

Applications réparties en Java

Ophélie FRAISIER

Semestre 4

Table des matières

1	Qu'est-ce qu'une application répartie ?	2
2	Concepts et outils préliminaires	3
2.1	La sérialisation	3
2.2	Le multi-threading	3
2.2.1	Rappels	3
2.2.2	Implémentation du multi-threading	4
	Héritage de la classe <code>java.lang.Thread</code>	4
	Utilisation de l'interface <code>Runnable</code>	4
2.3	La synchronisation	5
2.3.1	Les méthodes synchronisées	5
2.3.2	Synchronisation d'une partie de code	5
2.3.3	Accès à des ressources partagées	6
2.3.4	Efficacité de la synchronisation	6
3	Création de serveur Java	7
3.1	Conception d'une application client-serveur 2 tiers	7
3.2	Bibliothèques Java utilisées pour la création d'un serveur	7
3.2.1	La classe <code>ServerSocket</code>	7
3.2.2	La classe <code>Socket</code>	8
3.2.3	Exemple d'application	8
	Le serveur	8
	Le client	8
4	Création et manipulation d'objet distribués	9
4.1	Présentation	9
4.2	La conception d'une application distribuée	9
4.2.1	La création d'un objet distribué	9
4.2.2	La création de la partie serveur	10
4.2.3	L'accès à un objet distribué	10
4.2.4	Le service d'enregistrement des objets	10
4.2.5	Les étapes de la création d'un objet distribué	11
4.2.6	Les conventions de nommage	11
4.3	Les bibliothèques Java utilisées pour créer des objets distribués	11
4.3.1	La classe <code>java.rmi.Naming</code>	11
4.3.2	La classe <code>java.rmi.server.UnicastRemoteObject</code>	12
4.3.3	La classe <code>java.rmi.registry.LocateRegistry</code>	12
4.3.4	L'interface <code>java.rmi.Remote</code>	12
4.4	Un exemple d'application d'objet distribué	12
4.4.1	L'interface de l'objet	12
4.4.2	L'objet distribué	12
4.4.3	Le serveur d'objet distribué	12
4.4.4	Le client	13


Chapitre 1


Qu'est-ce qu'une application répartie ?

Il s'agit d'un ensemble composé d'éléments reliés par un système de communication.

Architectures basées sur la mise en série de plusieurs cartes graphiques pour faire du calcul

Les éléments ont des fonctions de traitement (processeurs), de stockage (mémoire) et de relation avec le monde extérieur (capteurs et actionneurs). Ils ne fonctionnent pas indépendamment mais collaborent à une ou plusieurs tâches communes.

 Une partie au moins de l'état global de l'application est partagée entre plusieurs éléments, sinon on aurait un fonctionnement indépendant.

 Les différents éléments d'une application distribuée collaborent toujours dans un but précis.

Java propose un ensemble de classes et d'interfaces pour simplifier la réalisation d'applications réparties, en particulier il est aisé de réaliser des applications serveurs et des objets distribués. Les classes et les interfaces permettant la réalisation de serveurs sont principalement regroupées dans le package `java.net` et celles pour les objets distribués dans `java.rmi` et les packages inclus.

Chapitre 2

Concepts et outils préliminaires

2.1 La sérialisation

La sérialisation consiste à prendre un objet en mémoire et à en sauvegarder l'état (= les attributs) sur un flux de données, vers un fichier par exemple. Ce concept permet aussi de reconstruire ultérieurement l'objet en mémoire à l'identique de ce qu'il pouvait être initialement (création d'un clone). Le problème réside dans le fait que les attributs ne sont pas tous de type scalaire. Certains sont de type agrégat (sous-objets ou tableaux) or en Java les agrégats sont obligatoirement stockés dans la mémoire hors de la pile d'exécution. Ils sont donc inévitablement référencés.

Par défaut les classes ne permettent pas de sauvegarder l'état d'un objet sur un flux de données (pour des raisons de performance). Par contre, pour une grande majorité des classes du JDK, elles ont été définies comme étant sérialisables : il suffit de les marquer comme étant sérialisables pour qu'elles le soient, en implémentant l'interface `java.io.Serializable`, qui ne contient aucune méthode.

Comme la mise en oeuvre de l'interface est explicite le compilateur travaille maintenant en confiance et va effectuer les traitements nécessaires. Toutefois ce marquage ne fonctionne que si tous les attributs de la classe sont eux-même sérialisable.

⚠ Ne sont pas sérialisables :

- les threads
- les streams
- les sockets

(Liste non exhaustive)

Dans certains cas il peut donc être nécessaire d'exclure un attribut de l'état sérialisé d'une classe. Dans ce cas il suffit de rajouter le qualificateur `transient` à l'attribut en question.

⚠ Un objet peut théoriquement avoir une taille phénoménale.

2.2 Le multi-threading

2.2.1 Rappels

Il est parfois nécessaire de découper un programme en unités d'exécution indépendantes. Chacune de ces unités est appelée un processus léger ou thread. Chaque thread est programmé comme s'il s'exécutait par lui-même et qu'il avait son propre CPU. Un mécanisme sous-jacent s'occupe de diviser le temps CPU pour le programmeur, mais en général il est inutile de gérer cet aspect. C'est ce qui rend la programmation multi-threads beaucoup plus simple.

Un thread est un flot de contrôles séquentiel à l'intérieur d'un processus. Un processus peut contenir plusieurs threads s'exécutant en concurrence.

Il y a plusieurs utilisations possibles du multi-threading, mais en général il y aura une partie du programme attachée à un évènement ou à une ressource particulière, et il ne faut pas que le reste de l'application ne dépende de cette partie.

Le traitement d'une demande client par le serveur ne doit pas affecter son fonctionnement, il faut donc déléguer le traitement demandé à un processus léger : création d'un thread associé à cet évènement ou à cette ressource que l'on laissera s'exécuter indépendamment du programme principal.

2.2.2 Implémentation du multi-threading

Héritage de la classe `java.lang.Thread`

Il s'agit du moyen le plus simple pour créer un thread. `Thread` contient toutes les méthodes nécessaires pour créer et faire fonctionner les threads.

La méthode la plus importante est `run()`, qui doit être redéfinie pour que le thread exécute le code attendu. Elle contient souvent virtuellement une boucle s'exécutant jusqu'à ce que le thread ne soit plus nécessaire, ainsi vous pouvez établir la condition sur laquelle vous pouvez arrêter cette boucle et donc le thread. `run()` est souvent transformée en boucle infinie : à moins d'un facteur extérieur ne cause sa terminaison elle continuera indéfiniment.

R Afin de minimiser les risques de famine, il est important d'insérer dans la boucle un appel à la méthode de classe `yield()`. Cette méthode entraîne la suspension temporaire du thread pour permettre aux autres processus légers de s'exécuter.

Dans la méthode `main()` vous pouvez créer et démarrer un certain nombre de threads. La méthode `start()` de la classe `Thread` procède à une initialisation spéciale du thread et appelle ensuite la méthode `run()`.

Les étapes sont donc :

- le constructeur est appelé pour construire l'objet,
- la méthode `start()` configure le thread et appelle `run()`.

⚠ Les threads ne sont pas exécutés dans l'ordre où ils sont créés. Cet ordre est indéterminé, à moins d'ajuster par programmation les priorités en utilisant `setPriority()`.

Utilisation de l'interface `Runnable`

Cette méthode est utilisée si l'héritage de `Thread` est impossible ou non souhaitable. Elle ne contient que la méthode `run()` à redéfinir comme lors de l'héritage de `Thread`.

Définir un thread par cette méthode n'est pas trivial. Lors de la création d'une classe implantant `Runnable` il y a création d'un objet `Runnable` mais le thread n'est pas lancé, car ceci doit être explicitement fait : `run()` est définie dans la classe mais reste en sommeil après la création. Pour produire un thread à partir d'un objet `Runnable` il faut créer un objet thread séparé et passer l'objet `Runnable` au constructeur spécial de `Thread`. L'initialisation usuelle est alors effectuée puis `run()` est appelée une fois `start()` invoquée.

```
1 public class MaClasse implements Runnable {
2     public void run() {
3         while(true) {
4             ...
5             Thread.yield();
```

```

6      ...
7    }
8  }
9}
10
11 public class Test {
12     public static void main(String argv[]) {
13         Runnable uneInstance = new MaClasse();
14         Thread th = new Thread(uneInstance);
15         th.start();
16     }
17 }

```

2.3 La synchronisation

2.3.1 Les méthodes synchronisées

Java possède un support intégré pour gérer des accès concurrents sur un type de ressources, en particulier sur les données en mémoire (pour Java la mémoire est un objet).

Alors que le programmeur rend les attributs **private** et accède aux données au travers de méthodes, il peut prévoir les collisions d'accès à ces données en rendant une méthode particulière **synchronized**. Une telle méthode assure qu'un seul thread à la fois peut l'appeler pour un objet particulier, bien que ce thread puisse appeler plus d'une des méthodes de l'objet.

Chaque objet contient un seul verrou, aussi appelé moniteur, qui fait automatiquement partie de l'objet. Quand une méthode **synchronized** est appelée cet objet est verrouillé et aucune autre méthode **synchronized** de cet objet ne peut être appelée jusqu'à ce que la première soit terminée et libère le verrou. Ce verrou protège la mémoire commune de l'écriture par plus d'une méthode à un instant donné.

Il existe aussi un verrou unique par classe. Ainsi les méthodes **synchronized** statiques peuvent verrouiller les autres objets, empêchant un accès simultané aux données statiques.

```

1 public class Wagon {
2     public synchronized void reserverPlace() {
3     }
4 }

```

2.3.2 Synchronisation d'une partie de code

La section de code que le programmeur veut isoler de cette manière est appelée une *section critique*, et il est possible d'utiliser le mot clé **synchronized** d'une manière différente pour créer une telle section. Java supporte les sections critiques à l'aide d'un **synchronized block**. Cette fois le mot-clé **synchronized** est utilisé pour spécifier l'objet sur lequel le verrou est utilisé pour synchroniser le code encapsulé.

Avant d'entrer dans un bloc synchronisé le verrou doit être acquis sur l'objet qui veut rentrer dans la section. Si un autre thread possède déjà le verrou l'entrée dans le bloc est impossible jusqu'à ce que le verrou soit libéré.

Bien sûr toutes les synchronisations dépendent de la diligence du programmeur.

Chaque morceau de code pouvant accéder à une ressource partagée doit être emballé dans un bloc **synchronized**.

```

1 public class Wagon {
2     public void reserverPlace() {
3         synchronized(this) {
4             ...
5             //section critique
6             ...
7         }
8     }
9 }


```

2.3.3 Accès à des ressources partagées

La pose de verrou n'est pas une solution forcément suffisante lorsque le thread doit accéder à une ressource indisponible. Dans ce cas il faut faire patienter le thread jusqu'à ce que la dite ressource soit enfin disponible. Pour gérer ces cas l'environnement Java propose aussi un support pour contrôler l'activité des threads.

Tout le support nécessaire à cette gestion est fourni dans la classe `Object`. Celle-ci propose notamment trois méthodes permettant d'endormir et de réveiller un thread.

- Pour endormir un thread dans le moniteur : `wait`. Il y a plusieurs prototypes fournis afin soit d'attendre indéfiniment soit durant un délai maximum.
`sleep()` permet d'endormir un processus mais cette dernière ne permet au thread de reprendre son activité qu'à la fin de la durée passée en paramètre de la méthode.
- Pour réveiller : `notify()` et `notifyAll()`. `notify()` permet de réveiller un unique thread endormi sur un objet sur lequel il faut se synchroniser, `notifyAll()` réveille tous les threads endormis.

 Lors de l'invocation d'une des méthodes ci-dessus l'objet qui les invoque doit être synchronisé. Dans le cas contraire une exception est générée.

```
1 public void m() {  
2     synchronized(o) {  
3         wait(o);  
4     }  
5 }
```

2.3.4 Efficacité de la synchronisation

Comme avoir deux méthodes écrivant dans le même morceau de données en même temps n'apparaît jamais être une bonne idée, il semblerait logique que toutes les méthodes soit automatiquement synchronisées, et donc d'éliminer le mot-clé `synchronized` ailleurs.

Mais il faut savoir qu'acquérir un verrou n'est pas une opération légère : cela multiplie par quatre le coût de l'appel de la méthode (c'est à dire l'entrée et la sortie de la méthode sans l'exécution du corps) et peut être très différent suivant la mise en oeuvre.

Donc si l'on sait qu'une méthode ne posera pas de problème particulier il est opportun de ne pas utiliser le mot-clé `synchronized`.

D'un autre côté supprimer le mot-clé `synchronized` pensant que c'est un goulot d'étranglement pour les performances en espérant qu'il n'y aura pas de collision est une invitation au désastre.

```
1 // Exemple ne fonctionnant pas et non pertinent  
2 public class MaClasse implements Runnable {  
3     public synchronized void run() {  
4         while(true) {  
5             ... // boucle infinie synchronisee  
6         }  
7     }  
8  
9     public synchronized int getBouh() {  
10         ... // non pertinent  
11     }  
12 }
```

Chapitre 3

Création de serveur Java

Toutes les demandes convergent vers un serveur qui les traite ou éventuellement délègue le traitement de celles-ci à des gestionnaires spécifiques.

3.1 Conception d'une application client-serveur 2 tiers

Un serveur est un programme qui écoute un port de la machine et qui se bloque jusqu'à ce qu'il détecte une demande de connexion. Une fois la demande détectée le serveur extrait un canal de communication permettant de recevoir le flux de données issu du client et un canal de communication permettant d'acheminer la réponse au client. Ces deux canaux devront être fermés lorsque les informations auront été acheminées.

Les bibliothèques fournies avec Java permettent de créer rapidement des applications client-serveur 2 tiers. Pour concevoir un serveur il est nécessaire de prendre en compte les points suivants :

- définir les services que doit rendre le serveur,
- définir les services pouvant s'exécuter indépendamment du serveur et les associer à un thread,
- définir les méthodes et les portions de code critique,
- concevoir l'architecture du serveur, en UML par exemple, indépendamment de tout langage,
- choisir une technologie et projeter la conception réalisée précédemment sur celle-ci,
- introduire les concepts propres au langage choisi, en particulier pour les problèmes de synchronisation du code et de sérialisation des données,
- introduire les bonnes bibliothèques et les intégrer à la conception.

3.2 Bibliothèques Java utilisées pour la création d'un serveur

Le langage Java propose le package `java.net` offrant différentes fonctionnalités pour réaliser un serveur, en particulier les classes `ServerSocket` et `Socket`.

3.2.1 La classe `ServerSocket`

Cette classe met en oeuvre un serveur basé sur les sockets. Un tel serveur attend une demande issue d'un client distant et effectue certaines opérations en fonction de la demande. Éventuellement le serveur retourne un résultat au demandeur. Elle contient deux méthodes intéressantes :

- `accept()` : cette méthode attend une demande d'un utilisateur en écoutant le port passé en paramètre au constructeur, et bloque son exécution jusqu'à détection d'une telle demande. Une fois celle-ci détectée elle retourne un socket permettant de dialoguer avec le serveur.
- `close()` : cette méthode clôture le serveur et tous les threads bloqués sur une instruction `accept()` génère une `SocketException`.

3.2.2 La classe Socket

Cette classe met en oeuvre un socket. Elle possède trois méthodes principales :

- `getInputStream()` : cette méthode fournit un `java.io.InputStream` permettant de recevoir les méthodes issues du client.
- `getOutputStream()` : cette méthode fournit un `java.io.OutputStream` permettant d'envoyer des messages au client.
- `close()` : cette méthode clôture le socket.

3.2.3 Exemple d'application

Le serveur

```
1 import java.io.*
2 import java.net.*
3
4 public class Serveur {
5     public static void main(String argv[]) throws IOException {
6         String str;
7         ServerSocket sos;
8         sos = new ServerSocket(9999);
9         try {
10             do {
11                 Socket soc = sos.accept(); //Attente d'une demande
12                 java.util.Scanner dos = new java.util.Scanner(soc.getInputStream());
13                 str = dos.nextLine();
14                 System.out.println("Message reçu : " + str);
15                 soc.close();
16             } while(true);
17         }
18         catch(UnknownHostException e) {
19             e.printStackTrace();
20         }
21         catch(IOException e) {
22             e.printStackTrace();
23         }
24         finally {
25             sos.close();
26         }
27     }
28 }
```

Le client

```
1 import java.io.*
2 import java.net.*
3
4 public class Client {
5     public static void main(String argv[]) throws IOException {
6         Socket soc = new Socket("www.iut-tlse3.fr", 9999); // Creation du socket
7         OutputStream os;
8         PrintWriter dos;
9         os = new soc.getOutputStream(); // Creation du canal d'ecriture dans soc
10        dos = new PrintWriter(os, true);
11        dos.println("Bonjour et bienvenue tout le monde");
12        dos.close();
13        soc.close();
14    }
15 }
```

Chapitre 4

Création et manipulation d'objets distribués

4.1 Présentation

Le but de RMI (Invocation de Méthodes Distantes) est de permettre l'appel, l'exécution et le renvoi de résultats d'une méthode exécutée dans une machine virtuelle différente de celle de l'objet l'appelant. Cette machine virtuelle peut être sur une machine différente pourvue qu'elle soit accessible par le réseau. La machine sur laquelle s'exécute la méthode distante s'appelle le serveur. L'appel coté client d'une telle méthode consiste à obtenir une référence sur l'objet distant puis à appeler la méthode à partir de cette référence. RMI se charge de rendre transparents la localisation de l'objet distant, son appel et le renvoi du résultat.

Bien qu'il soit possible d'utiliser des objets distribués pour réaliser des serveurs, l'intérêt majeur d'utiliser de tels objet est de créer un réseau d'entités pouvant communiquer entre elles pour répondre à une demande de l'utilisateur. Cela signifie que les objets jouent à la fois le rôle de serveur lorsqu'ils sont sollicités et le rôle de client lorsqu'ils sont demandeur. Pour être accessible un objet doit s'enregistrer dans un annuaire afin d'être connu par les autres objets de l'application.

La création de serveurs alliés à des objets distribués va permettre de définir des applications client-serveur 3 tiers.

4.2 La conception d'une application distribuée

La conception d'objets distribués nécessite la réalisation d'une partie serveur permettant de les contacter. La mise à disposition de la version 5 de Java a beaucoup simplifié l'utilisation de cette technologie par rapport à ce qui était de mise avant.

4.2.1 La création d'un objet distribué

Pour être distribué un objet doit être composé de trois parties :

- l'interface exportant les services (soit les méthodes publiques) fournis par l'objet. Cette interface doit étendre l'interface `java.rmi.Remote`. De plus toutes les méthodes de cette interface doivent être déclarées comme pouvant lever l'exception `java.rmi.RemoteException`. Cette erreur est levé lorsqu'une erreur issue du réseau se produit.
- la classe mettant en oeuvre l'interface. Cette classe doit soit étendre la classe `java.rmi.server.UnicastRemoteObject`, soit être exportée en utilisant la méthode statique `exportObject()` de la classe `UnicastRemoteObject` (pour des raisons de problème d'héritage notamment). Dans ce cas il convient d'instancier manuellement les objets du serveur et de les transmettre à la méthode statique.
- le serveur d'objets.

4.2.2 La création de la partie serveur

Pour être accessible un objet distribué doit être enregistré dans un annuaire. Actuellement deux solutions existent :


- utiliser l’annuaire `java.rmi.Naming` constituant l’annuaire par défaut.
 - utiliser les classes du package `javax.naming` permettant la définition de contextes personnalisés.
- Dans le cadre de cet enseignement nous utiliserons la première solution.

Pour enregistrer un objet il faut utiliser soit la méthode de classe `bind()` soit la méthode de classe `rebind()`. La première enregistre l’objet dans l’annuaire et lève l’exception `NameAlreadyBoundException` si l’objet est déjà lié dans l’annuaire, alors que la seconde écrase l’objet existant s’il est déjà lié dans l’annuaire.

4.2.3 L’accès à un objet distribué

Lorsqu’un client veut utiliser un objet distribué il doit récupérer la référence de l’objet distant en interrogeant l’annuaire. Cette interrogation se fait en invoquant la méthode de classe `lookup()` de la classe `java.rmi.Naming`. Cette méthode retourne un objet qu’il faudra transtyper avec le type attendu.

 Ce type attendu est fourni par l’interface exportant les services de l’objet distribué.

 Si l’on essaye d’accéder à un objet distant via la méthode `lookup()` et que cet objet n’est pas lié actuellement l’exception `NamingException` est levée.

4.2.4 Le service d’enregistrement des objets


Pour enregistrer un objet dans un annuaire il existe deux solutions :

- la plus élégante : l’utilisation de la classe prédéfinie `sun.rmi.registry.RegistryImpl` permettant d’exécuter l’adaptateur rmi vu comme un annuaire dans lequel on enregistre tous les objets distribués d’une même machine.

On passe en paramètre le numéro du port sur lequel les clients devront se connecter. Il faut aussi indiquer à l’annuaire le chemin d’accès aux paquetages contenant toutes les classes non prédéfinies utilisées en paramètres des méthodes distantes des objets distribués avec :

```
java-classpath ‘‘package1’’ sun.rmi.registry.RegistryImpl 9999
```

package1 étant le ou les chemins d’accès vers les paquetages, séparés par une virgule sous UNIX et deux-points sous Windows, et 9999 étant le numéro de port

 Si le port est déjà utilisé, l’exception `ExportException` ou `BindException` est générée.

Si vous définissez mal l’accès aux packages ou si les classes utilisées ne sont pas dans le package alors vous aurez une erreur d’exécution `RemoteException` ou `UnmarshalException` non pas lors du lancement de l’annuaire mais lors du lancement du serveur.

- la plus simple à mettre en oeuvre : exécuter l’annuaire dans le code du serveur en utilisant la classe `java.rmi.registry.LocateRegistry` et sa méthode de classe `createRegistry()`. Toutefois il faut que le serveur se trouve dans le même package que les objets distribués afin que l’annuaire puisse y accéder.


Cette solution évite le lancement de l’annuaire manuellement mais est plus contraignante d’un point de vue architecture logicielle.


Toutefois elle est suffisante pour les TPs.

4.2.5 Les étapes de la création d'un objet distribué


Pour concevoir un objet distribué il est nécessaire de prendre en compte les points suivants :

- isoler dans un même package la classe devant distribuer les objets ainsi que l'interface qu'elle réalise. Cette classe doit étendre la classe `java.rmi.server.UnicastRemoteObject` et l'interface doit étendre `java.rmi.Remote`,

 Toutes les méthodes de l'interface doivent être déclarées comme pouvant lever l'exception `java.rmi.RemoteException`

 L'interface de l'objet est aussi bien utilisée du côté client que du côté de l'objet distribué lui-même

- définir les classes devant être sérialisables, c'est-à-dire les classes apparaissant comme types de paramètres des méthodes de l'interface,
- définir une classe serveur chargée d'enregistrer l'objet distribué de l'annuaire. Cette classe peut éventuellement lancer l'annuaire. Si l'annuaire n'est pas exécuté il faut l'exécuter manuellement en lançant `sun.rmi.registry.RegistryImpl`,
- définir le client devant acquérir une référence sur l'objet distant en utilisant la méthode de la classe `java.rmi.Naming` et l'interface de l'objet,
- distribuer les packages sur les différentes machines.

 Lors de la distribution d'une classe il ne faut pas oublier de distribuer aussi toutes les classes dont l'objet distribué a besoin pour s'exécuter (soit les classes précédemment sérialisées).

4.2.6 Les conventions de nommage

Java préconise des conventions de nommage qu'il est conseillé d'utiliser bien qu'il n'y ait aucune obligation.

Soit **MaClasse** le nom de la classe dont les objets doivent être distribués.

- Son interface : **MaClasse**
- La classe réalisant cette interface : **MaClasseImpl**
- La classe serveur : **MaClasseServer**

4.3 Les bibliothèques Java utilisées pour créer des objets distribués

Les méthodes nécessaires pour réaliser des objets distribués se répartissent principalement dans trois classes et une interface.

4.3.1 La classe `java.rmi.Naming`

Cette classe offre trois méthodes pertinentes :

- `bind()` : cette méthode de classe lie un nom à un objet. Ce nom est de la forme `rmi://adresse:port/nom`. Si le nom existe déjà alors l'exception `NameAlreadyBoundException` est levée.
- `rebind()` : cette méthode est similaire et a les mêmes paramètres que la précédente. Toutefois si le nom existe alors l'ancien objet est écrasé.
- `lookup()` : cette méthode permet de récupérer la référence de l'objet distant dont le nom est passé en paramètre.

 Le nom doit être identique à celui utilisé dans la méthode `bind()` ou `rebind()`.

La valeur retournée par cette méthode est de type `Object` et doit donc être convertie en utilisant l'interface de l'objet distant.

4.3.2 La classe `java.rmi.server.UnicastRemoteObject`

Cette classe est utilisée comme marqueur lorsqu'elle est étendue par la classe des futurs objets distribués.

La seule méthode vraiment intéressante est la méthode statique `exportObject()` utilisée lorsque l'extension de la classe `java.rmi.server.UnicastRemoteObject` est impossible. Cette méthode nécessite deux paramètres :

- l'objet à distribuer,
- le numéro de port où se trouve l'annuaire.

4.3.3 La classe `java.rmi.registry.LocateRegistry`

Cette classe possède une méthode de classe permettant d'exécuter l'annuaire. Cette méthode a besoin d'un paramètre : le numéro de port sur lequel l'annuaire est enregistré.

4.3.4 L'interface `java.rmi.Remote`

Cette interface est un marqueur, elle ne possède aucune méthode.

4.4 Un exemple d'application d'objet distribué

4.4.1 L'interface de l'objet

```
1 public interface Personne extends java.rmi.Remote {
2     void setNom(String nom) throws java.rmi.RemoteException;
3     String getNom() throws java.rmi.RemoteException;
4 }
```

4.4.2 L'objet distribué

```
1 public class PersonneImpl extends java.rmi.server.UnicastRemoteObject
2     implements Personne {
3     private String nom;
4
5     public PersonneImpl(String nom) {
6         this.nom = nom;
7     }
8     public void setNom(String nom) throws java.rmi.RemoteException {
9         this.nom = nom;
10    }
11    public String getNom() throws java.rmi.RemoteException {
12        return this.nom;
13    }
14 }
```

4.4.3 Le serveur d'objet distribué

```
1 /* Ce programme de serveur cree (instancie) 2 objets distants, les enregistre aupres
2 du service de nom et attend les clients pour invoquer les methodes des objets distribues */
3 public class PersonneServer {
4     public static void main(String[] args) throws Exception {
5         java.rmi.registry.LocateRegistry.createRegistry(9999);
6         Personne dupont = new PersonneImpl("dupont");
7         Personne dulong = new PersonneImpl("dulong");
8         // declaration des objets dans l'annuaire
9         java.rmi.Naming.rebind("rmi:nacre.local:9999/manu", dupont);
10        java.rmi.Naming.rebind("rmi:nacre.local:9999/dulong", dulong);
11    }
12 }
```

4.4.4 Le client

```
1 public class Test throws Exception {  
2     public static void main(String [] args) {  
3         Personne unePersonne =  
4             (Personne)(java.rmi.Naming.lookup("rmi://nacre.local:9999/manu"));  
5         System.out.println("Bienvenue a " + unePersonne.getNom());  
6         // Bienvenue a dupont  
7     }  
8 }
```