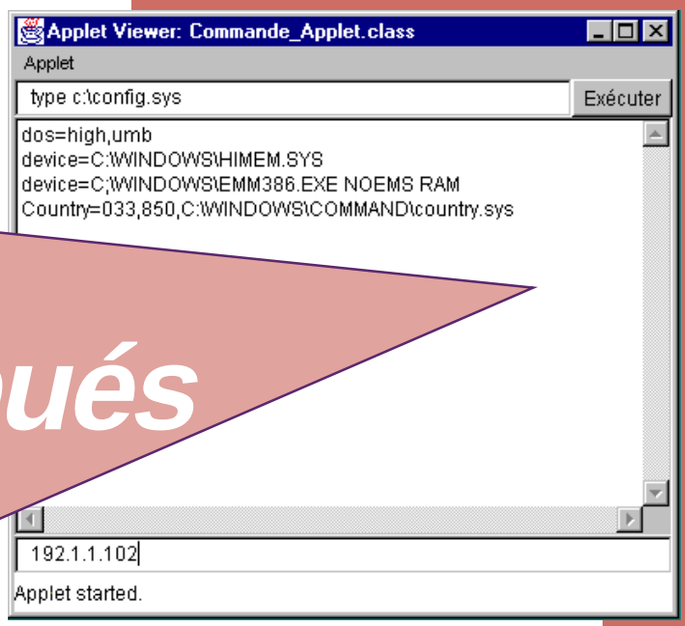

RMI, Objets distribués en Java

Par Jean-Marc Quéré

L'invocation de méthodes, sur laquelle est basé le fonctionnement de tout programme Java, peut aisément être assimilée à un mécanisme de communication par message. L'objet de RMI (Remote Method Invocation) est d'étendre simplement celle-ci à différents contextes d'exécution : machines virtuelles et ordinateurs différents. RMI permet donc l'implémentation d'un objet distant dont les méthodes peuvent être exploitées depuis une autre machine virtuelle située sur le même poste ou connecté à celui-ci via un réseau intra, extra, ou internet. La description de l'objet distant y est réalisée sous forme d'une interface qui en déclare les méthodes. Inspirez... Expirez.... Inspirez... En clair, le mécanisme abordé (RMI) permet d'utiliser à partir d'un programme Java un objet situé dans une autre machine virtuelle.

Temps de préparation :
½h (dont ¼h de cuisson : thermostat 5)
Ingrédients requis :
JDK1.1 (.7B recommandé) ou JDK 1.2, variables d'environnement PATH et CLASSPATH convenablement définies

Exemple JDK1.1.7B installé dans C:\JDK1.1.7B :

```
PATH=C:\WINDOWS\COMMAND;C:\WINDOWS\;C:\JDK1.1.7B\BIN
CLASSPATH=.;C:\JDK1.1.7B\LIB".
```

Il était une fois à l'heure H du jour J...

La classe JourEtHeure a pour seul objet d'afficher la date et l'heure au format : "aaaa-mm-jj hh:mm:ss.cc" (timestamp sql). Comportant la méthode "public void main(String argv[])", elle est autonome. Après compilation par "javac JourEtHeure", elle peut être activée par "java JourEtHeure".

JourEtHeure.java

```
import java.sql.Timestamp;
import java.util.Calendar;

class JourEtHeure {
  public String Maintenant() {
    return new
Timestamp(Calendar.getInstance().getTime().getTime()).toString();
  }

  static public void main(String argv[]) {
    System.out.println(new JourEtHeure().Maintenant());
  }
}
```

L'expression "Calendar . getInstance() . getTime() . getTime()" récupère l'instance usuelle de la classe Calendar (via la méthode getInstance()), puis la convertit en Date (premier getTime(), méthode de la classe Calendar), puis en long (second getTime(), méthode de la classe Date).

Et ailleurs ?

La réalisation proposée afin d'illustrer RMI, effectue le même traitement, au détail prêt, qu'il s'exécute dans un autre environnement (avec éventuellement un décalage horaire). A cet effet, trois sources distincts sont réalisés : une interface (JourEtHeure_Interface.java), le serveur RMI (JourEtHeure_Implementation.java) et son client (JourEtHeure_Client.java).

JourEtHeure_Interface.java

```
import java.rmi.*;

public interface JourEtHeure_Interface extends Remote {
  public String Maintenant() throws RemoteException;
}
```

L'interface se limite à la déclaration des méthodes devant être exploitées via RMI. Une seule dans le cadre de l'exemple : "Maintenant()". A noter que toutes les interfaces d'objets distants requièrent l'utilisation de l'interface "Remote". Elle ne comprend aucune déclaration ("interface remote { }"), mais permet de référencer les objets par un même type : "Remote". Enfin, nul n'étant parfait, d'éventuelles exceptions peuvent être levées par le processus RMI : serveur inconnu, déconnexion sauvage, etc. Elles doivent impérativement être traitées par l'application appelante d'où l'indication "throws RemoteException".

JourEtHeure_Implementation.java

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

import java.sql.Timestamp;
import java.util.Calendar;

public class JourEtHeure_Implementation extends UnicastRemoteObject
implements JourEtHeure_Interface {
  public JourEtHeure_Implementation() throws RemoteException {
    super();
  }

  public String Maintenant() throws RemoteException {
    return new Timestamp(Calendar.getInstance().getTime().getTime()).toString();
  }

  public static void main(String args[]) {
    System.setSecurityManager(new RMISecurityManager());
    try {
      JourEtHeure_Implementation JourEtHeure = new
JourEtHeure_Implementation();
      Naming.rebind("JourEtHeure", JourEtHeure);
      System.out.println("Prêt.");
    }
    catch (Exception e) {
      e.printStackTrace();
    }
  }
}
```

L'objet distant et ses méthodes sont implémentés sous forme d'un serveur RMI. La classe étend "UnicastRemoteObjet" et implémente l'interface précédemment réalisée. La classe "UnicastRemoteObjet" comporte toutes les fonctionnalités utiles à la réalisation de notre serveur d'objets, notamment la prise en charge des invocations distantes d'un client. Les objets distribués peuvent indifféremment être créés par un client ou le serveur. Le dialogue lié à leur emploi est géré par un flux sérialisé. Le constructeur dynamique se limite à l'appel du constructeur générique (celui de UnicastRemoteObjet) afin d'initialiser et d'activer le serveur. La méthode "Maintenant()" est définie conformément à l'interface. Elle effectue le même traitement que celle de "JourEtHeure" (retourne la date et l'heure au format timestamp sql). L'amorce du programme "main", incluse à la classe, commence par mettre en place un niveau de protection adapté (setSecurityManager) à la destination de l'application : serveur RMI. Ensuite, une instance de l'objet est créée (JourEtHeure). A ce stade le constructeur de "UnicastRemoteObjet" a été appelé et le serveur est prêt. Il ne reste plus qu'à le référencer pour en permettre l'utilisation. Cette opération est effectuée via l'appel de la méthode "rebind" de la classe "Naming". Elle l'associe à l'url : "rmi:// <adresse ip du micro hébergeant la machine virtuelle> / JourEtHeure". Ici, la méthode "rebind" est utilisée de préférence à "bind", car cette dernière "écrase" toute définition précédante, alors que "bind" lève une exception.

JourEtHeure_Client.java

```
import java.rmi.*;
import java.rmi.registry.*;
import java.rmi.RMISecurityManager;

public class JourEtHeure_Client {
  static public void main(String[] argv) {
    JourEtHeure_Interface JourEtHeure;
    try {
      JourEtHeure=(JourEtHeure_Interface)Naming.lookup("rmi://localhost/
JourEtHeure");
      System.out.println(JourEtHeure.Maintenant());
    }
    catch (Exception e) {
      e.printStackTrace();
    }
  }
}
```

Dernière pièce du puzzle, le client exécute deux opérations. **La première** consiste à récupérer l'objet distant via son url. "localhost" correspond à l'usage des différentes machines virtuelles sur le même ordinateur ; remplacez "localhost" par le nom ou l'adresse de la machine distante hébergeant le serveur RMI. **La seconde** affiche le résultat de l'invocation de la méthode "Maintenant()" de l'objet distant.

En pratique...

Si vous ne disposez par de plusieurs (au moins deux) ordinateurs reliés entre eux par un réseau tcp/ip, vous pouvez activer le serveur et son client sur une même machine. La démarche à suivre est explicitée ci-dessous. Compilez tous les sources :

```
javac JourEtHeure_Interface.java
javac JourEtHeure_Implementation.java
javac JourEtHeure_Client.java
```

Générez les amorces (JourEtHeure_Implementation_Stub/Skel) :

```
rmic JourEtHeure_Implementation
```

Les amorces Stub/Skeleton assurent l'interface entre les différents éléments de l'application et RMI, notamment en prenant en charge le système de description et d'invocation des objets distants. Elles sont générées de façon automatique, puis compilées. Sur le serveur, activez le service de dénomination par "rmiregistry" dans une première fenêtre "Commande MS-DOS", puis le serveur dans une seconde par "java JourEtHeure_Implementation". Le message "prêt" s'affiche. Si à ce stade vous obtenez un message du type :

```
java.rmi.ServerException:Server RemoteException; nested exception is :
java.rmi.UnmarshalException : error unmarshalling arguments; nested exception
is: java.lang.ClassNotFoundException : JourEtHeure_Implementation_Stub
```

c'est que "rmiregistry" n'a pas été démarré dans le répertoire ou est exécuté la commande "java JourEtHeure_Implementation" et qu'il ne trouve pas la classe "JourEtHeure_Implementation_Stub". Vous pouvez alors optez pour lancer ces deux commandes à l'invite MS-DOS à partir du même répertoire ou modifier votre variable d'environnement CLASSPATH afin qu'elle comporte le chemin d'accès aux classes "JourEtHeure_...".

Les classes requises (qui doivent donc être déployées) pour l'exécution de "JourEtHeure_Client" sur un poste client connecté au réseau sont : JourEtHeure_Client, JourEtHeure_Interface et JourEtHeure_Implementation_Stub. Activez l'appel au service par "java JourEtHeure_Client". L'heure du serveur s'affiche alors au format timestamp sql.

Envoyer une commande à l'invite MS-DOS... avec Windows 9x/NT

L'idée de base est d'exécuter à partir d'une applet une commande sur une machine distante, puis de récupérer et d'afficher le résultat. La classe "Commande", ci-dessous, illustre la procédure employée (localement) à cette fin *(voir le code Commande.java page suivante)*.

Suite à l'exécution de la commande dans un sous-processus via la méthode "exec()" de "Runtime", l'application attend 500ms, puis récupère le flux en entrée du sous-processus. Toutes les 500ms, le nombre de caractères disponibles est comparé à la dernière valeur mémorisée. S'ils sont égaux, les données issues du flux sont lues à partir d'un "InputStreamReader" sous la forme d'un tableau de caractères. Celui-ci est finalement converti en une chaîne de caractères puis retourné. Ne vous alarmez pas si la méthode employée vous paraît obscure *(ça fait bidouille...)*. En effet, dans les règles de l'art, le code employé *(en lieu et place du code en gras)* aurait du être :

```
// - - - théorie
dos.waitFor();
BufferedReader buf=
new BufferedReader(new InputStreamReader(dos.getInputStream()));
String mem;
while ((mem=buf.readLine())!=null)
lig=lig+mem;
// - - -
```

Hélas, cet @#% de "command.com" bloque l'application au niveau du "dos.waitFor()" et ne se termine jamais. Avec n'importe quelle autre application Windows *(notepad et consort),* cela fonctionne parfaitement. Autre souci lié à notre "command.com", la lecture du flux bloque au niveau du "readLine()", la valeur "null" n'est jamais retournée. La gymnastique proposée s'avère donc nécessaire. A noter que si vous utilisez un autre shell que "command.com", ce dernier peut très bien fonctionner avec la seconde méthode *(c'est le cas d'un sh95 en ma possession, c'est également le cas des autres OS : Linux, etc..)*.

Invocation distante...

Comme pour le premier exemple, trois sources sont nécessaires à l'application. Ils se nomment respectivement *(voir page suivante)* :

- Commande_Interface *(interface),*
- Commande_Implementation *(serveur)*
- Commande_Client *(client).*

Compilez tous les sources :

```
javac Commande_Interface.java
javac Commande_Implementation.java
javac Commande_Client.java
```

Générez les amorces (JourEtHeure_Implementation_Stub/Skel) :

```
rmic Commane_Implementation
```

Sur le serveur, activez le service de dénomination par "rmiregistry" dans une première fenêtre "Commande MS-DOS", puis le serveur dans une seconde par "java Commande_Implementation". Le message "prêt" s'affiche *(comme pour JourEtHeure)*.

Les classes à déployer pour l'exécution de "JourEtHeure_Client" sur un poste client connecté au réseau sont : Commande_Client, Commande_Interface et Commande_Implementation_Stub. Activez l'appel au service par "java Commande_Client". Le contenu du répertoire du serveur s'affiche.

L'applet... (réalisée avec JBuilder 2)

Elle comporte deux zones de saisie *(textField)*. La première située en haut à gauche comporte la ligne de commande à exécuter *(par défaut "DIR")*. Vous pouvez taper une commande quelconque, dès lors qu'elle ne requiert pas de dialogue avec l'utilisateur, par exemple : "TYPE C:\AUTOEXEC.BAT". La seconde zone en bas de l'applet doit comporter l'adresse *(ou la désignation)* du poste hébergeant le serveur RMI *(par défaut "localhost" : serveur et client sur le même poste, sinon utilisez l'adresse ip du serveur)*. Suite à la pression du bouton "Exécuter", la commande est exécutée sur le serveur et le résultat de celle-ci *(texte destiné à la console)* est affiché dans le textArea *(voir image en tête d'article, codes Applet Java et commande _html.html page suivante)*.

L'applet peut être activée soit à partir d'un navigateur en chargeant la page "Commande_Html.html", soit via l'appletviewer par la commande "appletviewer Commande_Html.html". Si vous utilisez un navigateur pour visualiser le document ne chargez pas celui-ci directement ("url du style file:///C|/.../Commande_Html.html") mais utilisez un serveur Http, sinon la connexion au serveur RMI sera refusée (pour cause d'insécurité).

Ca marche ?!

Et hop, un petit type du "config.sys"...- figure page 69 -

Pourquoi utiliser des objets distribués ?

Dans le cadre de l'utilisation d'application via internet, l'intérêt est évident. La mise en œuvre d'applet légère dialoguant avec un serveur RMI comportant l'ensemble des traitements diminue les temps de chargement et sécurise les traitements *(effectués sur le serveur)*. Pour les applications "strong java" ou d'entreprise, l'utilisation de ce procédé permet de grouper les traitements par machine en fonction de la puissance requise et de leur centre d'intérêt. L'actualisation des traitements effectués par le serveur *(tant que les méthodes invoquées ne sont pas modifiées)* ne requiert aucune intervention sur les clients. Enfin, aspect primordial, lors de l'emploi de ressources natives *(via JNI)* spécifiques à un contexte, la mise à disposition de celles-ci via un serveur RMI permet de conserver l'indépendance des autres postes à l'égard de celles-ci. Seul le contexte à l'origine de la contrainte subit cette mise à disposition. Cette pratique permet notamment d'ouvrir à Java un existant fort contraignant : type de machine et de système d'exploitation imposés, important développement réalisé de façon native, ... Un mécanisme d'objets distribués s'est largement répandu dans de nombreux contextes propriétaires afin de disposer de ces possibilités : IIOP / Corba.

Commande_Interface.java

```
import java.rmi.*;

public interface Commande_Interface extends Remote {
  public String Execute(String cde) throws RemoteException;
}
```

Commande_Implementation.java

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

import java.io.*;

public class Commande_Implementation extends UnicastRemoteObject implements
Commande_Interface {
  public Commande_Implementation() throws RemoteException {
```

Commande.java

```
import java.io.*;
class Commande {
  public String Execute(String cde) {
    String lig="";
    try {
      Process dos=
      Runtime.getRuntime().exec("COMMAND.COM /C "+cde);

      // - - - pratique
      int mem=0;
      java.lang.Thread.sleep(500);
      InputStream ins=dos.getInputStream();
      while (ins.available()!=mem) {
    mem=ins.available();
    java.lang.Thread.sleep(500);
      }
      char[] buf=new char[mem];
      new InputStreamReader(ins).read(buf,0,mem-1);
      lig=new String(buf);
      // - - -

    }
    catch (Exception e) {
      e.printStackTrace();
    }
    return lig;
  }

  static public void main(String argv[]) {
    System.out.println(new Commande().Execute("DIR"));
  }
}
```

Commande_Applet.java

```
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
import java.rmi.*;

public class Commande_Applet extends Applet {
  boolean isStandalone = false;
  Panel panel1 = new Panel();
  BorderLayout borderLayout1 = new BorderLayout();
  BorderLayout borderLayout2 = new BorderLayout();
  TextField textFieldCde = new TextField();
  Button buttonCde = new Button();
  TextArea textAreaCde = new TextArea();
  TextField textFieldUrl = new TextField();

  public Commande_Applet() {
  }

  public void init() {
    try {
    jbInit();
    }
    catch (Exception e) {
    e.printStackTrace();
    }
  }

  private void jbInit() throws Exception {
    panel1.setLayout(borderLayout1);
    this.setLayout(borderLayout2);
    textFieldCde.setText("DIR");
    buttonCde.setLabel("Exécuter");
    textFieldUrl.setText("localhost");
    buttonCde.addActionListener(new java.awt.event.ActionListener() {
      public void actionPerformed(ActionEvent e) {
        buttonCde_actionPerformed(e);
      }
    });
    this.add(panel1, BorderLayout.NORTH);
    panel1.add(textFieldCde, BorderLayout.CENTER);
    panel1.add(buttonCde, BorderLayout.EAST);
    this.add(textAreaCde, BorderLayout.CENTER);
    this.add(textFieldUrl, BorderLayout.SOUTH);
  }

  public String getAppletInfo() {
    return "Information applet";
  }

  public String[][] getParameterInfo() {
    return null;
  }

void buttonCde_actionPerformed(ActionEvent e) {
Commande_Interface Commande;
try {
Commande=(Commande_Interface)Naming.lookup("rmi://"+textFieldUrl.getText()+"/Commande");
textAreaCde.setText(Commande.Execute(textFieldCde.getText()));
}
    catch (Exception f) {
      f.printStackTrace();
    }
  }
}
```

Commande_Client.java

```
import java.rmi.*;
import java.rmi.registry.*;
import java.rmi.RMISecurityManager;

public class Commande_Client {
  static public void main(String[] argv) {
    Commande_Interface Commande;
    try {
      Commande=(Commande_Interface)Naming.lookup("rmi://localhost/Commande");
      System.out.println(Commande.Execute("DIR"));
    }
    catch (Exception e) {
      e.printStackTrace();
    }
  }
}
```

Commande_Html.html

```
<HTML>
<HEAD>
<META HTTP-EQUIV="Content-Type" CONTENT="text/html; charset=iso-8859-1">
<TITLE>
Page de test HTML de Commande_Applet
</TITLE>
</HEAD>
<BODY>
Commande_Applet appara&icirc;tra ci-dessous dans un navigateur Java.<BR>
<APPLET
  CODEBASE = "."
  CODE     = "Commande_Applet.class"
  WIDTH    = 400
  HEIGHT   = 300
  HSPACE   = 0
  VSPACE   = 0
  ALIGN    = middle
>
</APPLET>
</BODY>
</HTML>

    super();
  }

  public String Execute(String cde) throws RemoteException {
    String lig="";
    try {
      Process dos=Runtime.getRuntime().exec("COMMAND.COM /C "+cde);
      int mem=0;
      java.lang.Thread.sleep(500);
      InputStream ins=dos.getInputStream();
      while (ins.available()!=mem) {
    mem=ins.available();
    java.lang.Thread.sleep(500);
      }
      char[] buf=new char[mem];
      new InputStreamReader(ins).read(buf,0,mem-1);
      lig=new String(buf);
    }
    catch (Exception e) {
      e.printStackTrace();
    }
    return lig;
  }

  public static void main(String args[]) {
    System.setSecurityManager(new RMISecurityManager());
    try {
      Commande_Implementation Commande = new Commande_Implementation();
      Naming.rebind("Commande", Commande);
      System.out.println("Prˆt.");
    }
    catch (Exception e) {
      e.printStackTrace();
    }
  }
}
```