Bases de données et environnements distribués
Chapitre II : les objets distribués avec Java

Éric Leclercq

Département IEM / Laboratoire LE2i

Septembre 2015

émail : Eric.Leclercq@u-bourgogne.fr
http://ludique.u-bourgogne.fr/~leclercq
http://ufrsciencestech.u-bourgogne.fr

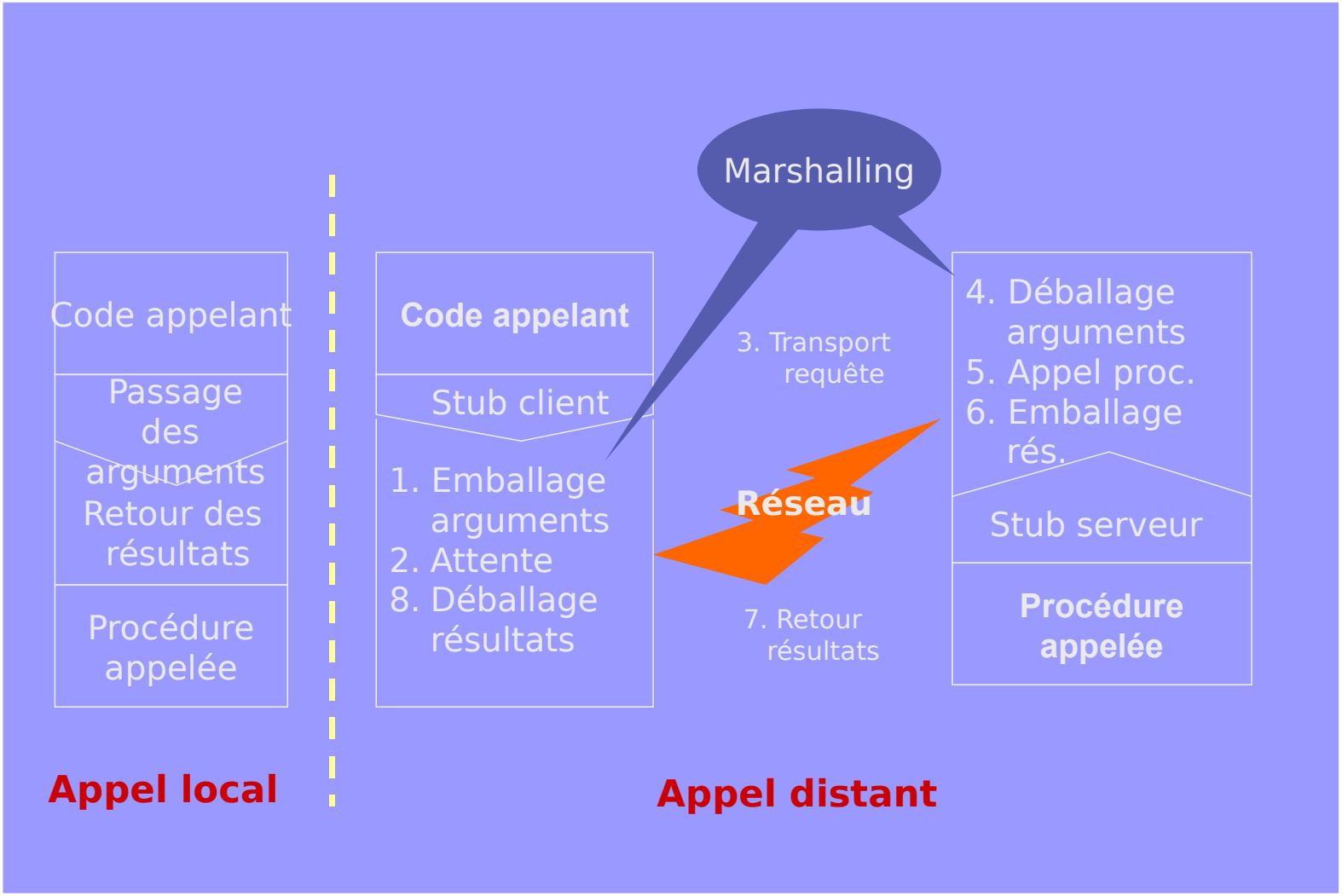
Plan du chapitre

- Du RPC à l'appel de méthode à distance
- Principe et motivations des OD
- Mise en œuvre
- Un exemple utilisant Java RMI
- Fonctionnement interne de Java RMI
- Objets mobiles
- Problématique de sécurité
- Le pattern fabrique d'objets (*Object Factory*)
- Le passage d'objets en paramètre
- Notion d'objet activable
- Appel en retour (*callback*)

Attention

Les exemples sont développés avec J2SDK/JDK 1.4 sauf spécification contraire.

Les principes du RPC



Les principes du RPC

Aucun intérêt si les stubs doivent être écrits (à la main) par le programmeur :

- la signature de la procédure est décrite à l'aide d'un langage de description d'interface (IDL)
- la description est traitée par un précompilateur qui génère le code des stubs client et serveur
- les stubs sont générés automatiquement

Traiter l'hétérogénité des architectures machine :

- principe : résoudre les problèmes d'hétérogénéité dans implantation automatique des stubs
- les données sont converties vers un format unique comme XDR (eXternal Data Representation)
- pour détails sur XDR - RFC 1832

Du RPC aux Intergiciels

Avantages du mécanisme RPC :

- Abstraction : les détails de la communication sont cachés
- Intégration dans un langage : portabilité et développement facilité
- Outils de génération : facilitent la mise en œuvre

Limitations :

- La structure de l'application est statique
- Pas de passage des paramètres par valeur
- La communication est synchrone

Conclusion

Technologie fortement intrusive dans le code de l'application

Du RPC aux Intergiciels

L'intergiciel implante l'abstraction de la communication :

- sous divers matériels et systèmes d'exploitation
- indépendance entre les applications et le système d'exploitation
- portabilité des applications
- partage des services distribués

Les services d'un intergiciel sont :

- Communication
- Localisation
- Transactions
- Sécurité
- Administration

Pas forcement basé sur une extension du mécanisme RPC

Du RPC aux Intergiciels

Des mécanismes plus évoluées visent à remédier aux limitations des RPC

- Objets répartis (Java RMI, CORBA) : abstraction, liaison dynamique, passage des paramètres (référence ou sérialisation)
- Intergiciel orientés messages (Message Oriented Middleware/MOM) : asynchronisme
- Composants répartis (ex : EJB, Corba CCM, .Net)

Remarque

Dans tous les cas, la persistance des données n'est pas prise en compte (il faut la réaliser explicitement par une sauvegarde des données)

Intérêt du paradigme des objets pour la construction d'applications réparties

- Encapsulation :
  - L'interface (méthodes + attributs) est le seul moyen d'accès à l'objet, l'état interne n'est pas directement accessible
- Classes et instances :
  - Offre un mécanisme de génération d'instances conformes conformes à un modèle
- Héritages :
  - Mécanisme de réutilisation de l'existant via un héritage fonctionnel
  - Mécanisme de spécialisation par un ajout de fonctionnalité
- Polymorphisme :
  - Multiples mises en œuvre diverses des servives d'une interface
  - Remplacement d'un objet par un autre si les interfaces compatibles
  - Facilite d'évolution et d'adaptation du code

Notion de référence distribuée

- Le passage d'élément par valeur induit des problèmes de cohérences dans les environnements distribués
- Une **référence** est moyen d'accès à un objet (distant ou pas)
- En général une référence est opaque : sa valeur ne peut pas (ne doit pas) être directement exploitéé
- Contenu d'une référence distribuée :
  - Toutes les informations nécessaires pour atteindre physiquement l'objet
  - Adresse du site qui héberge l'objet
  - Numéro de port sur le site
  - Localisation interne au serveur

Principe de Java RMI

- RMI permet de faire interagir des objets situés dans des espaces d'adressage distincts

Exemple :

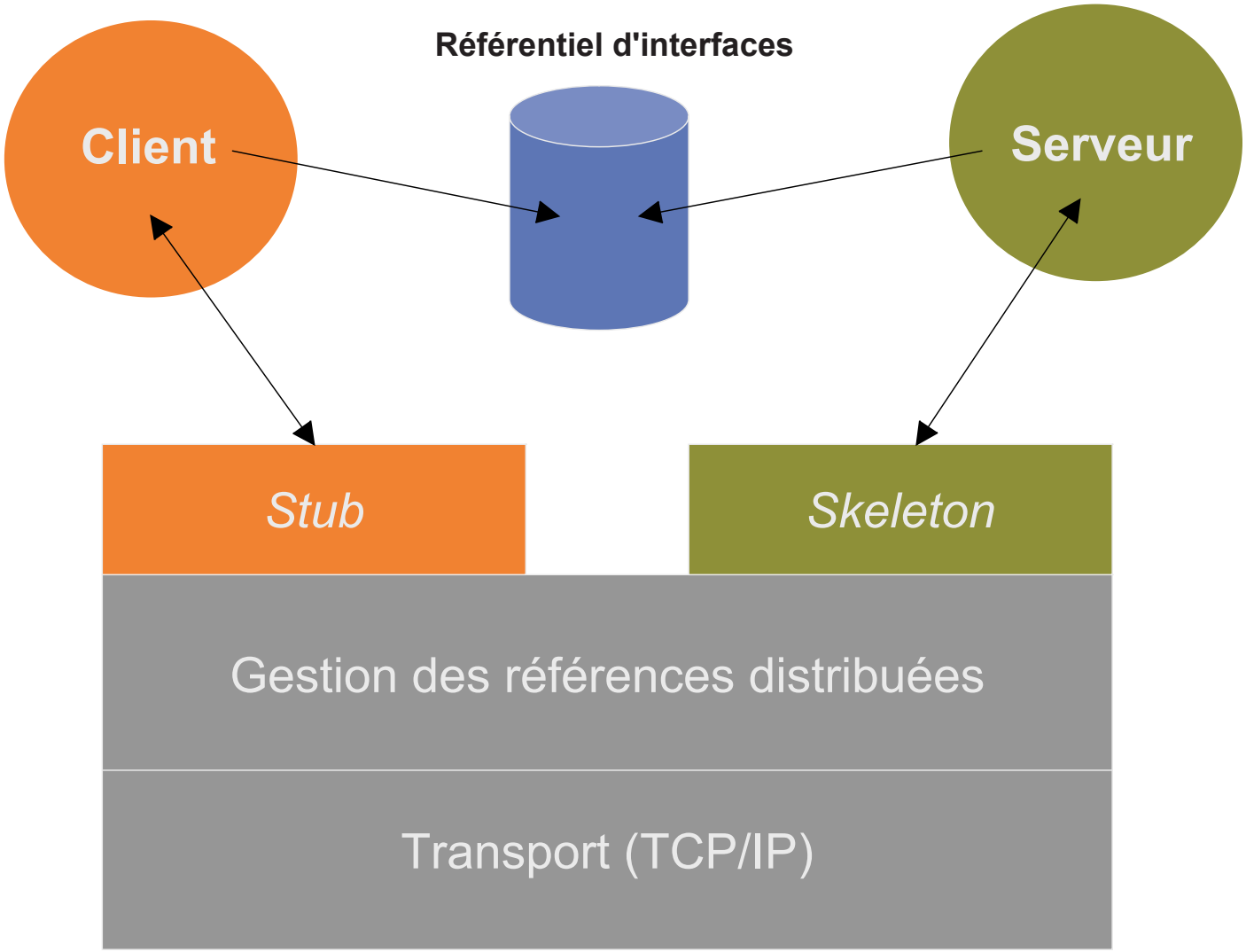
interaction entre objets situés sur des machines distinctes c'est-à-dire dans différentes machines virtuelles différentes

- RMI se veut simple à mettre en œuvre :
  - Définir l'interface de l'objet distribué (OD)
  - Du côté serveur : Implémenter l'OD Générer les stubs et skeletons (avec le compilateur rmic) Enregistrer l'objet distribué sur le bus
  - Côté client : s'attacher à l'OD et invoquer des méthodes
- Un OD se manipule ensuite comme tout autre objet Java

Principe de Java RMI

- Mono-langage et Multiplateforme (bénéfice JVM)
- Java : de JVM à JVM (une même représentation quelque soit la plateforme)
- RMI peut utiliser le mécanisme standard de sérialisation de Java pour l'envoi d'objets
- Propose un chargement dynamique des classes via HTTP
- Apporte en réponse en terme de sécurité : un gestionnaire de sécurité (`SecurityManager`) vérifie si certaines opérations sont autorisés par le serveur

Principe de Java RMI



Principe de Java RMI

- Souche ou *Stub* (sur le client) :
  - représentant local de l'objet distant qui implémente les méthodes 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

Principe de Java RMI

- Couche des références distantes :
  - traduit la référence locale au stub en une référence à l'objet distant
  - elle peut d'appuyer sur un processus tier (`rmiregistry`) pour rechercher des références
- Couche de transport :
  - écoute les appels entrants
  - établit et gère les connexions avec les sites distants : `java.rmi.UnicastRemoteObject` utilise les classes `Socket` et `SocketServer` (TCP) d'autres classes pourraient être utilisées par la couche transport (SSL, UDP par exemple)

Java RMI : règles d'usage

- L'interface d'un objet distant (*Remote*) est celle d'un objet Java, avec quelques contraintes :
  - L'interface distante doit être publique
  - L'interface distante doit étendre l'interface *java.rmi.Remote*
  - Chaque méthode doit déclarer l'exception `java.rmi.RemoteException`
- 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`
  - Une classe distante peut aussi avoir des méthodes appelables seulement localement (dans ce cas, elle ne font pas partie de l'interface)
- Passage d'objets en paramètre :
  - Les objets locaux sont passés par valeur et doivent être sérialisables (étendre l'interface `java.io.Serializable`)
  - Les objets distants sont passés par référence et sont désignés par leur interface (bénéficient de référence distribuée)

Classes et packages essentiels

- `java.rmi` : pour accéder à des objets distants
- `java.rmi.server` : pour créer des objets distants
- `java.rmi.registry` : lié à la localisation et au nommage des objets distants
- `java.rmi.dgc` : ramasse-miettes pour les objets distants
- `java.rmi.activation` : support pour l'activation d'objets distants

Écriture du serveur

Le serveur est une classe qui implémente l'interface de l'objet distant :

1. spécifier les interfaces distantes qui doivent êtres implémentées
2. définir une classe et un constructeur pour les objets distants
3. fournir la réalisation des méthodes des interfaces
4. configurer et instancier le gestionnaire de sécurité
5. créer au moins une instance de la classe serveur
6. enregistrer au moins une instance dans le service de nommage (`rmiregistry`)

Écriture de l'interface serveur

Spécification des interfaces distantes :

Interface : Hello.java

```
import java.rmi.*;
public interface Hello extends Remote {
 public String ditBonjour()
    throws java.rmi.RemoteException;
}
```

Écriture de l'implémentation du serveur

Implémentation : ImpServeurHello.java

```
import java.rmi.server.*;
import java.net.*;
import java.rmi.*;
public class ImpServeurHello extends UnicastRemoteObject implements Hello{
  public ImpServeurHello() throws RemoteException{super();}
  public String ditBonjour() throws RemoteException{return "bonjour a tous";}
  public static void main(String arg[]){
   try{
    ImpServeurHello s=new ImpServeurHello();
    String nom="nomdelobjet";
    Naming.rebind(nom,s); //enregistrement
    System.out.println("Serveur enregistré");
   }
   catch (Exception e){System.err.println("Erreur :"+e);}
 }
}
```

Compilation et lancement du serveur

- compilation du code du serveur :

```
> javac ImpServeurHello.java
```

- génèration stub et skeleton :

```
> rmic ImpServeurHello
```

- démarrer le référentiel d'interface :

```
> rmiregistry &
```

- Lancement du serveur :

```
> java ImpServeurHello
```

Implémentation du client

Implémentation : ClientHello.java

```
import java.rmi.*;
public class ClientHello{
 public static void main(String arg[]){
  try{
   Hello h=(Hello) Naming.lookup("nomdelobjet");
   String messageRecu=h.ditBonjour();
   System.out.println(messageRecu);
  }
  catch (Exception e){System.err.println("Erreur :"+e);}
 }
}
```

Compilation du client :

```
> javac ClientHello.java
```

Exécution :

```
> java ClientHello
```

Quels sont les fichiers `.class` nécessaires à l'exécution du client ?

Gestion de l'annuaire (registre)

- Le serveur peut crée et enregistrer plusieurs objets appartenant à une ou plusieurs classes l'enregistrement peut se faire auprès de plusieurs `rmiregistry`
- Prévoir l'arrêt des serveurs avec la méthode `unbind` de `UnicastRemoteObject`
- La classe `Naming` permet le dialogue avec différents services d'annuaire, l'URL de liaison peut avoir les formes suivantes :
  - `rmi://hostreg:port/Hello/World`
  - `rmi://:port/Hello/World`
  - `//:port/Hello/World`
  - `/Hello/World`
  - `HelloWorld`

Gestion de l'annuaire (registre)

La classe `Naming` propose les méthodes statiques suivantes pour l'enregistrement d'objets :

- `bind(String url, Remote r)`
- `rebind(String url, Remote r)`
- `unbind(String url)`

Pour la recherche dans l'annuaire, `Naming` propose :

- `Remote lookup(String url)` : retourne un stub
- `String[] list()` : liste les noms enregistrés

Gestion de l'annuaire (registre)

La classe `java.rmi.registry.LocateRegistry` propose les méthodes `getRegistry()` et `createRegistry()` pour la localisation et le création dynamique de l'annuaire (utilisée par `Naming`).

Création, localisation de l'annuaire

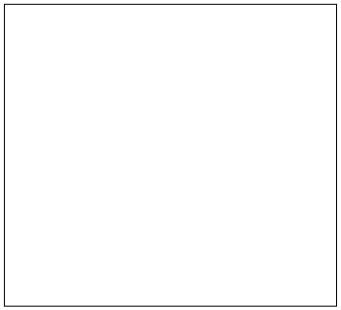
```
import java.rmi.registry.Registry;
import java.rmi.registry.LocateRegistry;
import java.rmi.RemoteException;
import java.rmi.server.UnicastRemoteObject;
public class Serveur {
 public static void main( String [] args) {
   try {
    ODImpl unOD = new ODImpl ();
    //Registry registry = LocateRegistry.createRegistry(1099);
    Registry registry = LocateRegistry.getRegistry();
    registry.bind("MonOD",unOD);
    System.out.println("Serveur␣pret"); }
    catch (Exception e) { System.out.println(e) ; }
       }
}
```

Chargement de classes

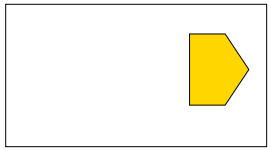
- Quand un client invoque une méthode d'un objet distant, le rmiregistry retourne une référence à cet objet (stub)
- Le rmiregistry doit pouvoir accéder aux classes correspondant à ce stub
- Le rmiregistry cherche d'abord dans son `CLASSPATH` la définition de la classe.
- Ensuite, la recherche est guidée par le propriété codebase, spécifié au lancement du serveur qui a enregistré enregistre l'objet : au lancement de la JVM `java -Djava.rmi.server.codebase=http://kundera.iem/login/mesclasses maclasse`

Cadre

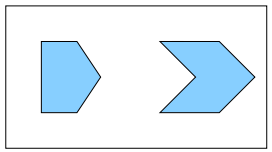
Application client



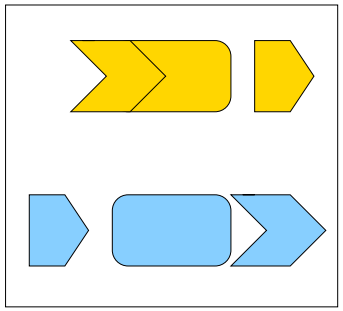
Registry

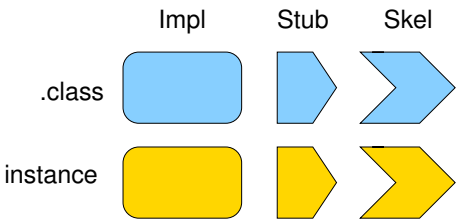


Serveur Web

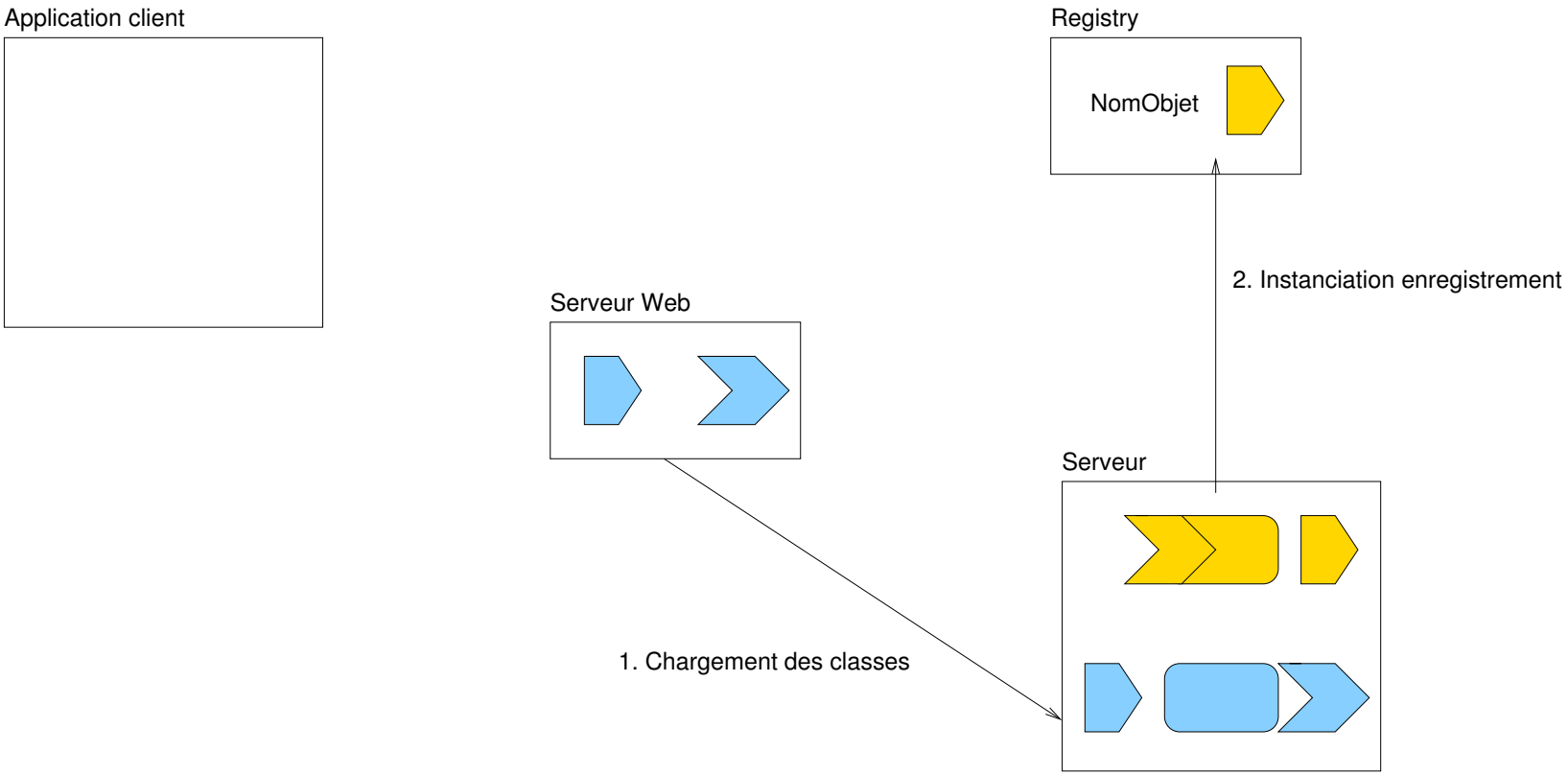


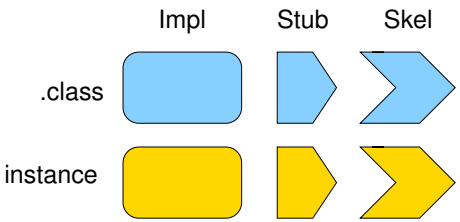
Serveur



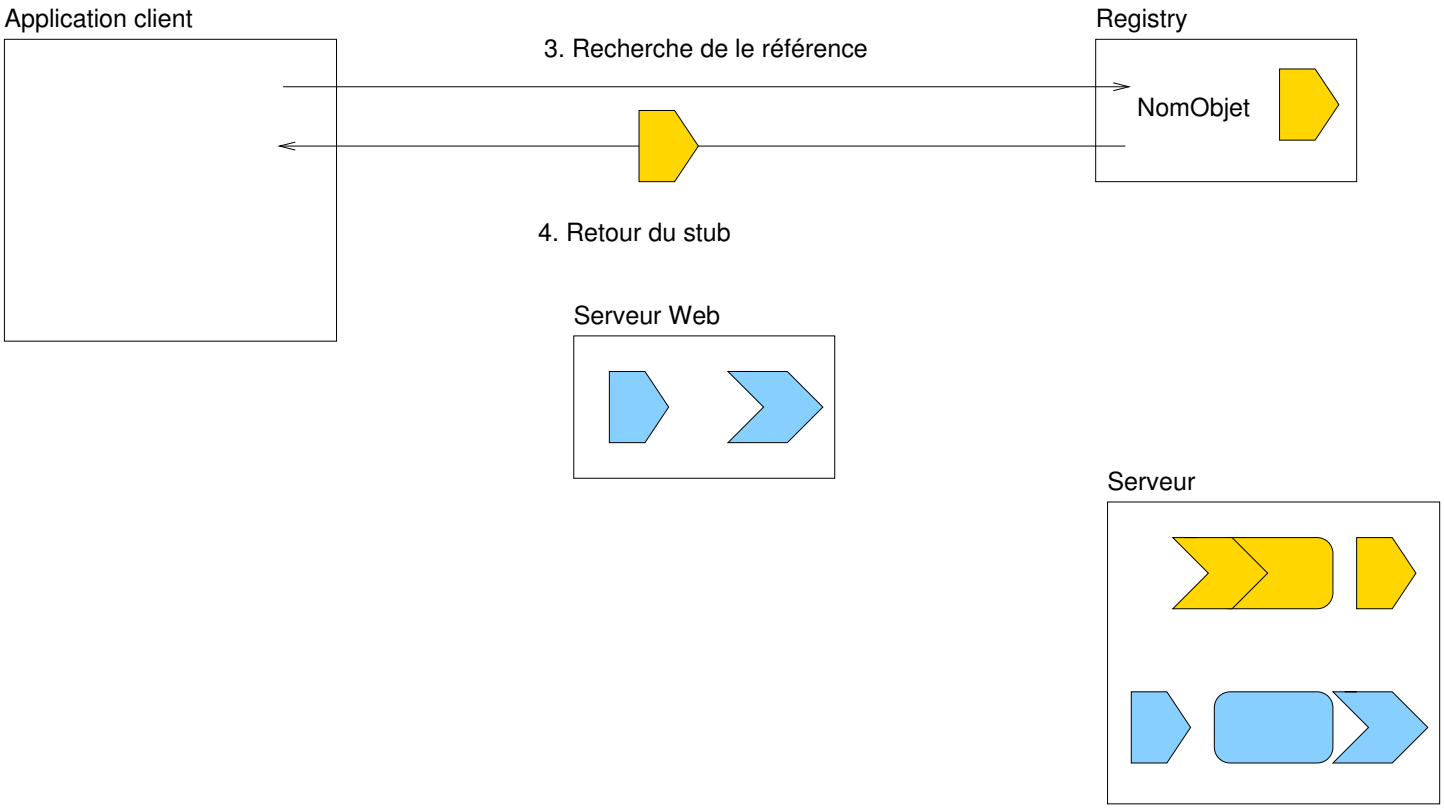


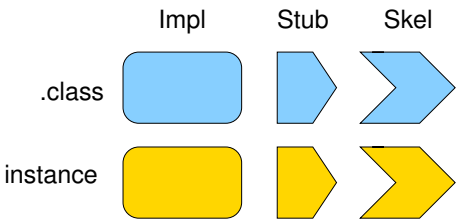
Publication dans l'anuaire



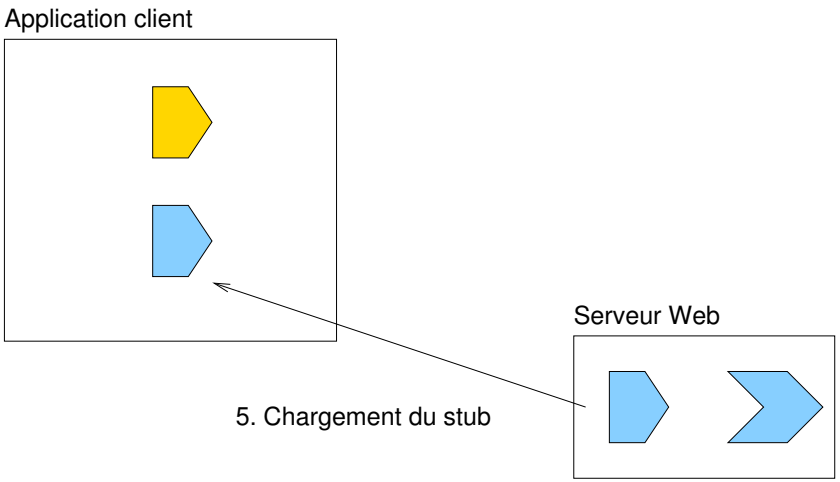


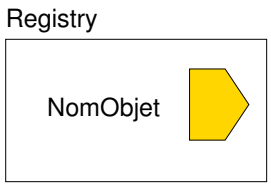
Recherche

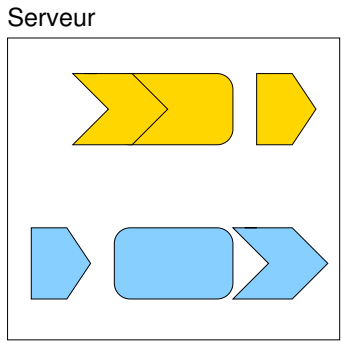

Application client
3. Recherche de le référence
Registry
NomObjet
4. Retour du stub
Serveur Web
Serveur

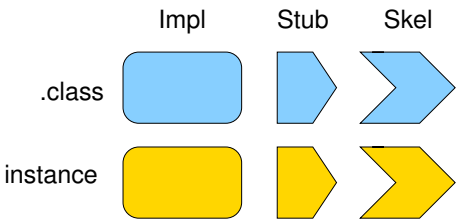

Impl
Stub
Skel
.class
instance

Chargement du stub

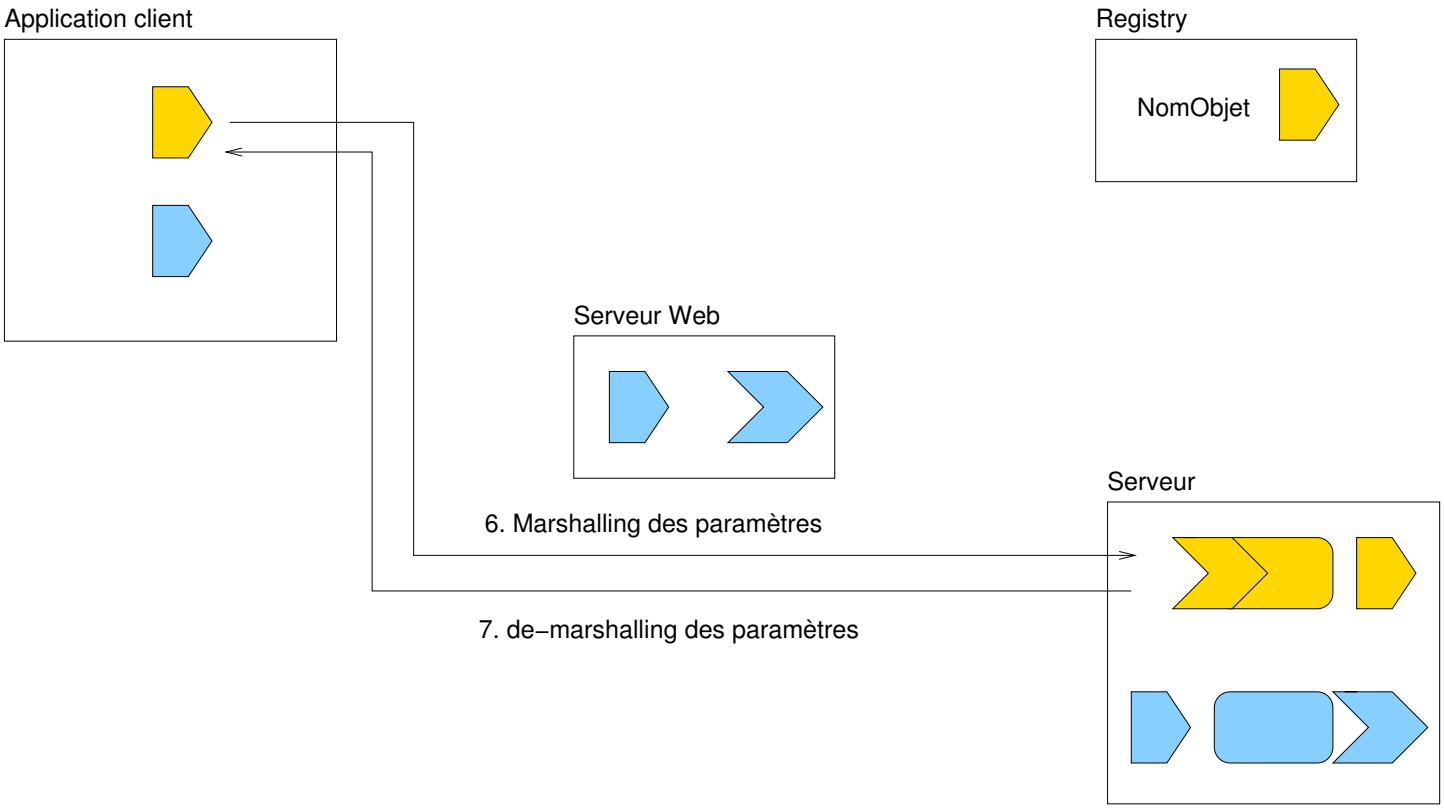


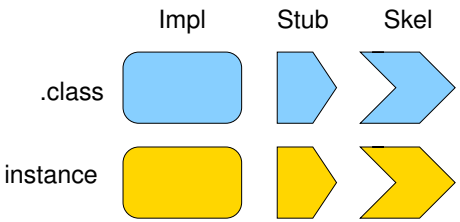






Invocation


Application client
Registry
NomObjet
Serveur Web
6. Marshalling des paramètres
Serveur
7. de−marshalling des paramètres


Impl
Stub
Skel
.class
instance

Passage par valeur

RMI permet de passer en paramètres des méthodes, : des types simples, mais aussi des objets complexes.

- les arguments passés doivent être serializable.
- la sérialisation écrit la structure de l'objet dans un flux binaraire transmissible sur le réseau (une copie de l'ensemble des variables d'état)

Inconvénients : mulitplication des copies d'un objet, traitement des objets référencés, version

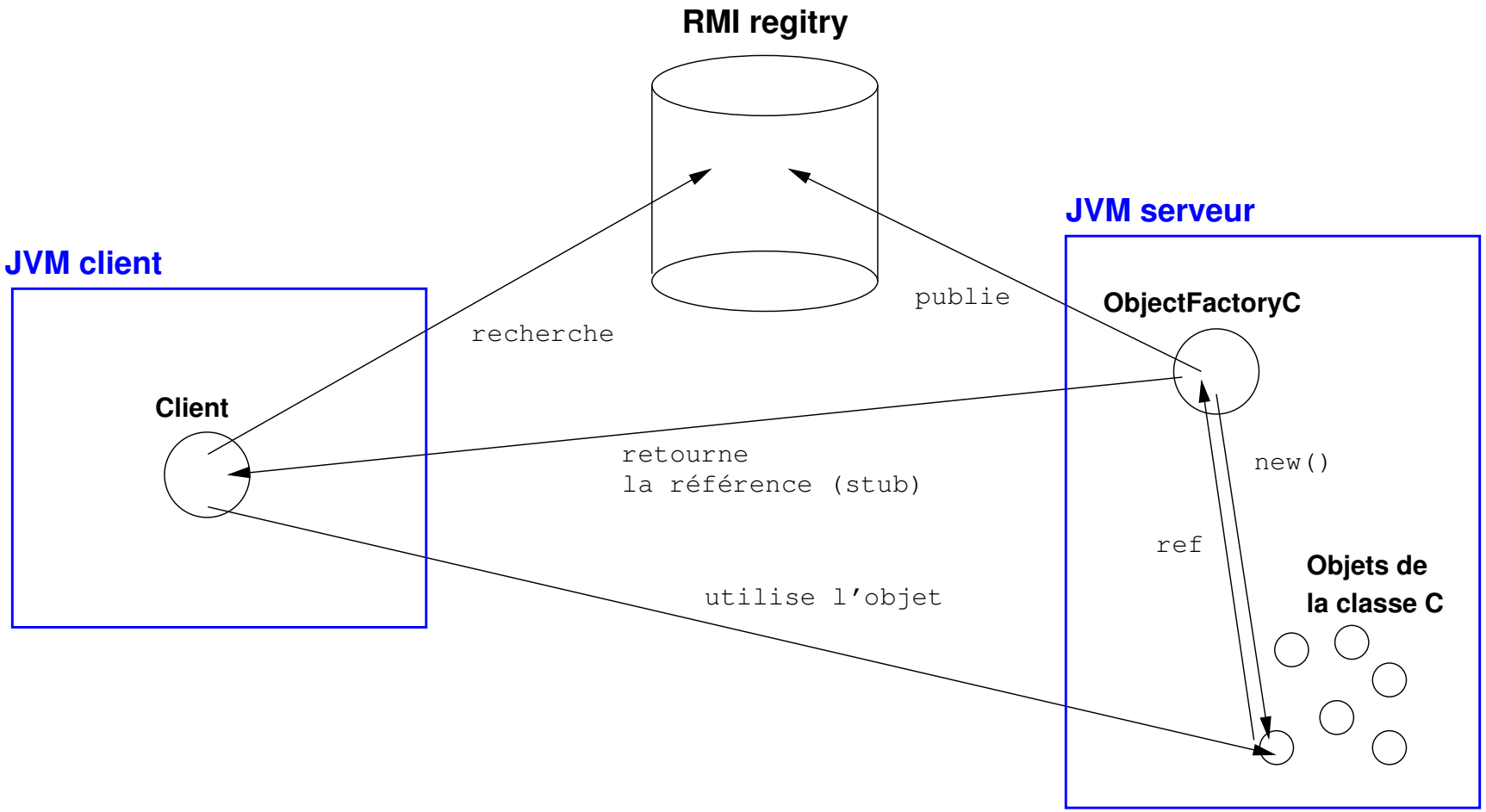
Passage par référence

- Une méthode d'un OD peut retourner une référence à un objet (plus précisément : un stub de l'objet) qui sera aussi utilisable
- Le destinataire utilisera ce stub pour appeler les méthodes de l'objet.
- Les OD passés par référence doivent satisfaire aux conditions suivantes : 1) étendre `Remote` 2) proposer une interface
- Inutile de les enregistrer dans le RMI registry

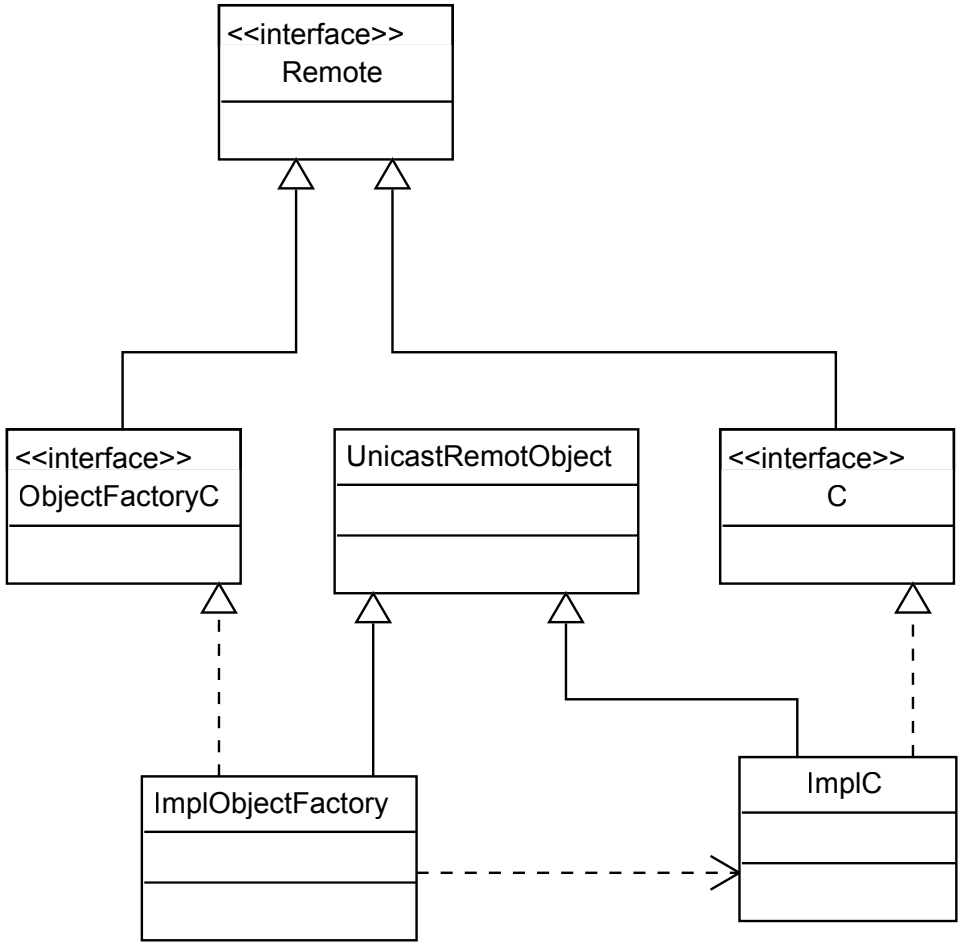
Example (Le pattern object factory)

- Objectif : permettre au client de construire de multiples instances d'une classe C sur le site du serveur
- `new`n'est pas utilisable directement car il ne gère que la mémoire locale (c-à-d celle du client)
- Solution : appeller une méthode d'un OD FabriqueC, qui crée localement (sur le serveur) les instances de la classe C (en utilisant new) et en retournant une référence (remote)

Pattern Object Factory



Pattern Object Factory



Pattern Object Factory

- Inutile d'enregistrer des milliers références dans le registry (exemple de compte bancaires)
- Les objets instancié comsomment la mémoire du serveur :
  - associer une base de données sauver et restorer les états
  - ne pas dupliquer plusieurs objets en mémoire
  - les références doivent être identiques (activation)

Passage par référence et objet mobile

Le serveur reçoit des tâches à exécuter sous la forme d'objets (mobiles)

- Plusieurs clients peuvent utiliser une machine serveur d'objets puissante pour effectuer une tâche rapidement par exemple dans les environnements mobiles
- Le serveur est générique :
  - la tâche n'a pas besoin d'être pré-définie,
  - il accepte en entrée un objet quelconque quipropose une méthode `execute()`
  - la méthode exécute est décrite dans une interface (`Task` l'interface correspondante)
- Le code d'implémentation est chargé dynamiquement depuis le client, vers le serveur par le système RMI
- invoque `execute()`
- retourne le résultat au client.

Interface serveur pour objets mobiles

compute.Compute

```
package compute;
import java.rmi.Remote;
import java.rmi.RemoteException;
public interface Compute extends Remote {
  Object executeTask(Task t) throws RemoteException; }
```

compute.Task

```
package compute;
import java.io.Serializable;
public interface Task extends Serializable {
    Object execute(); // méthode à retrouver sur tous les objets mobiles}
```

- La tâche peut retourner n'importe quel type d'objet
- Si on veut retourner des type primitifs (int, float) sous la forme d'objet, on doit passer par des wrappers de type (Integer, Float)
- L'interface Task étend Serializable afin de permettre le passage d'objet (par valeur) au travers du réseau (entre deux machines)

Implémentation du serveur pour objets mobiles

engine.ComputeEngine

```
package engine;
import java.rmi.*;
import java.rmi.server.*;
import compute.*;
public class ComputeEngine extends UnicastRemoteObject implements Compute{
    public ComputeEngine() throws RemoteException { super(); }
    public Object executeTask(Task t) {return t.execute(); }
    public static void main(String[] args) {
    if (System.getSecurityManager() == null) {
        System.setSecurityManager(new RMISecurityManager());
  }
    String name = "//host/Compute";
  try {
   Compute engine = new ComputeEngine();
   Naming.rebind(name, engine);
   System.out.println("ComputeEngine enregistré");
  } catch (Exception e) {
      System.err.println("ComputeEngine exception: " + e.getMessage());
      e.printStackTrace();
    }
  } // fin méthode main
}
```

Implémentation du client

Le client créer une tâche et l'envoie au serveur.

client.ComputePi

```
package client;
import java.rmi.*;
import java.math.*;
import compute.*;
public class ComputePi {
 public static void main(String args[]) {
  if (System.getSecurityManager() == null) {
   System.setSecurityManager(newRMISecurityManager()); }
 try {
            String name = "//" + args[0] + "/Compute";
            Compute comp = (Compute) Naming.lookup(name);
            Pi task = new Pi(Integer.parseInt(args[1]));
            BigDecimal pi = (BigDecimal) (comp.executeTask(task));
            System.out.println(pi);
        } catch (Exception e) {
            System.err.println("ComputePi exception: " + e.getMessage());
            e.printStackTrace();}
    }
}
```

Implémentation de la tâche

client.Pi

```
package client;
import compute.*;
import java.math.*;
public class Pi implements Task {
    private static final BigDecimal ZERO = BigDecimal.valueOf(0);
    private static final BigDecimal  ONE = BigDecimal.valueOf(1);
    private static final BigDecimal FOUR = BigDecimal.valueOf(4);
    private static final int roundingMode = BigDecimal.ROUND_HALF_EVEN;
    private int digits;
    public Pi(int digits) { this.digits = digits; }
    public Object execute() {return computePi(digits); }
    public static BigDecimal computePi(int digits) {
        int scale = digits + 5;
        BigDecimal arctan1_5 = arctan(5, scale);
        BigDecimal arctan1_239 = arctan(239, scale);
        BigDecimal pi = arctan1_5.multiply(FOUR).subtract(arctan1_239).multiply(FOUR);
        return pi.setScale(digits, BigDecimal.ROUND_HALF_UP);
    }
```

Implémentation de la tâche

client.Pi

```
public static BigDecimal arctan(int inverseX,  int scale)     {
        BigDecimal result, numer, term;
        BigDecimal invX = BigDecimal.valueOf(inverseX);
        BigDecimal invX2 =   BigDecimal.valueOf(inverseX * inverseX);
        numer = ONE.divide(invX, scale, roundingMode);
        result = numer;
        int i = 1;
        do {
            numer = numer.divide(invX2, scale, roundingMode);
            int denom = 2 * i + 1;
            term =  numer.divide(BigDecimal.valueOf(denom), scale, roundingMode);
            if ((i % 2) != 0) { result = result.subtract(term); }
           else {  result = result.add(term);   }
            i++;
        } while (term.compareTo(ZERO) != 0);
        return result;
    }
}
```

Asynchronisme : package concurrent

Java RMI ne supporte pas directement l'asychronisme, cependant plusieurs construction permettent de contourner le problème.
Java 5 introduit la notion de futur : les taches longues sont souvent bloquantes.

```
result = traitementLong();
```

Les threads ne peuvent pas renvoyer directement un résultat par `run()`

```
new Thread() {public void run() {traitementLong();}}
```

Au lieu de `Runnable` on peut utiliser `Callable <T>`. Cette interface a été créée pour compléter l'interface `Runnable`, la méthode run() qu'elle définit retourne une valeur générique, de type V, et peut déclencher une exception.

```
Callable<String> traitement = new Callable<String>(){
    public String call(){ return traitementLong();}
    public String traitementLong(){ return "val";));
String result = traitement.call();
```

Asynchronisme : `Future<V>`

L'interface `Future` permet de gérer la synchronisation du résultat différé d'un `Callable` :

- ne définit pas comment on peut lancer l'exécution de cette tâche
- la méthode `get()` pour récupérer le résultat : retourne le résultat que lorsque le calcul est terminé (appel bloquant)
- la méthode `get(long timeout, TimeUnit unit)` : retourne le résultat, si ce résultat n'est pas disponible au bout du temps précisé en paramètre, déclenche une exception de type `TimeoutException`
- `cancel()` : permet d'interrompre l'exécution de la tâche
- `isCancelled()` : retourne true si la tâche a été interrompue
- `isDone()` : retourne true si l'exécution de cette tâche est terminée, et que son résultat est disponible

Asynchronisme : lancement

- L'interface `RunnableFuture` est une extension des interfaces `Future` et `Runnable` (n'ajoute pas de méthode).
- `FutureTask<V>` implémente `Runnable, Future<V>, RunnableFuture<V>`. On construit une instance de cette classe en lui passant en paramètre une tâche à effectuer.
- Plusieurs possibilités pour lancer la tâche :

```
new Thread(future).start() ;
```

ou utiliser un *Thread Pool Executor*

```
import java.util.concurrent.Callable;
public class MyCallable implements Callable<String> {
    private long waitTime;
    public MyCallable(int timeInMillis){
        this.waitTime=timeInMillis;
    }
    public String call() throws Exception {
        Thread.sleep(waitTime);
        //return the thread name executing this callable task
        return Thread.currentThread().getName();
    }
}
```

Asynchronisme : lancement

```
import java.util.concurrent.*;

public class FutureTaskExample {
    public static void main(String[] args) {
        MyCallable callable1 = new MyCallable(1000);
        MyCallable callable2 = new MyCallable(2000);
        FutureTask<String> futureTask1 = new FutureTask<String>(callable1);
        FutureTask<String> futureTask2 = new FutureTask<String>(callable2);
        ExecutorService executor = Executors.newFixedThreadPool(2);
        executor.execute(futureTask1);
        executor.execute(futureTask2);
        while (true) {
            try {
                if(futureTask1.isDone() && futureTask2.isDone()){
                    System.out.println("Done");
                    //shut down executor service
                    executor.shutdown();
                    return;
                }
                if(!futureTask1.isDone()){
                //wait indefinitely for future task to complete
                System.out.println("FutureTask1 output="+futureTask1.get());
                }
                System.out.println("Waiting for FutureTask2 to complete");
                String s = futureTask2.get(200L, TimeUnit.MILLISECONDS);
                if(s !=null){
                    System.out.println("FutureTask2 output="+s);
                }
            } catch (InterruptedException | ExecutionException e) {
                e.printStackTrace();
            }catch(TimeoutException e){
```

Activation

Rappel du problème :

- Dans des application de SI plusieurs milliers d'objets distribués peuvent être nécessaires
- Le nombre d'objets distribués par JVM est important mais limité
- À un instant donné, seule une partie des objets distribués est utilisée
- En cas de problème sur le serveur, les références distribuées sont recréées
- Comment lancer à la demande des JVM hébergeant des objets distribués

Solution : le principe d'activation de Java

- Permettre d'enregistrer un service RMI sans l'instancier
- Un service RMI défini par cette méthode est inactif, est réveillé seulement quand un client y fait appel
- Le package `jjava.rmi.activation`
- Un démon/service `rmid` écoute les requêtes et active/reveille les objets à la demande

Activation : les étapes

- Inscription du service : à la place du service, une sorte de proxy est enregistré auprès du rmiregistry, Contrairement aux serveurs instances de UnicastRemoteObject, une fausse référence ne s'utilise que pendant de cours instant, pour inscrire le service auprès du rmiregistry, pour les appels au service.
- Activation du service : quand le client appelle un service, le rmiregistry fait appel à une fausse référence, qui vérifie si le vrai service est présent, sinon, elle fait appel au démon rmid pour créer une instance du service puis lui transmet l'appel.
- Transparence des accès : le client n'a en rien besoin de savoir comment le service est implémenté (activation à distance, ou comme serveur permanent). La création du service, surtout du serveur qui l'héberge est un peu différente, mais son code reste similaire.

Activation : définition et propriétés

Créer des objets activables

- Définir des classes d'objets distribués sous-classe de `Activatable`,
- ou bien passer un paramètre à la méthode statique `exportObject()` d la classe `Activatable`

Les propriétés à définir sont :

- Informations associées à chaque objet distribué via `ActivationDesc`
- Identification d'une JVM associée à un ensemble d'objets distribués activables `ActivationGroupID`
- Paramètres d'une JVM associée à un ensemble d'objets distribués `ActivationGroupDesc`

Activation : cycle de développement

- Créer l'interface de l'objet distribué
- Créer une implémentation de l'objet qui étend `Activatable` et qui possède un constructeur obligatoire `C(ActivationID id, MarshalledObject data)`
- Créer le programme qui lancera les objets distribués (le serveur) :
  - crée le/les groupes d'activation
  - crée les objets et les associe à un groupe d'activation
  - enregistre l'objet dans le registre
- Créer un client (pas de changement)
- Lancer le registre `rmiregistry` et le démon d'activation `rmid`
- Lancer le serveur et le client

Activation : code de l'OD

Interface : Hello.java

```
import java.rmi.*;
public interface Hello extends Remote {
 public String sayHello(String name) throws RemoteException;
}
```

Implémentation : HelloImpl.java

```
import java.rmi.*;
import java.rmi.activation.*;
import java.io.Serializable;
public class HelloImpl extends Activatable implements Hello,Serializable {
 public HelloImpl(ActivationID id, MarshalledObject data) throws RemoteException{
  super(id, 0);
 }
 public String sayHello(String nom) throws RemoteException{
  return "Hello␣"+nom;
 }
}
```

Activation : code de l'OD

Serveur : Server.java

```
imort jatil.
import java.rmi.registry.*;
import java.rmi.RMISecurityManager;
import java.rmi.activation.*;
public class Server{
 public static void main(String args[]) throws Exception{
 int port = 1099;
  try{
   System.setSecurityManager(new RMISecurityManager());
   Properties env = new Properties();
   env.put("java.security.policy","file://home/leclercq/.../Activation/helloworld.policy"
        );

  ActivationGroupDesc myGroupDesc = new ActivationGroupDesc(env, null);
  ActivationGroupID myGroupID = ActivationGroup.getSystem().registerGroup(myGroupDesc);
  ActivationGroup.createGroup(myGroupID, myGroupDesc, 0);
  ActivationDesc objectDesc = new ActivationDesc("HelloImpl","file://home/leclercq/.../
       Activation/HelloWorld-Activable", null);

  Hello myobject = (Hello)Activatable.register(objectDesc);

  Registry registry = LocateRegistry.getRegistry(port);
  registry.rebind("HelloActivable", myobject);

  System.out.println("HelloActivatable active et lie au registre");

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

Activation : code de l'OD

Paramètres : helloworld.policy

```
grant {
// Allow everything for now
permission java.security.AllPermission;
};
```

Activation : code de l'OD

Client : Client.java

```
import java.rmi.registry.LocateRegistry;
import java.rmi.registry.Registry;
public class Client {
 public static void main(String args[]) {
 String machine = "hostServ";
 int port = 1099;
 try {
  Registry registry = LocateRegistry.getRegistry(machine, port);
  Hello obj = (Hello) registry.lookup("HelloActivatable");
  System.out.println(obj.sayHello("toto");
 } catch (Exception e) {
 System.out.println("Client exception: " +e);
}
}
}
```

Exécution

Lignes de commandes

```
#lancement du registre et du demon d'activation
rmiregistry␣&
rmid␣-J-Djava.security.policy=helloworld.policy␣&


java␣-Djava.security.policy=helloworld.policy␣-Djava.rmi.server.codebase=...␣Server

java␣Client
```

- Comment intégrer la persistance et l'activation ?
- Arrêter le serveur et reprendre une interaction.