4.6 Ramasse miettes réparti

Reference counting without detection of dead cycles.

4.6.1 Principles

Remainder:
JRMP: Java Remote Method Protocol, more or less RMI runtime
Allows to define an interoperable protocol for the entire RMI messages,

DGC: principles

- Count the references (stubs) that point to a RMI remote object
- A stub is only valid for a limited time: the lease

A DGC collaborate (uses) the standard JVM local GC.

Lease: the amount of time a Stub is valid for, after that period, it has to be “renewed”.

4.6.2 Main actions

JRMP keeps track of all references to a stub in a JVM
+ Details of the Stubs to Activatable

JRMP keeps also a counter for each Remote Object (RO) in a JVM:

the number of remote stub towards that RO.

Upon the arrival of a new stub in a JVM:

JRMP send an increment message to the target JVM, for that RO.

When the stub is no longer referenced by a JVM:

JRMP send a decrement message to the target JVM, for that RO

When a RO counter reaches 0:

JRMP turns its reference to the RO into a weak reference

--> • Call to unreferenced

--> • it will be collected by the local GC when no other local references exist

---> PICTURE

4.6.3 Leasing: Time-To-Live stubs

Each stub is valid for a parameterized given period:

the lease

After a time inferior to the lease a stub tries to renew the lease:

sending a message to the RO site

If no renewal messages are received by a RO for a period superior to the lease:

then the lease counter can be set to 0

By default, the lease is set to 10 minutes, and stubs try to renew themselves:

after 5minutes.

The leases and the lease counters seem to be managed per JVM:

using the VMID.

Note that the RMIregistry is a RO itself that stores references to the ROs it registers in its hashtable. As a consequence, it counts as a reference, and any RO registered (bind) in the RMIregistry is not GCed.

4.6.4 Time-to-live stubs vs. counters

Questions: do we really need counters ?

Since the time-to-live stubs will eventually disappeared,

stop sending renewal messages,

the lease counter will be set to 0,

and the RO will be subject to GC.

So, why do we need counters ?

Answer: Reactivity

Without counter, a RO will be identified as “unreferenced” after the lease period (e.g. 10 minutes),

With counter, only after the time for the last decrement messages to be sent (a few ms).

4.6.5 API with RMI DGC

a) Property: java.rmi.dgc.leaseValue

Property: java.rmi.dgc.leaseValue

The value of this property represents the lease duration (in milliseconds) granted to other JVMs that hold remote references to objects which have been exported by this JVM.

Clients usually renew a lease when it is 50% expired, so a very short value will increase network traffic and risk late renewals in exchange for reduced latency in calls to Unreferenced.unreferenced.

The default value of this property is 600000 milliseconds (10 minutes).

Example:

-Djava.rmi.dgc.leaseValue=10000

Set the lease value to 10 seconds.

b) Listener on “no more remote reference”

There is a listener and a method that can be called when a RO is no longer referenced:

Interface java.rmi.Unreferenced method unreferenced ()

The method is called when RO counter reaches 0

c) Stopping a Remote Object

There is a method that can be called to explicitly “stop” a RO:

Packages java.rmi.server
Class UnicastRemoteObject

method public static boolean
unexportObject(Remote obj, boolean force)
throws NoSuchObjectException

Actually, “stop” means not callable from outside.

Upon a call to an un-exported RO, the exception

java.rmi.NoSuchObjectException: no such object in table

is thrown

CHAPITRE 5 Objets Réparties et Persistance

Dans un premier temps: les objets RMI Activables

5.1 Objets activables

Les objets RMI Activable permettent de faire des objets distants qui sont persistants (stockés sur disque), et qui sont activés automatiquement en cas d'appel.

En plus de l'écriture du client, il faut écrire deux classes pour faire un système avec activation:

- la classe activatable elle-même
- une classe de set-up

Quelques principes:

--> • les objets activables sont regroupés en un “GROUPE D’ACTIVATION”:

Tous les objets qui appartiennent à un groupe d’activation sont activés dans la même Machine Virtuelle Java

--> • des “DESCRIPTEURS” sont utilisés afin de représenter les entités:

- descripteur d’un groupe d’activation
- descripteur d’objet activable

Ils donnent toutes les informations nécessaires pour manipuler et reconstruire les entités qu’ils représentent, en particulier dans la Machine Virtuelle démon (rmid).

5.1.1 Class java.rmi.activation.Activatable

Les objets activable doivent étendre la classe

java.rmi.activation.Activatable

```
import java.rmi.*;
import java.rmi.activation.*;
public class ActivatableImplementation extends Activatable
      implements examples.activation.MyRemoteInterface {
```

Le code du côté client ne change pas

Décision du côté serveur uniquement, au niveau implémentation,

A priori sans implications du côté client.

Classes et interfaces RMI
pour les objets activables:

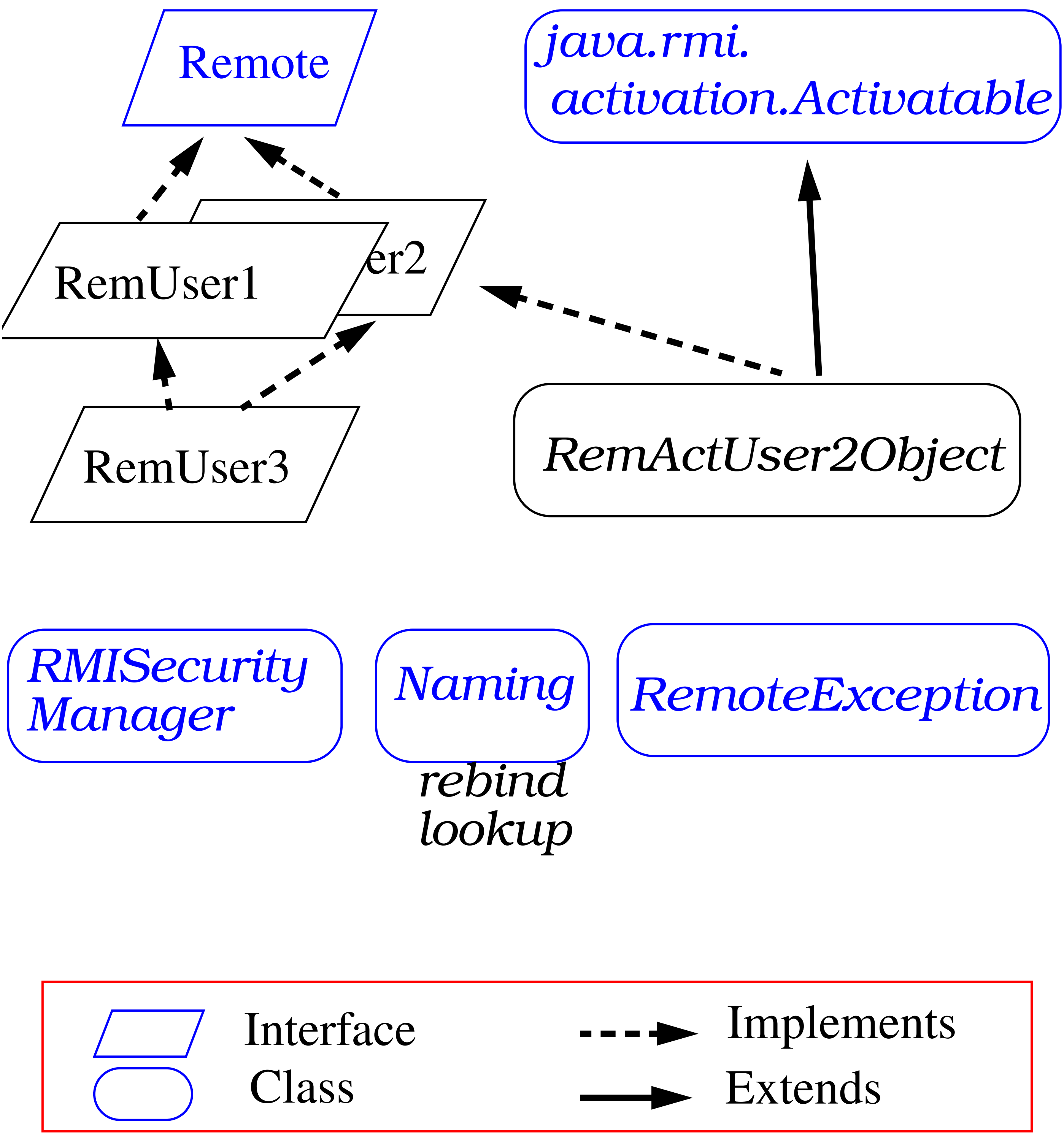

FIGURE 22 classes et interfaces RMI: objets activables

5.1.2 Implémentation d'une classe Activable

a) Importation

Les imports suivants sont indispensables:

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

b) Etendre Activatable

```
public class ActivatableImplementation extends
                                              Activatable
    implements
        examples.activation.MyRemoteInterface {
         ...
```

c) Constructeur spécial

Constructeur spécial à deux arguments (id)

```
public ActivatableImplementation(ActivationID id,
                                 MarshalledObject data)
    throws RemoteException {
    // Register the object with the activation system
    // then export it on an anonymous port
    super(id, 0);
 }
```

Ce constructeur sera utilisé par l'implémentation de la bibliothèque d'activation de Sun, pas par vous. Le constructeur sera appelé pour “ré-activer“ un objet sérialisé (**MarshalledObject data**)

d) Remote interface

Implémenter les routine déclarées dans l'interface distante.

```
public Object callMeRemotely() throws
                                        RemoteException {
   ...
    return "Success";
 }
```

A bien noter:

On n'étend plus le classe

java.rmi.server.UnicastRemoteObject

mais la classe

java.rmi.activation.Activatable

Cela permet de passer simplement d'une classe remote à une classe Remote-Activable.

D'autre part, il faut bien ajouter le constructeur à deux arguments.

5.1.3 Classe de set-up

Le rôle de la classe de setup est de :

créer toutes les informations nécessaires pour ultérieurement “activer” l’objet distant sur un appel,
sans pour autant créer effectivement l’objet distant tout de suite.

Le modèle d'activation répond à un certain nombre de caractéristiques:

- un objet activable appartient à un “Groupe d'Activation“
- Celui-ci est responsable :
  - de la création des OAs de son groupe
  - de l'activation: disque --> VM
  - de la désactivation: VM --> disque
- Il existe une VM par groupe d'activation
- C'est rmid qui va créer ou contacter cette VM (voir plus loin schéma global avec rmid).

5.1.4 Implémentation d'une classe "setup"

a) Importation

Les imports suivants sont indispensables:

```
import java.rmi.*;
import java.rmi.activation.*
import java.util.Properties;
```

b) SecurityManager

```
System.setSecurityManager(new
                                        RMISecurityManager());
```

c) Créer un groupe d'activation (instance)

Chaque objet activable appartient à un “Groupe d'Activation“, permet de retrouver la JVM où se trouve un objet activable, ou de la lancer,

Il vit dans la VM rmid

```
Properties props = new Properties();     // Specification d’un policy file
    props.put("java.security.policy", "/home/rmi_tutorial/activation/policy");

  ActivationGroupDesc.CommandEnvironment ace = null;
  ActivationGroupDesc exampleGroup = new ActivationGroupDesc(props, ace);

  // Once the ActivationGroupDesc has been created, register it
  // with the activation system to obtain its ID
  //
  ActivationGroupID agi =
    ActivationGroup.getSystem().registerGroup(exampleGroup);
```

d) Créer un descripteur de la classe activable

Un descripteur d'une classe activable permet de représenter la classe dans la VM rmid, avec toute les informations nécessaires à la création d'une instance (URL du code de la classe)

```
// The "location" String specifies a URL from where the
//class definition will come when this object is requested
// (activated).
// Don't forget the trailing slash at the end of the URL
// or your classes won't be found.
//
String location = "file:/home/rmi_tutorial/activation/";
```

location est utilisé pour identifier uniquement cette classe.

```
// Create the rest of the parameters that will be passed to
// the ActivationDesc constructor
//
MarshalledObject data = null;

// The location argument to the ActivationDesc constructor will be used
// to uniquely identify this class; it's location is relative to the
// URL-formatted String, location.
//
ActivationDesc desc = new ActivationDesc
   (agi, "examples.activation.ActivatableImplementation", location, data);
```

Ceci est en fait : une classe (ActivatableImplementation) ... qui pourra être activée (crée, ou chargée en mémoire).

Il se trouve qu'il n'y a pas de données sur le disque qui le représente (pour l'instant) car son constructeur n'a pas de paramètre.

Question:

lien groupe act.<--> Desc. classe activable ?

Les classes activables sont groupée dans un groupe d'activation (agi).

e) Enregistre la classe dans le rmid

A partir du descripteur de classe activable créé à l'étape précédente,

on fait un enregistrement dans le rmid de la machine (on rend une classe "activable")

Note: on n'a pas de création
(pas de new ActivatableImplementation ())

```
MyRemoteInterface mri = (MyRemoteInterface)Activatable.register(desc);
```

Cette opération returne un stub !!

f) Enregistre le stub dans le rmiregistry

Utilise mri retourné par l'enregistrement du descripteur.

Enregistre le stub dans le rmiregistry, associé à une chaîne de caractère

```
Naming.rebind("ActivatableImplementation", mri);
```

Attention:

Dans le cadre rmi sans activation, on enregistre dans le rmiregistry (rebind) l'objet accessible à distance lui-même.

Ici, on enregistre quelque chose (mri) que l'on VOIT comme une interface Remote.

En fait, RMI non activable fait la même chose de façon cachée.

Mais en fait, on ne sait pas exactement ce que l'on enregistre, cela est caché par l'implémentation:

c'est en fait qq chose qui se trouve dans le rmid

Une version modifiée des stubs RMI de façon à contacter rmid pour activer l'objet si nécessaire.

Après tout ça, le programme de setup peut se terminer:

- un objet persitent (descripteur) est créé
- il appartient à un groupe
- il est enregistré dans le rmiregistry

Note: Comment faire si on utilise un **rmiregistry** sur un autre port que celui par défaut ?

Lors de l'enregistrement du stub dans le rmiregistry, on ajoute une URL avec numéro de port:

```
Naming.rebind("//clio.unice.fr:2001/ActivatableImplementation", mri);
```

Idem lors du lookup chez le client.

5.1.5 Démon d'activation: rmid

rmid &

Permet d'activer les objets (en fait le groupe d'objet activable) si nécessaire.

C'est lui qui:

- Enregistre les groupes d'objets activables
- Pour chaque OA, garde les descripteurs (fait le lien entre le descripteur et le stub dans le **rmiregistry**

- Permet lors d'un appel à un objet activable:
  - de retrouver la VM où il se trouve
  - de lancer la VM si elle est arrêtée
  - de retrouver l'objet dans la VM si il s'y trouve
  - de charger l'objet dans cette VM s'il avait été sauvegardé sur disque

5.1.6 Schéma global

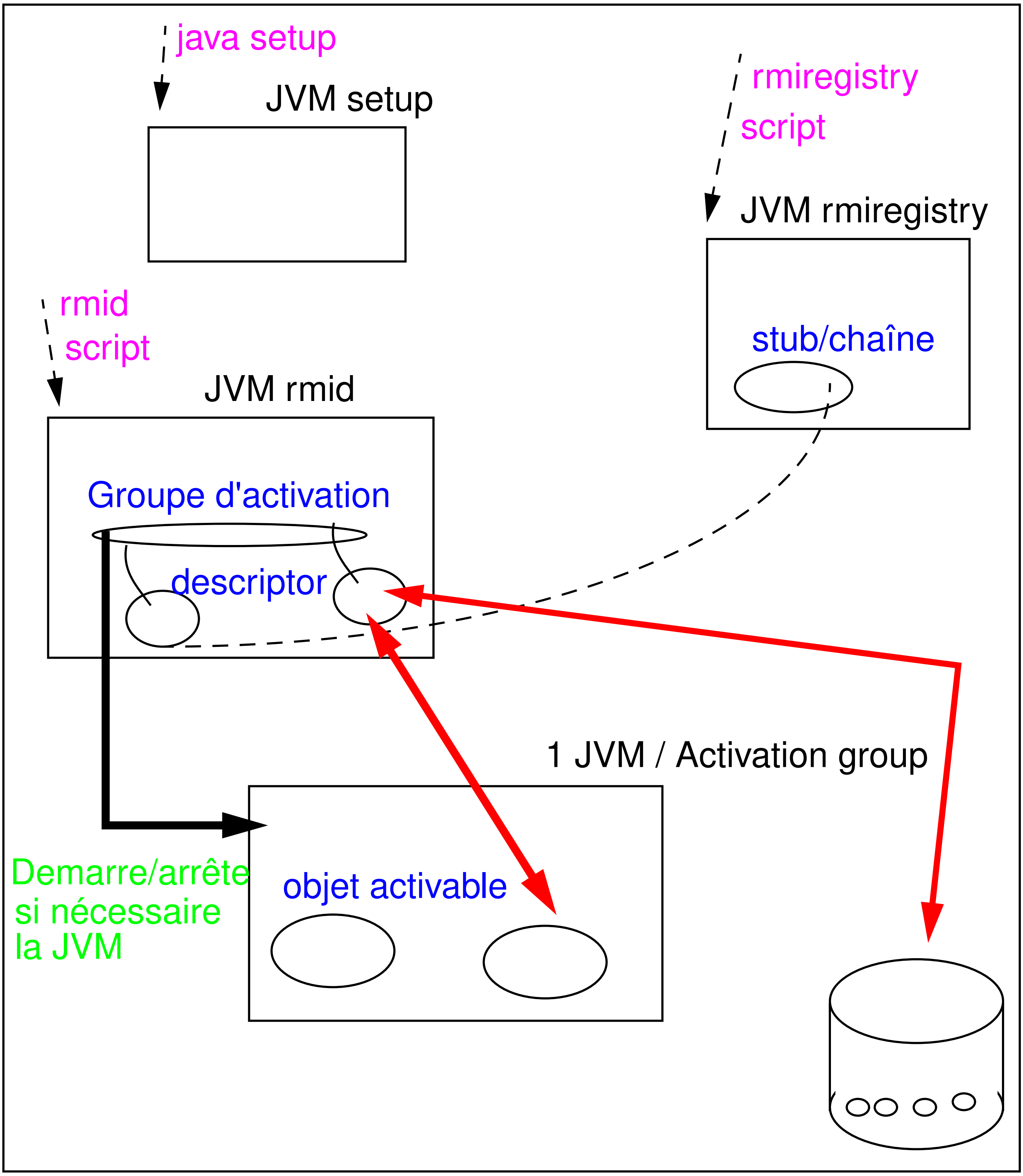

FIGURE 23 Schéma de fonction. des objets activables

5.1.7 Compilation et exécution

Ordre des opérations à exécuter:

1. Compilation des classes et interfaces, implémentation, client, setup

```
% javac -d . MyRemoteInterface.java
% javac -d . ActivatableImplementation.java
% javac -d . Client.java
% javac -d . Setup.java
```

2. rmic sur la classe d'implémentation

```
% rmic -d . examples.activation.ActivatableImplementation
```

3. Lancer le rmiregistry

```
% rmiregistry &
```

4. Lancer le rmid

```
% rmid &
```

5. Exécuter le programme setup

```
% java  -Djava.security.policy=/home/rmi_tutorial/activation/policy
          -Djava.rmi.server.codebase=file:/home/rmi_tutorial/activation/
              examples.activation.Setup
```

6. Exécuter le programme client

```
% java  -Djava.security.policy=/home/rmi_tutorial/activation/policy
                                examples.activation.Client clio
```

L'argument “clio“ est le nom de la machine où tourne le serveur (rmiregistry, rmid, où à été exécuté Setup)

5.1.8 Exemple

a) classe Activable

```
package examples.activation;

import java.rmi.*;
import java.rmi.activation.*;

public class ActivatableImplementation extends Activatable
    implements examples.activation.MyRemoteInterface {

  // The constructor for activation and export; this constructor is
  // called by the method ActivationInstantiator.newInstance during
  // activation, to construct the object.
  //
  public ActivatableImplementation(ActivationID id, MarshalledObject data)
    throws RemoteException {

    // Register the object with the activation system
    // then export it on an anonymous port
    //
    super(id, 0);
  }

  // Implement the method declared in MyRemoteInterface
  //
  public Object callMeRemotely() throws RemoteException {

    return "Success";

  }
}
```

b) class Setup

```
package examples.activation;

import java.rmi.*;
import java.rmi.activation.*;
import java.util.Properties;

public class Setup {

   // This class registers information about the ActivatableImplementation
   // class with rmid and the rmiregistry
   //
   public static void main(String[] args) throws Exception {

      System.setSecurityManager(new RMISecurityManager());

      // After JDK1.2Beta4, ActivationGroups are no longer created
      // implicitly as a side-effect of creating or registering the
      // first Activatable object
      //
      // Because of the 1.2 security model, a security policy should
      // be specified for the ActivationGroup VM. The first argument
      // to the Properties put method, inherited from Hashtable, is
      // the key and the second is the value
      //
      Properties props = new Properties();
      props.put("java.security.policy",
         "/home/rmi_tutorial/activation/policy");

      ActivationGroupDesc.CommandEnvironment ace = null;
      ActivationGroupDesc exampleGroup = new ActivationGroupDesc(props,
                                                                  ace);

      // Once the ActivationGroupDesc has been created, register it
      // with the activation system to obtain its ID
      //
      ActivationGroupID agi =
        ActivationGroup.getSystem().registerGroup(exampleGroup);

      // Now explicitly create the group
      //
      ActivationGroup.createGroup(agi, exampleGroup, 0);
```

```
        // The "location" String specifies a URL from where the class
        // definition will come when this object is requested (activated).
        // Don't forget the trailing slash at the end of the URL
        // or your classes won't be found.
        //
        String location = "file:/home/rmi_tutorial/activation/";

        // Create the rest of the parameters that will be passed to
        // the ActivationDesc constructor
        //
        MarshalledObject data = null;

        // The second argument to the ActivationDesc constructor will be used
        // to uniquely identify this class; it's location is relative to the
        // URL-formatted String, location.
        //
        ActivationDesc desc = new ActivationDesc
           ("examples.activation.ActivatableImplementation", location, data);

        // Register with rmid
        //
        MyRemoteInterface mri = (MyRemoteInterface)Activatable.register(desc);
        System.out.println("Got the stub for the ActivatableImplementation");

        // Bind the stub to a name in the registry running on 1099
        //
        Naming.rebind("ActivatableImplementation", mri);
        System.out.println("Exported ActivatableImplementation");

        System.exit(0);
    }
}
```

5.1.9 java.rmi.activation

```
package java.rmi.activation;
    public abstract class Activatable
            extends java.rmi.server.RemoteServer
    {
        protected Activatable(String codebase,
                              java.rmi.MarshalledObject data,
                              boolean restart,
                              int port)
            throws ActivationException, java.rmi.RemoteException;

        protected Activatable(String codebase,
                              java.rmi.MarshalledObject data,
                              boolean restart,
                              int port,
                              RMIClientSocketFactory csf,
                              RMIServerSocketFactory ssf)
            throws ActivationException, java.rmi.RemoteException;

        protected Activatable(ActivationID id, int port)
            throws java.rmi.RemoteException;

        protected Activatable(ActivationID id, int port,
                              RMIClientSocketFactory csf,
                              RMIServerSocketFactory ssf)
            throws java.rmi.RemoteException;

        protected ActivationID getID();

        public static Remote register(ActivationDesc desc)
            throws UnknownGroupException, ActivationException,
                    java.rmi.RemoteException;

        public static boolean inactive(ActivationID id)
            throws UnknownObjectException, ActivationException,
                    java.rmi.RemoteException;

        public static void unregister(ActivationID id)
            throws UnknownObjectException, ActivationException,
                    java.rmi.RemoteException;

        public static ActivationID exportObject(Remote obj,
```

```
                                        String codebase,
                                        MarshalledObject data,
                                        boolean restart,
                                        int port)
        throws ActivationException, java.rmi.RemoteException;

    public static ActivationID exportObject(Remote obj,
                                        String codebase,
                                        MarshalledObject data,
                                        boolean
restart,
                                        int port,
                                        RMIClientSocketFactory csf,
                                        RMIServerSocketFactory ssf)
        throws ActivationException, java.rmi.RemoteException;

    public static Remote exportObject(Remote obj,
                                      ActivationID id,
                                      int port)
        throws java.rmi.RemoteException;

    public static Remote exportObject(Remote obj,
                                      ActivationID id,
                                      int port,
                                      RMIClientSocketFactory csf,
                                      RMIServerSocketFactory ssf)
        throws java.rmi.RemoteException;

    public static boolean unexportObject(Remote obj, boolean force)
        throws java.rmi.NoSuchObjectException;
}
```