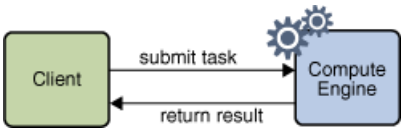
Programmation et modélisation orientées objet
(USAL5A & USAL2H)

Serveur d'exécution

Le but de ce TP est d'implanter un serveur d'exécution (*Compute Engine*)[1] qui permet de faire exécuter des « tâches » à distance le client crée une tâche, l'envoie au serveur, qui l'exécute et renvoie le résultat au client.



Une tâche devra simplement implanter l'interface suivante :

```
interface Task<T> extends Serializable {
    T execute();
}
```

1 Implantation

Le client et le serveur d'exécution s'exécutent de manière concurrente et communiquent à travers des canaux. On donnera 3 implantations complètes, dans des packages séparés :

1. **Version locale.** Le client et le serveur d'exécution tournent dans des *threads* de la JVM et les canaux sont implantés en utilisant la classe `BlockingQueue<A>` de la bibliothèque standard.

2. **Version distante.** Le client et le serveur d'exécution tournent dans des JVM différentes et les canaux utilisent les sockets et la sérialisation d'objet. On utilisera par défaut le port 55000.

3. **Version unifiée.** On introduit des interfaces pour s'abstraire de l'implantation des canaux (*subpackage* `channel` fourni), ce qui permet d'avoir un seul code pour le client et le serveur, qui peuvent être déployés soit localement soit à distance.

Le code se divise en plusieurs classes :

- Une classe `ComputeEngine<T>` qui correspond au serveur d'exécution. La méthode `run` attend une tâche, l'exécute, et renvoie le résultat.
- Une classe `RemoteComputeEngine<T>` qui correspond à l'API client, et contient la méthode suivante :

  ```
  public T executeTask(Task<T> t) throws IOException;
  ```

- Une classe `Client` qui implante `Runnable` et dont la méthode `run` exécutera une tâche simple (une instance de `SimpleTask`).
- Une classe `SimpleTask` qui implante `Task<Integer>` pour vos tests : une instance de `SimpleTask` permettra de faire la somme de deux entiers.
- (*optionnel*). Modifier le serveur d'exécution pour autoriser l'exécution de plusieurs tâches. Pour demander l'arrêt du serveur, le client enverra une instance d'une tâche spéciale appelée `Stop<T>` dont la méthode `execute` renvoie toujours `null`.

1. https://docs.oracle.com/javase/tutorial/rmi/designing.html

Et enfin les classes suivantes, selon les versions :

- **Version locale.** Une classe `Main` qui lance à la fois le client et le serveur.
- **Version distante.** Une classe `ClientMain` qui lance le client et une classe `ComputeEngineMain` qui lance le serveur le serveur.
- **Version unifiée.** Les classes `ClientMain`, `ComputeEngineMain` et `Main` adaptées pour utiliser le *subpackage* `channel`. Les classes `ClientMain` et `ComputeEngineMain` utilisent les adaptateurs « distants » et la classes `Main` utilise les adaptateurs « locaux ».

Remarques.

- Pour passer de la version locale à la version distante, il faut remplacer la classe `BlockingQueue` par les classes `ObjectInputStream` et `ObjectOutputStream` et les méthodes `put` et `take` de `BlockingQueue` par `writeObject` et `readObject` (et enfin `InterruptedException` par `IOException`).

  Il faut de plus introduire des *casts* à chaque utilisation de `readObject` car les classes `ObjectInputStream` et `ObjectOutputStream` ne sont pas génériques.

- Pour passer à la version unifiée, il faut remplacer les classes `ObjectInputStream` et `ObjectOutputStream` par les interfaces fournies `ObjectReader` et `ObjectWriter` qui sont génériques (les *casts* ne sont donc plus nécéssaires).

A Package channel