Maitrise d'Informatique
Module POD

Programmation Objet et Distribution
Documents autorisés

Université Pierre et Marie Curie
Année 1999–2000
Durée 3 heures

Examen du 26 septembre 2000

Exercice 1 : Listes d'associations en RMI

Le but de cet exercice est d'implanter des listes d'associations distantes sur un serveur RMI.
Pour cela, on définit la classe `Not_found` et l'interface `LA`.

```
class Not_Found extends Exception {};

interface LA {
  public void add(int cle, String val);
  public bool mem(int cle);
  public String find(int cle) throws Not_found;
  public void modify(int cle, String new_val) throws Not_found;
}
```

1. Définir l'interface `LADIST` qui étend l'interface `Remote` et décrit les mêmes méthodes en leur ajoutant les noms des exceptions déclenchables.

2. Écrire la classe ou la hiérarchie de classes Java, implantant les listes d'association et respectant l'interface `LADIST`.

3. Écrire un serveur RMI qui enregistre sous le nom `'maliste'` une instance de cette classe correspondant à une liste vide.

4. En indiquant les noms de fichiers de ces classes et interfaces, décrivez les étapes de compilation et de lancement du serveur.

5. Que fait le programme client suivant?

```
import java.rmi.*;

class Client {
  public static void main (String []args) {
    String url0="rmi://ippc23.infop6.jussieu.fr/maliste";
    try {
      laRMI la = (LADIST)Naming.lookup(url0);
      la#add(1,"MARCEL");
      if (la#mem(2)) { la#modify( (la#find(2))+"--"); }
      else { la#add(2,"ROBERT"); }
      la#add(3,"MIREILLE");
      System.out.println(la#search(1));
      System.out.println(la#search(2));
      System.out.println(la#search(3));
    }
    catch (Not_found e) {}
    catch (Exception e) {
      System.err.println("exception : " + e.getMessage())
    }
  }
```

6. Que se passe-t-il si ce client est lancé en même temps sur deux machines différentes avec `maliste` vide sur le serveur?

Exercice 2 : Gestion d'événements

Le but de cet exercice est de comparer différents modèles pour la gestion des événements issus des composants graphiques de Java. Pour cela on définit une première `applet`, qui contient deux composants graphiques de classe `BA`, de la manière suivante :

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

class BA extends Button implements ActionListener {
    BA target;
    BA(String s) {super(s); addActionListener(this);}

    BA getTarget() {return target;}
    void setTarget(BA n){target=n;}

    public void actionPerformed(ActionEvent e) {
        String s = getLabel();
        setLabel(target.getLabel());
        target.setLabel(s);
    }

}
class ev extends Applet {
    BA b1 = new BA("A");
    BA b2 = new BA("B");
    ev() {
        b1.setTarget(b2);
        b2.setTarget(b1);
        add(b1); add(b2);
    }
}
```

1. Indiquer ce que fait cette `applet` quand on clique sur le bouton `b1` puis sur le bouton `b2` en expliquant comment les composants graphiques communiquent entre eux.
2. Mêmes questions pour le programme suivant :

```
class Action implements ActionListener {
    Button a1,a2;
    Action (Button v1, Button v2){a1=v1; a2=v2;}
    void swapLabel(Button x1, Button x2) {
        String s = x1.getLabel();
        x1.setLabel(x2.getLabel());
        x2.setLabel(s);
    }
    public void actionPerformed(ActionEvent e) {
Object obj = e.getSource();
if ((obj  == a1) || (obj == a2)) {swapLabel(a1,a2);}
    }
}
class ev2 extends Applet {
    Button  b1 = new Button("A");
    Button  b2 = new Button("B");
    ActionListener al = new Action(b1,b2);
    ev2() {
        b1.addActionListener(al);
        b2.addActionListener(al);
        add(b1); add(b2);
    }
}
```

3. Ecrire une troisième version de ce programme en déclarant une classe `ev3` dérivant d'`Applet` en introduisant une classe locale dont une instance va traiter les clics des boutons. Indiquer en quoi votre solution est plus simple que la précédente.

4. Comparer ces trois solutions en justifiant votre préférence, en particulier sur la séparation graphisme et traitement des événements.

Problème : Partage de mémoire

Le but de cet exercice est de pouvoir partager une zone mémoire par plusieurs programmes tournant sur différentes machines. Pour simplifier, la zone mémoire partagée sera un tableau de flottants. Pour le serveur de tableau distant on définit les classes `connexion` et `serv_socket` suivantes :

```
class connexion (sd,sa) x  =
  object (self)
    val s_descr = sd
    val s_addr = sa
    val v = x

    method start () =  Thread.create (fun x -> self#run x ; self#stop x) ()

    method stop () =   Unix.close s_descr

    method private read () =
      let s = " "  and  r = ref ""  in
        while (ThreadUnix.read s_descr  s 0 1 > 0) &&
                (s.[0] <> '\n')
        do r := !r ^s
        done ;
        !r

    method private write s = let src = "s"^"\n" in
      ThreadUnix.write s_descr src 0 (String.length src); ()

    method  run () =
      let commande = self#read() in
      try
        match commande with
          "GET" ->   let i = int_of_string (self#read()) in
                       self#write(string_of_float v.(i))
        | "PUT" ->   let i = int_of_string (self#read()) in
                       let r = float_of_string (self#read()) in
                          v.(i)<-r
        |  _ ->  self#write("ERREUR 1")
      with _ ->  self#write("ERREUR 2")


  end ;;

class  serv_socket p  x =
  object (self)
    val port = p
    val mutable sock = ThreadUnix.socket Unix.PF_INET Unix.SOCK_STREAM 0
    val v = x

    initializer
      let mon_adresse = (Unix.gethostbyname(Unix.gethostname())).Unix.h_addr_list.(0)
      in Unix.bind sock (Unix.ADDR_INET(mon_adresse,port)) ;
         Unix.listen sock 3
```

```
  method private client_addr = function
      Unix.ADDR_INET(host,_) -> Unix.string_of_inet_addr host
    | _ -> "Unexpected client"

  method run () =
    while(true) do
      let (sd,sa) = ThreadUnix.accept sock in
        let connexion = new connexion(sd,sa) v  in
          (self#client_addr sa) ;
          connexion#start ()
    done
end ;;
```

1. Indiquer ce que fait le programme suivant :

   ```
   let main () =
     let t = Array.create 100 0.0 in
       let s = new serv_socket 12340 t in
       s#run();;
   main();;
   ```

   en précisant le rôle des méthodes employées de ces deux classes et en décrivant le protocole employé.

2. Ajouter au protocole du serveur une commande `LEN` qui retourne la longueur du vecteur.

3. Ecrire un client qui se connecte à ce serveur et calcule la somme des valeurs du vecteur partagé du serveur.

4. Ecrire la partie spécifique d'un client qui copie le contenu d'un vecteur local sur le serveur.

5. Que se passe-t-il si les deux clients précédents tournent en même temps?

6. On cherche à implanter un mécanisme d'exclusion mutuelle entre les différents clients. Proposer une extension du protocole pour pouvoir poser un verrou sur le vecteur du serveur à partir d'un client. Modifier les deux clients précédents pour qu'il n'y ait pas de modification du vecteur pendant sa lecture complète.

7. Implanter votre proposition en modifiant la classe `connexion`. Vous pouvez changer les arguments du constructeur de cette classe si nécessaire.

8. Que se passe-t-il si un client ayant pris le verrou se termine sans le libérer? Que faut-il modifier dans la classe `connexion` pour en tenir compte?

9. Pour éviter de verrouiller le vecteur entier, proposer sans l'implanter une solution où chaque élément du tableau possède son propre verrou.