Parallélisme

Eric Goubault (et Matthieu Martel)
Commissariat à l'Energie Atomique
Saclay

Reveil de serveur

Problèmes dans les méthodes précédentes:

- Même si les services sont peu souvent utilisés, il faut que les serveurs tournent en permanence,
- les serveurs doivent créer et exporter les objets distants: consommation inutile de mémoire et de CPU,

Depuis JAVA 2 on peut activer à distance un objet distant.

Activation à distance

- Permet d'enregistrer un service RMI sans l'instancier,
- Le service RMI défini par cette méthode est inactif,
- Est réveillé seulement quand un client y fait appel,
- Un processus démon est chargé d'écouter les requêtes et de réveiller les services: `rmid`.

Inscription du service

- A la place du service, une sorte de “proxy” est enregistré auprès du serveur de services RMI (**`rmiregistry`**),
- Contrairement aux serveurs instances de **`UnicastRemoteObject`**, cette “fausse” référence ne s’exécute que pendant un cours instant, pour inscrire le service auprès du **`rmiregistry`**, puis aux moments d’appels au service.

Activation du service

- Quand le client appelle ce service, le **`rmiregistry`** fait appel à cette fausse référence,
- Celle-ci vérifie si elle a un pointeur sur le vrai service,
- Si non, elle fait appel au démon **`rmid`** pour créer une instance du service (prend un certain temps, la première fois),
- Puis transmet l'appel au service nouvellement crée.

En pratique

- Transparence pour le client: le client n'a en rien besoin de savoir comment le service est implémenté (en activation à distance, ou comme serveur permanent).
- La création du service est un peu différente que dans le cas du serveur résident, mais son code reste similaire.

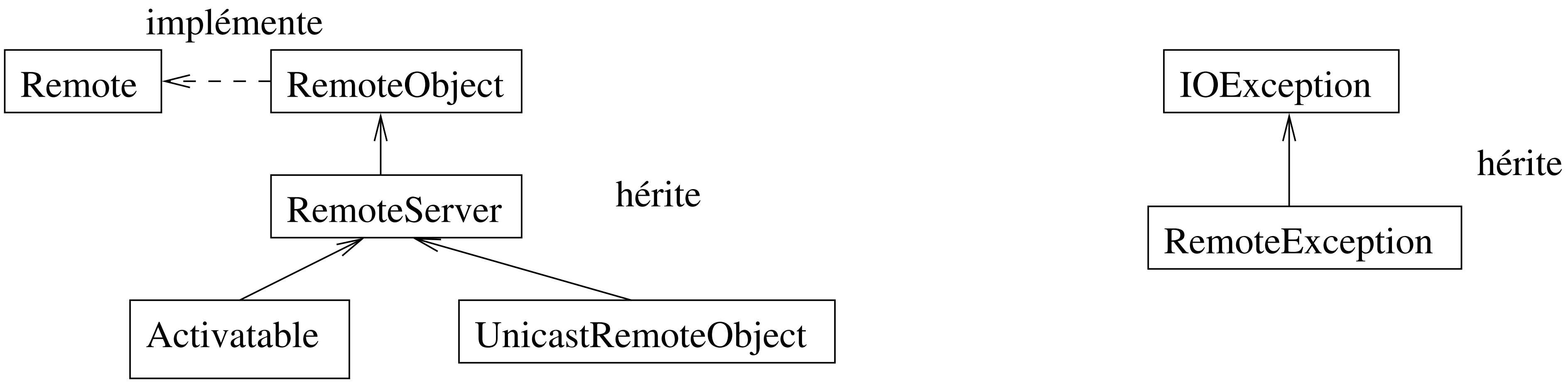
Pour en savoir plus: packages RMI

- `java.rmi` définit l'interface `RemoteInterface`, et les exceptions,
- `java.rmi.activation` (depuis JAVA2): permet l'activation à distance des objets,
- `java.rmi.dgc`: s'occupe du ramassage de miettes dans un environnement distribué,
- `java.rmi.registry` fournit l'interface permettant de représenter un `rmiregistry`, d'en créer un, ou d'en trouver un,
- `java.rmi.server` fournit les classes et interfaces pour les serveurs RMI.

Voir la documentation online:

`http://java.sun.com/j2se/1.4/docs/guide/rmi/package-use.html`

Les différentes classes et interfaces



En pratique - interface serveur activable à distance

L'interface du service ne contient que les méthodes désirées (sans argument, mais devant renvoyer une exception de type `java.rmi.RemoteException` - imposé par `rmic`):

```
public interface InterfaceObjetActivable
  extends java.rmi.Remote
{  public void MethodeA() throws java.rmi.RemoteException;
   public void MethodeB() throws java.rmi.RemoteException;
  ... }
```

Cette interface peut être en fait implémentée par un objet activable à distance ou un service permanent `UnicastRemoteObject`.

Création d'une implémentation du serveur

- Doit être une instance de `java.rmi.activation.Activatable`, (*en fait, il y a aussi moyen de faire autrement...*)
- Doit implémenter un constructeur particulier,
- Doit implémenter les méthodes désirées `MethodeA` etc.

Exemple - Création d'un objet activable à distance

```
public class ImplementationObjetActivable extends
             java.rmi.activation.Activatable
             implements InterfaceObjetActivable
{
  public ImplementationObjetActivable (
    java.rmi.activation.ActivationID activationID,
    java.rmi.MashalledObject data) throws
    java.rmi.RemoteException
    { super(activationID,0); }
```

(appelle le constructeur de la classe parent `Activatable(ID,0)`)

Création d'un objet activable à distance

Enfin, la méthode implémentant le service:

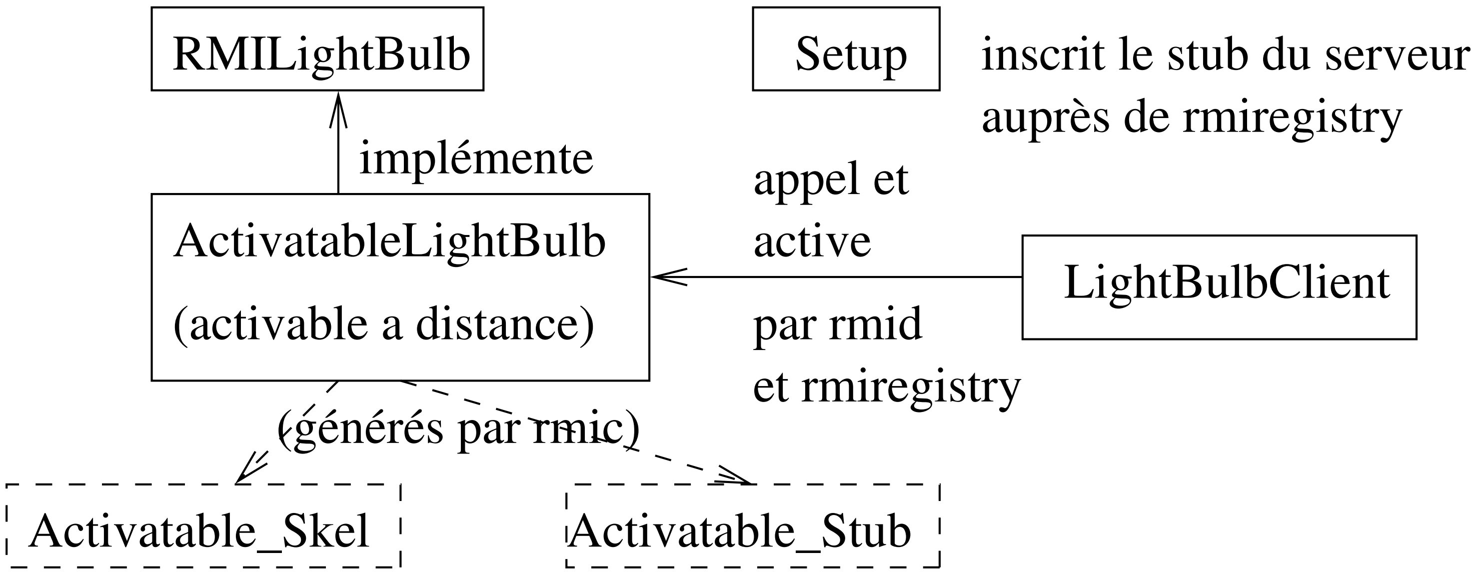
```
  public void doSomething()
  {
    ...
  }
}
```

Activation de l'objet

- En général inclus dans le `main` de l'implémentation de l'objet (service),
- Code assez compliqué par le fait que doivent être gérés:
  - la politique de sécurité,
  - les propriétés et droits du code exécutable,
  - l'inscription auprès de `rmid` et `rmiregistry`.

Exemple d'une "lampe"

(voir `~goubaul1/Cours03/RMI/Ex2`)



L'interface du serveur activable

```
package examples;
import java.rmi.*;

public interface RMILightBulb extends Remote {
    public void on() throws RemoteException;
    public void off() throws RemoteException;
    public boolean isOn() throws RemoteException;
}
```

Et un client possible...

(serait le même si on avait utilisé une implémentation du serveur comme instance de `UnicastRemoteObject`)

```
package examples;
import java.rmi.*;

public class LightBulbClient {
    public static void main(String args[]) {
System.out.println("Recherche d'une lampe...");
```

Utilisation du rmiregistry

```
  /* voir Server */
System.setSecurityManager(new RMISecurityManager());
try {
  String registry = "localhost";
  if (args.length >= 1)
    registry = args[0];
  String registration = "rmi://"+registry+
               "/ActivatableLightBulbServer";
  Remote remoteService = Naming.lookup(registration);
  RMILightBulb bulbService = (RMILightBulb) remoteService;
```

(permet de spécifier la machine où se trouve le serveur, et le `rmiregistry` correspondant)

Appel aux services

```
System.out.println("Appel de bulbservice.on()");
bulbService.on();
System.out.println("Lampe : "+bulbService.isOn());

System.out.println("Appel de bulbservice.off()");
bulbService.off();
System.out.println("Lampe : "+bulbService.isOn());
```

Récupération des exceptions

```
    } catch(NotBoundException nbe) {
      System.out.println("Pas de lampe dans le
                        repertoire de services!");
   } catch(RemoteException re) {
   System.out.println("RMI Error - "+re);
  } catch(Exception e) {
  System.out.println("Error - "+e);
  }
 }
}
```

Le serveur activable à distance

```
package examples;

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

Exemple d'une "lampe"

```
public class ActivatableLightBulbServer
extends Activatable implements RMILightBulb {
    public ActivatableLightBulbServer
            (ActivationID activationID,
             MarshalledObject data)
               throws RemoteException {
        super(activationID,0);
        setBulb(false); }
```

Services... enfin!

```
public boolean isOn() throws RemoteException {
  return getBulb();
}

public void setBulb(boolean value)
  throws RemoteException {
  lightOn = value; }

public boolean getBulb() throws RemoteException {
  return lightOn; } }
```

Services... enfin!

```
private boolean lightOn;

public void on() throws RemoteException {
  setBulb(true);
}

public void off() throws RemoteException {
  setBulb(false);
}
```

Le main de Setup et la sécurité

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

La sécurité

```
    Properties props = new Properties();
    props.put("java.security.policy",
"/users/profs/info/goubaul1/Cours03/RMI/Ex2/activepolicy");
```

La politique de sécurité doit être spécifiée à la ligne de commande - voir "Exécution".

Créer un descripteur de groupe

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

Permet d'associer des propriétés et des droits à l'exécution.

Inscrire le descripteur de groupe

Récupérer le descripteur du démon **`rmid`**, puis y enregistrer le descripteur de groupe:

```
ActivationGroupID agi =
  ActivationGroup.getSystem().registerGroup(exampleGroup);
```

Créer un descripteur d'activation du code

Au sein du groupe nouvellement crée; on passe le nom de la classe, l'endroit où se trouve le code, et une donnée optionnelle, une version "serialized" de l'objet distant (de type `MarshalledObject`):

```
  String location =
"file:/users/profs/info/goubaul1/Cours03/RMI/Ex2/";
  MarshalledObject data = null;
  ActivationDesc desc = new ActivationDesc
    (agi, "examples.ActivatableLightBulbServer",
     location, data);
```

Indiquer l'endroit où se trouve le code exécutable

On aurait pu aussi trouver le répertoire courant:

```
java.io.File location = new java.io.File(".");
String strLoc = "file://"
+URLEncoder.encode(location.getAbsolutePath(),"UTF-8");
System.out.println("Code \‘a ex\’ecuter : "+strLoc);
```

L'encodage `URLEncoder` permet d'être compatible avec les systèmes Windows etc. On peut aussi utiliser un serveur `http://...` pour le chargement dynamique de classes.

Chargement dynamique de classes

- sur un petit système, on peut gérer les stubs crées etc. (il faut recopier, ou avoir accès par NFS à tous ces fichiers sur toutes les machines considérées)
- on peut acccéder aux serveurs par adresse **`http`**:

```
java
-Djava.rmi.server.codebase=http://hostname:port/path
```

Enregistrer l'objet auprès du `rmid`

```
RMILightBulb stub = (RMILightBulb)Activatable.register(desc);
System.out.println("Got stub for ActivatableLightBulbServer");
```

Renvoie un "stub" que l'on peut enregistrer auprès du serveur de services **`rmiregistry`**.

Enregistrer le "stub" auprès du rmiregistry

```
Naming.rebind("ActivatableLightBulbServer", mri);
System.out.println("Exported ActivatableLightBulbServer");
System.exit(0); } }
```

(similaire au cas d'implémentation par `UnicastRemoteObject` - sauf que c'est le "stub" et pas l'objet lui-même qui est inscrit)

Complément: Sécurité

- Un programme JAVA peut dans certaines circonstances avoir à s'associer une "politique de sécurité" donnant les droits d'objets provenant de certaines autres machines.
- Typiquement le cas pour des objets `Serializable` transitant entre applications distantes,
- Exemple: la connection par `socket` à une machine distante passe par la méthode `checkConnect(String host, int port)` du `SecurityManager` (en fait, sous-classe `RMISecurityManager` pour les politiques concernant RMI) courant (définissant la politique de sécurité courante). En cas de non autorisation, messages du type:

```
java.security.AccessControlException: access denied
(java.net.SocketPermission 127.0.0.1:1099 connect,resolve)
```

Associer une politique de sécurité

Pour ce faire:

- construire un objet instance de `SecurityManager`,
- surcharger les fonctions `check...` dont on veut changer la politique de sécurité,
- invoquer la méthode de la classe `System`, `setSecurityManager`, en lui passant cet objet crée plus haut.

Exemple

```
System.setSecurityManager(new RMISecurityManager() {
  public void checkConnect(String host, int port) {}
  public void checkConnect(String host, int port,
                           Object context) {}
});
```

(utilise une classe anonyme, sous-classe de `RMISecurityManager` dans lesquelles les méthodes `checkConnect` retourne faux: aucune permission de créer des sockets pour communication avec d'autres machines)

Autre méthode

A partir de JAVA 1.2, une autre méthode est fournie en plus:

- toutes les méthodes `check...` font appel à la méthode `checkPermission` à laquelle on passe le type de permission souhaité,
- par exemple `checkConnect` appelle `checkPermission` sur un objet `SocketPermission`,
- la liste des permissions autorisées est gérée par la classe `Permissions`.

La classe `Policy`

- chaque programme a une politique de sécurité courante, instance de `Policy`, qui est `Policy.getPolicy()`,
- on peut modifier la politique de sécurité courante (comme avant avec le `SecurityManager`) en faisant appel à `Policy.setPolicy`,
- un fichier de permissions (lu au démarrage d'un programme) de la forme:

```
grant codeBase ``file:/home/goubault/-'' {
  permission java.security.AllPermission;
};
```

donnera tous les droits à tous les programmes dans `/home/goubault/`

La classe `Policy`

- Autre exemple (un peu plus fin...):

```
grant codeBase ``file:/home/goubault/-'' {
  permission java.net.SocketPermission
    "129.104.254.54:1024-", "connect, accept";
}
```

  (permet d'utiliser les *sockets* sur `loire.polytechnique.fr`, avec un numéro de port supérieur à 1024

- on peut utiliser `policytool` pour générer ces fichiers.

En pratique...

Pour utiliser le fichier de permissions en question (fichier), à l'exécution du `main` de la classe `maclasse`, tout en spécifiant le répertoire contenant le code:

```
java -Djava.security.policy=FICHIER
     -D-Djava.rmi.server.codebase=file:/LOCATION
     MACLASSE
```

Compilation

Il faut compiler, et créer les fichiers stubs et squelettes:

- `javac -d . RMILightBulb.java`,
- `javac -d . LightBulbClient.java`,
- `javac -d . Setup.java`
- `javac -d . ActivatableLightBulbServer.java`
- et enfin `rmic ActivatableLightBulbServer`)

(le `-d .` pour créer les répertoires correspondants aux packages)

Lancement des démons etc.

Sur la machine correspondant au serveur (ici `loire`):

- Lancer `rmiregistry` (avec un `CLASSPATH` ne contenant pas les `.class` appelés par le client)
- Spécifier la politique de sécurité. Par exemple ici, toutes les permissions (dans le fichier `activepolicy`):

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

- Lancer `rmid -J-Djava.security.policy=activepolicy`

Exécution

```
[goubaul1@loire Ex2]$ java
  -Djava.security.policy=activepolicy
  -Djava.rmi.server.codebase=file:/. examples.Setup
Got the stub for the ActivatableLightBulbServer
Exported ActivatableLightBulbServer
```

(pour spécifier la politique de sécurité)

Exécution d'un client

Sur `cher`:

```
[goubaul1@cher Ex2]$ java
  -Djava.security.policy=activepolicy
  examples.LightBulbClient loire
Looking for light bulb service
Invoking bulbservice.on()
Bulb state : true
Invoking bulbservice.off()
Bulb state : false
```

Comparaison avec UnicastRemoteObject

Le serveur lui-même (voir `~goubaul1/RMI/Ex1`):

```
public class RMILightBulbImpl extends
java.rmi.server.UnicastRemoteObject
                implements RMILightBulb {
    public RMILightBulbImpl()
    throws java.rmi.RemoteException {
      setBulb(false);
    }

    private boolean lightOn;
```

```
public void on()
  throws java.rmi.RemoteException {
    setBulb(true); }

public void off()
  throws java.rmi.RemoteException {
    setBulb(false); }

public boolean isOn()
  throws java.rmi.RemoteException {
    return getBulb(); }

public void setBulb(boolean value) {
    lightOn = value; }
public boolean getBulb() {
    return lightOn; } }
```

Mais...même interface de serveur

```
public interface RMILightBulb extends java.rmi.Remote {
    public void on() throws java.rmi.RemoteException;
    public void off() throws java.rmi.RemoteException;
    public boolean isOn() throws java.rmi.RemoteException;
}
```

Et même client...

Exécution

Sur cher (serveur):

```
[goubaul1@cher Ex1]$ java LightBulbServer
Loading RMI service
RemoteStub [ref: [endpoint:[129.104.254.54:1867](local),
            objID:[0]]]
```

EXÉCUTION (CLIENT)

Sur loire (client):

```
[goubaul1@loire Ex1]$ java LightBulbClient cher
Looking for light bulb service
Invoking bulbservice.on()
Bulb state : true
Invoking bulbservice.off()
Bulb state : false
```