Université Lille 1 Mai 2013
Master mention Informatique – M1
Module : Conception d'applications réparties (CAR)

Examen

Conception d'applications réparties (CAR)

Tous documents papier autorisés. Téléphones, calculatrices et ordinateurs interdits.
Le barème est donné à titre indicatif. 3 heures.
Ce sujet comporte 3 pages.

1 Questions de cours (2 points)

1.1 Expliquer ce que permet de faire le mécanisme dit de sérialisation en Java.
1.2 Quel type d'EJB est le plus naturellement associable à des services REST ? Justifier.

2 Invocation de méthodes distantes (6 points)

2.1 Dans le cadre d'un mécanisme d'invocation de méthodes distantes, donner la définition de la notion de souche cliente. (1 point)

Soit l'interface `CalcultatriceItf` suivante qui fournit deux méthodes permettant respectivement d'additionner et de soustraire deux octets passés en paramètre. Dans un premier temps, on ne s'intéresse pas au résultat dont on considère qu'il sera simplement affiché côté serveur.

```
interface CalculatriceItf {
      void plus( byte x, byte y );
      void moins( byte x, byte y );
}
```

On souhaite pouvoir accéder à distance en utilisant le protocole UDP à des objets implémentant cette interface. On considère que chaque souche cliente définit une instance de la classe `class RemoteRef { InetAddress ip; int port; }` contenant l'adresse IP et le numéro de port permettant de communiquer avec l'objet distant. Cette souche cliente doit pouvoir être exécutée en mode *multi-thread*.

2.2 Écrire en Java ou en pseudo-code la classe correspondant à cette souche cliente. (1,5 point)

On s'intéresse maintenant au côté serveur et on considère qu'il existe une classe `CalculatriceImpl` qui implémente `CalculatriceItf`. Soit le programme `Serveur` avec une méthode principale `main` qui permet de recevoir et traiter en mode *multi-thread* des invocations de méthodes distantes sur le port 1234 pour l'interface `CalculatriceItf`.

2.3 Écrire en Java ou en pseudo-code le programme `Serveur`. (1,5 point)

2.4 On souhaite maintenant prendre en compte le cas où les méthodes de l'interface `CalculatriceItf` renvoient un résultat de type `byte`. Proposer en français une solution pour cela. (1 point).

2.5 On souhaite maintenant prendre en compte le cas où les méthodes de l'interface `CalculatriceItf` déclarent une ou plusieurs exceptions. Proposer en français une solution pour cela. (1 point)

3 MapReduce (5 points)

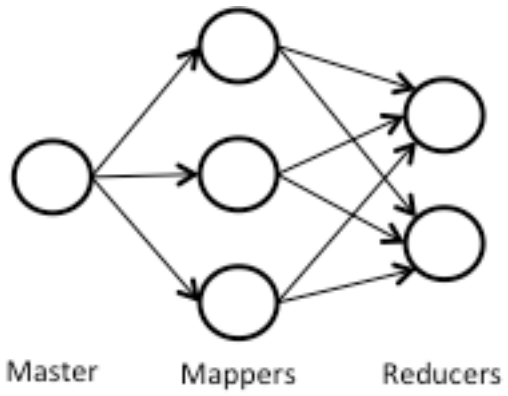
Cet exercice vise à étudier la mise en œuvre d'un service dit MapReduce qui permet d'indexer de façon efficace un grand volume de données. Un tel service est utilisé par des librairies comme Yahoo! Hadoop qui sert de base pour des moteurs de recherche et des systèmes de fichiers distribués.

Le but de l'exercice est de définir un service pour pouvoir dénombrer les occurrences des mots contenus dans `n` fichiers texte. On s'intéresse dans un premier temps à une solution centralisée. Soit l'interface `MapReduceItf` qui définit la méthode `count` prenant en paramètre un tableau de `java.io.File` et retournant une `java.util.Map` dont la clé est un mot et la valeur le nombre total d'occurrences de ce mot dans les `n` fichiers.

3.1 Écrire l'interface Java RMI `MapReduceItf`. (0,5 point)

3.2 Écrire la classe Java RMI implémentant cette interface. Le corps de la méthode peut être écrit en pseudo-code. (1 point)

On s'intéresse maintenant à une version répartie de la mise en œuvre de ce service. Pour cela on divise la tâche de comptage en deux sous-tâches dites `map` et `reduce`, implémentées respectivement par des objets `Mapper` et `Reducer`. Soit l'architecture illustrée dans la figure suivante et contenant 1 objet `Master`, 3 objets `Mapper` et 2 objets `Reducer`. L'objet `Master` distribue 1 fichier parmi `n` à chaque objet `Mapper`. Comme il y a potentiellement plus de 3 fichiers, chaque objet `Mapper` peut être sollicité plusieurs fois. À chaque sollicitation, l'objet `Mapper` compte les occurrences des mots du fichier qu'il reçoit et transmet pour chaque mot le nombre d'occurrences à un des objets `Reducer`. Le choix de l'objet `Reducer` se fait à l'aide d'une méthode `partition` dont on considère dans un premier temps qu'elle est fournie, et qui, à partir d'un tableau de `Reducer` et d'un mot, retourne la référence du `Reducer` à contacter pour ce mot. Chaque objet `Reducer` additionne les décomptes qu'il reçoit pour chaque mot et stocke le résultat dans une `java.util.Map`.



Le décompte est ainsi distribué parmi les objets `Reducer`. Chaque objet `Reducer` fournit une méthode `getMap` qui retourne sa `java.util.Map`. Plus le nombre d'objets `Mapper` est élevé, plus le volume de données traité efficacement va pouvoir être important.

3.3 Écrire les interface Java RMI `MapperItf` et `ReducerItf` des objets `Mapper` et `Reducer`. (1 point)

3.4 Écrire les classes Java RMI implémentant ces interfaces. Le corps des méthodes peut être écrit en pseudo-code. (2 points)

3.5 Proposer en français une implémentation pour la méthode `partition`. (0,5 point)

4 Agenda réparti (7 points)

Cet exercice porte sur la mise en œuvre d'un système d'agendas répartis, multi-utilisateurs et *multi-thread*. Soient des objets `Agenda` et des objets `Utilisateur` représentant respectivement des agendas et des utilisateurs de ces agendas. Les objets `Agenda` stockent des rendez-vous, représentés par des objets `RDV`, et comprennent un identifiant unique (entier), une date (`java.util.Date`), une heure de début (entier) et une heure de fin (entier). Un agenda fournit quatre services permettant respectivement de créer, modifier, lire et détruire un rendez-vous. Les `Agenda` sont des objets accessibles à distance en Java RMI. Les `RDV` sont des objets transmis par copie.

4.1 Proposer une interface Java RMI `AgendaItf` pour les objets `Agenda` et proposer une classe pour les objets `RDV`. Expliquer en français vos choix de conception en rapport avec l'écriture de cette interface et de cette classe. (1 point)

4.2 Dans le cas où ces quatre services sont accessibles via REST, quelles ressources proposez-vous de considérer ? Quelles actions proposez-vous pour ces ressources ? (1 point)

On souhaite pouvoir lister les rendez-vous stockés dans un `Agenda`. Pour cela, un service `list` est défini qui retourne un itérateur. L'itérateur permet de récupérer à distance et un à un les rendez-vous stockés dans un `Agenda`. Chaque itérateur fournit les services `hasNext` (aucun paramètre, retourne un booléen qui vaut vrai si il y a un rendez-vous suivant, faux sinon) et `next` (aucun paramètre, retourne le rendez-vous suivant, ou `null` si il n'y en a pas).

4.3 Proposer une ou plusieurs interfaces Java RMI permettant de mettre en œuvre cette fonctionnalité. (1 point)

On souhaite pouvoir inviter des utilisateurs à un rendez-vous. Chaque utilisateur doit pouvoir à tout moment notifier son acceptation ou son refus de participer au rendez-vous. Il doit pouvoir également changer d'avis. Enfin, si un rendez-vous est modifié ou détruit, les utilisateurs l'ayant accepté doivent être prévenus.

4.4 Proposer une ou plusieurs interfaces Java RMI permettant de mettre en œuvre ces fonctionnalités. (1 point)

4.5 Écrire la classe Java RMI implémentant un `Agenda`. Le corps des méthodes peut être écrit en pseudo-code. (3 points)