Programmation en Java/RMI

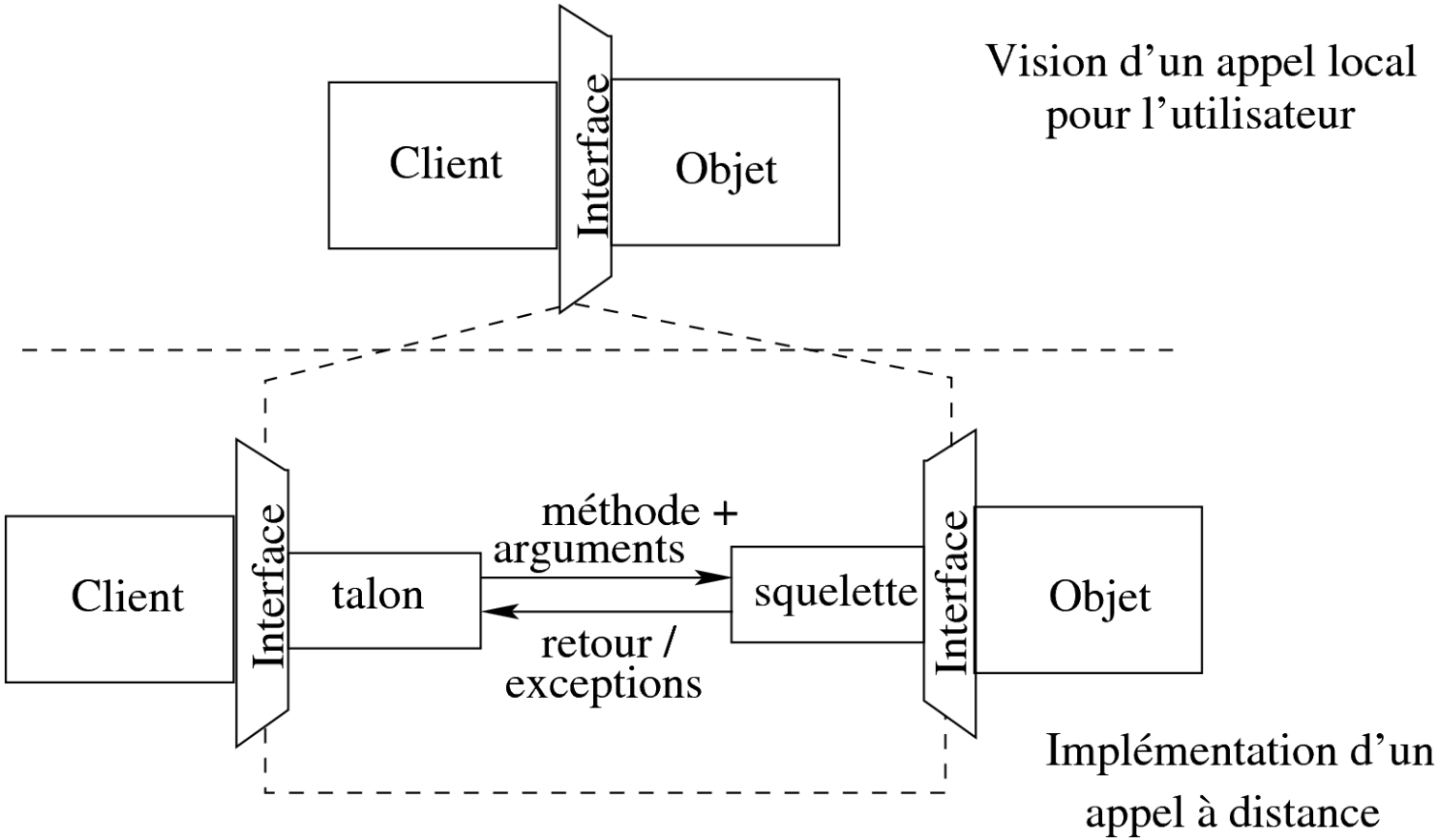
Module
Systèmes Communicants et Synchronisés
Master Informatique
1ère année

V. Felea & L. Philippe

Introduction

- principe
  - communication entre objets java distants (*remote*)
  - définition d'une interface
  - invocation de méthodes sur des objets distants presque comme en local
- depuis la version 1.1, quelques évolutions
- appelations
  - appel de méthode à distance (Remote Method Invocation : RMI)
  - modèle d'objets distribués Java

Principe d'invocation distante



Rôles

- interface : liste des méthodes (services rendus)
- implémentation de l'objet distant : implémente les méthodes
- serveur : met à disposition l'objet distant
- client : invocation des méthodes

- squelette : bibliothèque serveur
- talon : bibliothèque client

} souches générées automatiquement

Conception

- modèle client/serveur
  - 3/4 entités : client – interface/implémentation – serveur
- définition de l'interface
  - lien avec définition protocole
  - méthodes publiques (services rendus)
- implémentation de l'interface
  - donner le code de toutes les méthodes de l'interface
  - attention aux prototypes
  - plusieurs interfaces distantes

Interface (1)

- classique : classe purement virtuelle
  - liste de méthodes publiques
  - héritage
  - héritage multiple
- *distante*
  - hérite de l'interface *java.rmi.Remote*

Interface (2)

- méthodes supportées par le serveur
  - toutes les méthodes distantes de l'objet distant
  - d'autres méthodes de l'objet distant qui ne seront pas accessible à distance
- méthodes distantes : exception
  - *java.rmi.RemoteException*
  - ou super classes
    - *java.io.IOException*
    - *java.lang.Exception*
  - gestion de tous les cas d'erreur réseau et rmi
- exceptions liées à l'application

RemoteException

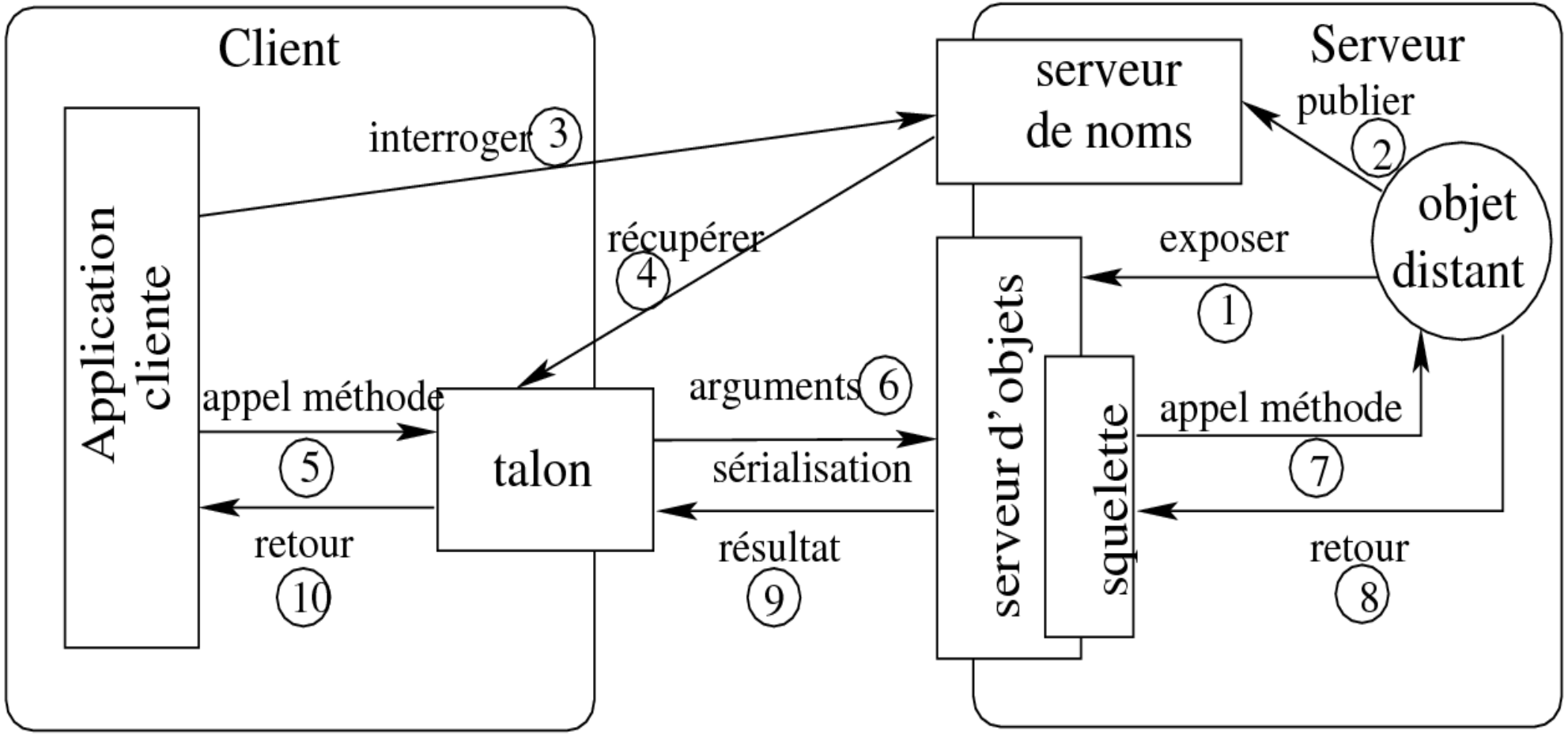
- robustesse des applications RMI
- étend la classe *IOException*
- déclenchée en cas d'erreur communication RMI
  - communication failure
    - serveur inaccessible, refus de connexion, connexion fermée, etc
  - erreur marshalling/unmarshalling
  - erreur de protocole
- traitement classique

```
try {... } catch (RemoteException re) { ... }
```

Exemple de code simple - interface

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

public interface HelloIntf extends Remote {
    // les méthodes distantes peuvent lever une RemoteException
    public void hello() throws RemoteException;
}
```

Fonctionnement RMI



Implémentation de l'objet distant

- implémentation des méthodes
  - **toutes** les méthodes de l'interface
  - d'autres méthodes (**locales**)
- hérite de *RemoteObject* (*UnicastRemoteObject*)
- implémente l'interface *Remote*

Exemple de code simple – interface et implémentation de l'objet distant

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

public interface HelloIntf extends Remote {
   // les méthodes distantes peuvent générer une RemoteException
   public void hello() throws RemoteException;
}
```

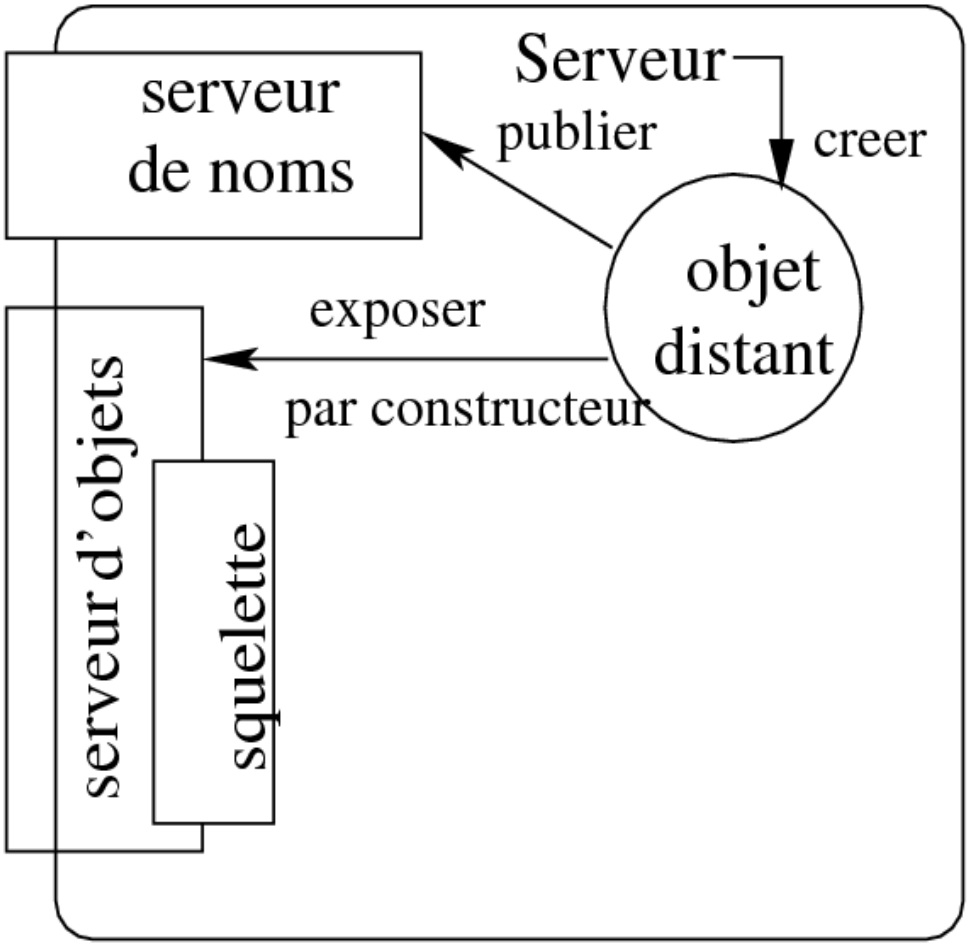
```
package exhello;
import java.rmi.server.UnicastRemoteObject;
import java.rmi.Remote;
import java.rmi.RemoteException;
public class HelloImpl extends UnicastRemoteObject
                       implements HelloIntf {
   public HelloImpl()  throws RemoteException  { super(); }
   // les méthodes distantes lèvent Remote Exception
   public void hello() throws RemoteException  {
      System.out.println("Hello world");
   }
}
```

Serveur

- création de l'objet → référence d'objet
- déclaration au serveur de noms avec un nom - *publication*

- attente de requêtes (pool de threads)
- peut faire autre chose en même temps (threads)

Serveur – actions



Référence d'objet

- serveur : type de l'implémentation
- client : type de l'interface
- enregistrée
  - répertoire: rmiregistry, ldap, jndi, etc
  - association
    - nom objet distant – référence d'objet distant
    - [rmi:]//nom_machine:port/nomObjDist
- peut être passée en paramètre (*Serializable*)

Le serveur de noms – rmi registry (1)

- enregistre les associations : «nomObjetDistant» - référence = publication (en *local*)
- connexion initiale entre le client et le serveur (*bootstrapping*)
- lancement
  - autonome (shell) - `rmiregistry [no_port]`
  - par programme - classe *java.rmi.registry.LocateRegistry*
- lancé sur adresse connue - host : localhost / port : 1099
- implémentation d'un fournisseur de service de résolution de nom au dessus de jndi (Java Naming and Directory Interface)
  - spécification Java masquant les différentes implémentations de services de résolution de noms

Le serveur de noms – rmi registry (2)

- classe *java.rmi.Naming*
- méthodes
  - *bind* : définition d'une association
  - *rebind* : redéfinition d'une association
  - *lookup* : recherche d'une référence
  - *list* : liste des noms
  - *unbind* : suppression d'une association
- exceptions
  - *AccessException* : accès distant non permis
  - *NotBoundException* : nom inexistant

Exemple de code simple – serveur

```
package exhello;
import java .rmi.Remote;

public interface HelloIntf extends Remote {
        // interface de l'objet distant
}
```

```
package exhello;
import java.rmi.server.UnicastRemoteObject;
import java.rmi.Remote;
public class HelloImpl extends UnicastRemoteObject
                        implements HelloIntf {
        // implémentation de l'objet distant
}
```

```
package exhello;
import java.rmi.Naming;
import java.rmi.RemoteException;
import java.net.MalformedURLException;
public class HelloServer {
    public static void main(String [] args) {
       try {
          HelloImpl objServHello = new HelloImpl();
          try{
             Naming.rebind("ObjHelloRemote", objServHello);
          }  // nécessaire pour utiliser le Naming
          catch(MalformedURLException e) {...}
          System.out.println("Serveur pret");
        } catch (RemoteException re) {...}
    }
}
```

Classe *LocateRegistry*

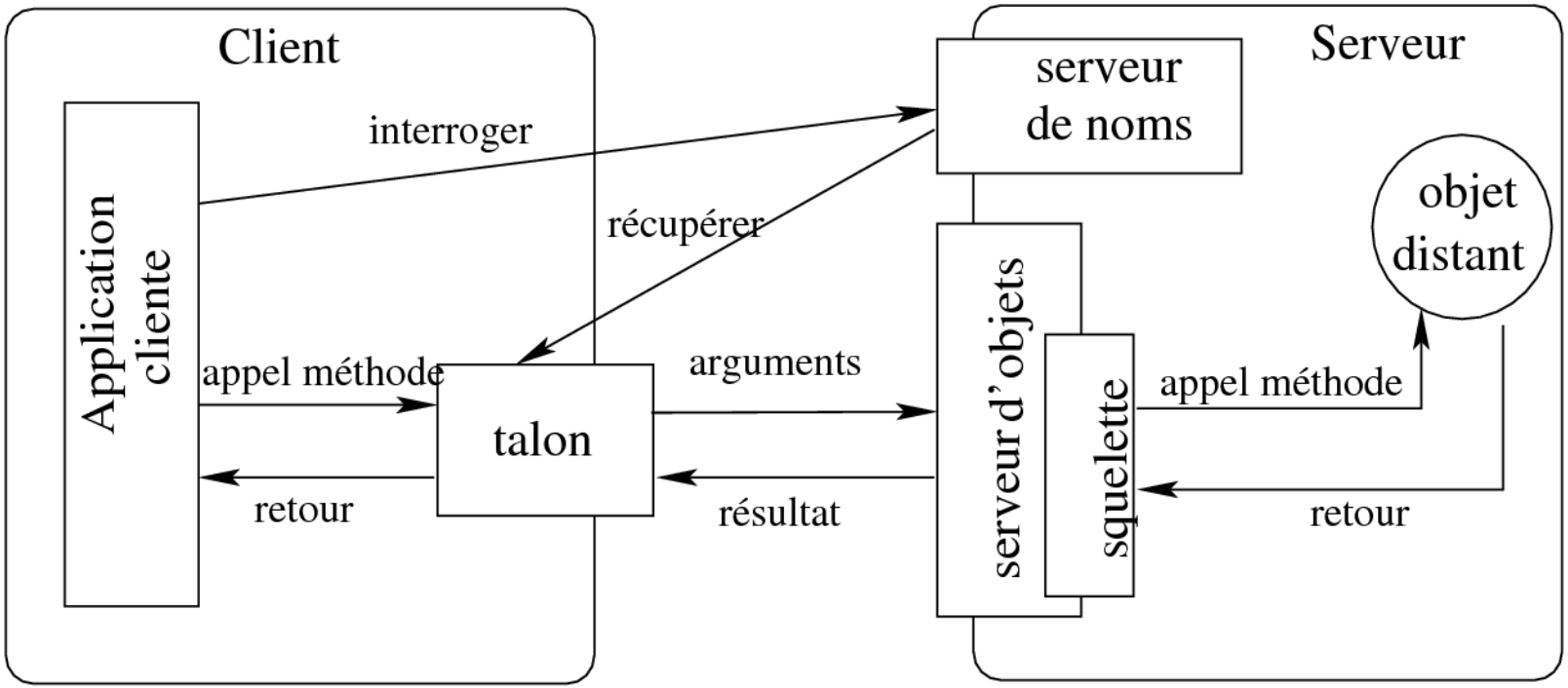
- méthodes statiques
  - *static Registry createRegistry(port)*
  - *static Registry getRegistry([port],[host])*
- gestion depuis le serveur
- port par défaut : 1099

```
// création d'un objet distant
HelloImpl objServHello = new HelloImpl();

// création dynamique de rmiregistry
Registry myRegistry = LocateRegistry.createRegistry(PORT);

//publier l'objet
myRegistry.rebind("ObjHelloRemote",objServHello);
```

Client

- accède à l'objet distant grâce au serveur de noms
- conversion explicite de la référence obtenue
  - *Object* → interface distante
- invocation « classique »
    - traitement de l'exception *RemoteException*

Client – actions



Exemple de code simple – client

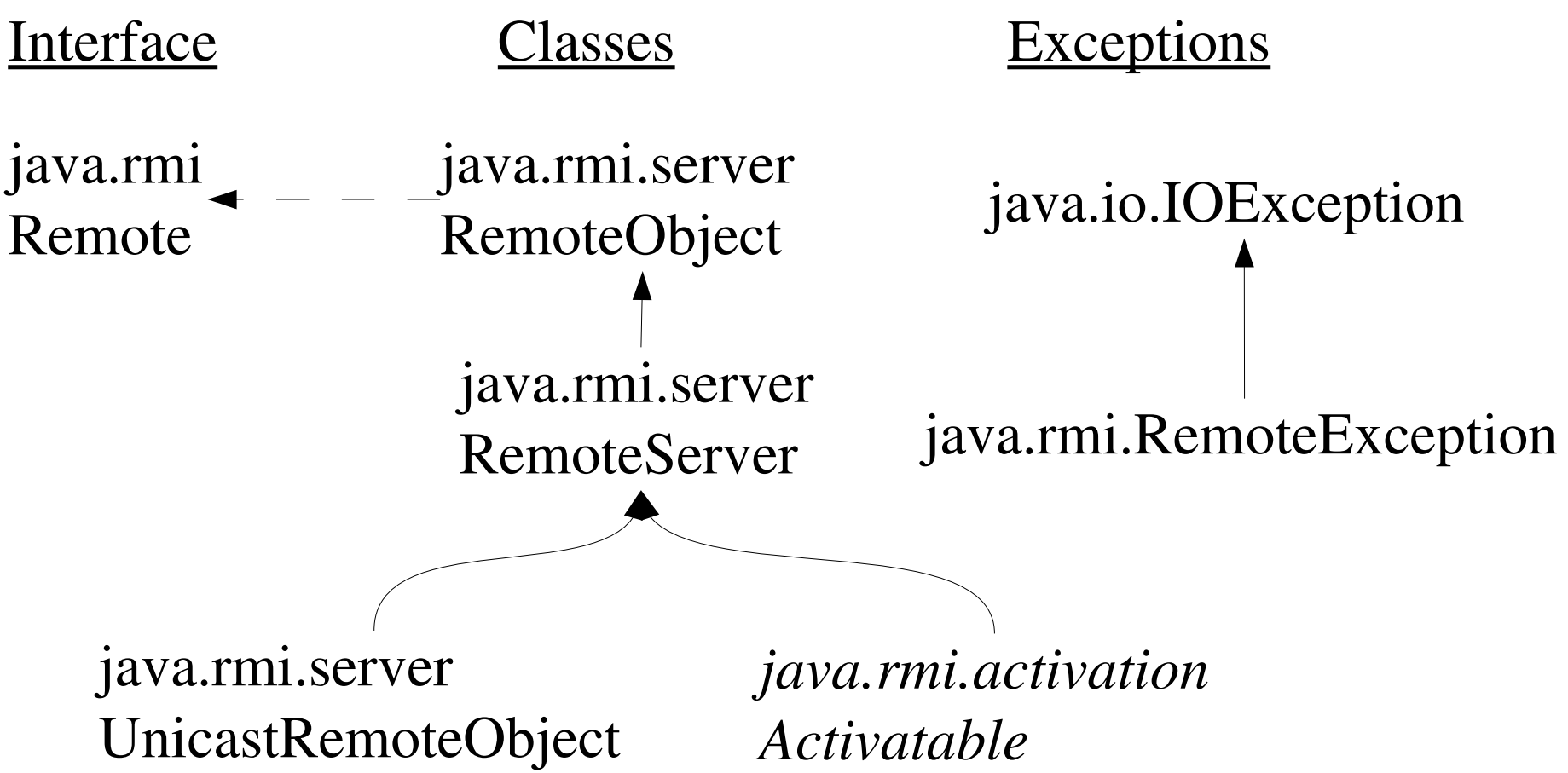
```
package exhello;
import java.rmi.Remote;

public interface HelloIntf extends Remote {
        // interface de l'objet distant
}
```

```
package exhello;
import java.rmi.server.UnicastRemoteObject;
public class HelloImpl extends UnicastRemoteObject
                        implements HelloIntf {
        // implémentation de l'objet distant
}
```

```
package exhello;

public class HelloServer {
   public static void main(String [] args) {
      ...
      // publication de l'objet distant
      Naming.rebind("ObjHelloRemote", objServHello);
      …
   }
}
```

```
package exhello;
import java.rmi.Naming;
import java.rmi.RemoteException;
import java.rmi.NotBoundException;
public class HelloClient {
   public static void main(String [] args)  {
      try {
         HelloIntf objHello =(HelloIntf) Naming.lookup(
               "//"+args[0]+"/ObjHelloRemote");

         // invocation de méthode sur l'objet distant
         objHello.hello() ;
      }// gestion des exceptions lookup et invocation
      catch (RemoteException e) { … }
      catch (NotBoundException e) { … }
      catch (java.net.MalformedURLException e) { … }
   }
}
```

Programmation

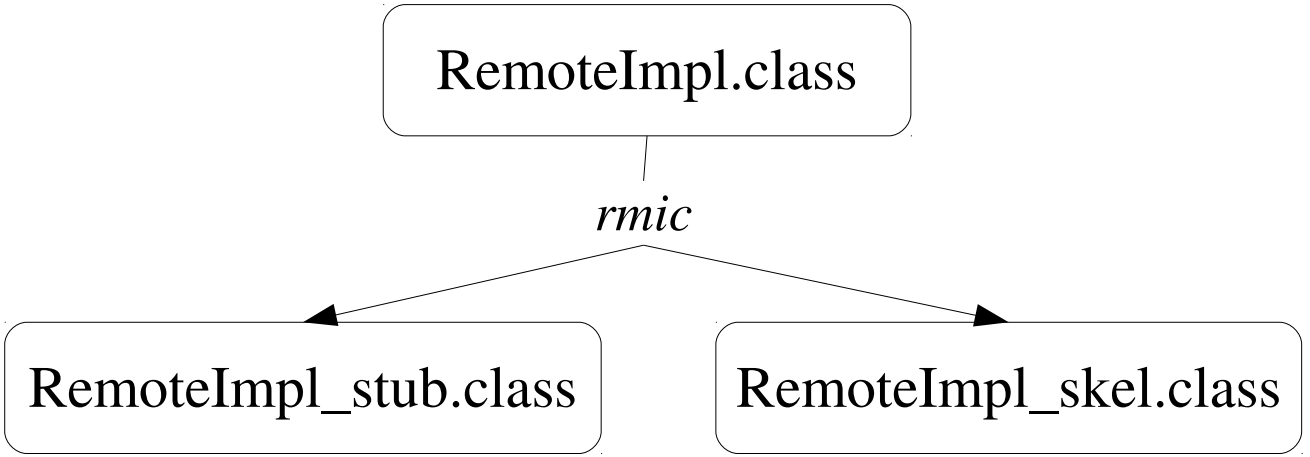


Utilisation

- compilation
  - javac -d . *.java
  - rmic -d . *paq.ImplObjDist* (jdk < 1.5)
- déploiement
  - classes nécessaires
- exécution
  - rmiregistry: accès aux classes (! CLASSPATH)
  - java *paq.classeServeur* [arguments] / *paq.classeClient* [arguments]

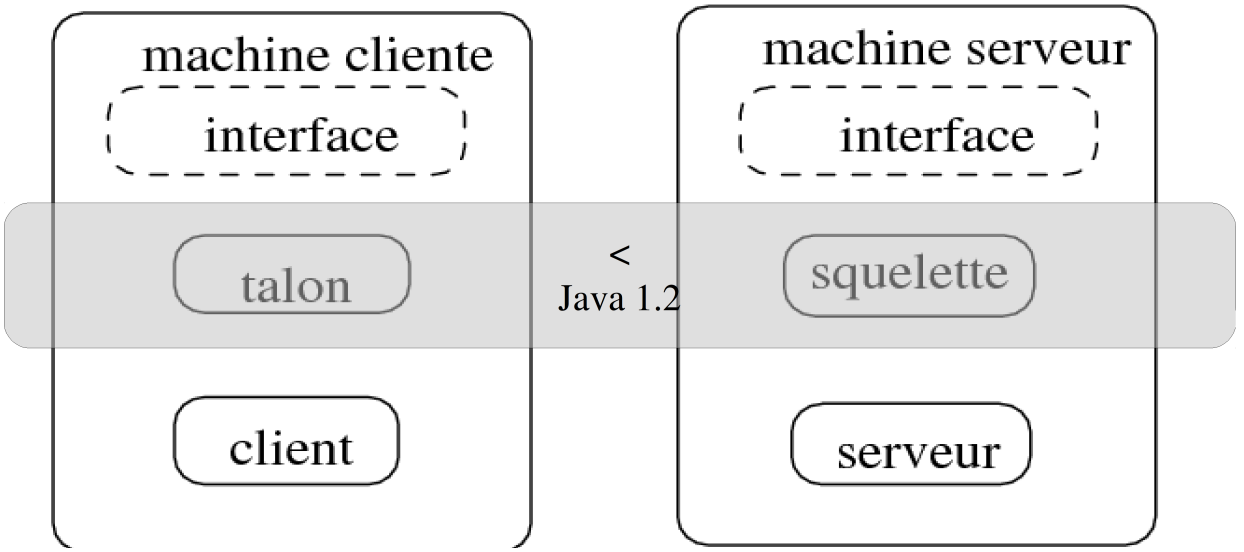
Génération des souches talon/squelette

- les souches (talon, squelette) sont générées par l'outil *rmic*
- le compilateur *rmic* prend en argument une classe d'implémentation d'objet distant

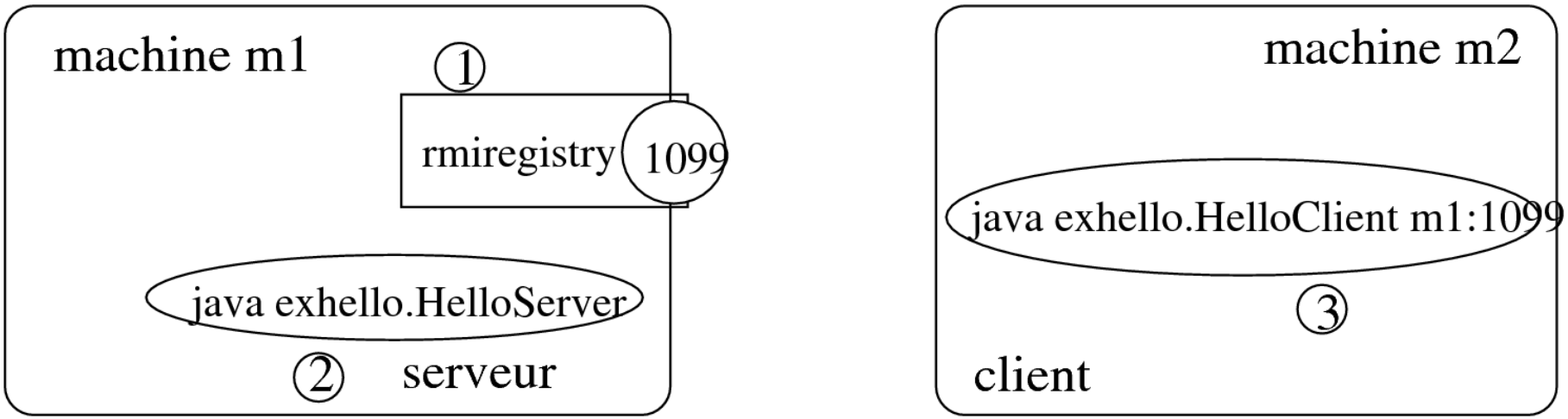


- avant Java 1.2 – souches nécessaires
- avec Java 1.2 – squelette optionnel (introspection)
- à partir de Java 1.5 – souches générées automatiquement (*java.lang.reflect.Proxy*)

Déploiement d'une application Java RMI



Exemple hello – exécution



Passage de paramètres (1)

- objets locaux (non distants)
  - par valeur
  - duplication
    - pas d'allocation
    - objet toujours disponible après envoi
- bas niveau : sockets
  - types primitifs : par défaut
  - types références : sérialisables (*java.io.Serializable*)

Passage de paramètre : objet local (1)

interface

```
public interface IntfDist extends Remote {
    public void passe(MonObjet objLoc)
        throws RemoteException;
}
```

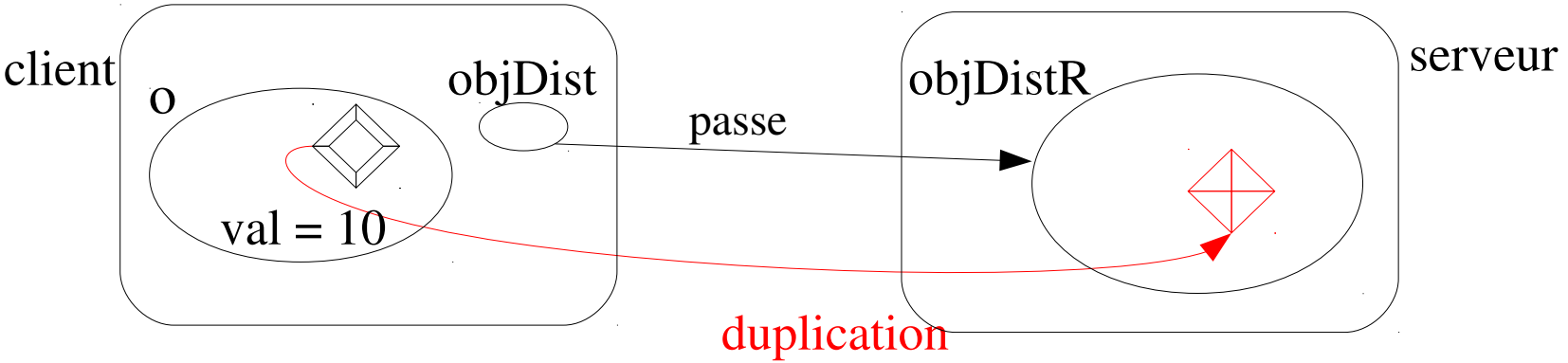

```
// allocation
MonObjet o = new MonObjet(10);
objDist.passe(o);
o.setVal(20);
```

```
void passe(MonObjet objLoc) {
    // pas d'allocation
    objLoc.setVal(30);
}
```

Passage de paramètre : objet local (2)

interface

```
public interface IntfDist extends Remote {
    public void passe(MonObjet objLoc)
      throws RemoteException;
}
```



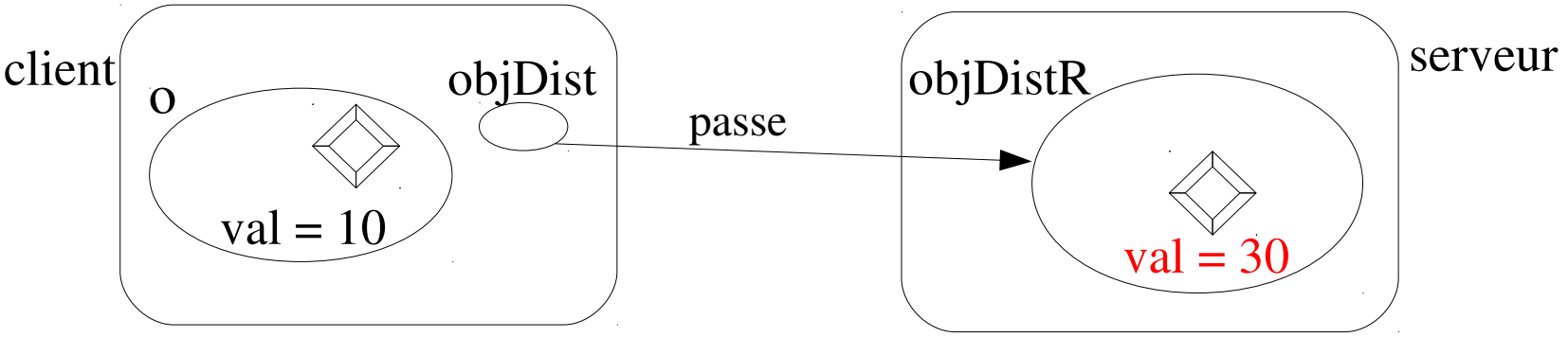

```
// allocation
MonObjet o = new MonObjet(10);
objDist.passe(o);
o.setVal(20);
```

```
void passe(MonObjet objLoc) {
   // pas d'allocation
   objLoc.setVal(30);
}
```

Passage de paramètre : objet local (3)

interface

```
public interface IntfDist extends Remote {
    public void passe(MonObjet objLoc)
      throws RemoteException;
}
```



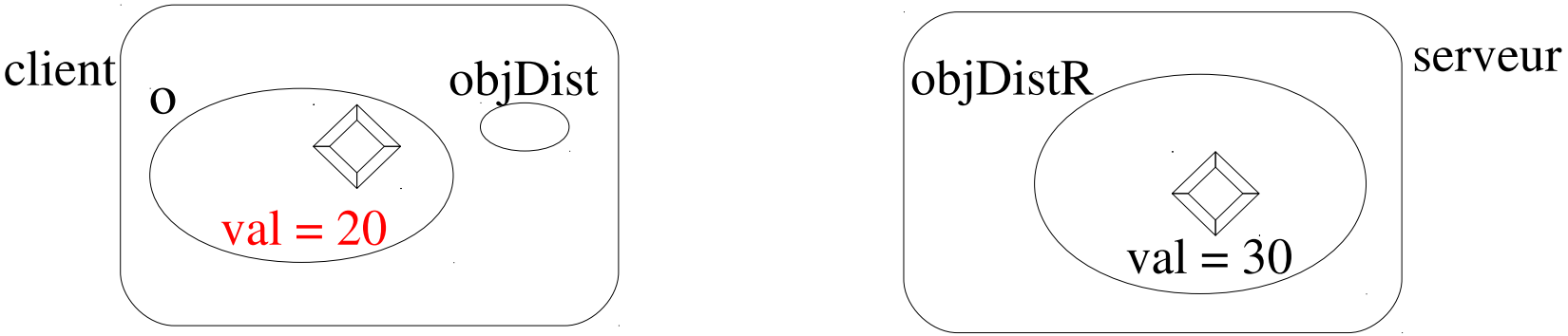

```
// allocation
MonObjet o = new MonObjet(10);
objDist.passe(o);
o.setVal(20);
```

```
void passe(MonObjet objLoc) {
   // pas d'allocation
   objLoc.setVal(30);
}
```

Passage de paramètre : objet local (4)

interface

```
public interface IntfDist extends Remote {
    public void passe(MonObjet objLoc)
      throws RemoteException;
}
```



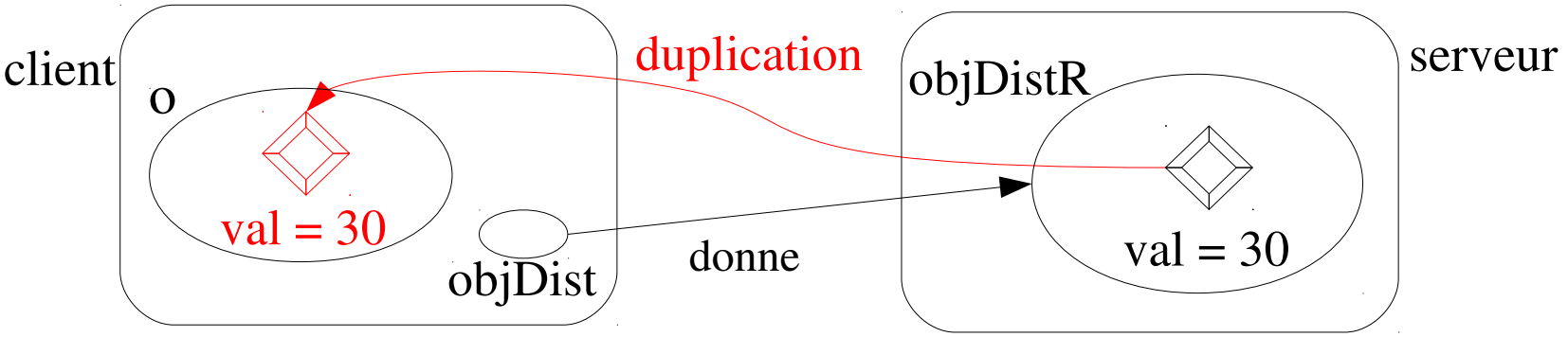

```
// allocation
MonObjet o = new MonObjet(10);
objDist.passe(o);
o.setVal(20);
```

```
void passe(MonObjet objLoc) {
   // pas d'allocation
   objLoc.setVal(30);
}
```

Passage de paramètre : objet local (5)

interface

```
public interface IntfDist extends Remote {
    public MonObjet donne( )
       throws RemoteException;
}
```



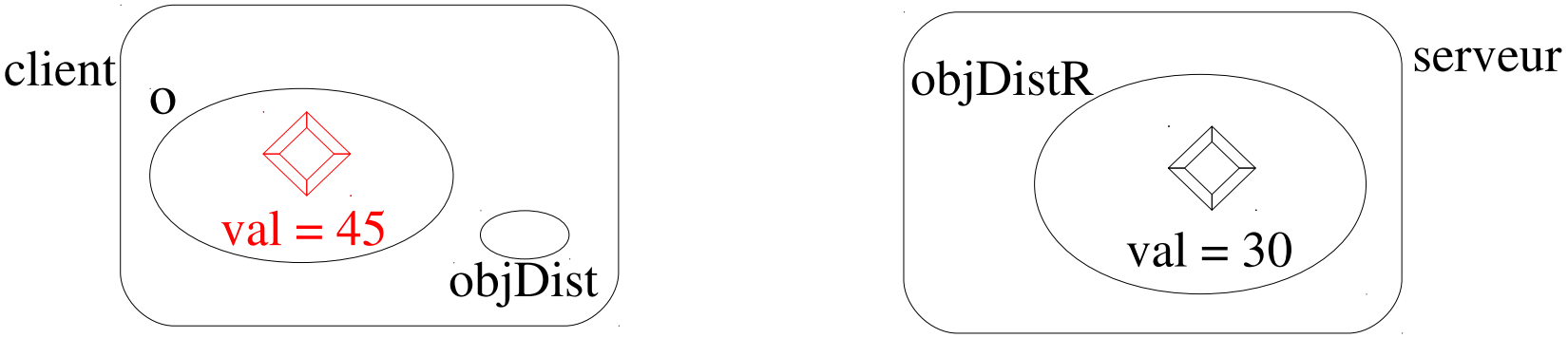

```
// pas d'allocation
MonObjet o = objDist.donne();
o.setVal(45);
```

```
MonObjet donne( ) {
    // allocation
    MonObjet objLoc = new MonObjet(30);
    return objLoc;
}
```

Passage de paramètre : objet local (6)

interface

```
public interface IntfDist extends Remote {
    public MonObjet donne( )
       throws RemoteException;
}
```




```
// pas d'allocation
MonObjet o = objDist.donne( );
o.setVal(45);
```

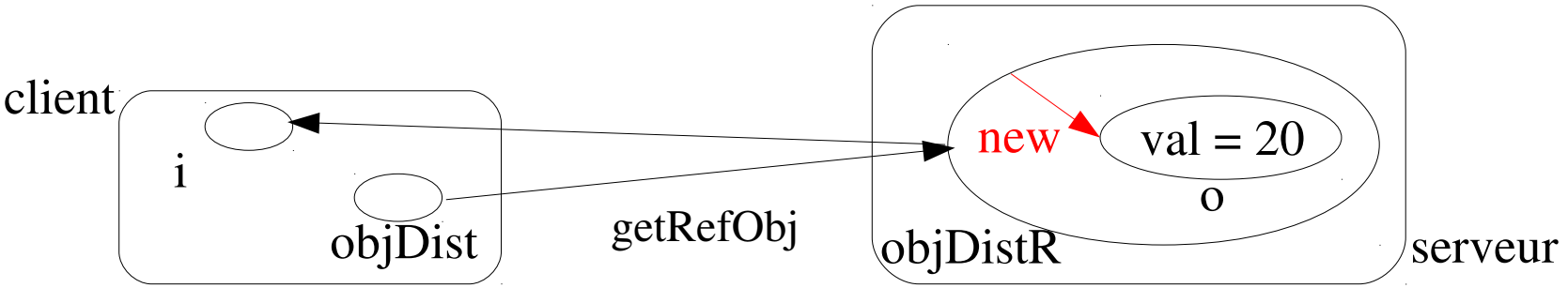
```
MonObjet donne( ) {
    // allocation
    MonObjet objLoc = new MonObjet();
    objLoc.setVal(30);
    return objLoc;
}
```

Passage de paramètres (2)

- objet distant (remote)
  - par référence (talon)
  - à partir de l'interface pas la classe
  - n'implémente que des interfaces distantes
- *UnicastRemotObject* étend *RemoteObject* (implémente *Serializable*)

Passage de paramètre : objet distant (1)

```
public interface IntfDist extends Remote {
   public ObjetIntf getRefObj( )
      throws RemoteException;
}
public interface ObjetIntf extends Remote {
      public int getVal() throws RemoteException;
      public void setVal(int v) throws RemoteException;
}
```



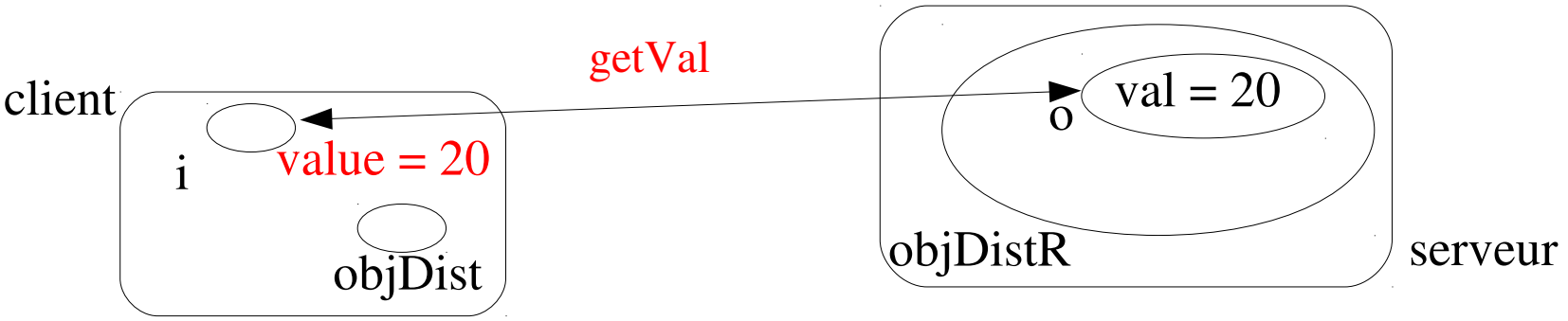

```
ObjetIntf i  = objDist.getRefObj();
```

```
ObjetIntf o = new ObjetImpl();
o.setVal(20);

ObjetIntf getRefObj( ) {
   return o;
}
```

constructeur IntfDistImpl

Passage de paramètre : objet distant (2)

```
public interface IntfDist extends Remote {
   public ObjetIntf getRefObj( )
      throws RemoteException;
}
public interface ObjetIntf extends Remote {
      public int getVal() throws RemoteException;
      public void setVal(int v) throws RemoteException;
}
```

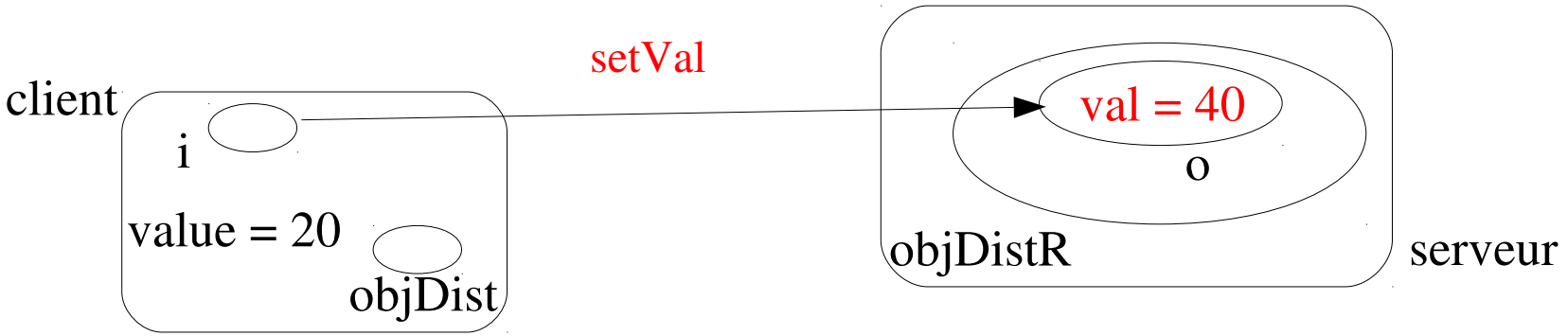



```
ObjetIntf i  = objDist.getRefObj( );
int value = i.getVal(); // 20
```

```
ObjetIntf o = new ObjetImpl();
o.setVal(20);
ObjetIntf getRefObj( ) {
   return o;
}
```

Passage de paramètre : objet distant (3)

```
public interface IntfDist extends Remote {
   public ObjetIntf getRefObj( )
      throws RemoteException;
}
public interface ObjetIntf extends Remote {
      public int getVal() throws RemoteException;
      public void setVal(int v) throws RemoteException;
}
```




```
ObjetIntf i  = objDist.getRefObj( );
int value = i.getVal();
i.setVal(40);
```

```
ObjetIntf o = new ObjetImpl();
o.setVal(20);
ObjetIntf getRefObj( ) {
   return o;
}
```

Passage de paramètres – referential integrity

RMI Specification section 2.6.3

- « If two references to an object are passed from one JVM to another JVM in parameters (or in the return value) in a single remote method call and those references refer to the same object in the sending JVM, those references will refer to a single copy of the object in the receiving JVM. »
- « More generally stated: within a single remote method call, the RMI system maintains referential integrity among the objects passed as parameters or as a return value in the call. »

Règles de conception pour le modèle distribué

- définition d'interfaces pour les objets distants
  - exception *RemoteException*
- implémentation des objets distants
  - étendre la classe *UnicastRemoteObject*
  - constructeurs qui lèvent l'exception *RemoteException*
  - implémentation des services de l'interface
- client/serveur : gérer les exceptions distantes
- paramètres : type et sémantique

Contraintes de conception pour le modèle distribué

- pas de gestion de la partie statique d'une classe
- pas de publication d'objets dans un serveur de noms distant
- appel distant bloquant

Modèles d'objets JVM locale/JVM distribuée - ressemblances

- la référence d'un objet distant peut être passée comme argument ou retournée comme résultat d'une invocation de méthode (locale ou distante)
- une référence à un objet distant peut être convertie vers toute interface implémentée par la classe de l'objet
- l'opérateur *instanceof* peut être utilisé pour vérifier les interfaces distantes supportées par l'objet distant

Modèles d'objets JVM locale/JVM distribuée - différences

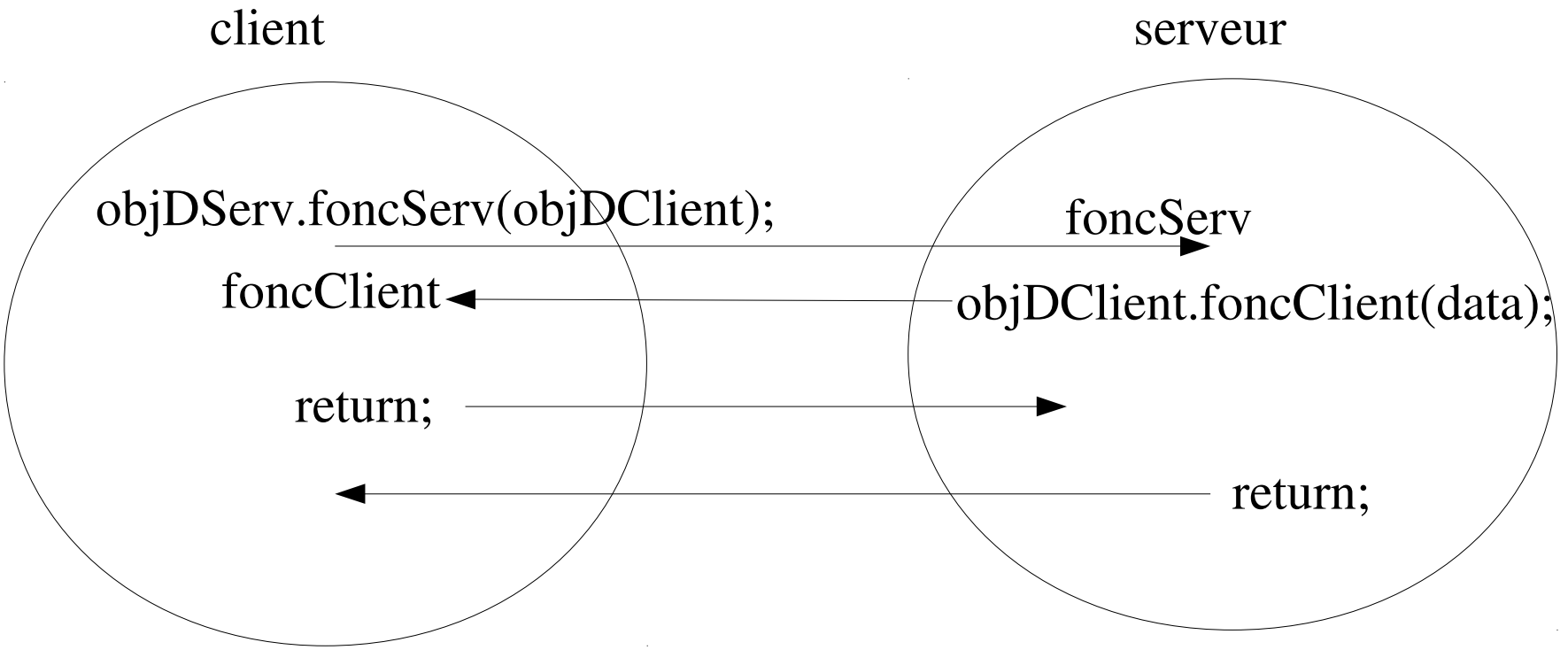
- les clients manipulent seulement les interfaces des objets distants, jamais leurs implémentations
- les paramètres/résultats – objets locaux – d'un appel de méthode distante sont passés par copie, et non par référence
- les objets distants sont passés par référence, pas par copie de l'implémentation distante
- le client doit traiter des exceptions supplémentaires (pour la recherche d'une référence d'objet distant ou l'appel de méthode distante)
- certaines méthodes de la classe *Object* sont redéfinies pour les objets distants

Déploiement

- localisation des différents bytecodes (fichiers .class)
- sur un site
  - chargement des classes dans le serveur de noms
  - CLASSPATH
- semi-local (partage NFS) ou distant
  - (local) différenciation des répertoires
    - client
    - serveur

→ *téléchargement dynamique* (pour le serveur de noms ou pour le client)

Principe de callback



objDServ = objet distant chez le serveur
objDClient = objet distant chez le client

RMI + thread (1)

- serveurs RMI multi-threadés
  - RMI peut utiliser plusieurs threads pour l'invocation des méthodes distantes
  - protéger les attributs et autres ressources partagés (*synchronized*)
  - pool de threads en attente
  - thread activé à la réception d'une requête
  - affectation transparente
- synchronisation
  - code thread-safe ?
  - enchaînement d'appels : libération des threads / des appelants

RMI + thread (2)

- libération du thread d'appel (débloquer l'appelant)
  - création d'un nouveau thread
  - retour
- exemple anneau
  - appel consomme un thread : en attente
  - appel suivant : ne libère pas le thread
  - boucle : ré-appelle le même objet = consomme un nouveau thread