TD 9 - Cooking with RMI / Asynchronous RMI

Exercice 1 – Homegrown RMI

Comme nous sommes au stade du hardcore coding tendance asociaux anarchiques, nous allons développer notre propre service RMI en Java, sans utiliser aucune des classes proposées par le JDK. Nous allons donc développer la couche faisant le lien entre le serveur et le client (`Remote Reference Layer`). Plus précisément, nous allons écrire le code permettant de déléguer l'appel fait depuis le client au travers d'un socket vers notre serveur maison. Nous allons devoir coder un équivalent au registre RMI basique ainsi que de quoi instancier nos stub et skeleton et enfin répondre aux appels distants.

Question 1

Faire la liste des modules et fonctions qui seront nécessaires pour effectuer notre implémentation maison du protocole RMI.

Question 2

Nous allons publier le même service qu'avec l'implémentation RMI de Sun : nous allons donc garder nos interfaces de la séance précédente. Comme nous faisons tout nous-même, notre implémentation ne nécessitera pas d'utiliser l'interface `Remote` ni d'expliciter `RemoteException` comme exception propageable. Réecrire les classes et interfaces dont nous nous servirons pour les appels distants.

Question 3

Dans un premier temps, on se rends compte que avant de pouvoir effectuer l'appel distant, il va falloir d'abord faire traverser un socket à notre appel. Le principe est donc de regrouper tout ce dont nous avons besoin dans un objet chez le client pour réaliser notre invocation (nom du service, méthode à invoquer, et les valeurs des paramètres). Nous allons regrouper cela dans une classe sérialisable qui traversera le socket. Cette classe sera instanciée par notre stub, envoyée dans le socket, et traitée par le skeleton.

Question 4

Lorsque le client effectue un appel sur un objet distant, nous devons d'une certaine manière attraper cet appel. Il nous faut donc définir une classe dérivant de `java.lang.reflect.InvocationHandler` (permettant des appels de fonctions sur objets de manière dynamique). Ceci produira une implémentation dynamique de notre stub, permettant ainsi de simuler un appel distant transparent pour l'utilisateur. Écrire un handler d'appel `RMIHandler`, créé sur un couple de flux d'objets (entrant et sortant) donné et qui instanciera simplement un `InvocationContext`, l'enverra dans le socket, et retournera ce qui revient par le socket.

Question 5

Lors des invocations, écriture des flux et sérialisation des objets, de multiples erreurs peuvent se produire (notamment au niveau des échanges résaux mais également des appels de fonctions). Comme nous sommes toujours dans l'optique

de faire notre implémentation maison, nous allons donc créer notre propre exception spécifique à notre système d'appel distant (semblable à `RemoteException`), permettant de réunir toutes les erreurs. Quelle serait l'héritage le plus approprié pour cette exception ? Écrire la classe `RemoteInvocationException`.

Question 6

Nous allons maintenant attaquer la couenne du problème, c'est à dire le registre. Comme l'écriture d'un registre RMI est relativement complexe, nous allons considérer ce problème fonction par fonction en supposant que toutes les fonctions feront finalement parties d'une classe `pc2r.Registry`. Écrire la fonction `register`, équivalente à la fonction RMI `bind`, en précisant bien quel type de structure de données vous comptez utiliser.

Question 7

Avant de passer à l'implémentation du serveur à proprement parler, il nous faut interpréter les arguments envoyés par les objets `InvocationContext`, notamment en termes de type (vu qu'on utilise ici une liste d'objets). On utilise pour ce faire la fonction `args2Class` permettant de construire la liste des types des arguments de l'invocation (le code vous est donné à la fin de la question). Ceci nous permettra d'utiliser les fonctionnalités d'appel dynamique JAVA (fonctions `getMethod()` et `invoke()`). Grâce à ces fonctions, écrire le code du serveur permettant de gérer les appels distants côté serveur (on considèrera qu'on réutilise le code d'un serveur précédent et on se contentera ici de modifier la fonction `run()` des threads clients)

```
private Class[] args2Class (Object[] objs)
{
 List<Class> classes = new ArrayList<Class>();
 for (Object o : objs)
  { classes.add(o.getClass());}
 return classes.toArray(new Class[]{});
}
```

Question 8

Il ne nous reste plus qu'à gérer la méthode `get` côté client qui lui permettra de récupérer le stub d'un objet de manière dynamique. Nous allons utiliser le `ClassLoader` Java pour permettre d'effectuer un lien dynamique avec notre `RMIHandler`. Écrire la fonction `get`.

Question 9

Après toutes ces péripéties, nous touchons au but et n'avons plus qu'à implémenter le serveur distant qui se servira de notre registre. Écrire la classe `RemoteServer`. Quelles différences faut-il prendre en compte vis-à-vis du RMI Sun ?

Question 10

Nous allons enfin pouvoir tester notre système d'appel distant. Écrire la classe `RemoteClient`. Quelles différences faut-il prendre en compte vis-à-vis du RMI Sun ?

TP - Remote Callback - Distant Process

Exercice 2 – Le Pattern `RemoteCallBack`

Les appels effectués par le biais de RMI peuvent etre long. Il est souhaitable pour un client de continuer à être réactif même en attendant la réponse d'une `Remote` méthode. Ainsi on souhaite qu'un serveur puisse appeller des `callback` sur le client pour le notifier d'évènements.

Afin d'implanter ce mécanisme de rappel(callback), il est nécessaire de définir un objet de rappel coté client dont une remote reference fournie au serveur lui permttra d'interagir avec le client. En d'autre termes cet objet de rappel est créé par le client comme objet distant et sert d'intermédiaire, *de messager*, portant la réponse du serveur au client. L'intérêt d'un tel mécanisme est de permettre de rendre la communication *asynchrone*. L'objet de rappel dispose typiquement de 5 méthodes :

- Deux pour le client : `boolean is_finished()` qui permet de savoir si le calcul est terminé et `TypeResultat getResult()` qui permet ensuite de récupérer le résultat.
- Trois pour le serveur : `void put(TypeResultat)` qui dépose le résultat à la fin du calcul `void finish()` qui indique que le résultat vient d'être déposé et prévient tous les threads en attente éventuelle du résultat `unreferenced` pour la gestion explicite des références d'objets distants, remplacant en quelque sorte le ramasse miette (GC) local L'interface `IRemoteCalculette` et la classe `RemoteCalculette` demeurent inchangées.

Question 1

La calculette est désormais considérée comme une sorte de caisse enregistreuse. A ce titre, tout nombre fourni à la méthode `cumuler` est stocké dans un vecteur (`class java.util.Vector`). Modifier la calculette pour que l'opération `getCumul` devienne *longue* (au sens de prendre beaucoup de temps).

A partir de maintenant nous allons définir un `Remote` objet sur lequel une `Remote` référence pourra être donnée à une autre JVM. Ca sera lui notre objet de callback.

Question 2

Ecrire la classe `RappelResRMI`, décrite plus haut, destinée à porter le résultat de la méthode `getCumul`, cette classe sera instanciée coté client et aura deux membres prives :

- `result`, un entier qui sera le resultat.
- `f` , un booleen qui sera positionné a vrai lorsque le calcul cote serveur sera terminé.

Écrire maintenant l'interface `IRappelResRMI` destinée à être transmise au serveur par le client (On prendra bien soin de n'exporter que les méthodes *vraiment nécessaires*)

Question 3

Créer une nouvelle interface `IRappelCalculette` ajoutant la methode `getCumulR (IRappelResRMI)` -pour passer l'objet temporaire de rappel- et créer aussi une nouvelle classe `RappelCalculette` implantant cette interface. Les clients distants de RappelCalculette appellent maintenant méthode `getCumulR (IRappelResRMI)` pour obtenir le résultat du cumul (et non plus la méthode `getCumul()`).

Question 4

Pour que le client puisse, en théorie, réaliser d'autres tâches sans attendre la réponse calculée par le serveur d'objet, la méthode `getCumulR` va démarrer un *thread* `TcalculCumul` qui prend en charge le travail. Pour ce faire, il devra nécessairement avoir accès à la `calculette` ainsi qu'au messager `IRappelResRMI` (qui lui seront donc passés en paramètre). Ecrire la classe de `Thread TcalculCumul` dont la méthode `run` :

- s'endort un peu (pour rallonger artificiellement l'opération)
- appelle `getCumul()` sur l'objet réel du serveur

– définit le résultat et marque l'opération comme terminée dans le messager coté client (par l'intermédiaire de son représentant coté serveur : méthode `RappelResRMI.finish()`)

Question 5

Quelle(s) modification(s) apporter dans le serveur ? Ecrire le serveur nommé `ServRappelCalculette.java`.

Question 6

Modifier le client pour appeler `getCumulR()` et attendre que le résultat soit disponible, c'est-à-dire que `isFinished()` du messager renvoie vrai. On portera une attention particulière au fait que l'attente active doit être évitée (sinon le mécanisme de rappel n'est pas utile).