Institut Informatique Claude Chappe

Master Informatique
Unité 178UD02
Programmation distribuée

TP n°2
Middleware Java RMI

Objectifs du TP :

- Introduction à la programmation des applications distribuées par objets distants.
- Mise en œuvre du middleware Java RMI
- Etudier une application existante et finir son développement

Sujet : réalisation d'un serveur de tâche

Présentation

Le but de ce TP est de développer un serveur de tâche. Une telle application distribuée est constituée :

- d'un serveur qui **effectue** des tâches exigeantes en ressource de calcul ;
- d'un ou plusieurs clients qui **définissent** des tâches, les **donnent à exécuter** au serveur et **récupère** de différentes manières le résultat de l'exécution.

Le client peut demander de faire exécuter une tâche de trois manières différentes :

- Soit immédiatement (exécution synchrone) : le client est bloqué en attendant que la tâche soit achevée d'être exécutée par le serveur qui lui retourne directement le résultat.
- Soit au plus tôt avec priorité : le client envoie une tâche avec un niveau de priorité devant être exécutée dès que le serveur est libre. Le serveur maintiens à jour une liste ordonnée de ce type de tâche selon le niveau de priorité. Il exécute la tâche la plus prioritaire en premier lieu.
- Soit de façon temporisée avec priorité : le client envoie une tâche à exécuter à une certaine heure et avec un certain niveau de priorité. Le serveur stocke les tâches de ce type dans une file ordonnée (selon l'heure et selon la priorité) en attendant d'exécuter la tâche la plus prioritaire de la file au moment indiqué. Entre deux tâches devant être exécutée au même moment, c'est celle qui est la plus prioritaire qui est exécutée en premier.

Dans les deux derniers types d'exécution nous avons un fonctionnement asynchrone. Le serveur enregistre la demande d'exécution puis, ultérieurement quand il est libre, exécute et renvoie le résultat de l'exécution au client.

L'application s'appuie sur l'exemple donné dans *The Java RMI Tutorial* (référence ci-après) qui a été étendu pour répondre aux spécifications du TP. En particulier, on a réutilisé le code de calcul du nombre Pi donné comme exemple de tâche dans le tutoriel.

Travail à réaliser

Il faut **finir le développement** de l'application distribuée en vous aidant des explications données ci-après, du diagramme de conception ci-joint et du code source déjà développé (accessible sur UMTICE).

Les classes à modifier sont : *DComputeEngine*, *FAPComputeEngine, TComputeEngine*.

Le compte-rendu de TP sera déposé sur UMTICE **en fin de séance** sous la forme d'un fichier compressé (format zip) contenant :

- L'ensemble des fichiers utiles pour compiler et exécuter l'application répartie ;
- Un document comportant les réponses aux trois questions suivantes :
  - Pourquoi l'interface *ITask* ne peut pas hériter de l'interface *ITaskObserver* ?
  - Pourquoi les instances de la classe *TaskObserverImpl* n'ont pas besoin d'être publiées alors que l'instance de la classe *ComputeEngineFactory* doit être publiée dans un registre RMI ?
  - Expliquez en quelques lignes le fonctionnement de l'exécution d'une tâche synchrone en indiquant les différents objets et méthodes utilisées.

Compilation de l'application

Compilation du serveur :

- Allez dans le répertoire *appserver*
- Compilez avec la commande : *javac compute/*.java engine/*.java*
- Créez un fichier archive avec la commande : *jar cf compute.jar compute/*.class*
- Déplacez le fichier *compute.jar* dans le répertoire *appclient*

Compilation du client :

- Allez dans le répertoire *appclient*
- Compilez avec la commande : *javac -cp compute.jar client/*.java*

Exécution de l'application

- Modifiez les fichiers *policy* pour qu'ils correspondent à votre environnement d'exécution.
- Dans un terminal, dans le répertoire *appserver,* lancez le service de nommage *rmiregistry*.
- Dans un 2ième terminal, dans le répertoire *appserver,* lancez : *sh golocal.sh*.
- Dans un 3ième terminal, dans le répertoire *appclient,* lancez : *sh golocal.sh 1 2 3 4 5*.
- Stoppez avec Ctrl + C : le client puis le serveur puis rmiregistry

Conception de l'application

Pour concevoir les applications client et serveur, il faut répondre aux spécifications techniques suivantes :

- Mettre en place un objet distant d'exécution des tâches. Cela est pris en compte avec les classes suivantes :
  - *ICompute* : interface distante d'exécution d'une tâche. Elle déclare trois services d'exécution correspondant aux trois types d'exécution possibles.
  - *ITask* : interface de déclaration d'une tâche. Elle déclare trois services dont celui d'exécution et d'obtention de son observateur.
  - *ComputeEngineFactory* : classe d'implémentation de l'interface distante *ICompute* sur le serveur.
  - *Client* : un exemple de classe d'implémentation du client. Il crée les tâches et leurs notificateurs associés puis demande au serveur de les exécuter. La tâche est alors passée par copie au serveur pour qu'il l'exécute. Le client peut choisir entre trois modes d'exécution.
  - *Pi* : un exemple de classe d'implémentation de l'interface *ITask*. Cette classe connaît son notificateur pour le donner au serveur.

- Mettre en place une fabrique d'exécution au niveau du serveur pour gérer facilement les différents types d'exécution : la classe *ComputeEngineFactory*. Cette classe crée à la volée les instances de classes qui implémentent les trois types d'exécution selon le service appelé par le client.

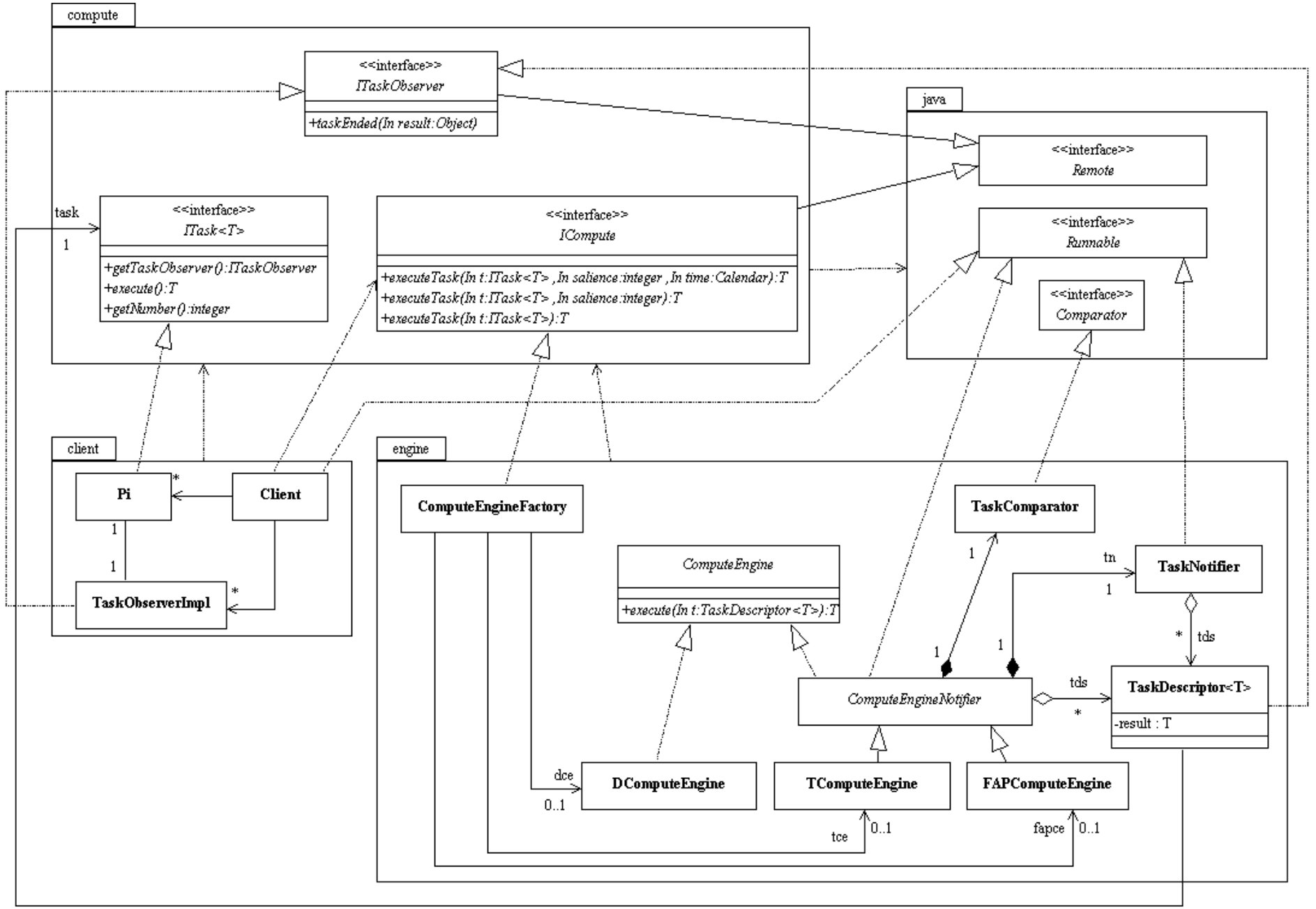
- Mettre en place les différents types d'exécution. Cela est pris en compte avec les classes suivantes :
  - *TaskDescriptor* : classe de description d'une tâche. Cette classe encapsule la tâche à exécutée ainsi que les informations liées à son exécution (priorité et moment d'exécution s'ils existent). Elle stocke également la valeur de retour de l'exécution de la tâche. En terme de comportement, elle a la charge de notifier le client de la fin d'exécution de la tâche qu'elle décrit.
  - *ComputeEngine* : interface générale à tout type d'exécution. Le service prend en argument un descripteur de tâche et retourne un résultat (qui vaut *null* dans le cas des tâches asynchrones).
  - *ComputerEngineNotifier* : classe abstraite pour les exécutions asynchrones. Elle possède un comparateur et une liste ordonnée de descripteurs de tâche. On utilise la classe *java.util.concurrent.PriorityBlockingQueue*.
  - *TaskComparator* : le comparateur utilisé pour ordonner la liste précédente.
  - *DComputeEngine* : classe d'implémentation de l'exécution synchrone.
  - *TComputeEngine* : classe d'implémentation de l'exécution asynchrone temporisée de tâche à priorité. Elle exécute la tâche du descripteur en tête de liste quand c'est le moment et après l'en avoir retiré. Elle stocke le résultat dans le descripteur puis l'ajoute au *TaskNotifier*. L'exécution temporisée est considérée comme juste si elle s'effectue dans un intervalle de plus ou moins 1 min par rapport au temps demandé.
  - *FAPComputeEngine* : classe d'implémentation de l'exécution asynchrone au plus tôt de tâche à priorité. Elle exécute la tâche du descripteur en tête de liste après l'en avoir retiré. Elle stocke le résultat dans le descripteur puis l'ajoute au *TaskNotifier*.

- Mettre en place un mécanisme de « callback » pour l'exécution asynchrone en appliquant le pattern *Observer*. Ce mécanisme permet au client d'être notifié de la fin de l'exécution d'une tâche par le serveur. Une fois notifié, le client peut récupérer le résultat de la tâche achevée. Ce pattern est mis en œuvre avec les classes suivantes :
  - *ITaskObserver* : interface de déclaration du service d'observateur de fin de tâche. Elle est implémentée à la fois du côté client et du côté serveur. Cette classe définit également un objet distant car le serveur doit exécuter ce code sur le client.
  - *TaskNotifier* : classe d'implémentation de l'algorithme de notification. Il enregistre ou enlève des observateurs (classes qui implémentent *ITaskObserver*) d'une liste (ici de descripteur de tâche). Il vide la liste au fur et mesure qu'il notifie le descripteur.
  - *TaskDescriptor* : classe qui notifie le client en passant comme argument le résultat de l'exécution de la tâche qu'il encapsule.
  - *TaksObserverImpl* : classe d'implémentation côté client de l'interface de notification de fin de tâche. Cette classe exécutée sur le client est appelée par le serveur et elle connaît la tâche qu'elle doit notifier.

Références web

The Java RMI Tutorial http://docs.oracle.com/javase/tutorial/rmi/overview.html

Dynamic code downloading using RMI
http://docs.oracle.com/javase/6/docs/technotes/guides/rmi/codebase.html


compute
<<interface>>
ITaskObserver
+taskEnded(In result:Object)
java
<<interface>>
Remote
<<interface>>
Runnable
<<interface>>
Comparator
task
1
<<interface>>
ITask<T>
+getTaskObserver():ITaskObserver
+execute():T
+getNumber():integer
<<interface>>
ICompute
+executeTask(In t:ITask<T>, In salience:integer, In time:Calendar):T
+executeTask(In t:ITask<T>, In salience:integer):T
+executeTask(In t:ITask<T>):T
client
Pi
Client
TaskObserverImpl
engine
ComputeEngineFactory
ComputeEngine
+execute(In t:TaskDescriptor<T>):T
TaskComparator
TaskNotifier
tn
tds
ComputeEngineNotifier
TaskDescriptor<T>
-result : T
dce
0..1
DComputeEngine
TComputeEngine
FAPComputeEngine
tce
fapce