Conception de logiciels intranet : intergiciels à objets répartis (NSY102)

RPC et RMI

Les RPC (*Remote Procedure Call*) sont des mécanismes d'appel de procédure ou fonction à distance. Ces mécanismes ont spécifiés et implanté au départ pour des langages procéduraux (par exemple, XDR/RPC pour le C sous Unix est déjà décrit dans la RFC 1057 de Juin 1988).

Les RMI (*Remote Method Invocation*) correspondent à la généralisation des RPC aux langages objets, donc à l'invocation de méthode à distance, et nous allons voir qu'il ne s'agit pas simplement d'une simple adaptation de la syntaxe des appels.

En effet, un objet distant est représenté par un « mandataire », ou *proxy* (qui généralise la notion de « souche » ou *stub* de la terminologie RPC), sur la machine locale, et ce *proxy* peut lui-même être transmis à travers une communication réseau (car il est aussi sérialisable). Par conséquent de nouvelles abstractions deviennent possible :

- invoquer une méthode d'un objet se trouvant sur une autre machine (équivalent RPC) :

```
someData = remoteObject.method(anotherData);
```

- passer un objet distant en argument d'un appel local ou distant :

```
someObject = localObject.method(remoteObject);
someData = remoteObject.method(anotherRemoteObject);
```

- récupérer un objet distant comme résultat d'un appel distant :

```
anotherRemoteObject = remoteObject.method(someData);
```

- enregistrer/rechercher un objet distant dans un « registre » (pour l'initialisation) :

```
remoteObject = Registry.lookup("someObject");
```

Le but de ce TP est d'implanter une version simplifiée (et partielle) des RMI de la JDK, autorisant toutefois les différents types d'appels distants décrits ci-dessus, et incluant un registre. Cette implantation devra permettre, en particulier, de tester l'exemple du serveur d'exécution (*Compute Engine*)[1] présenté dans le tutoriel d'Oracle sur les RMI.

Implantation des RMI [8 pts]

1. **(version simple).** Dans cette première version, nous supposons qu'un objet accessible à distance implante nécessairement l'interface `Remote` suivante :

```
public interface Remote {
   default void close() throws Exception {}
   Object method(Object arg) throws Exception;
}
```

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

Compléter le code du serveur (classe `RemoteServer`) et le code du *proxy* (classe `RemoteObjectInvocationHandler`) et le méthodes statiques de `UnicastRemoteObject` :

```
static Remote exportObject(Remote o, int port)
static void unexportObject(Remote obj) throws Exception
```

La méthode `exportObject` permet de créer un nouvel objet distant : elle crée à la fois le *proxy* et le serveur, lance le serveur et renvoie le *proxy*) et la méthode `unexportObject` permet de terminer l'objet distant (i.e. de mettre fin aux appels distants sur cet objet et donc terminer la boucle du serveur). Par convention, pour terminer, la méthode `close` invoquera `method` avec l'argument `null`.

2. **(version intermédiaire)**. Afin d'autoriser plusieurs méthodes (ayant éventuellement plusieurs arguments) dans une interface distante, on transforme l'interface `Remote` ainsi :

```
public interface Remote {
   default void close() throws Exception {}
   Object invoke(String name, Class<?>[] types, Object[] args)
          throws Exception;
}
```

Modifier le code du serveur et du *proxy* de manière à communiquer aussi le nom de la méthode, les types des arguments puis les arguments. Par convention, pour terminer, la méthode `close` invoquera la méthode `invoke` avec `"close"` comme nom de méthode (et `null` pour les autres arguments).

3. **(version finale)**. Pour autoriser une syntaxe conventionnelle pour les appels distants côté *proxy*, il faut soit utiliser un outil pour générer des « souches » (*stubs*) dans le style du générateur `rmic` (maintenant déprécié), soit utiliser l'introspection. Nous utiliserons l'introspection dans cette version, le code correspondant est fourni (il s'appuie sur les classes de la JDK `java.lang.reflect.Proxy` côté *proxy* et `java.lang.reflect.Method` côté serveur). L'interface de `Remote` devient maintenant simplement :

```
public interface Remote {
   default void close() throws Exception {}
}
```

Remarques.

- La méthode `close` n'est pas présente dans l'interface `Remote` des RMI de la JDK, nous la conservons toutefois pour simplifier le protocole.

- La méthode `toString` est nécéssaire pour les logs du serveur (elle est donc traitée par le *proxy*). Pour compléter l'implantation, il faudrait aussi ajouter `equals` et `hashCode` en se conformant à la sémantique des objets distant décrite dans la documentation de la classe `RemoteObject` de la JDK (cette extension sort du cadre de ce projet).

- Le code à compléter pour cette partie est *le même que pour la partie* 2. Cette version est là pour vous simplifier l'écriture des programmes d'exemples.

Implantation du registre [2 pts]

Le registre est un dictionnaire distant qui associe un nom (clé de type `String`) à un objet distant (valeur de type `Remote`). Il est donc possible de réutiliser l'implantation d'un dictionnaire distant (`RemoteMap`) vu dans les TP précédents pour implanter (partiellement) l'interface `Registry` du package `java.rmi.registry` (méthodes `rebind` et `lookup`).

Une classe `Binding<K, V>` utilisant un délégué de type `RemoteMap<K, V>` (version distante, qui réalise une seule opération par connection) est aussi fournie dans le package `binding`. Les classes `Registry` et `RMIRegistry` sont ensuite définie ainsi :

```
class Registry extends Binding<String, Remote> {...}
class RmiRegistry extends Server<String, Remote> {...}
```

On se donnera aussi une classe `LocateRegistry` avec une méthode statique `getRegistry` qui renvoie le registre distant en spécifiant éventuellement le nom du serveur (qui sera `localhost` par défaut) et/ou le numéro de port (qui sera 55000 par défaut).

Programmes d'exemples [10 pts]

- Tester votre implantation sur l'exemple du serveur d'exécution (*Compute Engine*), qui est fourni pour les 3 versions. Compléter les tests pour la version 3 avec des nouveaux exemples de tâches différentes.

  Remarque. Une classe `Main` qui réalise les différentes étapes permettant tester l'exemple du serveur d'exécution est donnée (et vous pouvez l'adapter à vos propres exemples). Les differents serveurs sont lancés localement sous forme de *threads*, mais les communications utilisent les *sockets* (et donc la sérialisation).

- Proposer d'autres exemples (que le serveur d'execution) utilisant vos RMI (version 3) permettant d'exploiter la possibilité de passer des objets distants en paramètre ou en résultat d'un appel distant.