SCIENCES SORBONNE UNIVERSITÉ

SRCS : Systèmes Répartis Client/Serveur

TME 6 : Verrouillage d'objet partagé en RMI

Objectifs pédagogiques :

- **Java RMI**
- **synchronisation distante**
- **Déploiement d'objets répartis**

On souhaite mettre en place un mécanisme d'accès en exclusion mutuelle à un objet partagé par un ensemble de clients distants. Le service offert permet de verrouiller ou de relâcher un objet de type quelconque. Lors du verrouillage de l'objet, le service doit renvoyer au client la dernière version de l'objet (donc transmission d'une copie de l'objet). Tant que l'objet n'est pas relâché par un client, les autres demandes de verrouillage non satisfaites doivent être mises en attente. La file d'attente doit obéir à une politique FIFO.

Lorsqu'un client déverrouille l'objet, il envoie au serveur la dernière version de l'objet qu'il a potentiellement modifié. Lors du déverrouillage, l'objet est envoyé au prochain demandeur en attente.

Exercice 1 – Implémentation avec clients fiables

Dans cet exercice, nous allons considérer que les clients relâchent toujours l'objet en temps fini.

Question 1

Dans un package `srcs.rmi.concurrent` écrire l'interface `SharedVariable` paramétrée par une variable de type `T` (généricité) qui sera le type de l'objet manipulé. Quelle contrainte faut-il ajouter sur `T` ?

Question 2

Dans le même package écrire une classe `SharedVariableClassical` qui implémente l'interface `SharedVariab` toujours paramétrable par une variable de type `T` respectant les spécification décrites ci-dessus. La valeur initiale de l'objet sera renseignée via le constructeur. Vous privilégierez la méthode `notifyAll()` et vous vous assurez que l'ordre de d'accès respecte un ordre FIFO.

Afin de pouvoir utiliser les fichiers de tests unitaires JUnit4 qui vous sont fournis, il est nécessaire de programmer une classe mère `SystemDeployer` qui permet de déployer et d'annuler le déploiement l'environnement de chaque test. Pour cela il est nécessaire d'y programmer deux méthodes :

- une méthode annotée `@Before` qui sera exécutée avant chaque méthode de test JUnit. Cette méthode devra lancer un processus pour le rmiregistry (voir la documentation de `ProcessBuilder` et `Process`), déployer une variable partagée de type `Integer` avec une valeur initiale à 0 et enfin qui enregistrera sa référence dans le registry sous le nom indiqué dans le fichier de test.
- une méthode annotée `@After` qui sera exécutée après chaque méthode de test JUnit. Cette méthode annulera le déploiement de la variable partagée et tuera le processus exécutant le rmiregistry.

N.B : pour être certain d'utiliser le processus registry lancé dans la méthode `@Before`, vous devez vous assurer qu'il n'y ai pas d'ancien processus registry résiduel d'une éventuelle ancienne exécution (par exemple si on a tué la JVM avec un Ctrl-C). Pour celà deux solutions s'offrent à vous :

- Ajouter un hook pour l'extinction de la JVM avec la méthode `Runtime.getRuntime().addShutdownHook`
- Exécuter la commande `killall -q rmiregistry` préalablement à la commande `rmiregistry` toujours via l'API `ProcessBuilder` et `Process`.

Question 3

Écrire la classe `SystemDeployer`

Question 4

Faire en sorte que la classe `TestSharedVariableClassical` compile et s'assurer que le test est correct.

Exercice 2 – Implémentation avec clients non fiables

Nous allons à présent considérer que les clients puissent pour une raison quelconque ne **jamais** relâcher l'objet.

Question 1

Quel serait le comportement du système avec l'implémentation de `SharedVariable` de l'exercice précédent ?

Question 2

Coder une nouvelle classe `SharedVariableReliable` implantant `SharedVariable` toujours paramétrable par une variable de type `T` et résolvant le problème identifié à la question précédente. On supposera que décidant de ne pas relâcher la variable ne fera jamais appel à la méthode de libération.

Question 3

Modifier la classe `SystemDeployer` pour pouvoir tester cette nouvelle implantation et s'assurer que le test JUnit `TestSharedVariableReliable` compile et soit correct.