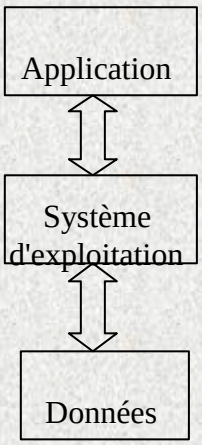
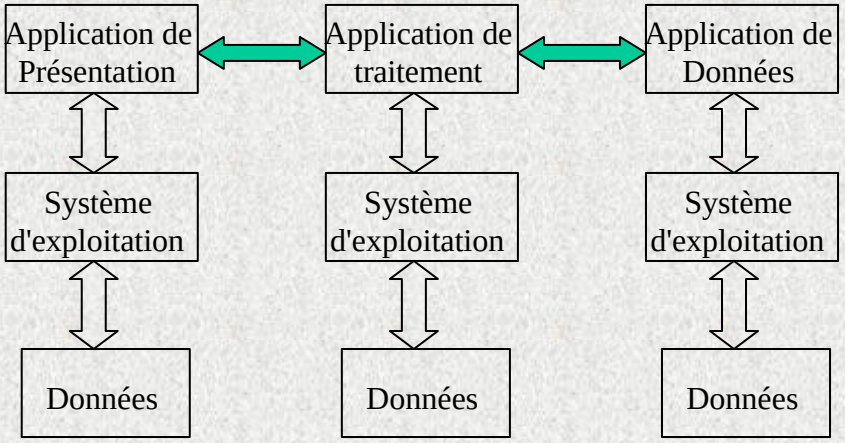
Conception de serveurs d'applications ouverts

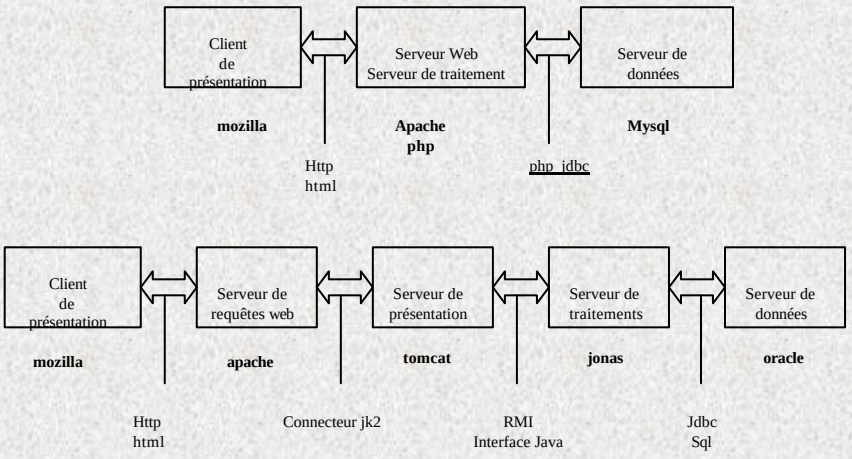
Un modèle d'exécution standard



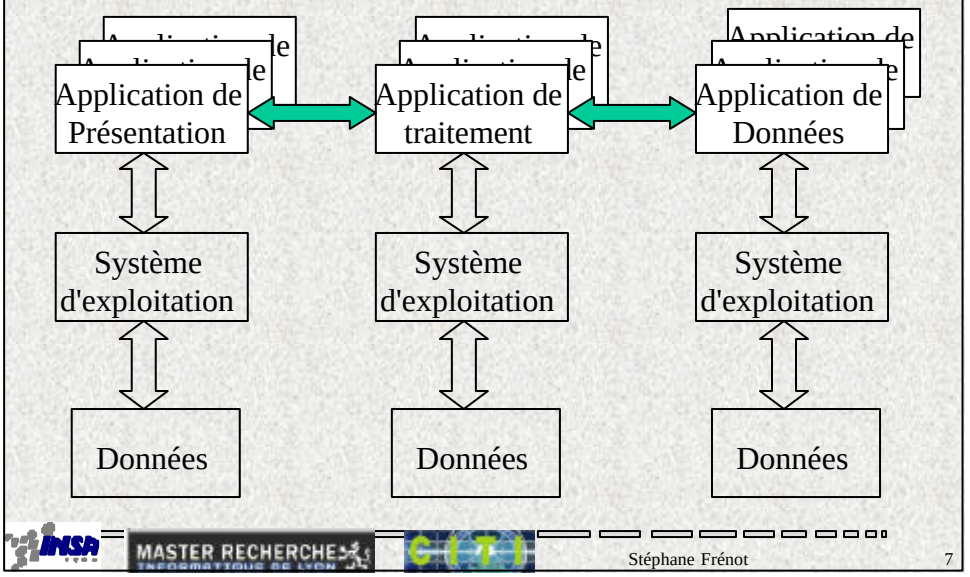
Répartition "horizontale" d'une application

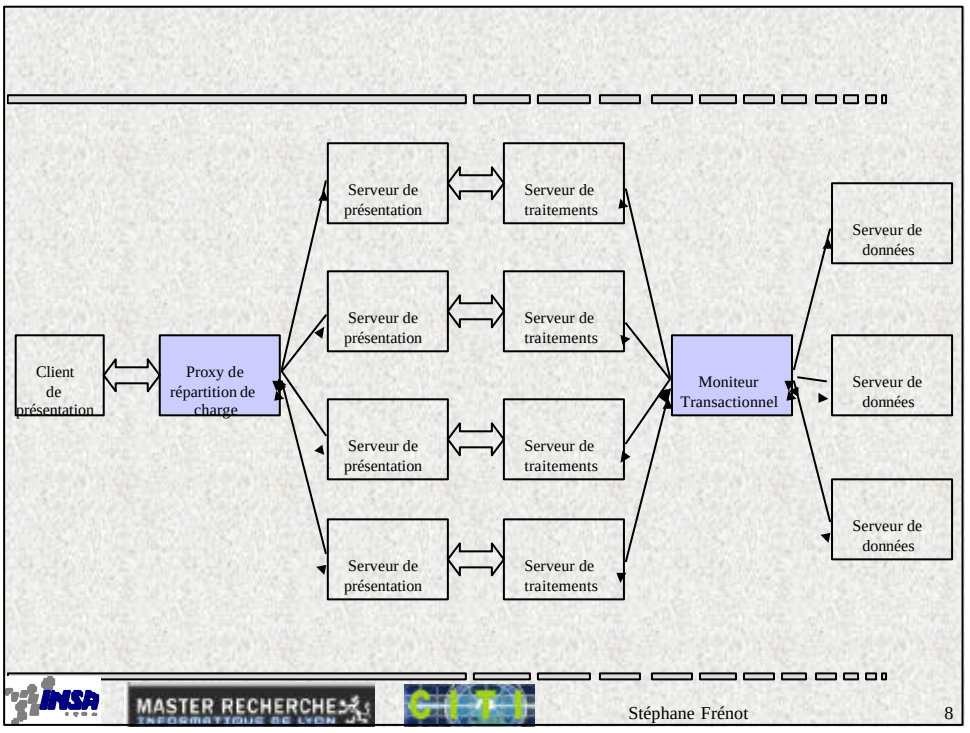


Répartition d'une application : **e-**

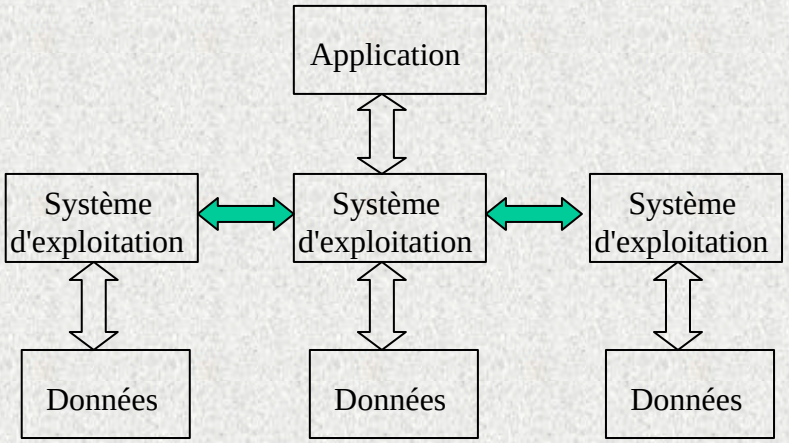


Répartition "verticale" d'une application



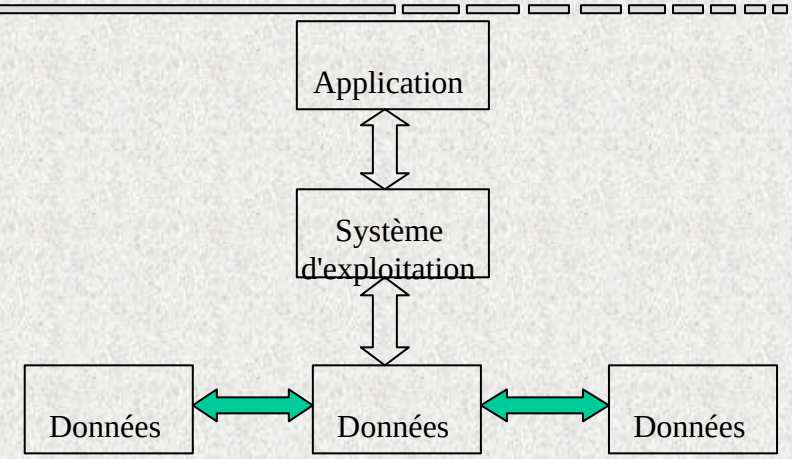


La distribution du SE (Processus)



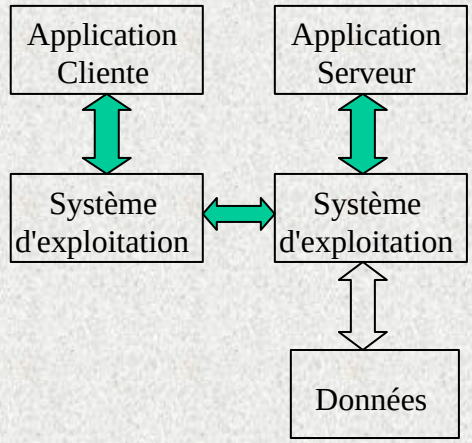
Clusters : Beowulf,
SE distribués : Mach, Mosix...
Grid Computing : http://www.gridforum.org

La distribution du SE (les données)



Disques RAID
Mémoire partagée

La distribution du SE (l'affichage)



Telnet
X11
Ica/Vnc

La distribution indépendante du SE

- Base de données :
  - Répliquées / Réparties
- Moniteurs transactionnels
- Web
- Serveurs d'applications

Défis pour les systèmes distribués d'une manière générale

- Modèles
- Gestion de l'hétérogénéité
- Ouverture
- Sécurité
- Capacité de croissance
- Gestion des pannes
- Gestion de la concurrence
- Gestion de la transparence

Modèles de communication

- C/S : Le client serveur, le modèle de base
- Evénements : Principes de canaux de communication
- P2P : les entités d'exécution sont identiques
- Orienté message : Synchrone/Asynchrone
- Orienté flux
- Communication de groupes
- Espace partagés : tuple space
- Base de données : jdbc / SQL
- Liaison ftp
- Tube unix : IPC
- Protocoles de vote, 2phase commit
- Communication de groupe
- copier/coller des ihm
- Modèle MVC
- Jeux on-line multijoueurs

Modèle de programmation

- Réseau,
- Matériel,
- Système d'exploitation

==> Langages de programmation (objet)

==> Génie logiciel (composants)

==> Le Middleware

Machines virtuelles

Plates-formes à composants

Interpréteurs

Le Client / Serveur

la brique de base

Caractéristiques de la communication InterProcessus

- Primitives : send/receive
- Synchrone / Asynchrone
- Destinataire des messages
- Fiabilité
  - Validité : même si perte de messages
  - Intégrité : Non corrompu, Sans duplication
- Ordre : Ordre de l'émetteur
- TCP / UDP

Caractéristiques du modèle client-serveur

- Notion de service
  - réalisé par un serveur
  - demandé par un client
  - définie par une interface (API) entre client et serveur
- Communication par messages
  - Requête : paramètre d'appel, spécification du service requis
  - Réponse : résultat, indicateur d'exécution/d'erreur
  - Synchrone : Client et serveurs sont bloqués
- Avantages
  - Structuré,
  - Protégé (espaces d'exécution distincts),
  - Partagé

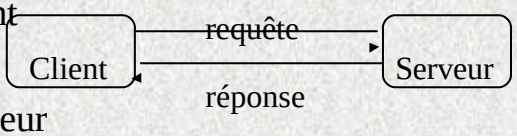
Client / serveur "echo" en java

- L'appel client :
  - java Client test
- La chaîne "test" est récupérée par le client

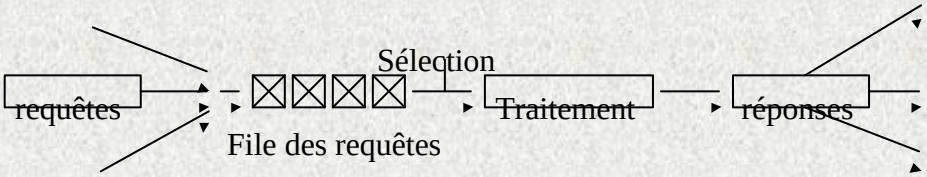
- ==> Commentaires !
- ==> Améliorations
- ==> Trois codes possibles

Modèle client/serveur : partage

- Vu du client



- Vu du serveur
  - Gestion des requêtes (priorité)
  - Exécution du service (séquentielle, concurrent)
  - Mémorisation ou non l'état du client



Modèle Client-Serveur
Gestion des processus

- Client et serveur exécutent des processus distincts
  - Mise en œuvre par les primitives TCP/IP
  - le client est suspendu (appel synchrone)
  - éventuellement plusieurs requêtes peuvent être traitées concurremment par le serveur
    - parallélisme réel (multiprocesseur)
    - pseudo-parallélisme
  - La concurrence peut prendre plusieurs formes
    - plusieurs processus
    - plusieurs processus légers dans le même espace virtuel

Modèle Client-Serveur

- Protocoles sans état :
  - Chaque requête est indépendante de la suivante
    - http, echo, X11 (à détailler), jdbc/sqlNet, …
  - Pas de causalité entre les requêtes
  - Pas d'ordre d'arrivé
  - Le modèle est résistant aux pannes du client et du serveur
    - ==> Il suffit de ...

Serveur / Protocole à état

- Certaines requêtes dépendent des autres pour un même client
  - Identification, Echange de clés de cryptage, Sélection de Menus…
- Le serveur doit mémoriser la connexion du client
  - Mode connecté : la connexion est identifiée par la socket
  - Mode déconnecté : le protocole doit véhiculer l'information d'identification du client
- Problèmes :
  - Panne
  - Sécurité

Serveur à état : panne

- Panne du client :
  - Le serveur peut repérer qu'il s'agit du même client
    - Comment ?
  - Le client peut "rejouer" toute la session.
- Panne du serveur
  - Le serveur doit enregistrer les états du clients
    - Notion de journalisation ou de persistance
    - Mais il doit pouvoir ré-identifier le client
  - Le client peut "rejouer" toute la session.
    - Ex : sncf

Serveur à données rémanentes

- Les requêtes manipulent des données persistantes accessibles par plusieurs clients
  - Exemple base de données
    - Banque, Bourse
  - L'ordre des requêtes est important :
    - Délais d'internet,
    - Inconsistance des données
      - Exemple :
        - » Crédit --> Débit --> ok
        - » Débit --> Agios --> Crédit --> ok - agios

Serveurs inter-dépendant

- Pour fournir une réponse un serveur dépend d'autre serveurs.
  - Nécessité de synchronisation
  - Panne des serveurs esclaves
  - Transactions, 2PC

Du client/serveur à la suite...

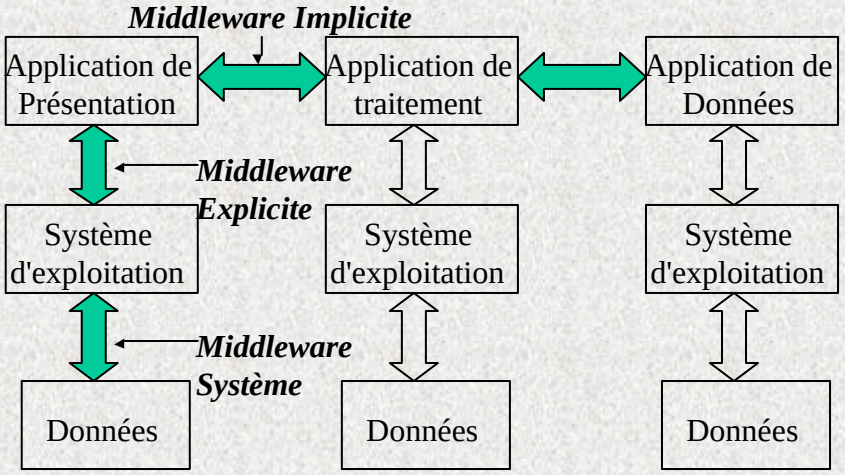
- Trois questions à résoudre
  - Comment casser la relation forte entre client et serveur, comment rendre le client indépendant du serveur pour l'appel
    - ==> ?
  - Comment éviter au client d'avoir du code non réseau dans son appel
    - ==> ?
  - Comment augmenter la performance sur plusieurs machines en parallèles
    - ==> ?

  ==> La solution miracle **le middleware**

Middleware

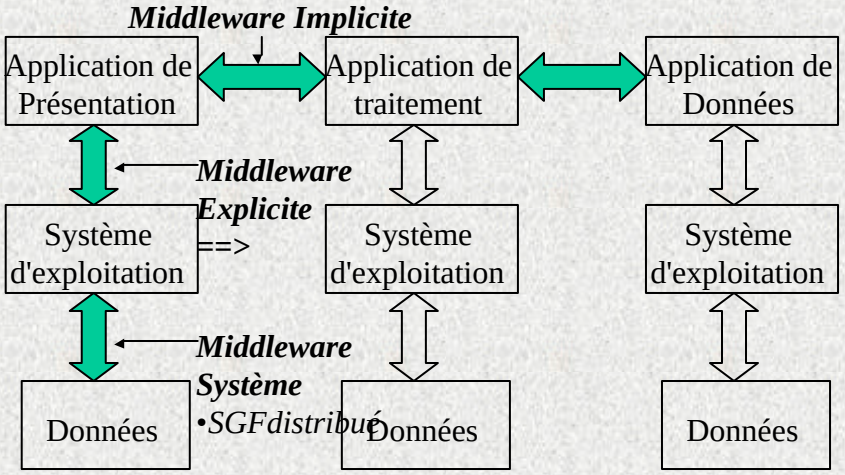
- Le middleware est une couche d'interposition entre un client et un service, qui réalise certaines opérations au compte du client ou du serveur

Répartition d'une application



Java - *RMI*
Remote Method Invocation

Répartition d'une application



Objectif

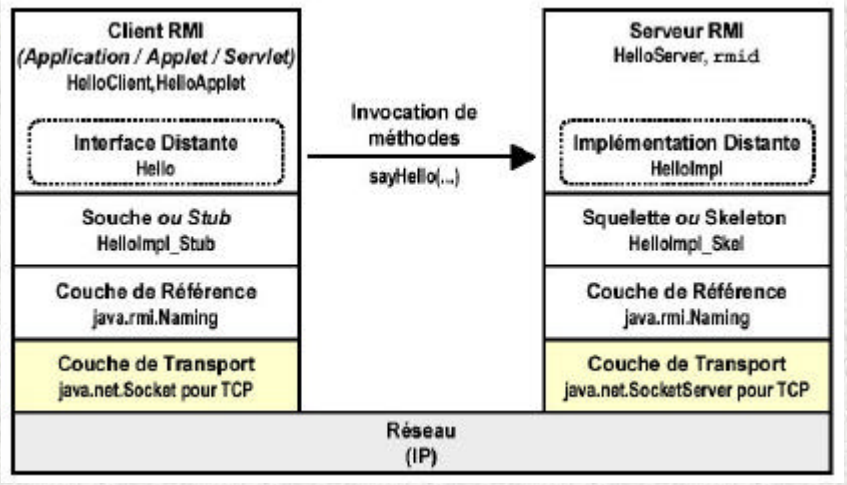
- Rendre transparent la manipulation d'objets situés dans un autre espace d'adressage
- Les appels suivants doivent être transparents, que l'objet soit local ou distant

```
Toto toto=new Toto();
String x=z.calcul(toto);
y=z.verifie(x, 30);
```

Objectifs

- RMI est une *core* API
- RMI permet de faire interagir des objets situés dans des espaces d'adressage distincts situés sur des machines distinctes
- RMI repose sur les classes de sérialisation
- Simple à mettre en oeuvre :
  - Définir l'interface de l'objet distribué (OD)
  - Côté serveur : implémenter l'OD et l'attacher à une URL
  - Côté serveur : générer les *stubs* et *skeletons* (rmic) et diffuser l'OD
  - Côté client : s'attacher à l'OD (URL) et l'invoquer
- Un OD se manipule comme tout autre objet Java

Architecture de l'invocation de méthodes



Points clé

- Java distingue deux familles d'objets
  - Les objets locaux
  - Les objets distants :
    - ils implantent une interface
    - l'interface étends l'interface marker java.rmi.Remote
- Les passages par référence fonctionnent de la même manière que dans le cas classique
  - Si l'objet passé en référence n'est pas remote, il est sérialisé, sinon le client obtient une référence distante désignée par une interface
- ==> ? Comment un client obtient t'il l'accès au service ?

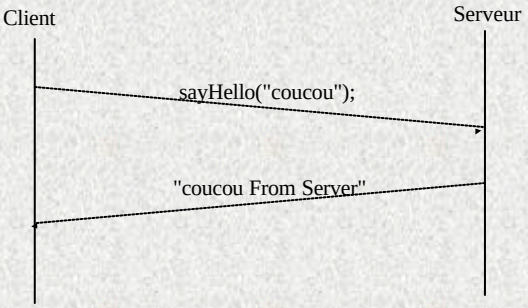
Le service de nommage

- RMI registry : annuaire des services qui tournent sur le serveur distant
- rmi://host:port/name

`bind (name, obj)`	Lie l'objet distant (remote object) à un nom spécifique
`rebind (name, obj)`	Lie l'objet distant même s'il existe déjà l'ancien est supprimé
`unbind (name)`	Retire l'association entre un nom et un objet distant
`lookup(url)`	Renvoie l'objet distant associé à une URL
`list(url)`	Renvoie la liste des associations sur la registry spécifiée dans l'URL

Un service exemple

- Le service Hello :
  - Le serveur reçoit une requête Hello du client
  - Il répond par une chaîne augmentée d'une phrase



Le client

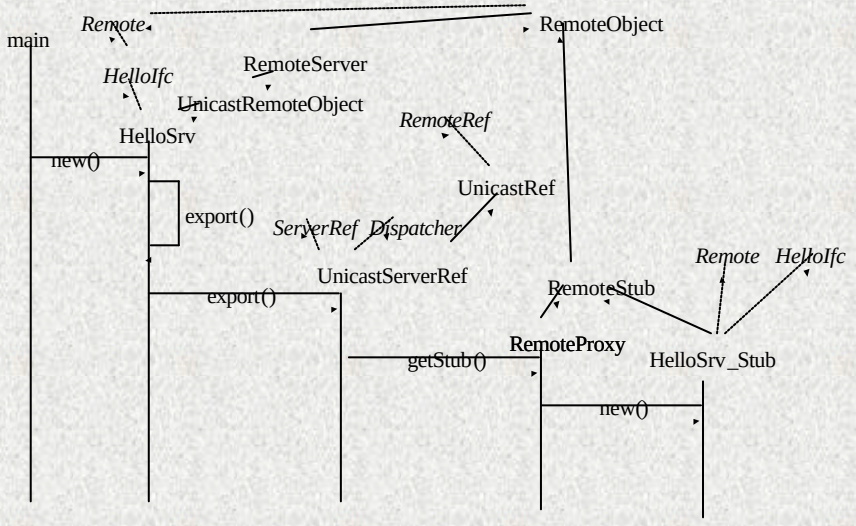
- Le client doit rester "simple"
  - ==> Pas de code non-fonctionnel

```
package rmi;

import java.rmi.Naming;

public class HelloClient  {
  public static void main(String [] arg) throws Exception {
    HelloIfc hello=(HelloIfc)Naming.lookup("rmi://ares-frenot-
3/"+HelloIfc.class.getName());
    String s1="coucou";
    String s2=hello.sayHello("coucou");
    System.out.println("Envoyé : "+s1);
    System.out.println("Reçu : "+s2);
  }
}
```

Le serveur reste simple aussi !

```
package rmi;

import java.rmi.server.UnicastRemoteObject;
import java.rmi.Naming;
import java.rmi.RemoteException;

public class HelloSrv extends UnicastRemoteObject implements HelloIfc {
  private static String FROMSERVER;

  public HelloSrv(String serverPart) throws RemoteException {
    this.FROMSERVER=serverPart;
  }

  public String sayHello(String hello){
    return hello+this.FROMSERVER;
  }

  public static void main(String [] arg){
    try{
      HelloSrv srv=new HelloSrv(" From Server");
      Naming.bind(HelloIfc.class.getName(), srv);
    }catch(Exception e){
      e.printStackTrace();
    }
  }
}
```


La création du serveur



Exemple 2: Le gestionnaire de comptes

- Serveur de comptes
  - Hashtable
  - Position d'un compte
  - Méthode ajout/retrait/position

Exemple (1)

- Définir l'interface de l'OD (Contrat entre le client et le serveur)

```
package banque;

public interface Banque extends java.rmi.Remote {

  public void ajouter(String id, double somme)throws
   java.rmi.RemoteException;

  public void retirer(String id, double somme)throws
   java.rmi.RemoteException;

  public Position position(String id) throws java.rmi.RemoteException;

}
```

- Implante Remote,
- Lève des RemoteExceptions,
- Les arguments sont sérialisables

Exemple (2)

- Ecrire une implémentation de l'OD (1)

```
public class BanqueImpl extends java.rmi.server.UnicastRemoteObject
   implements Banque {
  Hashtable clients;
  public BanqueImpl(Hashtable clients) throws java.rmi.RemoteException {
    super();
    this.clients = clients;}

  public void ajouter(String id, double somme) throws
   java.rmi.RemoteException {((Compte)clients.get(id)).ajouter(somme); }

  public void retirer(String id, double somme) throws
   java.rmi.RemoteException {((Compte)clients.get(id)).retirer(somme); }

  public Position position(String id) throws java.rmi.RemoteException {
    return ((Compte)clients.get(id)).position(); }
}
```


Exemple (3)

- Ecrire une implementation de l'OD (2)

```
public class Position implements java.io.Serializable {
  public double solde;
  public Date dateDerniereOperation;

  public Position(double solde, Date dateDerniereOperation) {
    this.solde = solde;
    this.dateDerniereOperation = dateDerniereOperation;
  }
}
```

Exemple (4)

- Ecrire une implementation de l'OD (3)

```
public class Compte {
  private Position position;

  Compte(double solde, Date date) {position = new Position(solde,
   date);  }

  void ajouter(double somme) {
    position.solde += somme;
    position.derniereOperation = new Date(); }

  void retirer(double somme) {
    position.solde -= somme;
    position.derniereOperation = new Date(); }

  Position position() { return position; }
}
```


Exemple (5)

- Ecrire un programme enregistrant une instance de l'OD

```
public class BanqueServeur {
  public static void main(String[] args) {

    Hashtable clients = initComptesClients();

    try {
      BanqueImpl obj = new BanqueImpl(clients);
      java.rmi.Naming.rebind("//falconet.inria.fr/MaBanque", obj);
      System.out.println("Objet distribue 'Banque' est enregistre");
    }catch(Exception e) {
      e.printStackTrace();
    }
  }
}
```

Exemple (6)

1) Compiler les sources
javac Position.java Compte.java Banque.java BanqueImpl.java BanqueServeur.java

2) Générer les *stub* et les *skeletons*
rmic banque.BanqueImpl
==> BanqueImpl_Stub.class et BanqueImpl_Skel.class

3) Lancer le RMI registry
rmiregistry &

4) Lancer l'OD
java banque.BanqueServeur
(ou java -**Djava.rmi.server.codebase=http://falconet.inria/codebase** banque.BanqueServeur)

Exemple (7)

- Ecrire un programme client

```
public class Client {
  public static void main(String[] args) {
    try {
      Banque b =
   (Banque)java.rmi.Naming.lookup("//falconet.inria.fr/MaBanque");
       b.ajouter(args[0], 100.00);
       b.retirer(args[0], 20.00);
       Position p = b.position(args[0]);
       System.out.println("Position au "+p.dateDerniereOperation+":"+
    p.solde);
     }catch(Exception e) {e.printStackTrace();}
  }
}
```

Distributed Garbage Collector (DGC)

- Le DGC interagit avec les GC locaux et utilise un mécanisme de *reference-counting*
- Lorsqu'un OD est passé en paramètre à un autre OD → `ref_count++`
- Lorsqu'un *stub* n'est plus référencé → *weak reference*
- Lorsque le GC du client libère le *stub*, sa méthode `finalize` est appelée et informe le DGC → `ref_count--`
- Lorsque le nombre de références d'un OD = 0 → *weak reference*
- Le GC du serveur peut alors libérer l'OD
- Le client doit régulierement renouveler son bail auprès du DGC.
- Si référence à un OD libérée → `RemoteException`

Comparaison objet local / objet distant

- Définition de l'objet
  - Définit par sa classe de description
  - Définit par une interface de description qui étend Remote
- Implantation
  - Implanté par la classe
  - Le comportement est implanté par une classe qui implante l'interface « remote »
- Création
  - Opérateur new
  - Une instance distante est fabriquée avec l'opérateur new. Un objet ne peut pas créer à distance un objet.
- Accès
  - Un objet est accédé directement par une variable qui le référence
  - Un objet distant est accédé par une variable qui référence un talon d'implantation de l'interface distante

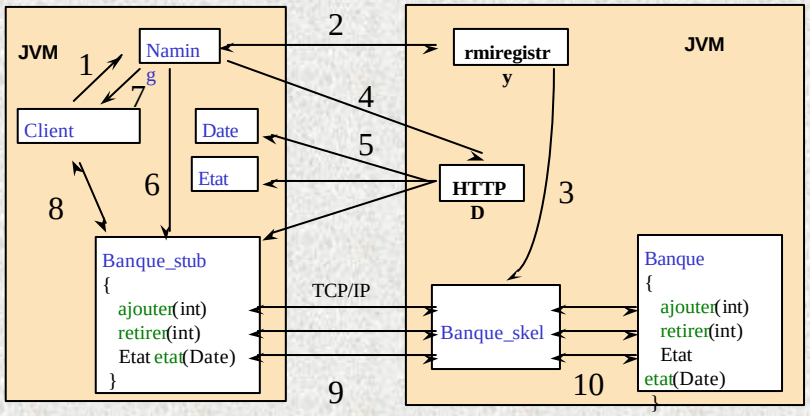
Comparaison objet local / objet distant

- Référence
  - Une référence sur un objet pointe directement dans la pile
  - Une « remote référence » est un pointer sur un mandataire (proxy, talon) dans la pile. Ce stub contient les informations qui lui permettent de se connecter à l'objet distant qui contient les implantations des méthodes
- Référence actives
  - Un objet est considéré vivant si au moins une variable le référence
  - Dans un environnement distribué les MV peuvent « crasher ». Un objet distant est actif s'il a été accédé avant une certaines durée de vie (lease period). Si toutes les références distantes ont été explicitement relachées ou si toutes les références distantes ont dépassées leur période de leasing l'objet est disponible pour le GC
- Finalisation
  - Si un objet implante la méthode finalize(), elle est appelée avant que l'objet soit GC
  - Si un objet distant implante l'interface Unreferenced, la méthode unreferenced

Comparaison objet local / objet distant

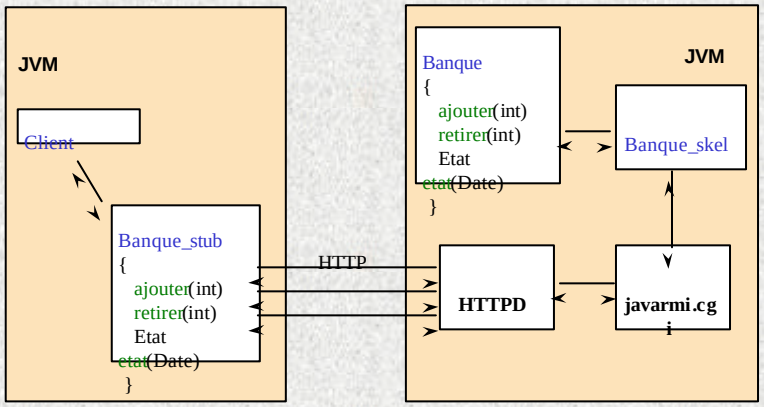
- Garbage collection
  - Quand toutes les références locales sont perdues l'objet est candidat au ramassage
  - Le GC distribué fonctionne avec les GC locaux
- Exceptions
  - Les exceptions sont soit de type Runtime soit Exception. Le compilateur force le développeur à traiter toutes les exceptions
  - RMI force les programmes à traiter toutes les RemoteExceptions qui peuvent être levées.

Récupération dynamique de classes



Encapsulation http (post)

***Tunneling* HTTP**



Conclusion sur Java RMI

- Extension du RPC aux objets
  - Permet l'accès homogène à des objets distants
  - Permet d'étendre l'environnement local par chargement dynamique de code
  - Pas de langage séparé de description des composants
- Limitations
  - Environnement restreint à un langage unique (Java)
  - Pas de services additionnels
    - Duplication d'objets
    - Transactions...

Autres Systèmes à objets répartis

- RPC : Remote Procedure Call
  - Interface d'appel c
  - Générateur de stub et skeleton (genrpc)
  - Sun rpc
  - Base pour nfs, nis, rwall, rsh...
- CORBA 2: La référence
  - Indépendant du langage de programmation

	RPC	RMI	CORBA	.NET Remoting	SOAP
Qui	SUN/OSF	SUN	OMG	MicroSoft/ECMA	W3C
Plate-formes	Multi	Multi	Multi	Win32, FreeBSD, Linux	Multi
Langages de Programmation	C, C++, ...	Java	Multi	C#, VB, J#, ...	Multi
Langages de Définition de Service	RPCGEN	Java	IDL	CLR	XML
Réseau	TCP, UDP	TCP, HTTP, IIOP customisable	GIOP, IIOP, Pluggable Transport Layer	TCP,HTTP, *IIOP*	RPC,HTTP
Marshalling		Sérialisation Java	Représentation IIOP	Formatteurs Binaire, SOAP	SOAP
Nommage	IP+Port	RMI, JNDI,JINI	CosNaming	IP+Nom	IP+Port, URL
Intercepteur	Non	depuis 1.4	Oui	Oui CallContext	Extension applicative dans le header
Extra		Chargement dynamique des classes	Services Communs Services Sectoriels	Pas de Chargement dynamique des classes	

Activation d'objets distants

- Rappel (JDK1.1)
  - l 'objet distant est actif au démarrage du serveur RMI
- Motivation
  - principe du démon inetd d'Unix
  - Le démon rmid démarre une JVM qui sert l'objet distant seulement au moment de l'invocation d'une méthode (à la demande) ou au reboot.
- JDK1.2 introduit
  - un nouveau package java.rmi.activation
  - un objet activable doit dériver de la classe Activatable
  - un démon RMI rmid qui active les objets à la demande ou au reboot de la machine
- Info : jdk1.2.2\docs\guide\rmi\activation.html
- Remarque : l'activation est très utilisée par JINI !

Créer une implémentation activable

- La classe doit étendre **`java.rmi.activation.Activable`** et implémenter l'interface distante
- La classe doit déclarer un constructeur avec 2 arguments

  java.rmi.activation.ActivationID, java.rmi.MarshalledObject

```
public class HelloActivatableImpl
      extends java.rmi.activation.Activatable implements Hello {
  private int defaultlang;
  public ActivatableImplementation(ActivationID id, MarshalledObject data)
    throws java.rmi.RemoteException {
              super(id, 0);
              this.defaultlang=((Integer)data.get()).intValue();
  }
   // implémentation des méthodes
   public String sayHello() throws java.rmi.RemoteException { ... } ...
}
```


Créer le programme d'enregistrement

- Rôle
  - passer l'information nécessaire à l'activation de l'objet activable au démon rmid puis enregistrer l'objet auprès de rmiregistry
- Descripteur
  - ActivationDesc: description des informations nécessaires à rmid
- Groupes d'objets activables
  - rmid active une JVM par groupe
  - Les objets du même groupe partagent la même JVM
    - ActivationGroup: représente un groupe d'objets activables
    - ActivationGroupDesc: description d'un groupe
    - ActivationGroupID: identifiant d'un groupe

Code d'enregistrement

```
import java.rmi.*; import java.rmi.activation.*; import java.util.Properties;
public class SetupActivHello {
   public static void main(String[] args) throws Exception {
    System.setSecurityManager(new RMISecurityManager());

// création d'un groupe d'objets activables
    Properties props = new Properties(); props.put("java.security.policy", "./helloactiv.policy");
     ActivationGroupDesc.CommandEnvironment ace = null;

// descripteur du groupe
    ActivationGroupDesc agroupdesc = new ActivationGroupDesc(props, ace);
//
    ActivationSystem asystem = ActivationGroup.getSystem();
    ActivationGroupID agi = asystem.registerGroup(agroupdesc);
// enregistrement du groupe
    ActivationGroup.createGroup(agi, agroupdesc , 0);
```


Code d'enregistrement 2

```
// le descripteur doit contenir le codebase pour chercher les classes
    String classeslocation = "http://hostwww/hello/myclasses/";
// le descripteur peut contenir un objet sérialisable pour l'initialisation de l'objet ; data peut
   être null

    MarshalledObject data = new MarshalledObject (new Integer(Home.ES));
// création d'un descripteur pour l'objet activable
    ActivationDesc adesc = new ActivationDesc
       (agi, "examples.hello.HelloActivatableImpl", classeslocation, data );

// enregistrement de l'objet auprès du démon rmid : récupération d'un stub
    Hello obj = (Hello)Activatable.register(adesc);
// enregistrement du stub auprès du rmiregistry
    Naming.rebind("//hostreg/HelloActiv", obj);
    System.exit(0);
   }
}
```

Répartition d'une application