

Programmation Réseau

# RMI



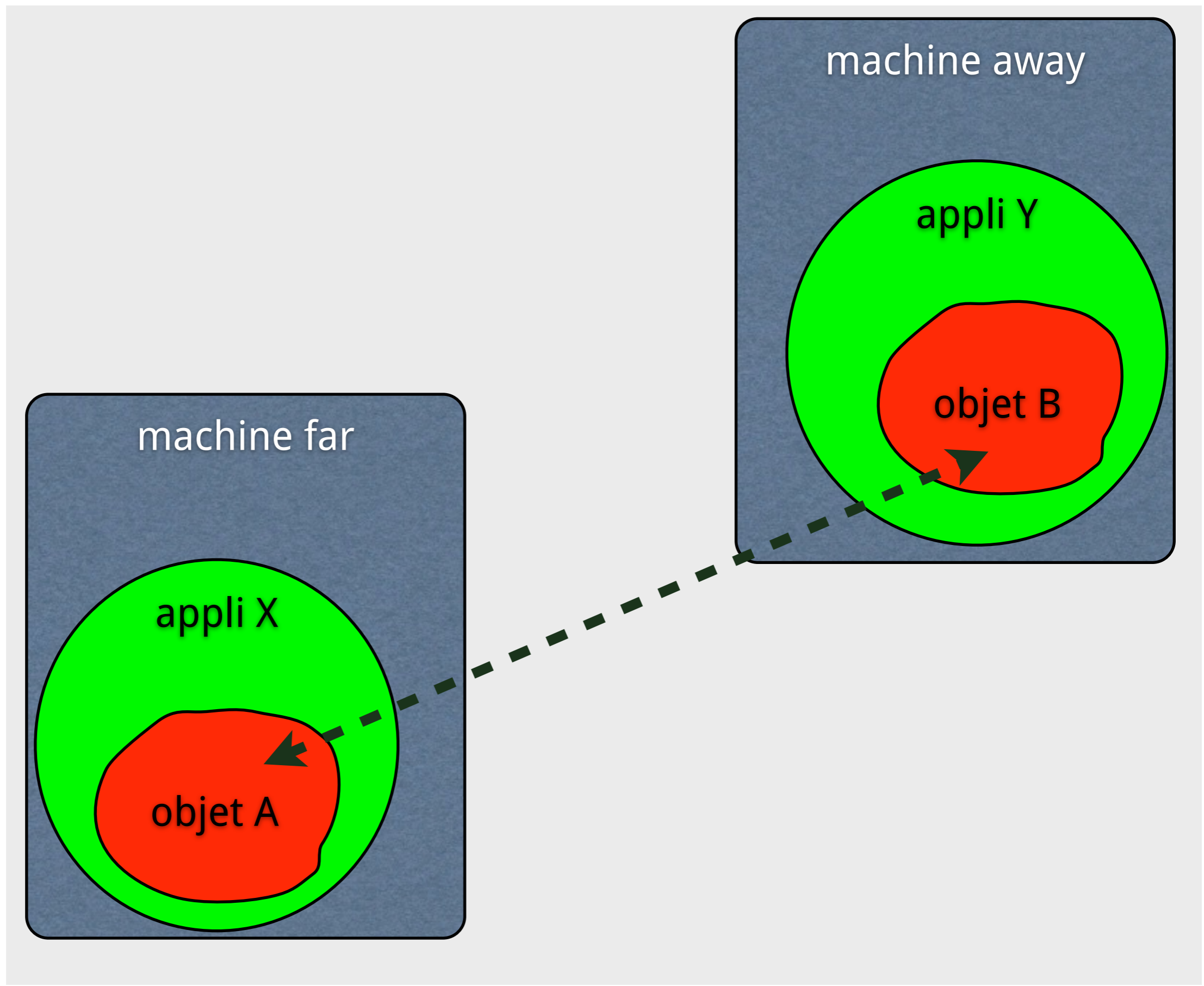
Jean-Baptiste.Yunes@univ-paris-diderot.fr  
armand@informatique.univ-paris-diderot.fr

UFR Informatique

2014

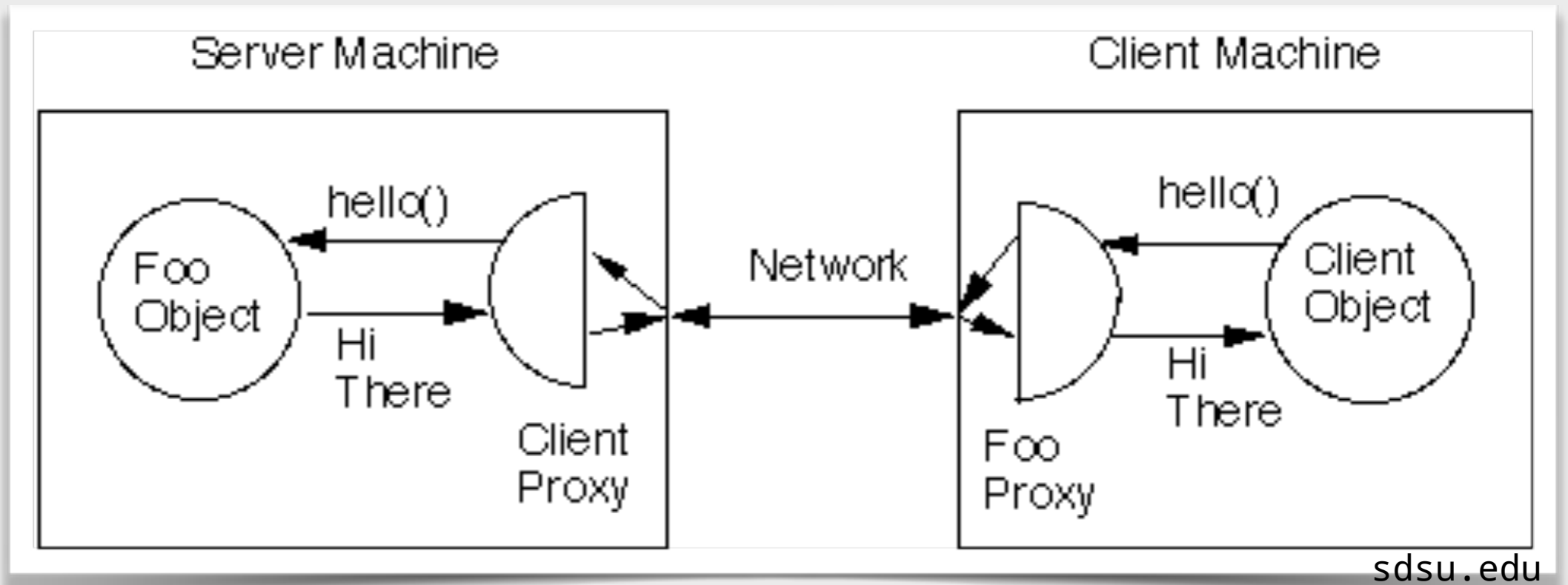
# Les RMI de Java

- Les applications RMI sont des applications bâties sur le modèle objet de Java et dans lesquelles les objets sont répartis dans différents processus (en général sur différentes machines)...
- on comprend donc :
  - l'usage fait du réseau pour communiquer entre objets
  - la difficulté mais aussi l'intérêt de faciliter les appels de méthodes



- L'idée est de rendre **transparente** la manipulation d'objets distants
- Un appel de méthode sur un objet distant doit être **syntactiquement** le même qu'un appel de méthode sur un objet local
- Idée : masquer (au programmeur) les communications nécessaires dans un objet :
  - dont l'interface est exactement celle de l'objet distant
  - qui délègue tous les appels, à travers le réseau, à l'objet distant...

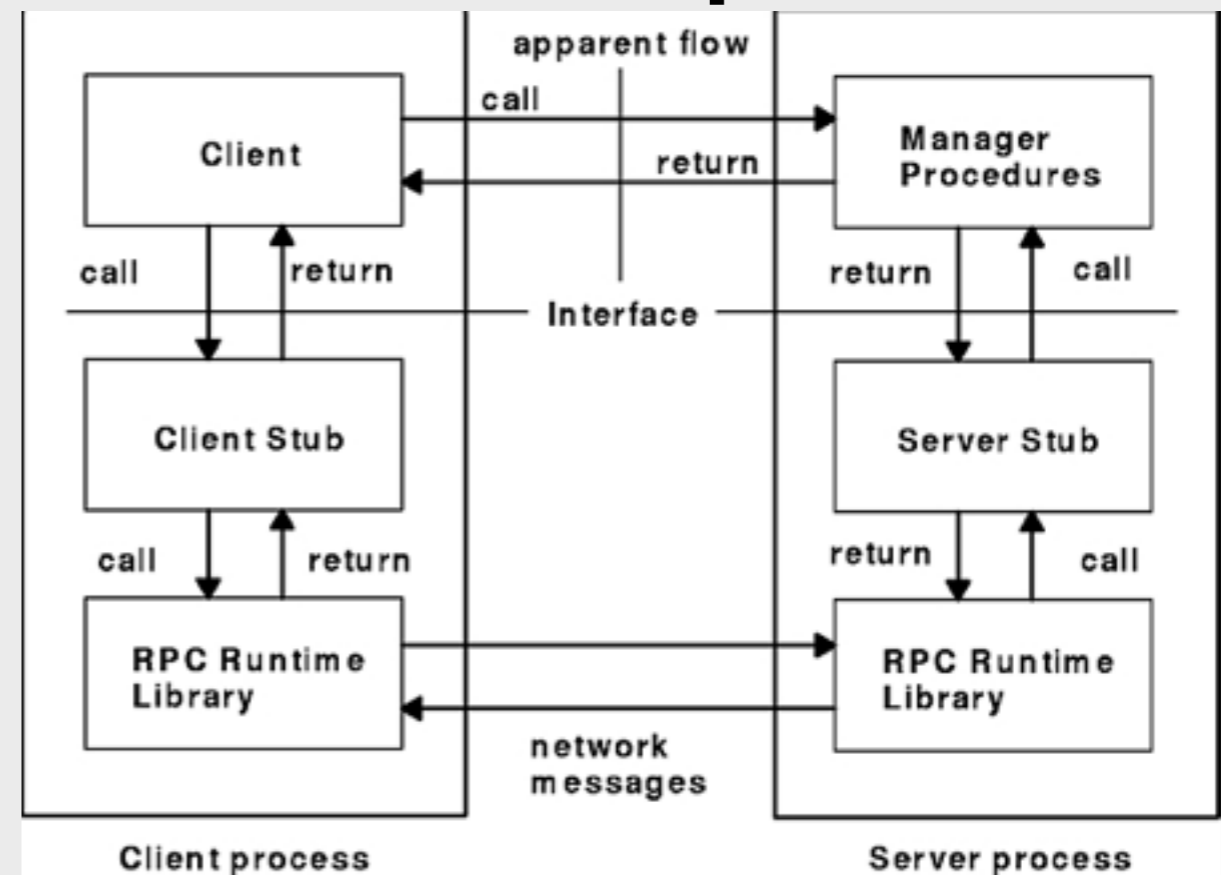
- Le schéma général est alors le suivant :



- Ce mécanisme repose sur une technique bien connue de délégation :
- Le design pattern **proxy**



- Pour les RMI, la terminologie est différente (mais le principe reste le même) et s'inspire de celle des RPC :
- le proxy côté client s'appelle un **talon** (stub)
- le proxy côté serveur est un **squelette** (skeleton)



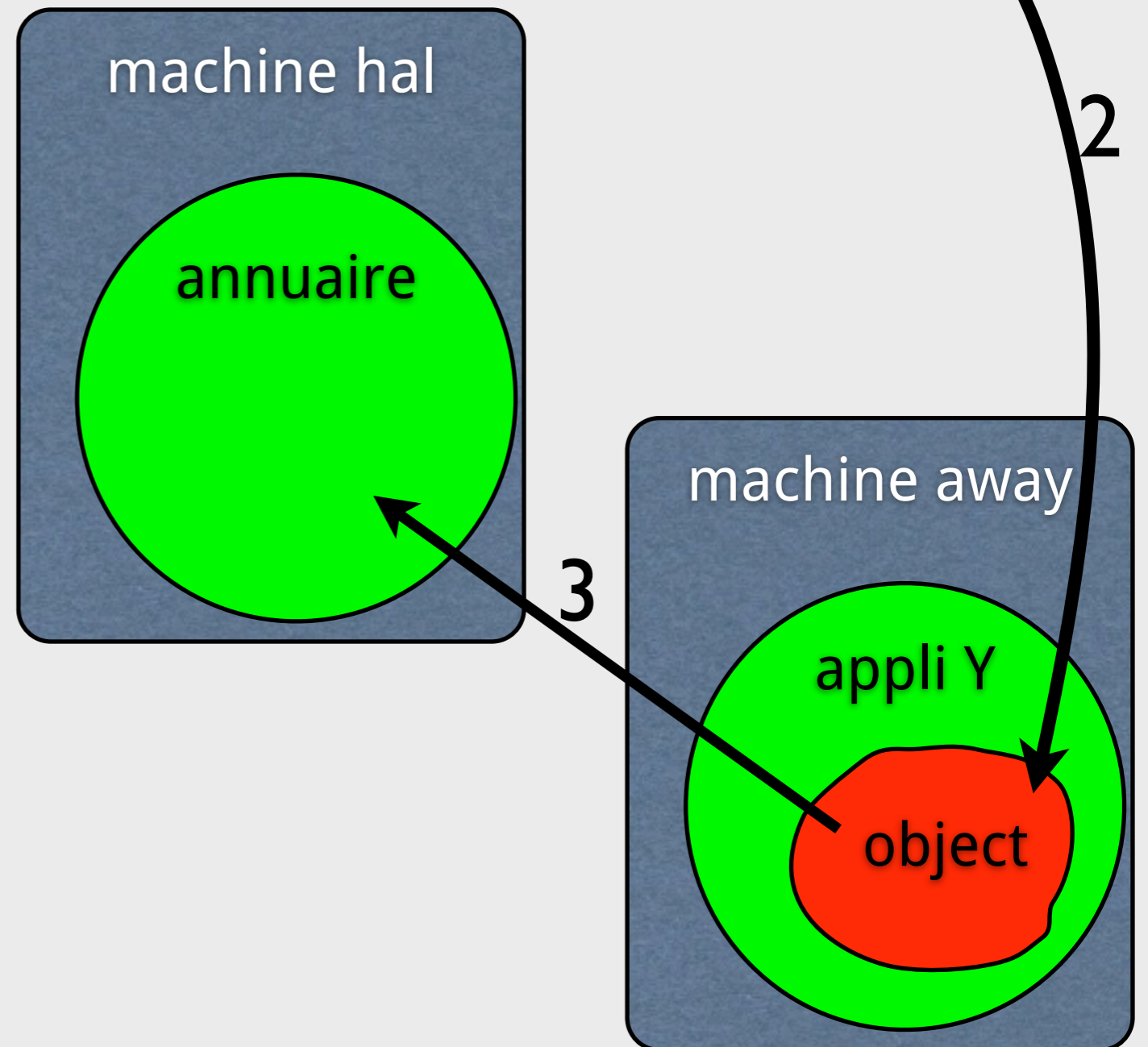
<http://www.javvin.com/protocolRPC.html>

Remote Procedure Call Flow

- Bien entendu de nombreux problèmes sont à résoudre :
- Comment créer les talons et squelettes ?
- Comment localiser un objet distant ?
- ...

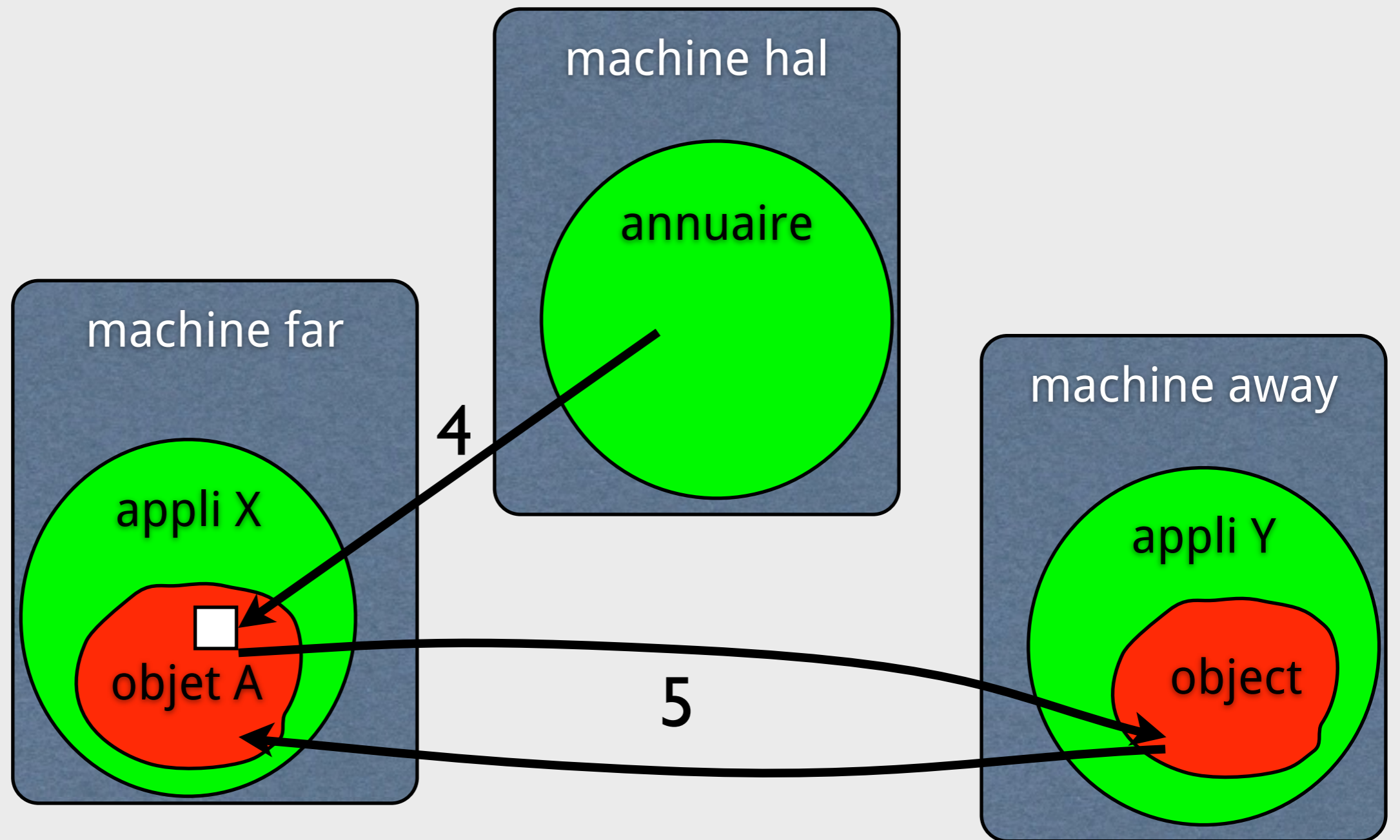
- Schéma général

1. définir une interface pour l'objet
2. créer un objet qui l'implémente
3. le nommer et le faire apparaître dans un annuaire





- Schéma général
  4. récupérer une référence sur l'objet
  5. invoquer une méthode sur l'objet via la référence



- Tout d'abord, côté serveur :
  - Tout objet qui désire être exposé à travers les RMI doit :
    - implémenter une interface qui elle-même doit :
      - spécialiser l'interface `java.rmi.Remote`
      - contenir des méthodes dont la signature contient une clause `throws` faisant apparaître l'exception `java.rmi.RemoteException`
      - dont tous les paramètres ou valeurs de retour doivent être sérialisables, i.e. implémenter l'interface `java.io.Serializable`
    - spécialiser la classe `java.rmi.server.UnicastRemoteObject`

```
// L'interface qui sera exposée
```

```
import java.rmi.*;
```

```
import java.io.*;
```

```
public interface ODI extends Remote {
```

```
    public int getRandom() throws RemoteException;
```

```
    public int getCalls() throws RemoteException;
```

```
}
```

```
// Une implémentation de l'interface...
import java.rmi.*;
import java.rmi.server.*;
import java.io.*;
import java.util.*;
public class OD extends UnicastRemoteObject implements ODI {
    private Random alea;
    private int calls;
    public OD() throws RemoteException {
        log("ctor");
        alea = new Random();
        calls = 0;
    }
    public int getRandom() throws RemoteException {
        log("getRandom() "+this);
        calls++;
        return alea.nextInt();
    }
    public int getCalls() throws RemoteException {
        log("getCalls() "+this);
        return calls;
    }
}
```

- Pour que l'objet soit atteignable par un client :
- il faut l'enregistrer dans un annuaire des objets exposés
- cela suppose qu'un annuaire soit disponible
  - un tel annuaire s'appelle un **registre RMI** (RMI registry)
  - le service réseau correspondant est fourni par défaut avec l'environnement Java; la commande s'appelle `rmiregistry`

- il faut lancer un annuaire
- attention, car l'annuaire peut-être conduit à charger les classes nécessaires...

A screenshot of a terminal window titled "Terminal — rmiregistry — 80x24". The terminal shows the command `rmiregistry` being executed in a directory `[Ciboulette:Programmation reseau/sources/RMI]` by a user named `yunes`. The terminal background is blue, and the text is white. The window has standard macOS window controls (red, yellow, green buttons) in the top-left corner and a close button in the top-right corner.

```
Terminal — rmiregistry — 80x24
[Ciboulette:Programmation reseau/sources/RMI] yunes% rmiregistry
```



- L'enregistrement d'un objet peut simplement s'effectuer à l'aide de la méthode statique :

```
void bind(String nom, Remote o);
```

de la classe

```
java.rmi.Naming
```

- où :
  - nom est de la forme //machine:port/id (machine et port sont optionnels)
  - o est l'objet à exposer...

- ou alors à l'aide en retrouvant un annuaire via :

```
LocateRegistry.getRegistry(...)
```

- puis en utilisant :

```
void bind(String nom, Remote o);
```

de la classe

```
java.rmi.registry.Registry
```

- Attention, `bind()` peut échouer si un objet a déjà été enregistré sous ce nom,
- on peut alors utiliser `rebind()`

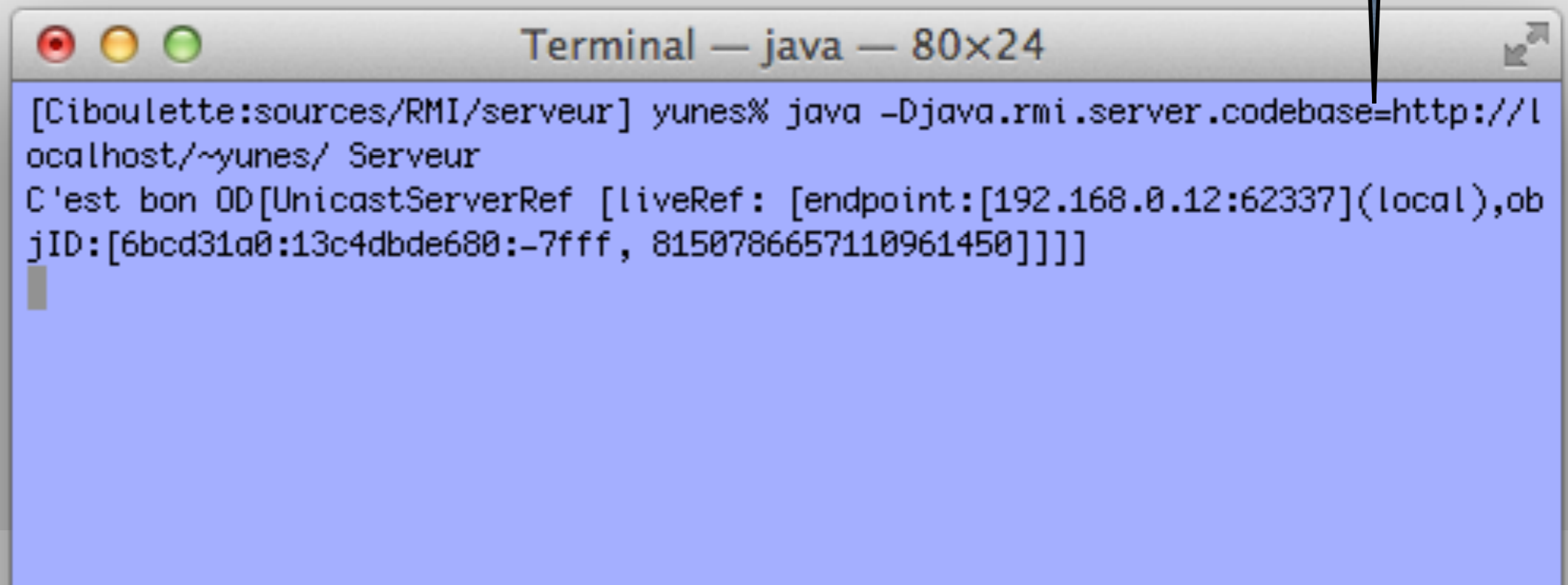
```
// Le serveur qui expose l'objet
import java.rmi.*;
import java.io.*;

public class Serveur {
    public static void main(String []args) {
    try {
        OD o = new OD();
        Naming.bind("rmi://localhost/od",o);
        System.out.println("C'est bon "+o);
    } catch(Exception e) {
        System.out.println("ERREUR");
        e.printStackTrace();
    }
    }
}
```

URL de l'annuaire  
en pratique jamais rien d'autre que  
localhost

- il faut lancer le serveur

d'où les classes utiles  
seront-elles obtenues ?

A terminal window titled "Terminal — java — 80x24" with a blue background. The terminal shows the execution of a Java command to start an RMI server. The output indicates that the server is running and provides details about the UnicastServerRef, including the endpoint and object ID.

```
[Ciboulette:sources/RMI/serveur] yunes% java -Djava.rmi.server.codebase=http://localhost/~yunes/ Serveur
C'est bon OD[UnicastServerRef [liveRef: [endpoint:[192.168.0.12:62337](local),objID:[6bcd31a0:13c4dbde680:-7fff, 8150786657110961450]]]]
```

- Pour atteindre l'objet, côté client, il suffit :
  - d'interroger le registre via la méthode statique  
`(Object)lookup(String nom);`  
de la classe `Naming`
  - d'utiliser l'objet ordinairement pour y appeler des méthodes
  - attention, il est plus prudent de ne considérer que le type de l'interface pour le type de l'objet renvoyé



```
// Le client...
import java.rmi.*;
import java.io.*;

public class Client {
    public static void main(String []args) {
        try {
            ODI o = (ODI)Naming.lookup("//localhost/od");
            System.out.println("OD="+o);
            for (int i=0; i<10; i++) {
                System.out.println("Rand(i) = "+o.getRandom());
            }
            System.out.println("Calls : "+o.getCalls());
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

quel est l'objet que  
l'on souhaite ?

```
Terminal — tcsh — 80x24
[Ciboulette:sources/RMI/client] yunes% java Client
OD=Proxy[ODI,RemoteObjectInvocationHandler[UnicastRef [liveRef: [endpoint:[192.1
68.0.12:62399](remote),objID:[-63c4b233:13c4dc2450c:-7fff, -616177607043352892]
]]]
Rand(i) = 1002368058
Rand(i) = 2142141056
Rand(i) = -1616353082
Rand(i) = 1222364157
Rand(i) = 1053606475
Rand(i) = -1129491520
Rand(i) = -686251917
Rand(i) = -146429651
Rand(i) = 667518928
Rand(i) = -569169992
Calls : 10
[Ciboulette:sources/RMI/client] yunes%
```

- Rappelons que l'univers Java est protégé par différents mécanismes de **sécurité**
- les RMIs sont aussi sous le couvert d'une politique de sécurité, deux façons de les paramétrer :
  - installer en interne un `SecurityManager` adéquat
  - surcharger en externe le paramétrage de la politique de sécurité, via la propriété `java.security.policy`

- pour le paramétrage externe :
- lancer la JVM en spécifiant quel fichier contient les paramètres de sécurité avec la commande

```
java -Djava.security.policy=fichier ...
```

- créer un fichier spécifiant les valeurs des paramètres, comme (ici la politique la moins restrictive) :

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

- Il est possible de créer un annuaire depuis le serveur lui-même

```
// Un serveur-annuaire...

import java.rmi.*;
import java.rmi.registry.*;
import java.io.*;

public class ServeurAutoRegistry {
    public static void main(String []args) {
        try {
            Registry a = LocateRegistry.createRegistry(
                Registry.REGISTRY_PORT);

            OD o = new OD();
            a.rebind("od",o);
            System.out.println("C'est bon "+o);
        } catch(Exception e) {
            System.out.println("ERREUR");
            e.printStackTrace();
        }
    }
}
```



- On notera :
  - que l'enregistrement d'un `UnicastRemoteObject` crée un `Thread` côté serveur; ce `Thread` est dédié à la gestion de la communication côté serveur
  - l'utilisation intensive des exceptions (beaucoup de communications sont cachées et les erreurs doivent être traitées)
  - que des problèmes de concurrence peuvent apparaître
  - que des problèmes de fiabilité peuvent apparaître (un client - un serveur peut disparaître brutalement)
  - que le garbage collector est particulier (dgc + interface `Unreferenced`)

# L'activation d'objets

- Dans le modèle des objets distribués précédents les objets préexistent côté serveur
- Si l'on souhaite obtenir quelque chose qui se rapproche plus du modèle des services (à la mode du super-démon d'Unix) on peut utiliser l'activation à la demande d'objets distants
- Les objets ne sont instanciés qu'à réception d'une requête
- Il s'agit du mécanisme d'activation d'objets

- il est important de savoir que
  - les objets activables sont regroupés par **groupe d'activation**
  - qu'une JVM prendra en charge un groupe
  - qu'un démon se chargera d'activer une JVM lorsque nécessaire

- Du côté serveur, un objet activable doit :
  - étendre la classe `Remote`
  - étendre la classe `java.rmi.activation.Activatable`
  - implémenter un constructeur à deux arguments de type
    - `java.rmi.activation.ActivationID`
    - `java.rmi.MarshalledObject<T>`

- Ensuite il est nécessaire de créer un programme permettant la mise en place qui consiste à :
  - positionner une politique de sécurité adaptée pour la JVM qui sera lancée
  - création d'un groupe d'activation ou rejoindre un groupe d'activation (une JVM par groupe d'activation)
  - déclaration après du système de nommage

- La commande `rmid` correspond au démon qui prend en charge les (groupes d') objets activables
- cette commande doit être utilisée en partenariat avec le registre RMI `rmiregistry`



```
// Interface de l'objet activable

import java.rmi.*;

public interface Hello
    extends Remote {
    String hello(String name)
        throws RemoteException;
}
```

```
// Implémentation de l'interface
import java.rmi.*;
import java.rmi.activation.*;
import java.io.*;

class ConcreteHello extends Activatable implements Hello {
    private String name;
    public ConcreteHello(ActivationID id,
                        MarshalledObject<String> o)
        throws RemoteException, IOException,
        ClassNotFoundException {

        super(id,0);
        name = o.get();
        System.out.println("ctor ConcreteHello("+id+"
name="+name);
    }
    public String hello(String name) throws RemoteException {
        System.out.println("hello called");
        try { Thread.sleep(20000); } catch(Exception e) { }
        return "I am "+this.name+", I say hello "+name;
    }
}
```

## Création du groupe...

```
public class Startup {
    public static void main(String[] args)
        throws Exception {
        // Politique de sécurité de la JVM activée
        Properties props = new Properties();
        props.put("java.security.policy", "grouppolicy");

        // paramètres de la JVM
        ActivationGroupDesc.CommandEnvironment ace = null;

        // Création d'un descripteur du groupe
        ActivationGroupDesc group =
            new ActivationGroupDesc(props, ace);

        // Enregistrement du groupe auprès du rmid
        ActivationSystem as = ActivationGroup.getSystem();
        ActivationGroupID agi = as.registerGroup(group);
    };
    ...
}
```

```
...  
    // objet qui sera passé en paramètre du constructeur  
    MarshalledObject<String> data =  
        new MarshalledObject<String>("serveur");  
  
    // Création du descripteur de l'objet activable  
    ActivationDesc desc =  
        new ActivationDesc(agi, "ConcreteHello",  
            "file:/Users/yunes/Desktop/activation/", data);  
  
    // Enregistrement de l'objet activable auprès du rmid  
    // et Récupération de la souche  
    Hello obj = (Hello)Activatable.register(desc);  
  
    // Enregistrement de la souche auprès du rmiregistry  
    Naming.rebind("HelloServer", obj);  
}  
}
```

```
// Un client invoque un objet activable...
import java.rmi.*;
import java.rmi.registry.*;
import java.io.*;

public class Client {
    public static void main(String []args) {
        try {
            Registry registry =
                LocateRegistry.getRegistry("localhost");
            System.out.println("Registry is "+registry);
            Hello o = (Hello) registry.lookup("HelloServer");
            System.out.println("Hello="+o);
            System.out.println(o.hello(args[0]));
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```