RMI

Fabrice Rossi

4 juin 2003

1 Connexion personnalisée

Quand on accède à un serveur représenté par un objet RMI identifié par un nom dans le *registry*, la connexion obtenue ne tient pas compte de l'identité du client. Quand on doit personnaliser les services fournis, il faut construire un moyen d'identifier le client. Il ne s'agit pas ici de fournir un grand niveau de sécurité, mais simplement de faciliter la programmation du client et du serveur en permettant des connexions personnalisées.

Pour illustrer ce problème, on propose un modèle très simple dans lequel le serveur affiche (seulement côté serveur) le message qui lui est transmis pas le client, en préfixant le texte par l'identité du client (par exemple son nom d'utilisateur).

1.1 Solution par identifiant

Exercice 1

La solution la plus classique (mais la moins objet) se base sur l'attribution d'un identifiant temporaire. On considère l'interface suivante pour le serveur :

IServeurIdent

```
1 import java.rmi.*;
2 public interface IServeurIdent extends Remote {
3   public long login(String user,String pass) throws RemoteException;
4   public void logout(long id) throws RemoteException;
5   public void postMessage(long id,String msg) throws RemoteException;
6 }
```

Le principe est très simple : quand un client veut se connecter, il obtient une référence distante sur une implémentation de `IServeurIdent` par l'intermédiaire du *registry*. Il appelle ensuite la méthode `login` pour obtenir un identifiant temporaire, représenté par un `long`. Si le login est impossible, la méthode renvoie naturellement une exception.

Une fois le client connecté, toute interaction avec le serveur passe par l'utilisation de l'identifiant temporaire, en particulier la méthode `postMessage` qui provoque l'affichage du message sur le serveur. La méthode `logout` invalide l'identifiant temporaire.

Questions :

1. Implémenter une classe `UserInfo` qui sera utilisée par le serveur pour représenter un utilisateur connecté. Elle devra donc comporter le nom de l'utilisateur, l'identifiant temporaire associé à celui-ci et la date d'attribution de cet identifiant (pour gérer le *time out*).
2. Implémenter le serveur, en prenant soin de respecter les consignes suivantes :
   - les identifiants doivent être uniques (deux utilisateurs connectés ne peuvent pas avoir le même identifiant) ;
   - on doit pouvoir associer facilement un identifiant à un utilisateur ;
   - le serveur doit vérifier l'age de la connexion avant d'autoriser une action par le client.

Le serveur devra proposer une méthode locale permettant d'ajouter des utilisateurs enregistrés (sous forme de couples nom d'utilisateur et mot de passe). Au démarrage du serveur, on ajoutera quelques utilisateurs pour les tests.

3. Implémenter un client en ligne de commande permettant de tester le système. L'idéal est d'afficher l'identifiant obtenu au login afin de pouvoir relancer plusieurs fois le client sans avoir à se reconnecter à chaque lancement.

1.2 Solution par objet dédié

Exercice 2

La solution proposée dans l'exercice précédent n'est pas orientée objet et expose un peu trop les détails techniques. Une solution plus simple s'obtient en créant des objets dédiés. Pour ce faire, on sépare la connexion des services proprement dits. On obtient ainsi deux interfaces :

IServeurLogin

```
1 import java.rmi.*;
2 public interface IServeurLogin extends Remote {
3   public IServeur login(String user,String pass) throws RemoteException;
4 }
```

IServeur

```
1 import java.rmi.*;
2 public interface IServeur extends Remote {
3   public void postMessage(String msg) throws RemoteException;
4 }
```

Du côté client, la nouvelle solution s'utilise de la façon suivante : on récupère grâce au *registry* une référence distante sur une implémentation de `IServeurLogin` (il y a une seule instance de cet objet). Le client appelle la méthode `login` qui lui renvoie une référence distante sur une implémentation de `IServeur`. Toute l'interaction se fait alors avec `IServeur`. L'idée est que chaque appel de `login` produit un nouvel objet implémentant `IServeur`.

Questions :

1. Écrire une classe `Display` qui sera utilisée sur le serveur pour produire les affichages (par l'intermédiaire d'un singleton). Cette classe n'a aucun intérêt pratique (on pourrait faire un appel direct à `System.out.println`), mais elle permet de simuler la mise en place d'un serveur complexe.
2. Écrire une implémentation de `IServeur` qui conserve l'identité du client qui l'utilise et qui passe par le singleton `Display` pour faire ses affichages.
3. Écrire une implémentation de `IServeurLogin`. Comme dans l'exercice précédent, on préparera quelques utilisateurs enregistrés pour les tests.
4. Implémenter un client en ligne de commande permettant de tester le système. Comme il faut conserver ici la référence sur l'objet distant pour que le mécanisme fonctionne, il faudra que le client effectue une série d'interactions pour tester réellement le serveur.

2 Communication asynchrone

RMI est un mécanisme d'appel de méthodes à distance, ce qui correspond par défaut à un modèle synchrone : le client attend que le serveur lui renvoie une réponse. Ce modèle fonctionne parfaitement mais est très restrictif, en particulier dans les situations où le traitement sur le serveur peut être relativement long. Nous allons étudier dans cette section les différentes techniques qui permettent de construire un modèle asynchrone au dessus de RMI.

2.1 Solution par *threads* sur le client

Si le serveur n'offre aucun support pour un appel asynchrone, la seule solution est de passer par un *thread* sur le client. Nous allons donc commencer cette section par quelques exercices sur les threads.

2.1.1 Manipulations élémentaires de threads

Il y a deux façons de créer un thread en Java :

1. en écrivant une classe qui hérite de la classe `Thread` et en plaçant le code à exécuter dans la méthode `run`, comme dans l'exemple suivant :

ThreadClasse
```
1  public class ThreadClasse extends Thread {
2    public void run() {
3      // code à exécuter dans le thread
4    }
5  }
```

On exécute le thread en écrivant :

MainThreadClasse
```
1  public class MainThreadClasse {
2    public static void main(String[] args) {
3      ThreadClasse monThread = new ThreadClasse();
4      monThread.start();
5    }
6  }
```

2. en écrivant une classe qui implémente l'interface `Runnable`, en plaçant le code à exécuter dans la méthode `run` et en construisant un objet `Thread` de support, comme dans l'exemple suivant :

ThreadInterface
```
1  public class ThreadInterface implements Runnable {
2    public void run() {
3      // code à exécuter dans le thread
4    }
5  }
```

On exécute le thread en écrivant :

MainThreadInterface
```
1  public class MainThreadInterface {
2    public static void main(String[] args) {
3      ThreadInterface monCode = new ThreadInterface();
4      Thread monThread = new Thread(monCode);
5      monThread.start();
6    }
7  }
```

Dans les deux cas, la méthode `start` rend tout de suite la main au thread appelant (le thread principal dans le cas de la méthode `main`). Il est impossible de changer le prototype de `run` ou de `start`, ce qui signifie qu'un thread ne peut pas communiquer directement avec un autre, il faut passer par des moyens détournés.

Exercice 3

Écrire un programme Java comportant deux threads (en plus du thread principal) qui affichent chacun la liste des entiers compris entre 1 et `n`, où `n` désigne le paramètre de leur constructeur. On pourra introduire une pause aléatoire entre deux affichages pour bien visualiser l'ordonnancement entre les threads.

2.1.2 Synchronisation basique

Quand on appelle la méthode `start` d'un thread, celui-ci démarre et la méthode rend tout de suite la main, sans attendre la fin de l'exécution de la méthode `run`. Dans certaines circonstances, il est utile qu'un thread puisse se synchroniser avec un autre. Excepté pour des cas très simples, cela nécessite une coopération entre les threads, les méthodes `stop`, `suspend` et `resume` de la classe `Thread` ayant été abandonnées depuis longtemps.

La primitive la plus simple pour la synchronisation est la méthode `join`. Si `t` est un `Thread`, l'appel `t.join()` bloque le thread appelant (à ne pas confondre avec `t`) jusqu'à ce que `t` se termine. La méthode `join` possède des variantes permettant d'indiquer un temps maximal d'attente. De plus toutes les versions peuvent renvoyer l'exception `InterruptedException` qu'il faut donc prendre en compte.

Exercice 4

Modifier le programme de l'exercice précédent pour faire afficher un message à la méthode `main` quand les deux threads de comptage sont terminés.

Une synchronisation moins rudimentaire demande un support complet par le programmeur. Il faut par exemple que la méthode `run` du thread teste régulièrement la valeur d'une variable (accessible depuis un autre thread) pour savoir si elle doit continuer à fonctionner. Dans un premier temps, nous allons réaliser une attente active (ce qui est une mauvaise idée, cf la section suivante).

Exercice 5

Modifier le thread de l'exercice 3 pour incorporer les fonctionnalités suivantes :
- une variable privée du thread, `actif`, indique si celui-ci doit tourner ou non. Cette variable est initialisée à `true` (le thread tourne) par le constructeur. La classe propose une méthode permettant de suspendre le thread et une autre pour le redémarrer, en changeant la valeur de `actif` ;
- la méthode `run` affiche toujours les entiers compris entre 1 et `n`, mais en testant entre chaque affichage la valeur de `actif`. Si `actif` vaut `false`, le thread fait une pause (d'une durée à déterminer), puis teste de nouveau la variable, etc.

Écrire un programme qui réalise une démonstration du mécanisme de suspension ainsi obtenu (avec au moins deux threads en plus du thread principal).

2.1.3 Synchronisation évoluée

Le problème de l'exercice précédent est l'utilisation de l'attente active (le thread tourne à vide en attendant d'être réveillé) qui gaspille des ressources et correspond à un grain de synchronisation très grossier. Heureusement, Java propose un mécanisme subtil mais efficace pour éviter l'attente active.

Le système est basé sur les méthodes `wait`, `notify` et `notifyAll` de la classe `Object`. Tout objet Java hérite de `Object` et possède de ce fait un verrou utilisé quand on travaille avec plusieurs threads. Dans certaines circonstances, l'utilisation d'un objet par un thread oblige ce dernier à acquérir le verrou de l'objet. Si le verrou est déjà possédé par un autre thread, le nouvel arrivant est bloqué (de façon passive) jusqu'à ce que le propriétaire du verrou le relâche.

Deux situations imposent l'acquisition du verrou :

1. une méthode peut être déclarée `synchronized` : dans ce cas, elle n'est utilisable qu'après obtention du verrou par le thread appelant. Comme le verrou est global à l'objet, aucun autre thread ne peut appeler une autre méthode `synchronized` du même objet (les autres méthodes sont accessibles normalement) ;
2. on peut utiliser la construction suivante :

```
1 synchronized(truc) {
2   // instructions du bloc
3 }
```

   Le bloc ainsi construit ne peut être exécuté qu'après acquisition du verrou de l'objet désigné par `truc`.

Quand un thread possède un verrou sur un objet, il peut le relâcher d'une façon particulière en appelant la méthode `wait` (de cet objet). Le principe de cette méthode est que le thread s'endort (de façon passive) après avoir relâché le verrou. Le seul moyen de réveiller le thread est alors qu'un autre thread devienne propriétaire du verrou de l'objet concerné et appelle la méthode `notify` (ou `notifyAll`) de ce dernier. Cette

méthode réveille l'un des threads endormis "sur" l'objet (la méthode `notifyAll` les réveille tous). Au niveau des détails techniques, la méthode `wait` possède des variantes avec temps maximal d'attente et toutes les versions peuvent renvoyer l'exception `InterruptedException`.

Exercice 6

Modifier le thread de l'exercice 5 afin de remplacer l'attente active par un mécanisme d'attente passive basé sur `wait` et `notify`. On procédera de la façon suivante :
- l'attente active sera remplacée par un bloc `synchronized` dans lequel la pause est remplacée par un `wait` ;
- les méthodes de contrôle du thread seront déclarées `synchronized` et la méthode de redémarrage utilisera `notify` pour réveiller le thread.

Réutiliser le programme de démonstration pour tester le mécanisme de suspension ainsi obtenu.

2.1.4 Appel RMI asynchrone

Pour réaliser un appel RMI asynchrone au niveau client, il "suffit" d'encapsuler l'appel dans un thread comme l'illustre l'exercice suivant :

Exercice 7

Écrire un serveur RMI qui propose une méthode `echo` classique (elle renvoie au client la chaîne de caractères paramètre de l'appel), mais avec une pause importante au début de la méthode, pour simuler un calcul complexe (on pourra tirer au hasard la durée de la pause). Écrire un client pour ce serveur qui réalise l'appel de façon asynchrone en procédant comme suit :
- écrire un objet `Thread` (ou `Runnable`) qui réalise l'appel puis affiche le résultat ;
- démarrer l'appel dans le thread principal et exécuter en parallèle (toujours dans ce thread principal) des opérations quelconques (comme par exemple quelques affichages entrecoupés de pauses).

Le problème de cet exercice est que c'est le thread qui exécute l'appel qui décide de faire l'affichage du résultat. Dans certaines situations, cela n'est pas vraiment acceptable, d'où l'exercice suivant :

Exercice 8

Modifier l'objet d'appel asynchrone de l'exercice précédent afin d'obtenir les fonctionnalités suivantes :
- la méthode `run` ne doit plus afficher le résultat mais se contenter de le stocker dans une variable adaptée ;
- l'objet doit proposer une méthode permettant de savoir si le calcul est terminé ;
- l'objet doit proposer une méthode bloquante renvoyant le résultat de l'appel (le blocage sera basé sur le couple `wait`/`notify`)

Proposer une démonstration des fonctionnalités de ce nouvel appel asynchrone.

2.2 Solution par *pulling*

Si le serveur propose des services permettant des appels asynchrones, la programmation du client est simplifiée. Nous commençons la partie serveur par une API de type *pulling* : le client lance une opération sur le serveur, puis demande périodiquement à celui-ci si l'opération est terminée. Le client est donc actif et réalise en général une attente active, comme dans l'exercice suivant :

Exercice 9

On considère l'objet distant suivant :

IPullServer

```
1 import java.rmi.*;
2 public interface IPullServer extends Remote {
3   public IDoSomethingResult doSomething(String param) throws RemoteException;
4 }
```

La méthode `doSomething` est asynchrone, c'est-à-dire qu'elle doit lancer un calcul côté serveur (en utilisant un thread) et rendre la main le plus rapidement possible au client. Pour permettre à celui-ci d'obtenir le résultat de l'opération, la méthode renvoie une référence vers un objet distant `IDoSomethingResult` spécifique à l'appel. Voici l'interface en question :

IDoSomethingResult

```
1 import java.rmi.*;
2 public interface IDoSomethingResult extends Remote {
3   public boolean isDone() throws RemoteException;
4   public String getResult() throws RemoteException;
5 }
```

La méthode `isDone` renvoie `true` si et seulement si le serveur a terminé les calculs correspondant à l'opération. La méthode `getResult` renvoie le résultat de l'opération (`null` si celle-ci n'est pas terminée).

Questions :

1. Implémenter `IDoSomethingResult` par une classe qui implémente aussi `Runnable` (et qui hérite de `UnicastRemoteObject`) : l'idée est d'avoir le code à exécuter (la méthode `run`) avec le dispositif de stockage (pour les méthodes `isDone` et `getResult`) et le serveur RMI (grâce à `UnicastRemoteObject`). Pour les calculs de `doSomething`, on se contentera d'un écho avec un délai de réponse aléatoire.
2. Implémenter `IPullServer`.
3. Implémenter un client utilisant `IPullServer` pour faire un appel asynchrone avec une attente active.

On peut aussi fournir une solution avec attente passive :

Exercice 10

On reprend l'exercice précédent en ajoutant à l'interface `IDoSomethingResult` la méthode suivante :

```
1 public void waitForResult() throws RemoteException;
```

Cette méthode permet au client d'éviter l'attente active : elle bloque le client jusqu'à ce que le résultat soit disponible.
Implémenter cette nouvelle version en utilisant le couple `wait`/`notify` pour implémenter le blocage sur le serveur.

2.3 Solution par *callback* (*push*)

Dans certaines situations, l'aspect asynchrone doit être poussé à l'extrême au sens où le temps qui peut s'écouler entre un appel de méthode et l'obtention d'une réponse n'est pas borné. En fait, il s'agit alors plus d'un mécanisme d'abonnement que d'un appel de méthode à distance. La solution classiquement employée est celle du *push* : le client s'inscrit auprès du serveur et c'est ce dernier qui appelle le client pour l'informer de la disponibilité du résultat. On peut d'ailleurs imaginer plusieurs résultats arrivant séquentiellement.

Avec RMI, un mécanisme de *push* est facile à implémenter. Il suffit que le client crée un objet serveur (une *callback*) qui est transmis (sous forme de référence distante) au serveur proprement dit lors de l'inscription. Quand le serveur souhaite prévenir le client, il appelle une méthode distante de l'objet *callback*. Pour simplifier le déploiement, on peut passer par le *design pattern* décorateur pour l'implémentation de la *callback*.

Commençons par une mise en œuvre simple :

Exercice 11

On considère l'objet distant suivant :

IPushServer

```
1 import java.rmi.*;
2 public interface IPushServer extends Remote {
3   public void doSomething(String param,IDoSomethingCallback callback)
4     throws RemoteException;
5 }
```

La méthode `doSomething` est asynchrone : elle lance le calcul côté serveur et rend la main le plus rapidement possible. La référence distante `IDoSomethingCallback` correspond à un objet créé sur le client que le serveur va utiliser pour informer le client de la complétion de l'opération. L'interface est très simple :

IDoSomethingCallback

```
1 import java.rmi.*;
2 public interface IDoSomethingCallback extends Remote {
3   public void reportCompletion(String result) throws RemoteException;
4 }
```

Questions :

1. Implémenter `IPushServer` : l'opération est un simple écho avec un délai de réponse aléatoire.
2. Implémenter `IDoSomethingCallback` de la façon la plus simple possible, par exemple en faisant réaliser un affichage (sur le client) par la méthode `reportCompletion`.
3. Implémenter un client permettant de tester le mécanisme. On pourra se contenter de quelque chose de très simple en utilisant le fait qu'une JVM ne s'arrête que si les objets distants qu'elle propose ne sont pas utilisés (grâce au mécanisme de *garbage collecting* réparti).
4. Tester le comportement du serveur quand le client se termine s'en avoir attendu la complétion de l'appel.

Le défaut de l'exercice précédent est qu'aucune synchronisation n'est mise en place au niveau du client. Pour obtenir quelque chose de plus réaliste, il faudrait que l'objet qui implémente `IDoSomethingCallback` s'inspire des mécanismes présentés dans la section 2.1 pour permettre une telle synchronisation :

Exercice 12

Ajouter à l'implémentation de `IDoSomethingCallback` construite dans l'exercice précédent, des méthodes inspirées des exercices 9 et 10 permettant au client d'attendre de façon active et/ou passive l'appel du serveur. Modifier le programme de test pour réaliser une démonstration des nouvelles fonctionnalités.

Pour finir, voici un exercice qui regroupe la plupart des techniques étudiées dans la série d'exercices.

Exercice 13

Le but de l'exercice est de réaliser un prototype pour un système de discussion de type *chat*, en ligne de commande. On prend comme ossature la solution de l'exercice 2, mais on remplace l'interface `IServeur` par la version suivante :

IServeur

```
1 import java.rmi.*;
2 public interface IServeur extends Remote {
3   public ISalon join(ISalonCallback callback) throws RemoteException;
4 }
```

avec :

ISalon

```
1 import java.rmi.*;
2 public interface ISalon extends Remote {
3   public void postMessage(String msg) throws RemoteException;
4   public void leave() throws RemoteException;
5 }
```

et :

ISalonCallback

```
1 import java.rmi.*;
2 public interface ISalonCallback extends Remote {
3   public void receiveMessage(String msg) throws RemoteException;
4 }
```

Les éléments à prendre en compte pour l'implémentation sont les suivants :
– l'interface `IServeurLogin` permet la mise en place d'une connexion personnalisée entre le client et le serveur, grâce à l'objet dédié `IServeur`, exactement comme dans l'exercice 2 ;

- l'interface `IServeur` permet au client de rejoindre un salon de discussion (pour simplifier, on a ici un seul salon). La connexion au salon est représentée par deux objets distants :
  - l'interface `ISalon` représente un objet distant dédié à l'utilisateur, résidant sur le serveur. Il permet au client de poster des messages dans le salon. Les messages s'affichent instantanément sur le serveur. La méthode `leave` permet au client de quitter le salon ;
  - l'interface `ISalonCallback` permet une communication asynchrone entre le serveur et le client. Elle correspond à un objet distant tournant sur le client, comme dans les exercices 9 et 10. Elle est utilisée par le serveur pour publier les messages des membres du salon. Le message envoyé par le serveur (paramètre `msg` de la méthode `receiveMessage`) est préfixé par le nom de l'utilisateur qui a posté ce message.

L'exercice consiste donc à implémenter un serveur et un client basé sur la conception décrite au dessus, puis à tester le résultat avec aux moins 2 clients. On fera particulièrement attention à rendre le serveur robuste aux déconnexions sauvages.