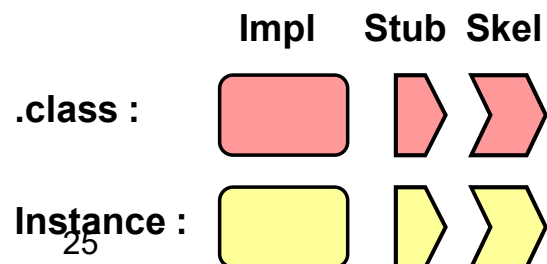


# L'enregistrement de l'objet

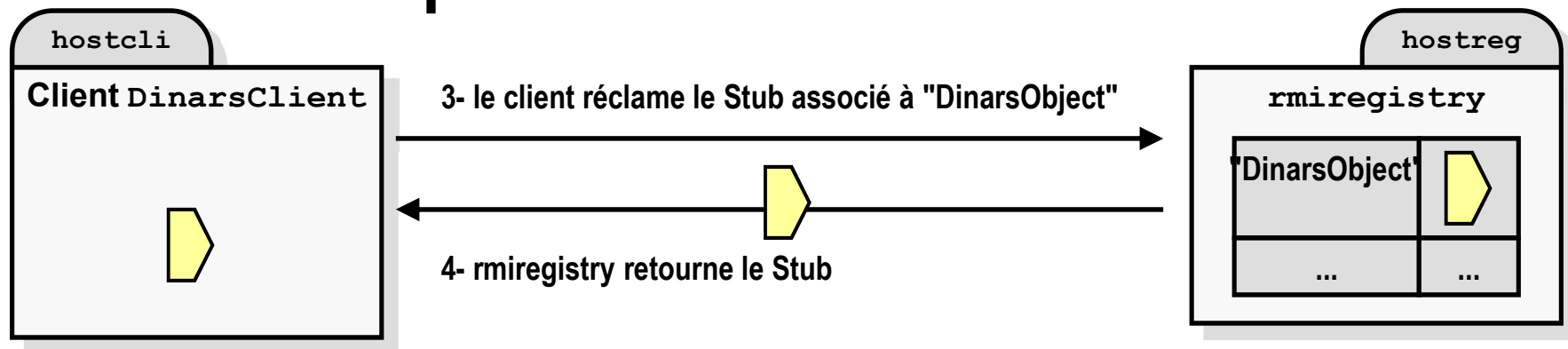


1- le serveur charge les classes  
Stub et Skel

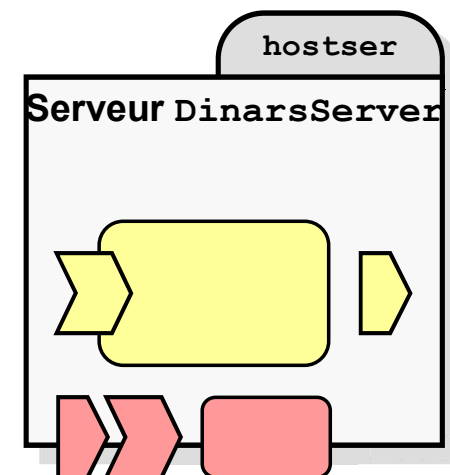
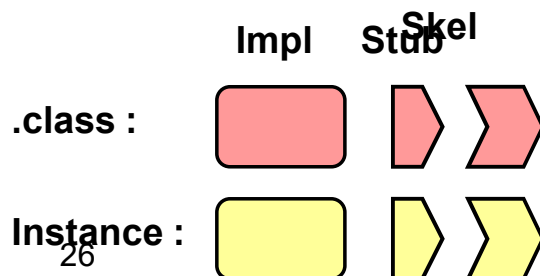
Socket →  
http ou nfs →



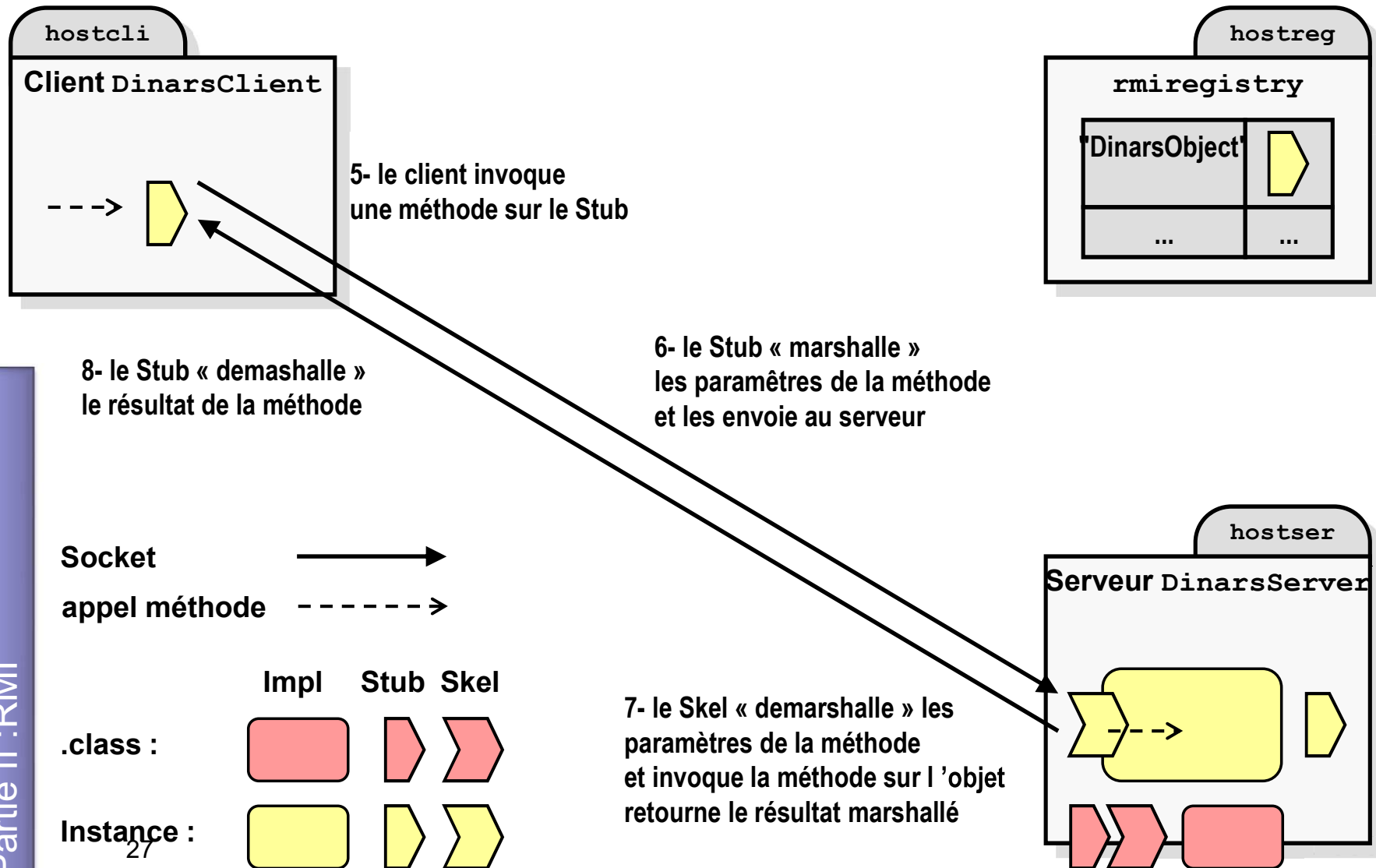
# La récupération du Stub



Socket →  
http ou nfs →



# Invocation d'une méthode





# Etapes de création : Serveur

1. Ecrire l'interface de l'objet distant :
  - ☐ interface : "extends java.rmi.Remote"
  - ☐ méthodes : "throws java.rmi.RemoteException »
  - ☐ paramètres sérialisables : "implements Serializable »
2. Ecrire une implémentation de l'objet serveur
  - ☐ classe : extends java.rmi.server.UnicastRemoteObject
3. Générer les Stub/Skeleton correspondants. (outil rmic)
4. Publier le Stub et attendre l'invocation par les clients

# Etapes de création : serveur

## Interface objet distant

```
1 package testrmi ;
2 import java.rmi.*;
3 public interface CalculInterface extends java.rmi.Remote {
4     public int additionner (int a , int b ) throws RemoteException ;
5     public int soustraire (int a, int b) throws RemoteException;
6 }
```

CalculInterface.java

```

import java.rmi.RemoteException;
import java.rmi.Naming;
public class CalculImpl extends java.rmi.server.UnicastRemoteObject
    implements CalculInterface {
    public CalculImpl () throws RemoteException {
        super();
    }
    public int additionner(int a, int b) throws RemoteException {
        return a + b;
    }
    public int soustraire(int a, int b) throws RemoteException {
        return a - b;
    }
    public static void main (String args[])
    {
        try {
            System.out.println("Création objet distant...\n");
            CalculImpl serveurCalcul = new CalculImpl();
            System.out.println("Creation succes...\n");
            System.out.println("Enregistrement objet distant");
            Naming.rebind("rmi://localhost:1099/Calcul", serveurCalcul);
            System.out.println("Enregistrement réussi");
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}

```

CalculImpl.java

# Explications

## ■ Le service de nommage : `Java.rmi.Naming`

Fonction	Rôle
<code>bind (name, obj)</code>	Lie l'objet distant (remote object) à un nom spécifique
<code>rebind (name, obj)</code>	Lie l'objet distant même s'il existe déjà
<code>unbind (name)</code>	Retire l'association entre un nom et un objet distant
<code>Obj lookup (url)</code>	Renvoie l'objet distant associé à une URL
<code>String [] list(url)</code>	Renvoie la liste des associations sur la registry spécifiée dans l'URL

# Etapes de création : serveur

Génération des stub et skeleton

- Compilation des fichiers : **Javac**
  - Génération des stub et skeleton
- > **Rmic testrmi.CalculImpl**

```
C:\>"c:\Program Files\java\jdk1.6.0_03\bin\rmic" testrmi.CalculImpl
C:\>cd testrmi
C:\testrmi>dir
Le volume dans le lecteur C n'a pas de nom.
Le numéro de série du volume est 66E9-BDD1

Répertoire de C:\testrmi

24/09/2008  10:41    <REP>
24/09/2008  10:41    <REP>
24/09/2008  10:40             424 CalculImpl.class
24/09/2008  10:28             379 CalculImpl.java
24/09/2008  10:41             2 055 CalculImpl_Stub.class
24/09/2008  10:40             255 CalculInterface.class
24/09/2008  10:25             233 CalculInterface.java
```





# Lancement du serveur

- Lancement du service de nommage :

```
> rmiregistry <port>
```

- Lancement du serveur

```
java -Djava.rmi.server.codebase=<URL>  
-Djava.rmi.server.hostname=<hôte>  
-Djava.security.policy=java.policy <serveur>.
```

# Ecriture de la classe client

```
public class ClientCalcul{
    public static void main(String args[])
    {
        try {
            CalculInterface objc = (CalculInterface)java.rmi.Naming.lookup
            ("rmi://localhost:1099/Calcul");
            int i = objc.soustraire(5,3);
            int j = objc.additioner(5,5) ;
            System.out.println( i ); System.out.println( j );
        }catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```



# Lancement du client

```
>java -Djava.rmi.server.codebase=<URL>  
-Djava.security.policy=java.policy  
<client>  
>2  
>10
```



# Passage d'arguments

- Il y a 2 possibilités de passage d'arguments lors de l'invocation d'une méthode distante :
  1. Le paramètre est d'un type primitif : passage par valeur
  2. Le paramètre est un objet :
    1. Il est sérialisé et envoyé au serveur
    2. Une référence distante est envoyée
- Idem pour la valeur de retour d'une méthode distante



# Sérialisation d'objets (1)

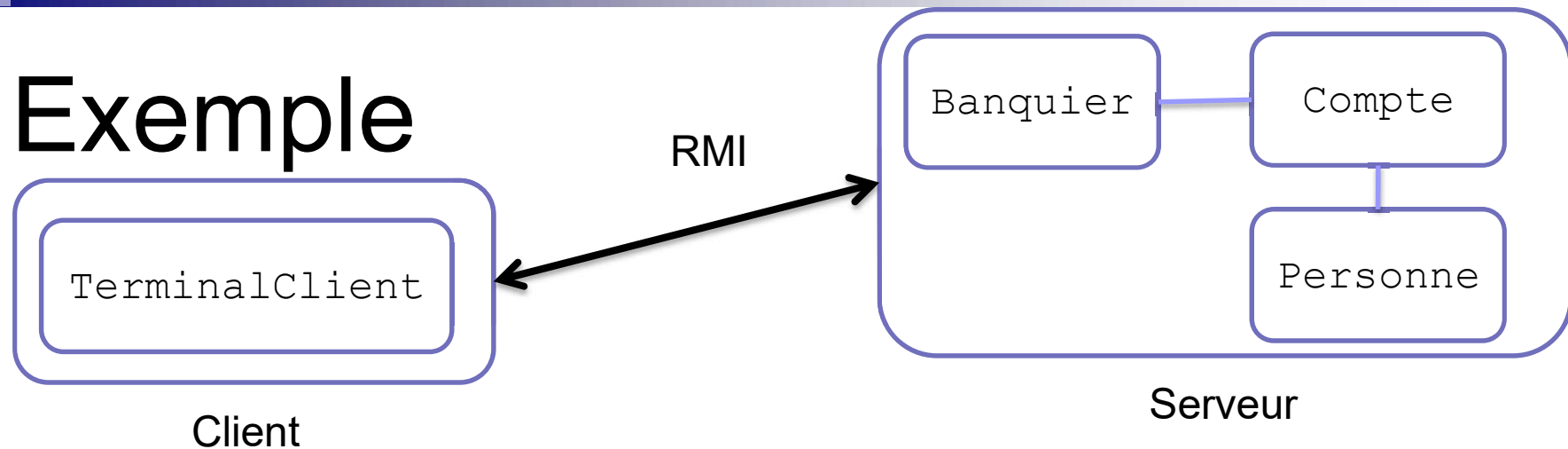
- La sérialisation est une opération (on parle aussi de *marshalling* et *d'unmarshalling*) qui consiste à transformer un objet dans un format transférable par un flux de données (cf. les classes *ObjectInputStream* et *ObjectOutputStream* du package *java.io*). On s'en sert principalement dans 2 cas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



# 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.

# Exemple



```
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;
        }
    }
}
```

# Exemple

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

Est-ce que la classe compte doit elle aussi être serialisable ??



# 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* :

```
public static void main (String args[])
{
    try {
        System.setSecurityManager(new RMISecurityManager());
        ...
    }
}
```

# 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]  
  ...  
};
```



# Sécurité et RMI (3)

```
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";  
};
```