Master d'informatique | **module PC2R (4I507)** | **Université Pierre et Marie Curie**
Documents autorisés | *Durée 2 heures* | **Année 2017-2018**

Examen réparti du 31 mai 2018

Exercice 1 : Eclairage de serres en Esterel

Important : L'utilisation des instructions adaptées est appréciée. Elles évitent des codes longs et souvent propices aux erreurs.

Le but de cet exercice est de simuler l'éclairage dans une serre. Pour compenser la faible luminosité, on allume une ou des lampes.

La serre (module `serre`) est composée de
- un détecteur de lumière du soleil (module `soleil`),
- trois lampes indépendantes (module `lampe`) avec les puissances respectives 2, 1 et 1.

Le système fonctionne de la manière suivante : (voir l'exemple de session ci-dessous)
- On le démarre (ou le redémarre) avec le signal `START`.
- On l'arrête avec le signal `STOP`.
- Au (re)démarrage, le module `soleil` émet le premier signal valué `SOLEIL` avec une valeur aléatoire comprise entre 0 et 10 (voir `alea()`).
  Il l'émettra de nouveau tous les 10 `tick` avec une nouvelle valeur aléatoire.
- A chaque signal `SOLEIL` émis, la première lampe est actionnée instantanément si la valeur de `SOLEIL` est strictement inférieure à 9.
  Dans ce cas, elle émet d'abord le signal `LAMPE1`, puis en continu le signal `LUMIERE` avec comme valeur associée sa puissance (ici 2).
- A chaque signal `SOLEIL` émis, la 2ème (resp. 3ème) lampe est actionnée instantanément si la valeur de `SOLEIL` est strictement inférieure à 8 (resp. 7).
  Dans ce cas, elle émet d'abord le signal `LAMPE2` (resp. `LAMPE3`), puis en continu le signal `LUMIERE` avec comme valeur associée sa puissance (ici 1).
- Si le signal `LUMIERE` est émis, on souhaite voir la valeur associée comme la somme de toutes les lumières émises par le ou les lampes.

Question 1. Ecrire le module `soleil`.
Question 2. Ecrire le module `lampe`.
Question 3. Ecrire la module `serre`.

Les pannes du détecteur de lumière du soleil et/ou des lampes sont possibles.
Pour simuler ces pannes, on propose, par exemple, dans les modules `soleil` et `lampe` un test d'égalité `alea(50) = 25`. Si ce test est vrai :
- dans `soleil`, le signal `KO_SOLEIL` est émis en continu et aucun autre signal est émis.
  On ne pourra redémarrer le système qu'en redonnant à nouveau le signal `START`. Le temps de réparer le capteur.
- dans la lampe 1 (resp. 2 et 3), le signal `KO_LAMPE1` (resp. `KO_LAMPE2` et `KO_LAMPE3`) est émis en continu et aucun autre signal concernant la lampe est émis.
  On ne pourra redémarrer la lampe qu'en donnant le signal `REP_LAMPE1` (resp. `REP_LAMPE2` et `REP_LAMPE3`). Le temps de réparer la lampe.

Pour gérer ces pannes :

Question 4. Modifier le module `soleil`.
Question 5. Modifier le module `lampe`.
Question 6. Modifier la module `serre`.

```c
#include <stdlib.h>
#include <stdio.h>


/* alea() genere un nombre aleatoire compris dans [0, maximum] */
int alea (int maximum) {

  return (int)((rand() + 1.0) / RAND_MAX * maximum);
}
```

```
$ serre
serre> START;;;;;;;;
--- Output: SOLEIL(8) LUMIERE(2) LAMPE1
--- Output: LUMIERE(2)
--- Output: LUMIERE(2)
--- Output: LUMIERE(2)
--- Output: LUMIERE(2)
--- Output: LUMIERE(2)
--- Output: LUMIERE(2)
--- Output: LUMIERE(2)
serre> ;;;;;;;;
--- Output: LUMIERE(2)
--- Output: LUMIERE(2)
--- Output: SOLEIL(3) LUMIERE(4) LAMPE1 LAMPE2 LAMPE3
--- Output: LUMIERE(4) KO_SOLEIL
--- Output: KO_SOLEIL
--- Output: KO_SOLEIL
--- Output: KO_SOLEIL
--- Output: KO_SOLEIL
serre> START;;;;;;;;;
--- Output: SOLEIL(0) LUMIERE(4) LAMPE1 LAMPE2 LAMPE3
--- Output: LUMIERE(4)
--- Output: LUMIERE(4) KO_LAMPE3
--- Output: LUMIERE(3) KO_LAMPE3
--- Output: LUMIERE(3) KO_LAMPE3
--- Output: LUMIERE(3) KO_LAMPE3
--- Output: LUMIERE(3) KO_LAMPE3
--- Output: LUMIERE(3) KO_LAMPE3
--- Output: LUMIERE(3) KO_LAMPE3
serre> REP_LAMPE3;;;;;;;;
--- Output: LUMIERE(3)
--- Output: SOLEIL(3) LUMIERE(4) LAMPE1 LAMPE2 LAMPE3
--- Output: LUMIERE(4)
--- Output: LUMIERE(4)
--- Output: LUMIERE(4)
--- Output: LUMIERE(4)
--- Output: LUMIERE(4)
--- Output: LUMIERE(4)
serre> ;;;;;;;;;;;;;;;
--- Output: LUMIERE(4) KO_LAMPE1
--- Output: LUMIERE(2) KO_LAMPE1
--- Output: LUMIERE(2) KO_LAMPE1
--- Output: SOLEIL(7) LUMIERE(1) LAMPE2 KO_LAMPE1
--- Output: LUMIERE(1) KO_LAMPE1
--- Output: LUMIERE(1) KO_LAMPE1
--- Output: LUMIERE(1) KO_LAMPE1
--- Output: LUMIERE(1) KO_LAMPE1
--- Output: LUMIERE(1) KO_LAMPE1
--- Output: LUMIERE(1) KO_LAMPE1
--- Output: LUMIERE(1) KO_LAMPE1
--- Output: LUMIERE(1) KO_LAMPE1
--- Output: LUMIERE(1) KO_LAMPE1
--- Output: SOLEIL(8) KO_LAMPE1
--- Output: KO_LAMPE1
serre> REP_LAMPE1;;;;;;;;
--- Output:
--- Output:
--- Output:
--- Output:
--- Output:
--- Output:
--- Output:
--- Output:
--- Output:
serre> ;;;;;;;;;;
--- Output: SOLEIL(6) LUMIERE(4) LAMPE1 LAMPE2 LAMPE3
--- Output: LUMIERE(4)
--- Output: LUMIERE(4)
--- Output: LUMIERE(4)
--- Output: LUMIERE(4)
--- Output: LUMIERE(4)
--- Output: LUMIERE(4)
--- Output: LUMIERE(4)
--- Output: LUMIERE(4)
--- Output: LUMIERE(4)
serre>
```

Exercice 2 : RMI asynchrone / Garbage collector réparti

RMI est un mécanisme d'appel de méthodes à distance, ce qui correspond par défaut à un modèle *synchrone* : le client attend que le serveur lui renvoie une réponse. Ce modèle fonctionne parfaitement mais est très restrictif, en particulier dans les situations où le traitement sur le serveur peut être relativement long. Nous allons étudier dans cet exercice différentes techniques qui permettent de construire un modèle *asynchrone* au dessus de RMI. Cette construction nous permettra également d'envisager la construction d'un système de *Garbage Collector* réparti en RMI.

Partie 1. Construction d'un RMI Asynchrone

1. Solution par threads sur le client

Si le serveur n'offre aucun support pour un appel asynchrone, la seule solution est de passer par un thread sur le client. Pour réaliser un appel RMI asynchrone au niveau client, il "suffit" d'encapsuler l'appel dans un thread

Question 1. Ecrire un serveur RMI qui propose une méthode echo classique (elle renvoie au client la chaîne de caractères paramètre de l'appel), mais avec une pause importante au début de la méthode, pour simuler un calcul complexe (on pourra tirer au hasard la durée de la pause). Ecrire un client pour ce serveur qui réalise l'appel de façon asynchrone en utilisant un thread.

Le problème de cette solution est que c'est le thread qui exécute l'appel qui décide de faire l'affichage du résultat. Dans certaines situations, cela n'est pas vraiment acceptable

Question 2. Modifier l'objet Thread d'appel asynchrone afin d'obtenir les fonctionnalités suivantes **sans utiliser les objets Futures**

- la méthode run ne doit plus afficher le résultat mais se contenter de le stocker dans une variable adaptée
- l'objet doit proposer une méthode permettant de savoir si le calcul est terminé
- l'objet doit proposer une méthode bloquante renvoyant le résultat de l'appel

Proposer une démonstration des fonctionnalités de ce nouvel appel asynchrone.

2. Solution par consultation

Ici nous allons directement modifier le serveur pour proposer des services permettant des appels asynchrones. Nous commençons la partie serveur par une API de type *consultation* : le client lance une opération sur le serveur, puis demande périodiquement à celui-ci si l'opération est terminée. Le client est donc actif et réalise en général une attente active. On considère ainsi l'objet distant suivant

```
1 import java.rmi.*;
2 public interface IPullServer extends Remote {
3     public IDoSomethingResult doSomething(String param) throws RemoteException;
4 }
```

La méthode `doSomething` est asynchrone, c'est-à-dire qu'elle doit lancer un calcul côté serveur (en utilisant un thread) et rendre la main le plus rapidement possible au client. Pour permettre à celui-ci d'obtenir le résultat de l'opération, la méthode renvoie une référence vers un objet distant `IDoSomethingResult` spécifique à l'appel. Voici l'interface en question

```
1 import java.rmi.*;
2 public interface IDoSomethingResult extends Remote {
3     public boolean isDone() throws RemoteException;
4     public String getResult() throws RemoteException;
5 }
```

La méthode `isDone` renvoie true si et seulement si le serveur a terminé les calculs correspondant à l'opération. La méthode `getResult` renvoie le résultat de l'opération (null si celle-ci n'est pas terminée).

Question 3. Implémenter `IDoSomethingResult` par une classe qui implémente aussi `Runnable` (et qui hérite de `UnicastRemoteObject`) : l'idée est d'avoir le code à exécuter (la méthode run) avec le dispositif de stockage (pour les méthodes `isDone` et `getResult`) et le serveur RMI (grâce à `UnicastRemoteObject`). Pour les calculs de `doSomething`, on se contentera d'un écho (d'ou le paramètre param) avec un délai de réponse aléatoire.

Question 4. Implémenter `IPullServer`.

Question 5. Implémenter un client utilisant `IPullServer` pour faire un appel asynchrone avec une attente active.

On veut maintenant fournir une solution avec attente passive

Question 6. On reprend l'exercice précédent en ajoutant à l'interface `IDoSomethingResult` la méthode suivante :

```
1 public void waitForResult() throws RemoteException;
```

Cette méthode permet au client d'éviter l'attente active : elle bloque le client jusqu'à ce que le résultat soit disponible. Implémenter cette nouvelle version en utilisant un mécanisme de synchronisation permettant d'implémenter le blocage sur le serveur.

Partie 2. Garbage Collector

On remarque que dans notre solution de RMI asynchrone précédente, on crée des objets spécifiques à *chaque appel* effectué par un client. On se retrouve donc rapidement avec un ensemble d'objets transitoires qui peuvent rapidement devenir encombrant. On doit donc gérer un système de Garbage Collector (GC) réparti au niveau du serveur. On rappelle rapidement quelques propriétés d'un GC.

1. Chaque objet doit contenir un compteur de références (`obj.ref_count`)
2. A chaque affectation, (`a = obj`) on augmente ce compteur de références
3. On considèrera ici par simplicité que `obj = null` représente la suppression de l'objet par notre GC maison

Question 7. Décrire les modifications à apporter au système précédent et les implémenter (succintement) pour permettre de gérer le GC dans notre cas asynchrone.

Question 8. bonus D'une manière générale, l'utilisation d'un serveur RMI pose de grosses questions quand à l'utilisation d'un GC réparti. Décrivez la *modélisation possible* d'un système de GC réparti utilisable au dessus de la couche RMI (donc sans modification des classes RMI fournies par Java).