

# **Variables de cluster**

## ***Un couteau suisse***

Philippe PRADOS

[pp@philippe.prados.name](mailto:pp@philippe.prados.name)

Les langages de programmations proposent différents environnements pour y placer des variables. Nous pouvons synthétiser les différents besoins pour regrouper les différentes variables par un schéma (Figure 1).

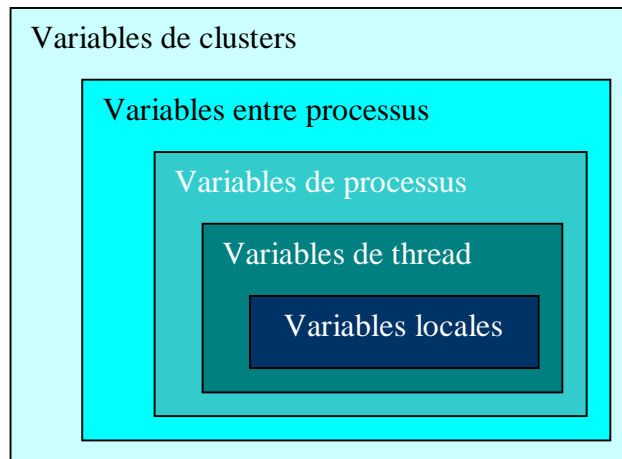


Figure 1

## 1. VARIABLES LOCALES

Le cadre le plus utilisé est sans conteste la pile d'exécution. On y retrouve les variables locales aux fonctions ou aux méthodes, ainsi que les paramètres de chaque traitement. Certaines langages comme le Pascal, possèdent des mécanismes permettant à une fonction d'accéder directement aux variables locales de la méthode appelantes (Source 1).

```
PROCEDURE MaProcedure;
  VAR uneVariable: integer;

  PROCEDURE ProcedureInterne;
    VAR uneAutreVariable: integer;
  BEGIN
    uneVariable := 1;
    uneAutreVariable := 2;
  END

BEGIN
  ProcedureInterne;
END.
```

### Source 1

Java propose une approche similaire, mais qui ne fonctionne qu'en lecture. Une méthode d'une classe interne peut accéder à une variable d'une autre méthode si celle-ci est `final` (Source 2).

```
class MaClass
{
  public void maMethode(final int monParametre)
  {
    final int maVariable=3;
    class ClasseInterne
    {
      void methode()
      {
        if (monParametre==maVariable)
          ...
      }
    }
    new ClasseInterne().methode();
  }
}
```

### Source 2

La méthode `ClasseInterne.methode()` peut accéder au paramètre `monParametre` et à la variable `maVariable`. En faite, les valeurs de ces deux paramètres sont dupliquées dans des variables cachés de `ClasseInterne`. Ce code est équivalent à celui Source 3. Le compilateur génère automatiquement cette version.

```
class MaClass
{
  public void maMethode(final int monParametre)
  {
```

```

final int maVariable=3;
class ClasseInterne
{
    private final int monParametre$;
    private final int maVariable$;
    public ClasseInterne(int i,int j)
    {
        monParametre$=i;
        maVariable$=j;
    }
    void methode()
    {
        if (monParametre$==maVariable$)
            ...
    }
}
new ClasseInterne(monParametre,maVariable).methode();
}
}

```

Source 3

## 2. VARIABLES DE TACHES

Il est parfois nécessaire d'avoir des variables globales à une tâche. Cela permet d'éviter de transférer des paramètres dans toutes les méthodes. Par exemple, le contexte transactionnel est généralement une variable globale associée à une tâche. Plusieurs tâches peuvent fonctionner en même temps, chacune possède son contexte transactionnel.

Certains compilateurs C/C++ proposent une extension permettant de qualifier une variable globale pour qu'elle soit associée à une tâche. Le problème de ces variables est qu'elles ne peuvent généralement pas être initialisées lors de la déclaration. Elles ont une valeur par défaut lors de leur première utilisation.

Pour offrir ce service en java, manipulez des dictionnaires indexés par la [Thread](#) courante (Source 4).

```

import java.util.*;
public class ThreadMap
{
    private static Map threadMap_=new HashMap();
    private ThreadMap() { }
    public static Map get()
    {
        return get("<default>");
    }
    public static Map get(String name)
    {
        Thread thread=Thread.currentThread();
        Map namedMap=(Map)threadMap_.get(thread);
        if (namedMap==null)
            threadMap_.put(thread,namedMap=new HashMap(3));
        Map map=(Map)namedMap.get(name);
        if (map==null)
            namedMap.put(name,map=new HashMap(7));
        return map;
    }
    public static void release()
    {
        release("<default>");
    }
    public static void release(String name)
    {
        Thread thread=Thread.currentThread();
        Map namedMap=(Map)threadMap_.get(thread);
        if (namedMap==null) return;
        Map map=(Map)namedMap.get(name);
        if (map==null) return;
        namedMap.remove(name);
        if (namedMap.size()==0)
            threadMap_.remove(thread);
    }
    public static void releaseAll()
    {
        Thread thread=Thread.currentThread();
        threadMap_.remove(thread);
    }
}

```

Source 4

Il faut faire très attention à bien libérer le dictionnaire lorsque la tâche meurt (Source 5).

```
public void run()
{
    try
    {
        Map map=ThreadMap.get("monContext");
        map.put("abc","def");
    }
    finally
    {
        ThreadMap.release("monContext");
    }
}
```

#### Source 5

Si la libération n'est pas effectuée, l'instance `Thread` présente dans le dictionnaire reste vivante. Il lui est associé une pile d'exécution de 64 Ko ! C'est une belle fuite mémoire.

### 3. VARIABLES DE PROCESSUS

Un autre cadre pour mémoriser les variables est le contexte global. Des variables sont déclarées pour un processus et sont visibles par toutes les fonctions et toutes les méthodes. Ce concept est très vieux et se retrouve dans pratiquement tous les langages. Le BASIC n'offre que ce niveau pour déclarer les variables. Avec java, ces variables correspondent aux attributs statiques des classes. Il y a, avec java, un cadre particulier car ces variables ne sont visibles que par l'intermédiaire d'un même `ClassLoader`. Il est donc possible d'avoir deux variables statiques identiques de la même classe, mais chargées par des `ClassLoader` différents (Figure 2). Suivant le `ClassLoader` ayant chargé la classe, différentes variables statiques sont disponibles.

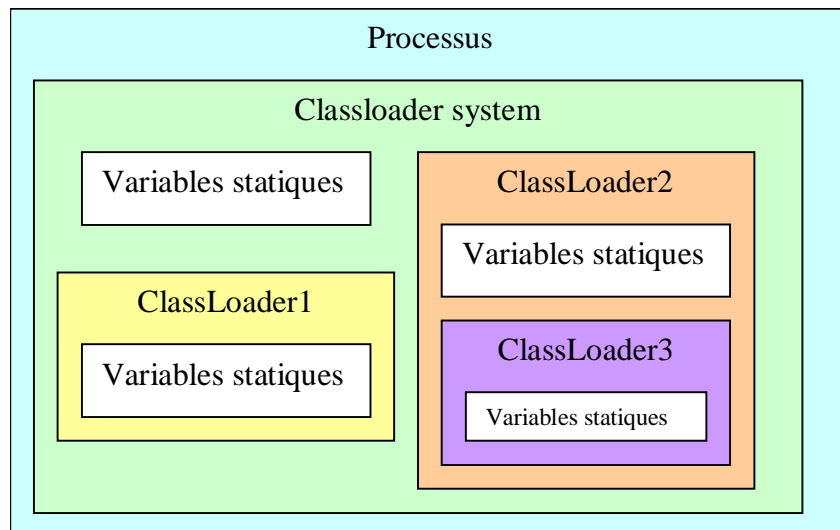


Figure 2

## 4. VARIABLES ENTRE PROCESSUS

Dans un environnement multi-processus, il est parfois nécessaire d'avoir des variables partagées entre eux. Les systèmes d'exploitations offrent généralement des technologies permettant de partager des zones mémoires entre les processus (Figure 3). Cela permet d'y placer des variables accessibles simultanément à plusieurs processus. Le disque local peut être une technique pour partager des informations entre différents processus.

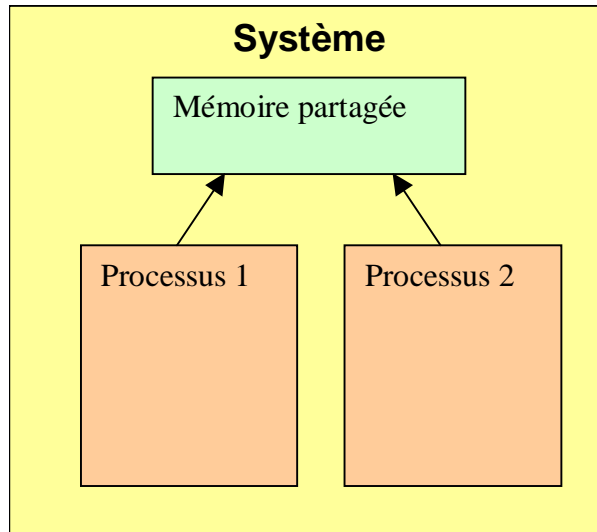


Figure 3

## 5. VARIABLES DE CLUSTERS EN JAVA

L'environnement moderne dans lequel nous travaillons maintenant propose un autre cadre pour les variables : les clusters. Les serveurs d'applications doivent gérer un nombre très important de clients. Les applications exposées sont stratégiques pour les entreprises. Il faut garantir une performance et une disponibilité optimale. Pour répondre à ce besoin, les architectures proposent généralement d'utiliser plusieurs serveurs en grappe et d'exploiter des technologies de répartition de charge, d'affinités, ou autres. Cela permet de considérer qu'une grappe de serveur est équivalente à un gros serveur virtuel. Certaines parties peuvent être indisponibles, les autres prennent le relais. Il est nécessaire de partager des variables entre plusieurs serveurs.

Quels sont les différentes approches pour partager des variables entre plusieurs serveurs ? Est-ce qu'elles répondent aux objectifs de traitements-répartis et de disponibilité ?

### 5.1 Les grappes de serveurs

Commençons par étudier les différentes technologies permettant de partager des serveurs dans une architecture réseau. Plusieurs serveurs sont disponibles. Ils ont chacun une adresse IP différente. Une boîte (WebSphere Edge Server d'IBM par exemple) s'occupe de dispatcher les requêtes entrantes vers les différents serveurs (Figure 4).

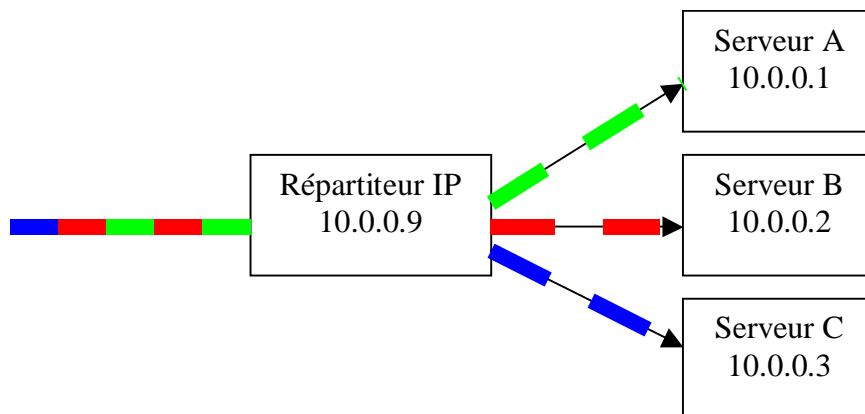


Figure 4

Cette approche est simple à mettre en place. Chaque nouvelle connexion IP est dirigée vers une machine particulière de la grappe. Le serveur cible est choisi par un algorithme de répartition de charge, ou plus simplement, chacun son tour. Cela fonctionne bien, mais ne permet pas d'associer un utilisateur particulier à une machine.

Les risques de cette architecture sont les suivants : surcharge du répartiteur ou indisponibilité totale de l'application si celui-ci tombe. WebSphere Edge Server propose différentes technologies pour utiliser un répartiteur de secours et ajouter des fonctionnalités de cache des pages HTML.

Le protocole HTTP est un protocole sans connexion persistante. Cela veut dire qu'une nouvelle connexion TCP/IP est ouverte pour chaque demande de page. Pour une page possédant deux images par exemple, la page et les deux images peuvent être servies par trois machines différentes. Tant qu'il s'agit d'informations statiques comme des images, cela ne pose pas de problèmes.

Par contre, s'il s'agit de maintenir le contexte de l'utilisateur, les différents serveurs doivent s'échanger des informations. Certains serveurs d'applications mémorisent le contexte de l'utilisateur dans une base de données. Par exemple, le serveur WebSphere d'IBM est capable de sauvegarder toute la session de l'utilisateur d'un seul coup ou chaque clef de la session HTTP individuellement dans une base de donnée relationnelle. Cette approche permet de garantir une forte disponibilité mais à un impact sur les performances (30%). Avant chaque calcul d'une page, le serveur doit récupérer l'état de la session de la base de donnée. Une extension proposée par Tangosol (voir plus bas) permet un partage des sessions entre les serveurs sans utilisation de la base de donnée.

Pour éviter le partage des sessions des utilisateurs, on utilise souvent une technique d'affinité. Un utilisateur est destiné à une et une seule machine. Les critères d'affinités peuvent être : l'adresse IP source, la valeur d'un cookie, un pattern de l'URL ou la session SSL. Si une machine tombe, tous les utilisateurs associés perdent leurs sessions. Les autres ne sont pas impactés. Les utilisateurs perdus doivent se reconnecter pour utiliser l'application. Le sacrifice de certains utilisateurs dans un cas limite permet d'améliorer notablement les performances au quotidien.

Les critères permettant de sélectionner une machine pour un utilisateur sont variables. Le répartiteur peut maintenir un état de la charge de chaque serveur pour répartir les connexions. D'autres critères peuvent être utilisés. Citons le type de document demandé (gif, jpeg, html, jsp ?), une distribution en Round-Robin ou d'autres algorithmes plus complexes.

D'autres stratégies permettent de répartir la charge entre les serveurs, sans points de faiblesses. Les connexions TCP/IP sont parfois capable de sauter d'un serveur vers un autre.

Une adresse IP virtuelle est utilisée pour référencer la grappe de serveur. Cette adresse est publique et arrive sur un brin de réseau ou se trouve des Edge servers et les différents serveurs de l'application. En écoutant le réseau, les répartiteurs capturent l'ouverture de la communication, commence à récupérer les premiers paquets jusqu'à l'obtention d'un critère de sélection du serveur cible. Les règles peuvent être : l'adresse IP source, le port cible, la session SSL, le type de document demandé, la valeur ou la présence de paramètres dans la requête, etc. Une fois que le serveur cible est identifié, la communication est transférée vers celui-ci. La communication est alors capturée et la réponse repart vers le client, directement à partir de la cible. Le répartiteur n'intervient plus dans la communication (Figure 5).

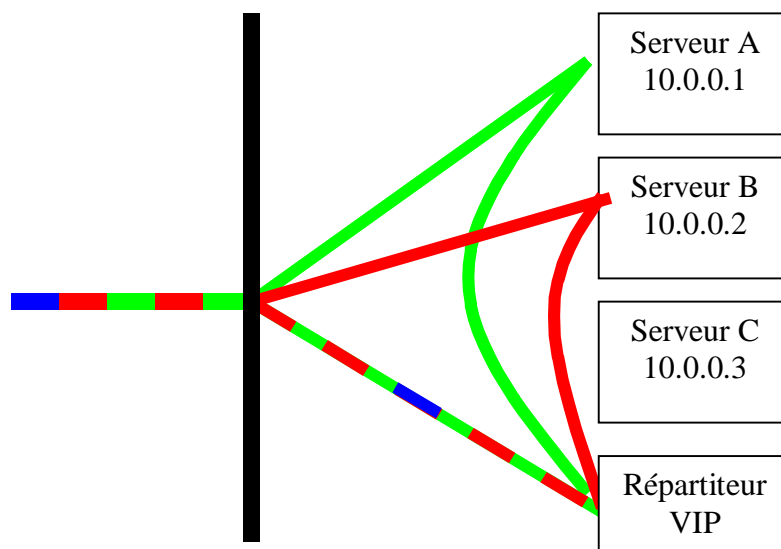


Figure 5

Cette approche permet d'annuler la défaillance du serveur de répartition car il est possible d'en placer plusieurs sur le réseau. D'autre part, les communications ne sont pas ralenties, car une fois que le serveur cible est identifié, le répartiteur n'intervient plus.

## 5.2 Le référentiel

Pour partager les informations entre différents serveurs, une des approches possibles consiste à utiliser un référentiel unique utilisé par tous les serveurs. Ce référentiel peut être :

- Un répertoire partagé
- Une base de donnée relationnelle ou objet
- Un serveur LDAP
- Un objet RMI ou CORBA

Suivant les choix d'architectures physiques de l'application ces différentes approches seront plus ou moins efficaces. Les situations étant très variables, pourquoi ne pas offrir une panoplie de solution ? Cela permet d'effectuer le choix technologique le plus tard possible, après un test

de performance en grande nature dans l'environnement d'exploitation. C'est ce que nous nous proposons de faire. Nous allons rédiger un « couteau suisse » pour gérer les variables clusters.

### 5.3 L'utilisation

Nous allons, dans un premier temps, proposer une utilisation de ce service compatible avec les différentes approches. Nous allons fournir l'accès aux variables clusters à l'aide d'une `java.util.Map`. Cette interface de l'API java est suffisante pour maintenir le référentiel. Les variables sont récupérées par valeur à l'aide de leurs noms, servant de clef.

Il faut ensuite offrir une approche pour retrouver la Map partagée lors du démarrage des serveurs. Pour cela, proposons de nommer les Maps. Une méthode permet de retrouver une Map par son nom. Si elle n'est pas présente, elle est construite avant d'être donnée à l'application. Quelle que soit la technologie choisie, l'application commence par récupérer une référence vers la Map des variables clusters. Celle-ci est gardée dans une variable de processus, une variable statique.

```
static java.util.Map varcluster=factory.getMap("varcluster");
```

L'accès aux variables partagées étant maintenant disponible, il faut étudier comment les manipuler. En théorie, il faut protéger l'accès à ces données car plusieurs serveurs peuvent les modifier simultanément. En pratique, et pour des raisons de performances, ces variables ont généralement une volatilité faible. Elles évoluent peu. Les applications les consultent régulièrement mais les modifient peu. Une approche optimiste consiste alors à ignorer les effets de bords d'une modification simultanée. Le dernier serveur qui modifie une donnée écrase la valeur précédente.

## 6. LES TECHNOLOGIES

Les variables étant peu volatiles par principe, il peut être judicieux de maintenir un cache des différentes valeurs et de propager les modifications à tous les serveurs. Cela permet de maintenir le cache à jour et améliore notablement les performances en lecture. Malheureusement, suivant le choix technique effectué pour partager les contextes, ce n'est pas toujours possible. Il faut en effet recevoir un événement lors de la modification d'une valeur. Certaines technologies le permettent, d'autres non.

### 6.1 Partage de répertoire

Les systèmes d'exploitations offrent généralement la possibilité de partager un ou plusieurs répertoires. Des protocoles exploitent les réseaux pour identifier les différents serveurs accessibles et pour retrouver les ressources partagées. Un serveur publie un répertoire de travail auquel tous les membres du groupe peuvent accéder (Figure 6).

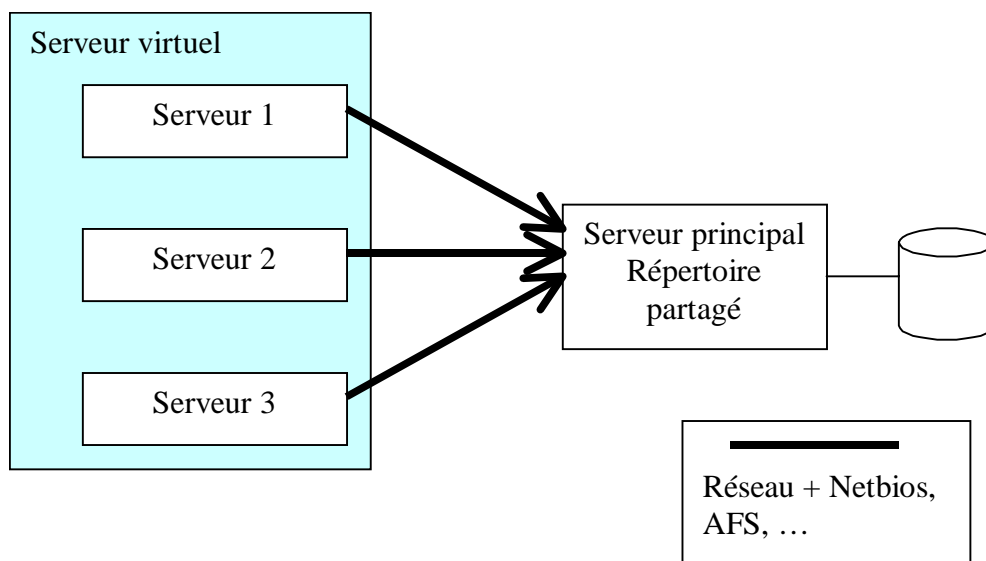
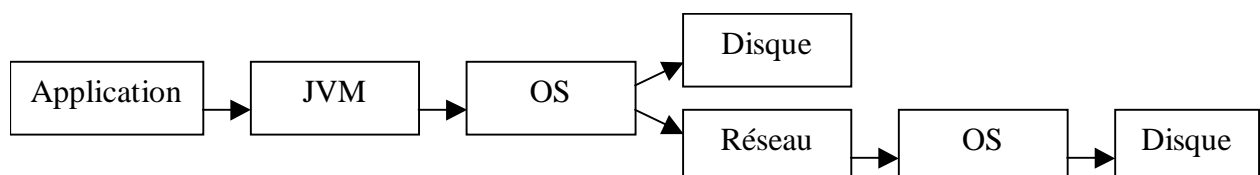


Figure 6

Cette approche permet également d'offrir une communication entre plusieurs processus sur le même serveur. Il ne s'agit plus d'utiliser un répertoire publié sur le réseau, mais un répertoire sur le disque local. C'est le système d'exploitation qui offre alors la technologie de communication entre les processus. Cette approche est intéressante avec java car ce langage est incapable de manipuler des mémoires partagées.

La communication suit le schéma suivant :



Avantages :

- Cette approche est simple à installer. Les protocoles nécessaires sont présents dans les différents systèmes d'exploitations.
- Il n'est pas nécessaire d'avoir une base de donnée.

- Les serveurs peuvent être redémarrés, ils savent reprendre la situation en cours de route.

**Inconvénients :**

- Pour communiquer, les différents serveurs doivent exploiter systématiquement le réseau et générer un trafic dépendant du protocole sous-jacent utilisé.
- Il faut publier un répertoire sur le serveur principal. Cela est une porte souvent exploitée par les pirates.
- Si le serveur principal tombe, l'application ne fonctionne plus.
- Il est nécessaire de nettoyer à la main le répertoire partagé lorsque tous les serveurs sont arrêtés.
- Il n'est pas possible de propager les modifications dans les caches des différents serveurs car le système de fichier n'émet pas d'évènement.
- Il faut configurer les droits d'accès pour l'application.

## 6.2 Base de donnée

Les bases de données relationnelles sont accessibles par tous les serveurs. Une grande partie des applications en ont besoin pour fonctionner. Pourquoi ne pas les utiliser comme référentiel ? Pour chaque Map, une table est créée automatiquement dans la base. Celle-ci propose deux colonnes : une clef et un champ binaire pour y stocker les objets javas sérialisés (Figure 7).

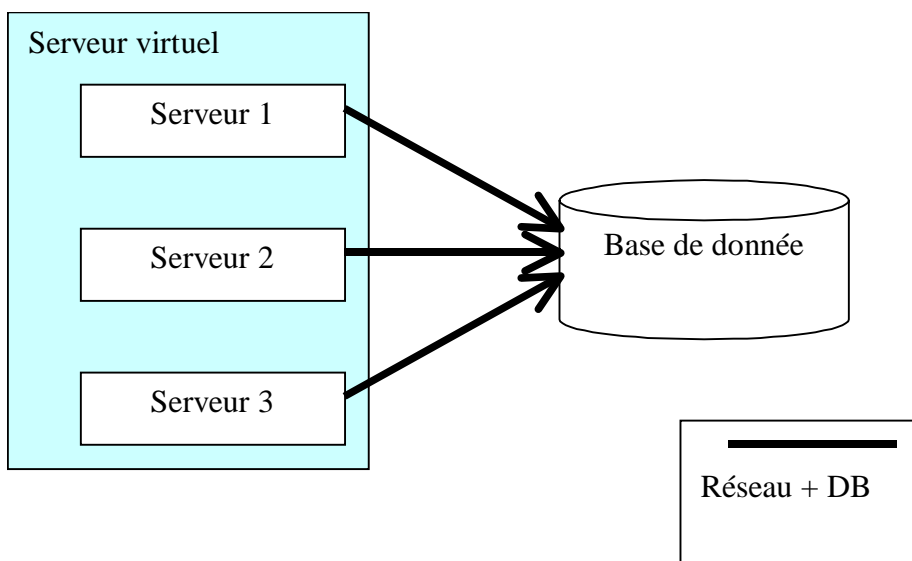


Figure 7

La communication suit le schéma suivant :



**Avantages :**

- Si l'application a besoin par ailleurs de la base de donnée, cette approche est simple à installer.
- Les technologies de tolérances aux pannes offertes par les fournisseurs de base de données sont exploitables.
- Les serveurs peuvent être redémarrés, ils savent reprendre la situation en cours de route.

**Inconvénients :**

- Pour communiquer, les différents serveurs doivent exploiter systématiquement le réseau et générer un trafic dépendant du protocole sous-jacent pour communiquer avec la base de donnée.
- Si le serveur de base de donnée tombe, l'application ne fonctionne plus. C'est généralement le cas de toute façon.
- Il est nécessaire de nettoyer à la main les tables lorsque tous les serveurs sont arrêtés, sauf si elles sont montées en mémoire.
- Il n'est pas possible de propager les modifications dans les caches des différents serveurs.

## 6.3 LDAP

Les spécifications JNDI permettent d'enregistrer des objets dans un référentiel quelconque (LDAP, base de registre Windows, référentiel RMI, etc.) Il existe de nombreux drivers JNDI permettant de s'interfacer avec divers référentiels. Citons des drivers pour LDAP, DNS, NIS, NDS, RMI, CORBA ou le gestionnaire de fichier. Les serveurs LDAP peuvent accepter de recevoir des objets javas sérialisés. Cela peut servir à partager des informations entre les différents serveurs (Figure 8).

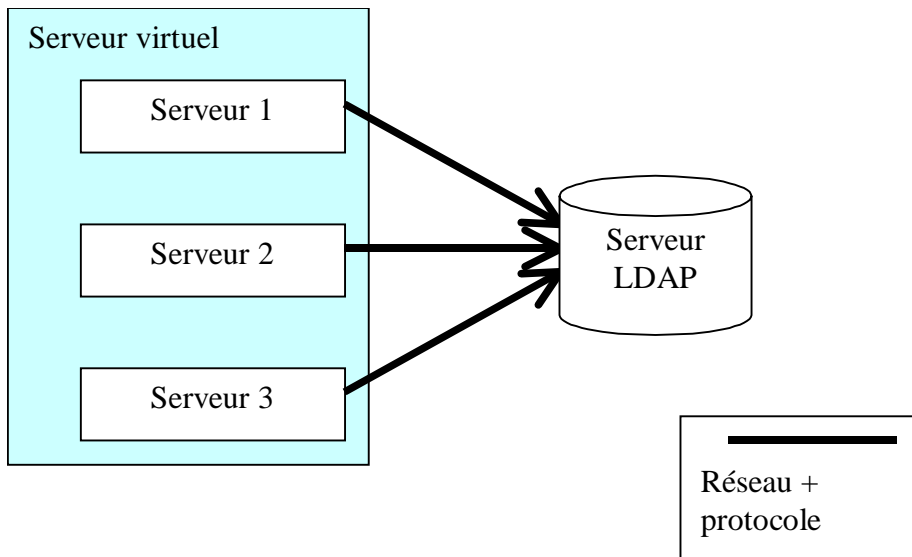
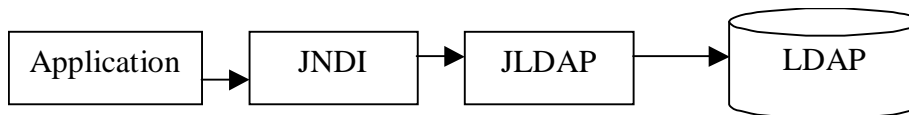


Figure 8

La communication suit le schéma suivant :



**Avantages :**

- Le référentiel est souvent présent et nécessaire pour l'application (généralement pour l'authentification ou pour retrouver les Homes EJB).
- Les serveurs peuvent être redémarrés, ils savent reprendre la situation en cours de route.

**Inconvénients :**

- Il faut posséder un référentiel. Celui-ci est généralement partagé par l'ensemble de l'entreprise. L'utilisation qui en est faites risque de dégrader les performances des autres applications. Des architectures plus complexes à l'aide de cache peuvent corriger ce problème.
- Pour communiquer, les différents serveurs doivent exploiter le réseau et générer un trafic dépendant du protocole sous-jacent. Les référentiels d'entreprises sont généralement éloignés des serveurs de l'application. Les temps réseaux sont importants. Des serveurs LDAP intermédiaires peuvent rapprocher les données de l'application.
- Si le référentiel tombe, l'application ne fonctionne plus.
- Il est nécessaire de nettoyer à la main le référentiel lorsque tous les serveurs sont arrêtés.

**6.4 RMI/CORBA**

Java offre une technologie permettant d'invoquer un objet distant. Un objet RMI or CORBA peut servir de référentiel aux variables partagées. Pour obtenir la valeur d'une clef, le serveur RMI/CORBA est invoqué.

Les services RMI de java utilisent le protocole Java Remote Method Protocol (JRMP) pour communiquer. Ils proposent un ramasse miette répartie. Une base d'enregistrement RMI doit être lancée sur le réseau ([rmiregistry](#)).

Une autre option consiste à utiliser le protocole Internet Inter-ORB Protocol (IIOP) pour invoquer des objets CORBA. Ces objets peuvent être rédigés dans n'importe quels langages (C, C++, etc.). Une base d'enregistrement CORBA doit être lancée sur le réseau (Common Object Services Name Service)

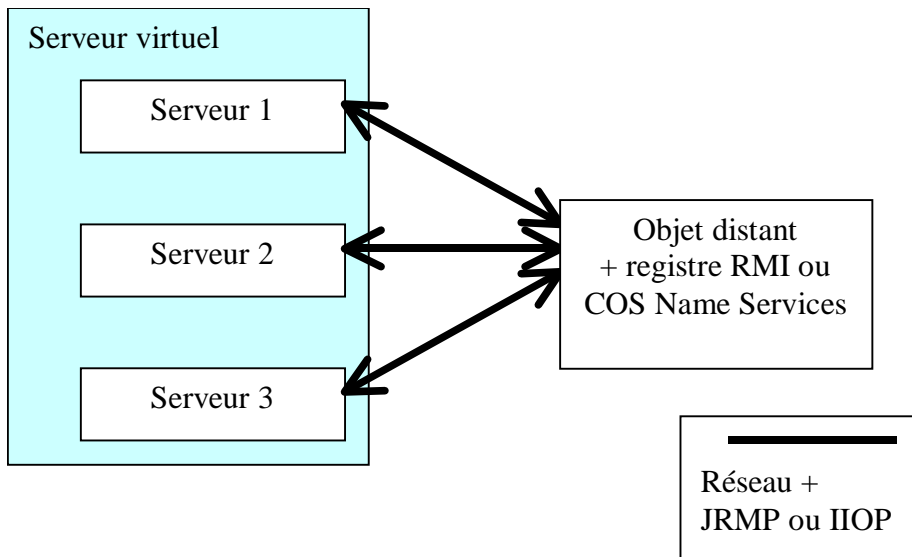


Figure 9

L'objet distant peut également prévenir tous les serveurs lors de la modification d'une information. Ainsi, chaque serveur peut maintenir un cache des valeurs à jour, réduisant ainsi le trafic réseau (Figure 9).

La communication suit le schéma suivant :



**Avantages :**

- Les serveurs peuvent maintenir un cache des variables partagées.
- Les informations étant uniquement en mémoire, il n'est pas nécessaire de faire le ménage lors de l'arrêt de tous les serveurs.
- Les serveurs peuvent être redémarrés, ils savent reprendre la situation en cours de route.

**Inconvénients :**

- Il faut lancer le serveur RMI/CORBA.
- Pour trouver le serveur distant, il faut généralement utiliser une technologie périphérique comme un fichier partagé, une base d'enregistrement RMI ou CORBA, un serveur de nom (JNDI, LDAP, etc.).
- Pour communiquer, les différents serveurs doivent exploiter le réseau et générer un trafic dépendant du protocole sous-jacent.
- Si l'objet distant tombe, l'application ne fonctionne plus. Les serveurs peuvent continuer à exploiter les informations présentes dans leurs caches, mais ne peuvent plus les modifier.

### 6.5 Coherence de Tangosol

Tangosol propose la technologie Coherence™ permettant de résoudre ce besoin de façon très élégante. En exploitant les adresses IP de broadcast du protocole TCP/IP (adresse IP de la forme 224.M.m.b), il est possible de signaler à tous les serveurs des évolutions d'une variable partagée. Chaque serveur adapte son cache locale en conséquence. Plusieurs stratégies sont disponibles pour gérer les Map.

1. La modification d'une donnée est propagée sur tous les serveurs du cluster (Figure 10).

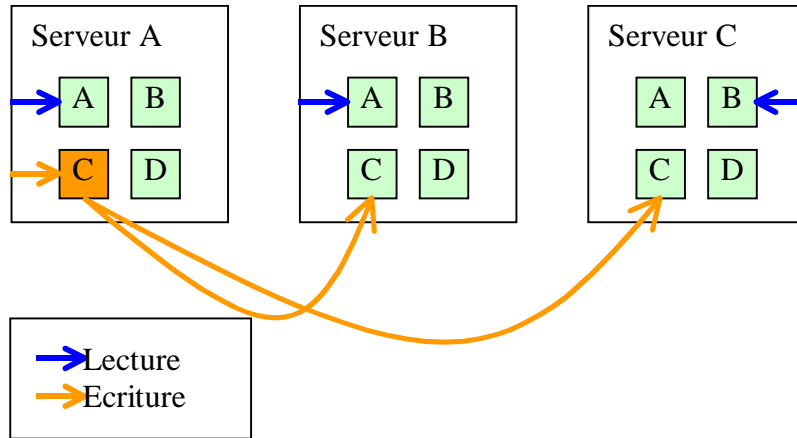


Figure 10

2. Les données sont centralisées sur un serveur avec une possibilité de backup (Figure 11).

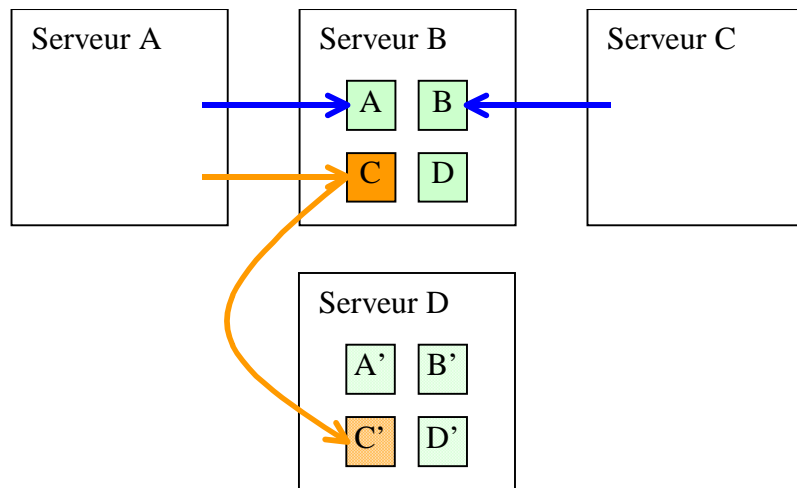


Figure 11

3. Les données sont réparties sur les différents serveurs avec une possibilité de backup (Figure 12).

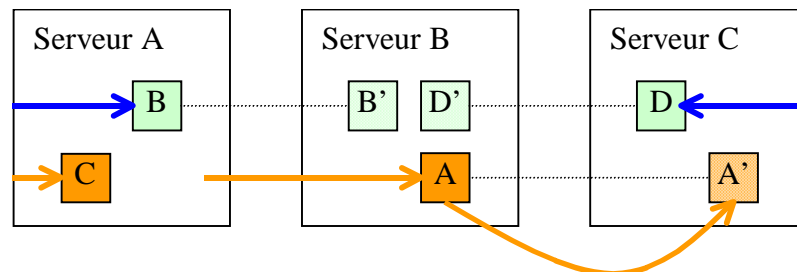


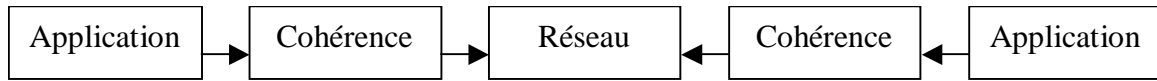
Figure 12

La première approche permet d'améliorer la lecture car les données sont immédiatement disponibles. Elle est préconisée dans le cas où les données sont peu volatiles.

Le deuxième approche permet d'exporter un gros cache dans une machine virtuelle spécifique. Cela réduit la mémoire nécessaire à un serveur d'application par exemple. Le ramasse miette est alors plus efficace. Le serveur Coherence de cache peut être sur le même serveur mais dans une machine virtuelle différente pour supprimer les trafics réseaux.

La troisième approche est préconisée lors d'un cache volumineux où les données sont utilisées généralement par un seul serveur à la fois. Des mécanismes de purge des caches permettent de déplacer les objets où ils sont nécessaires. Des sauvegardes permettent de garantir une disponibilité sans faille si un serveur tombe.

L'approche 1 correspond à la situation choisie dans ce document : peu de volatilité des données. La communication suit le schéma suivant :



#### Avantages :

- Il n'y a pas de référentiel global. Celui-ci est dilué entre tous les serveurs de l'application.
- Il n'est pas nécessaire de faire le ménage lors de l'arrêt de tous les serveurs.
- Le trafic réseau est réduit au minimum.
- Tant qu'il existe un serveur vivant, l'application continue à fonctionner.
- Les serveurs peuvent être redémarrés, ils savent reprendre la situation en cours de route.

#### Inconvénients :

- Il s'agit d'un produit dont il faut acquérir la licence.
- Tous les serveurs doivent être sur le même brin réseau ou les routeurs doivent propager les messages TCP/IP broadcasts.

## 6.6 Synthèse

Différentes approches sont possibles. Une interface normalisée permet de passer d'une approche à une autre sans modification majeure de l'application. Munie de ces outils, un test en grandeur nature permet de sélectionner la meilleure technologie suivant les différents objectifs : performance, trafic réseau, disponibilité et coût.

Parmi les exemples d'utilisation de ces services, citons la gestion d'une liste noire. Si un utilisateur identifié effectue des traitements s'apparentant à du piratage, il est judicieux de l'enregistrer dans une liste noire pour une période donnée. Celle-ci doit être propagée vers les différents serveurs afin qu'ils puissent tous réagir à une nouvelle connexion de cet utilisateur. Cette communication peut prendre différentes approches comme nous l'avons montré.

Une autre utilisation consiste à partager des caches entre serveurs. Par exemple, un cache peut maintenir le contenu d'une table de la base de donnée. Si une nouvelle donnée est ajoutée à la table par l'intermédiaire d'un serveur, les autres membres de la grappe de serveur ne le savent pas. Leurs caches locaux ne sont plus à jour. Notre couteau suisse pour gérer les variables clusters nous permettra de gérer ces situations sans sacrifier les caches locaux.

## 7. CODAGE

Maintenant que les spécifications sont bien en place, que l'interface est spécifiée (`java.util.Map`) il est temps de coder tout cela. Ce sera l'occasion de parcourir de nombreuses technologies. Nous allons commencer par rédiger la version Fichier.

### 7.1 Version Fichier

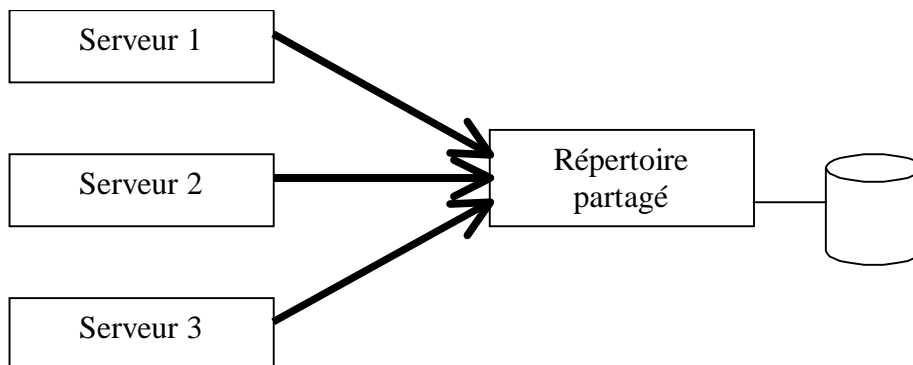


Figure 13

Nous souhaitons permettre à différents serveurs de partager des informations comme si s'agissait de variables globales. Pour offrir ce service, nous avons décidé d'utiliser l'interface `java.util.Map` pour permettre l'ajout et la consultation d'objets Java. Un nom est associé à une Map, puis elle est recherchée par l'application. Le nom identifie le groupe de cluster ayant besoin de partager des informations. La Map est en relation avec un répertoire partagé par tous les serveurs.

À première vue, cette version est simple à coder. Le package `java.io` offre toutes les classes nécessaires à la manipulation de fichiers. En fait, ce n'est pas le cas. En effet, plusieurs processus peuvent manipuler simultanément le même fichier (Figure 13). L'un peut le modifier pendant que l'autre y sérialise un nouvel objet. Comment contrôler cela ? Comment contrôler l'accès aux fichiers entre plusieurs processus ?

À la surprise générale, il est impossible de le faire avant le JDK 1.4. En effet, les classes du package `java.io` ne permettent pas de bloquer l'accès à un fichier. Si plusieurs processus écrivent dans le même fichier, les données sont mélangées !

Le JDK 1.4 offre un nouveau package, `java.nio.channels`, permettant de contrôler l'accès aux fichiers. Les classes `FileInputStream`, `FileOutputStream` et `RandomAccessFile` offrent la nouvelle méthode `getChannel()` permettant d'obtenir un canal d'accès au fichier. Ce canal propose des méthodes pour bloquer le flux ou une partie du flux du fichier. Il est alors possible d'obtenir un accès exclusif sur un fichier en écriture.

```
FileOutputStream out=new FileOutputStream("monFichierPartagé.txt");
out.getChannel().lock(); // JDK 1.4
```

Ainsi, un seul processus peut écrire dans le fichier. Par contre, pour obtenir un accès exclusif en lecture, c'est plus compliqué. En effet, le code suivant ne fonctionne pas :

```
FileInputStream in=new FileInputStream("monFichierPartagé.txt");
in.getChannel().lock(); // JDK 1.4
```

Si le fichier est bloqué par un autre processus, une exception `NonWritableChannelException` est émise lors de l'invocation de la méthode `lock()`. Il n'est pas possible d'obtenir directement un accès exclusif sur un fichier en lecture.

Il faut alors demander un accès en lecture et écriture. La classe `RandomAccessFile` permet cela. Par contre, elle ne permet plus de créer une instance `ObjectInputStream` pour lire une instance. Il faut alors utiliser une astuce pour transformer une instance `RandomAccessFile` en `ObjectInputStream`. Utilisons pour cela l'instance `FieldDescriptor` connectée au fichier.

```
RandomAccessFile rin=new RandomAccessFile("monFichierPartagé.txt", "rw");
FileInputStream in=new FileInputStream(rin.getFD());
in.getChannel().lock(); // JDK 1.4
ObjectInputStream oin=new ObjectInputStream(in);
```

Nous pouvons maintenant décider d'une implémentation. Nous allons créer une classe `FileDistributedMap` qui implémente l'interface `java.util.Map`. À partir d'un répertoire racine, nous allons créer un sous-répertoire pour chaque Map à publier et un fichier `clef.ser` pour chaque clef. Les objets associés aux clefs sont sérialisés dans ces fichiers. Nous pouvons ainsi utiliser les fonctionnalités de manipulation des répertoires pour répondre aux différentes méthodes de la classe `java.util.Map`. La méthode `Map.remove()` utilise la méthode `File.delete()`. La méthode `Map.containsKey()` utilise la méthode `File.exists()`. La méthode `Map.get()` utilise une instance `ObjectInputStream` pour lire l'instance présente dans le fichier `.ser` correspondant. La méthode `Map.put()` utilise une instance `ObjectOutputStream` pour écrire l'instance.

La méthode la plus complexe est la méthode `entrySet()`. Elle doit retourner un tableau d'instance `Map.Entry`, en connexion directe avec le conteneur. Nous allons utiliser une classe interne pour garder un contact avec l'instance `FileDistributedMap` associé. Cela permet d'utiliser la méthode `put()` lors de la méthode `Map.Entry.setValue()`. Le code de cette version est présenté source 6.

```
import java.io.*;
import java.nio.channels.*;
import java.util.*;

public class FileDistributedMap implements Map
{
    private final File base_;
    static File home_;
    FileDistributedMap(String name)
    {
        super();
        base_=new File(home_,name);
        base_.mkdir();
    }
    public void clear()
    {
        File[] list=base_.listFiles();
        for (int i=list.length-1;i>=0;--i)
            list[i].delete();
    }
    public boolean containsKey(Object key)
    {
        return getFile(key).exists();
    }
    public boolean containsValue(Object value)
    {
        File[] list=base_.listFiles();
        for (int i=list.length-1;i>=0;--i)
        {
            String name=list[i].getName();
            name=name.substring(0,name.lastIndexOf('.'));
            Object v=get(name);
            if (value==null ? v==null : value.equals(v))
                return true;
        }
        return false;
    }
    public java.util.Set entrySet()
    {
        class Entry implements Map.Entry
```

```

{
  private Object key_;
  private Object value_;
  public Object getKey()
  {
    return key_;
  }
  public Object getValue()
  {
    return value_;
  }
  public Object setValue(Object value)
  {
    Object old=put(key_,value);
    value_=value;
    return old;
  }
  public boolean equals(Object o)
  {
    if (!(o instanceof Map.Entry))
      return false;
    Map.Entry m=(Map.Entry)o;
    return (getKey()==null ? m.getKey()==null
           : getKey().equals(m.getKey())) &&
           (getValue()==null ? m.getValue()==null
           : getValue().equals(m.getValue()));
  }
  public int hashCode()
  {
    return (getKey()==null ? 0 : getKey().hashCode()) ^
           (getValue()==null ? 0 : getValue().hashCode());
  }

  public String toString()
  {
    return "["+key_+":"+value_+"];
  }
};
Set result=new HashSet();
File[] list=base_.listFiles();
for (int i=list.length-1;i>=0;--i)
{
  String name=list[i].getName();
  name=name.substring(0,name.lastIndexOf('.'));
  Entry entry=new Entry();
  entry.key_=name;
  entry.value_=get(name);
  result.add(entry);
}
return result;
}
public Object get(Object key)
{
  try
  {
    Object data=null;
    File file=getFile(key);
    try
    {
      RandomAccessFile rin=new RandomAccessFile(file,"rw");
      FileInputStream in=new FileInputStream(rin.getFD());
      rin.getChannel().lock(); // JDK 1.4
      data=new ObjectInputStream(in).readObject();
      in.close();
    }
    catch (FileNotFoundException x)
    {
      // Ignore
    }
    return data;
  }
  catch (ClassNotFoundException x)
  {
    throw new DistributedMapException(x);
  }
  catch (IOException x)
  {

```

```

        throw new DistributedMapException(x);
    }
}
private File getFile(Object key)
{
    String skey=(String)key;
    if (skey.indexOf('.')!=-1)
        throw new SecurityException();
    if (skey.indexOf(File.pathSeparatorChar)!=-1)
        throw new SecurityException();
    return new File(base_,skey+".ser");
}
public boolean isEmpty()
{
    return (base_.list().length==0);
}
public java.util.Set keySet()
{
    Set result=new HashSet();
    File[] list=base_.listFiles();
    for (int i=list.length-1;i>=0;--i)
    {
        String n=list[i].getName();
        result.add(n.substring(0,n.length()-4));
    }
    return result;
}
public Object put(Object key, Object value)
{
    try
    {
        Object old=get(key);
        FileOutputStream out=new FileOutputStream(getFile(key));
        out.getChannel().lock(); // JDK 1.4
        new ObjectOutputStream(out).writeObject(value);
        out.close();
        return old;
    }
    catch (IOException x)
    {
        throw new DistributedMapException(x);
    }
}
public void putAll(java.util.Map t)
{
    Set set=t.keySet();
    for (Iterator i=set.iterator();i.hasNext();i)
    {
        String key=(String)i.next();
        Object data=t.get(key);
        put(key,data);
    }
}
public Object remove(Object key)
{
    Object old=get(key);
    getFile(key).delete();
    return old;
}
public static void setLink(String uri) throws java.net.MalformedURLException
{
    home_=new File(new java.net.URL(uri).getFile()
        .replace('/',File.separatorChar));
}
public int size()
{
    return base_.list().length;
}
public java.util.Collection values()
{
    Collection result=new ArrayList();
    File[] list=base_.listFiles();
    for (int i=list.length-1;i>=0;--i)
    {
        String name=list[i].getName();
        name=name.substring(0,name.lastIndexOf('.'));
        result.add(get(name));
    }
}

```

```

    }
    return result;
  }
}

```

#### Source 6

La méthode statique `setLink()` permet d'indiquer à l'aide d'une URL la racine du répertoire servant de référentiel.

```

FileDistributedMap.setLink("file://master/shared");
varcluster=new FileDistributedMap("varcluster");

```

Nous verrons plus tard comment utiliser une instance fabricante pour lier toutes les versions.

Voilà la première version de nos variables de clusters. Plusieurs serveurs peuvent manipuler une Map pour partager des informations. Nous allons décliner ce concept à l'aide d'autres technologies. Nous avons usiné notre première lame de notre couteau suisse de variables clusters.

## 7.2 Version JDBC

Nous souhaitons proposer un mécanisme standard pour partager des variables entre différents serveurs. Un mécanisme de répartition de charge s'occupe de diriger les utilisateurs vers les serveurs de la grappe. Ceux-ci doivent partager quelques informations et pouvoir les mettre à jour. Pour des raisons évidentes de performance, ces données sont peu volatiles. Elles sont rarement modifiées, mais consultées par tous les serveurs. Nous avons proposé une première version qui utilise un répertoire partagé entre tous les serveurs. Ce répertoire sert de référentiel entre les différents serveurs.

Les applications importantes, celle nécessitant la mise en place de cluster, utilisent généralement une base de donnée. Java offre l'API normalisée JDBC pour pouvoir invoquer une base de donnée relationnelle. Cela permet d'ouvrir une connexion avec la base, de lui envoyer des requêtes et d'y obtenir des informations.

Nous allons utiliser cette API pour implémenter l'interface `java.util.Map`. Nous souhaitons sauver des objets dans une table de la base, associé à une clef. Ainsi, tous les serveurs partageant la même base de données auront accès aux mêmes informations (Figure 14).

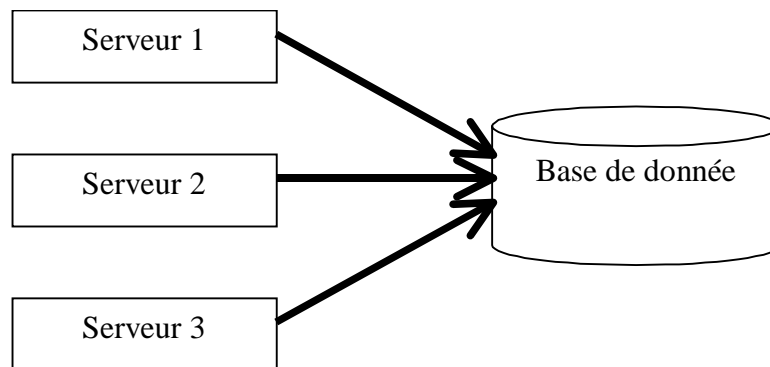


Figure 14

Dans un premier temps, nous devons régler le problème de la connexion avec la base de donnée. Il existe deux approches pour cela : Utiliser la classe `DriverManager` ou récupérer une instance `DataSource` d'un serveur JNDI (Java Naming and Directory Interface).

La classe `DriverManager` est en relation avec les différents drivers JDBC. La méthode `getConnection()` attend une URI pour identifier : le type de base de donnée à utiliser, le serveur où elle se trouve et le nom de la base. D'autres paramètres permettent d'indiquer le nom et le mot de passe de l'utilisateur.

```

DriverManager.getConnection("jdbc:oracle:thin:@localhost:madb","sa","");

```

Cet exemple ouvre une base oracle `madb` sur le serveur local.

Il faut auparavant enregistrer les différents drivers JDBC possible. Une approche consiste à valoriser la variable d'environnement `jdbc.drivers` pour y indiquer les différents drivers.

```

jdbc.drivers=COM.ibm.db2.jdbc.app.DB2Driver|org.hsql.jdbcDriver

```

La deuxième technique de connexion à la base de donnée utilise l'API JNDI pour obtenir une instance `DataSource`. Cette instance s'occupe de maintenir un cache des connexions avec la base de donnée. L'ouverture d'une connexion prend du temps et consomme des ressources. Pour optimiser cette phase, une instance `DataSource` s'occupe de maintenir un pool de connexion ouverte. Cela est particulièrement utile lors d'application Internet ou Intranet. Les servlets peuvent se partager les connexions avec la base.

D'autres part, les bases de données peuvent préparer une stratégie d'exécution d'une requête sans pour autant l'exécuter. Cette préparation prend du temps qui peut, dans certains cas, être supérieur à l'exécution de la requête elle-même. L'instance `DataSource` est capable de garder un cache des `preparedStatements` associés à une connexion. Ainsi, la base de donnée peut préparer le travail lors du premier accès, et n'effectuer que les recherches lors de l'invocation des transactions SQL.

JNDI est une API permettant de communiquer avec un annuaire. Il existe différents drivers JNDI. À l'URL <http://java.sun.com/jndi> vous trouverez plusieurs drivers pour manipuler un serveur DNS, les fichiers sur le disque, des objets CORBA, RMI ou un serveur LDAP. Dans certains annuaires, il est possible d'y mémoriser des objets java sérialisés.

En consultant l'annuaire, l'application peut obtenir une copie d'une instance en mémoire. Ainsi, avec l'aide d'une interface utilisateur spécifique, un administrateur peut modifier le comportement de l'application en mémorisant des objets dans l'annuaire. Il peut indiquer les différents paramètres d'accès à la base de donnée, les différents paramètres de caches, etc. Une instance dérivée de `DataSource` est alors construite, puis mémorisée dans l'annuaire.

Pour obtenir cette instance, l'application doit commencer par ouvrir l'annuaire. Pour cela, il faut créer une instance `javax.naming.InitialContext()` et lui demander de consulter une clef sous la branche « `jdbc` ».

```
ds=(DataSource)new InitialContext().lookup("jdbc/DistributedMap");
```

L'instance `DataSource` est alors utilisée pour communiquer avec la base. Le paramétrage d'un serveur d'application permet généralement d'utiliser un serveur JNDI par défaut. Si ce n'est pas le cas, ou si vous désirez utiliser un annuaire spécifique, il faut valoriser les paramètres d'environnements `java.naming.factory.initial` et `java.naming.provider.url`.

Une autre approche consiste à construire une instance `Hashtable()`, d'y valoriser les deux paramètres et de l'utiliser pour initialiser le contexte initial.

```
Hashtable env=new java.util.Hashtable();
env.put(Context.INITIAL_CONTEXT_FACTORY,
        "com.ibm.ejs.ns.jndi.CNInitialContextFactory");
env.put(Context.PROVIDER_URL,
        "iiop://localhost:900");
```

Avec JNDI 1.1, il est possible de valoriser ces paramètres ou de déclarer les différents annuaires disponibles à l'aide d'un fichier `jndi.properties`.

```
java.naming.factory.initial=com.sun.jndi.ldap.LdapCtxFactory
java.naming.provider.url=ldap://localhost:389/o=jnditutorial
```

Ce fichier est à placer dans la racine d'une des archives de l'application ou dans le répertoire `/${JAVA_HOME}/lib`.

Avec l'instance `DataSource`, nous pouvons maintenant ouvrir une connexion avec la base de données. Plus exactement, nous obtenons une connexion disponible.

```
DataSource ds=
    (DataSource)new InitialContext().lookup("jdbc/DistributedMap");
Connexion cnx=ds.getConnection("sa","pass");
```

Attention, il ne faut pas oublier de libérer la connexion lorsqu'elle n'est plus nécessaire à l'aide d'un `cnx.close()`. Elle est alors remise dans le cache et peut servir à une autre tâche. Cette approche est très utilisée dans les serveurs d'applications. Les servlets peuvent se partager les connexions et les statements. Le serveur d'application se charge d'instancier et d'enregistrer une instance `DataSource` dans le serveur JNDI.

Nous allons rédiger la classe `JDBC DistributedMap` pour utiliser une base de donnée JDBC comme référentiel des différents serveurs de la grappe.

La méthode `getConnexion()` de la classe se charge d'utiliser l'approche `DriverManager` ou `DataSource` suivant les cas.

Les drivers JDBC savent invoquer la base de donnée suivant deux approches :

- Soit le développeur construit une chaîne de caractère avec sa requête et demande à la base de donnée de l'exécuter ;

```
Connection cnx= getConnection();
cnx.createStatement().executeQuery("select * from user where name="+nom)
```

- Soit le développeur déclare sa requête en laissant quelques variables vides. Il demande alors à la base de donnée de préparer le traitement. Ensuite, il suffit de valoriser les variables pour pouvoir lancer l'exécution de la requête.

```
Connection cnx= getConnection();
PreparedStatement stm=
    cnx.prepareStatement("select * from user where name=?");
stm.setString(1,nom);
stm.executeQuery();
```

La deuxième approche est préférable pour deux raisons :

- La valorisation des variables est faite par le driver JDBC. On est alors certains qu'il n'y a pas de caractères spéciaux dans la variable `nom` qui pourrait modifier l'interprétation de la requête. Les pirates exploitent souvent cette faille pour modifier la base de donnée lorsque les requêtes sont construites trop naïvement.
- La base de donnée peut analyser et préparer le traitement sans connaître la valeur du paramètre `name` de la clause `where`. Cette préparation peut être recyclée par l'application. Ainsi les performances sont supérieures à partir d'un nombre important de requête.

Nous allons préparer toutes les requêtes dans le constructeur de notre classe. Nous souhaitons utiliser une table différente pour chaque Map nécessaire à l'application. Le nom d'une table ne peut pas être un paramètre d'un `preparedStatement`. Nous construisons alors toutes les chaînes de caractères qui seront nécessaire à notre classe, et les mémorisons dans des variables d'instances.

Lors de l'invocation du constructeur, il faut vérifier si la table SQL correspondant à la Map est présente. Si ce n'est pas le cas, nous allons tenter de la construire. Nous souhaitons une table dont le nom est construit à partir du nom de la Map. Cette table doit avoir deux colonnes : la première possède la clef. Elle est du type `VARCHAR(32)`. Nous y stockerons une chaîne de caractères ; La deuxième colonne est du type `BINARY`. En effet, nous souhaitons y placer des objets sérialisés.

```
PreparedStatement stm=cnx.prepareStatement("CREATE TABLE "+name_+" (key VARCHAR(32) PRIMARY
KEY, data BINARY )");
stm.executeUpdate();
```

Nous profitons de cette création pour ajouter un index à notre table.

```
stm=cnx.prepareStatement("CREATE UNIQUE INDEX key ON "+name_+" (KEY)");
stm.executeUpdate();
```

Si l'utilisateur n'a pas les droits pour construire ces tables, elles doivent être construites à la main avant l'utilisation de l'application.

Voilà, tout est prêt pour décliner les différentes méthodes de l'interface `java.util.Map` à l'aide d'un driver JDBC. Cette interface ne retourne pas d'exceptions. Nous devons alors transformer les exceptions JDBC en exception `RuntimeException`.

La méthode `put()` utilise l'instruction SQL `INSERT` pour ajouter une nouvelle valeur. L'objet à mémoriser est d'abord sérialisé dans un tampon avant d'alimenter la requête.

```
ByteArrayOutputStream buf=new ByteArrayOutputStream();
new ObjectOutputStream(buf).writeObject(value);
stm.setBytes(2,buf.toByteArray());
```

La méthode `get()` exécute un `SELECT` puis manipule l'instance `ResultSet` pour déplier l'instance.

```
rs=stm.executeQuery();
if (!rs.next()) return null;
return new ObjectInputStream(rs.getBinaryStream("data")).readObject();
```

La méthode `size()` utilise habilement l'instruction `COUNT()` de SQL.

```
"SELECT COUNT(*) FROM "+name_
```

Afin d'être certain de bien fermer tous les `ResultSet` et tous les `PreparedStatement`, les méthodes utilisent l'instruction `finally` et capture individuellement les exceptions `SQLException` qui pourraient apparaître. En effet, il est possible de recevoir une exception lors de la fermeture d'un résultat ou d'une connexion. Ces erreurs ne sont plus importantes à ce niveau.

```
PreparedStatement stm=null;
ResultSet rs=null;
try
{
    ...
}
finally
{
    try
    {
        if (rs!=null) rs.close();
    }
    catch (SQLException x) { } // Ignore
    try
    {
        if (stm!=null) stm.close();
    }
    catch (SQLException x) { } // Ignore
}
```

Le code de cette version est présenté source 7.

```
import java.io.*;
import java.util.*;
import java.sql.*;
import javax.sql.*;

final public class JDBCdistributedMap implements Map
{
    private String name_="test";
    private final String get_;
    private final String put_;
    private final String size_;
    private final String remove_;
    private final String clear_;
    private final String containsKey_;
    private final String keySet_;
    private final String entrySet_;
    private final String values_;

    private static String jdbcUser_;
    private static String jdbcPassword_;

    // Use direct
    private static String jdbcUri_;
```

```

// Use JNDI
private static DataSource dataSource_; // Connexion via jndi
JDBCDistributedMap(String name)
{
    name_="shared_"+name;
    get_="SELECT key,data FROM "+name_+" WHERE key=?";
    put_="INSERT INTO "+name_+" VALUES(?,?)";
    size_="SELECT count(*) FROM "+name_;
    remove_="DELETE FROM "+name_+" WHERE key=?";
    clear_="DELETE FROM "+name_;
    containsKey_="SELECT COUNT(*) FROM "+name_+" WHERE key=?";
    keySet_="SELECT key FROM "+name_;
    entrySet_="SELECT key,data FROM "+name_;
    values_="SELECT data FROM "+name_;
    try
    {
        Connection cnx=getConnection();
        PreparedStatement stm=cnx.prepareStatement("CREATE TABLE "+
            name_+" (key VARCHAR(32) PRIMARY KEY, data BINARY )");
        stm.executeUpdate();
        stm=cnx.prepareStatement("CREATE UNIQUE INDEX key ON "+
            name_+" (KEY)");
        stm.executeUpdate();
    }
    catch (SQLException x)
    {
        throw new DistributedMapException(x);
    }
}
public void clear()
{
    PreparedStatement stm=null;
    try
    {
        Connection cnx=getConnection();
        cnx.setReadOnly(false);
        stm=cnx.prepareStatement(clear_);
        stm.executeUpdate();
    }
    catch (SQLException x)
    {
        throw new DistributedMapException(x);
    }
    finally
    {
        try
        {
            if (stm!=null) stm.close();
        }
        catch (SQLException x) { } // Ignore
    }
}
public boolean containsKey(Object key)
{
    PreparedStatement stm=null;
    ResultSet rs=null;
    try
    {
        Connection cnx=getConnection();
        cnx.setReadOnly(true);
        stm=cnx.prepareStatement(containsKey_);
        stm.setString(1, (String)key);
        rs=stm.executeQuery();
        rs.next();
        return rs.getInt(1)==1;
    }
    catch (SQLException x)
    {
        throw new DistributedMapException(x);
    }
    finally
    {
        try
        {
            if (rs!=null) rs.close();
        }
        catch (SQLException x) { } // Ignore
    }
}

```

```

        try
        {
            if (stm!=null) stm.close();
        }
        catch (SQLException x) { } // Ignore
    }
}
public boolean containsValue(Object v)
{
    Collection set=values();
    for (Iterator i=set.iterator();i.hasNext();)
    {
        Object value=i.next();
        if (value==null ? v==null : value.equals(v))
            return true;
    }
    return false;
}
public Set entrySet()
{
    class Entry implements Map.Entry
    {
        private Object key_;
        private Object value_;
        Entry(Object key,Object value)
        {
            key_=key;
            value_=value_;
        }
        public Object getKey()
        {
            return key_;
        }
        public Object getValue()
        {
            return value_;
        }
        public Object setValue(Object value)
        {
            Object old=put(key_,value);
            value_=value;
            return old;
        }
        public boolean equals(Object o)
        {
            if (!(o instanceof Map.Entry))
                return false;
            Map.Entry m=(Map.Entry)o;
            return (getKey()==null ? m.getKey()==null
                    : getKey().equals(m.getKey())) &&
                (getValue()==null ? m.getValue()==null
                    : getValue().equals(m.getValue()));
        }
        public int hashCode()
        {
            return (getKey()==null ? 0 : getKey().hashCode()) ^
                (getValue()==null ? 0 : getValue().hashCode());
        }
        public String toString()
        {
            return "["+key_+"':" +value_+"']";
        }
    };
    PreparedStatement stm=null;
    ResultSet rs=null;
    try
    {
        Set result=new HashSet();
        Connection cnx=getConnection();
        cnx.setReadOnly(true);
        stm=cnx.prepareStatement(entrySet_);
        for (rs=stm.executeQuery();rs.next();)
        {
            result.add(new Entry(rs.getString(1),
                new ObjectInputStream(rs.getBinaryStream(2)).readObject()));
        }
    }
}

```

```

        return result;
    }
    catch (ClassNotFoundException x)
    {
        throw new DistributedMapException(x);
    }
    catch (IOException x)
    {
        throw new DistributedMapException(x);
    }
    catch (SQLException x)
    {
        throw new DistributedMapException(x);
    }
    finally
    {
        try
        {
            if (rs!=null) rs.close();
        }
        catch (SQLException x) { } // Ignore
        try
        {
            if (stm!=null) stm.close();
        }
        catch (SQLException x) { } // Ignore
    }
}
public Object get(Object key)
{
    try
    {
        return get(getConnection(),key);
    }
    catch (SQLException x)
    {
        throw new DistributedMapException(x);
    }
}
private Object get(Connection cnx,Object key)
{
    PreparedStatement stm=null;
    ResultSet rs=null;
    try
    {
        stm=cnx.prepareStatement(get_);
        cnx.setReadOnly(true);
        stm.setString(1,(String)key);
        rs=stm.executeQuery();
        if (!rs.next()) return null;
        return new ObjectInputStream(
            rs.getBinaryStream("data")).readObject();
    }
    catch (IOException x)
    {
        throw new DistributedMapException(x);
    }
    catch (ClassNotFoundException x)
    {
        throw new DistributedMapException(x);
    }
    catch (SQLException x)
    {
        throw new DistributedMapException(x);
    }
    finally
    {
        try
        {
            if (rs!=null) rs.close();
        }
        catch (SQLException x) { } // Ignore
        try
        {
            if (stm!=null) stm.close();
        }
        catch (SQLException x) { } // Ignore
    }
}

```

```

    }
}
private Connection getConnection() throws SQLException
{
    if (dataSource_==null)
    {
        return DriverManager.getConnection(jdbcUri_, jdbcUser_, jdbcPassword_);
    }
    else
    {
        return dataSource_.getConnection(jdbcUser_, jdbcPassword_);
    }
}
public boolean isEmpty()
{
    return size()==0;
}
public Set keySet()
{
    PreparedStatement stm=null;
    ResultSet rs=null;
    try
    {
        Set set=new HashSet();
        Connection cnx=getConnection();
        cnx.setReadOnly(true);
        stm=cnx.prepareStatement(keySet_);
        for (rs=stm.executeQuery();rs.next(); )
        {
            set.add(rs.getString(1));
        }
        return set;
    }
    catch (SQLException x)
    {
        throw new DistributedMapException(x);
    }
    finally
    {
        try
        {
            if (rs!=null) rs.close();
        }
        catch (SQLException x) { } // Ignore
        try
        {
            if (stm!=null) stm.close();
        }
        catch (SQLException x) { } // Ignore
    }
}
public Object put(Object key, Object value)
{
    PreparedStatement stm=null;
    try
    {
        Connection cnx=getConnection();
        cnx.setAutoCommit(false);

        Object old=get(cnx,key);
        cnx.setReadOnly(false);
        stm=cnx.prepareStatement(put_);
        stm.setString(1,(String)key);
        ByteArrayOutputStream buf=new ByteArrayOutputStream();
        new ObjectOutputStream(buf).writeObject(value);
        stm.setBytes(2,buf.toByteArray());
        stm.executeUpdate();
        cnx.commit();
        return old;
    }
    catch (IOException x)
    {
        throw new DistributedMapException(x);
    }
    catch (SQLException x)
    {
        throw new DistributedMapException(x);
    }
}

```

```

    }
    finally
    {
        try
        {
            if (stm!=null) stm.close();
        }
        catch (SQLException x) { } // Ignore
    }
}
public void putAll(Map t)
{
    PreparedStatement stm=null;
    try
    {
        Connection cnx=getConnection();
        cnx.setAutoCommit(false);
        cnx.setReadOnly(false);

        Set set=t.keySet();
        stm=cnx.prepareStatement(put_);
        for (Iterator i=set.iterator();i.hasNext();)
        {
            String key=(String)i.next();
            Object value=t.get(key);
            stm.setString(1,(String)key);
            ByteArrayOutputStream buf=new ByteArrayOutputStream();
            new ObjectOutputStream(buf).writeObject(value);
            stm.setBytes(2,buf.toByteArray());
            stm.executeUpdate();
        }
        cnx.commit();
    }
    catch (IOException x)
    {
        throw new DistributedMapException(x);
    }
    catch (SQLException x)
    {
        throw new DistributedMapException(x);
    }
    finally
    {
        try
        {
            if (stm!=null) stm.close();
        }
        catch (SQLException x) { } // Ignore
    }
}
public Object remove(Object key)
{
    PreparedStatement stm=null;
    try
    {
        Connection cnx=getConnection();
        Object old=get(cnx,key);
        cnx.setReadOnly(false);
        stm=cnx.prepareStatement(remove_);
        stm.setString(1,(String)key);
        stm.executeUpdate();
        return old;
    }
    catch (SQLException x)
    {
        throw new DistributedMapException(x);
    }
    finally
    {
        try
        {
            if (stm!=null) stm.close();
        }
        catch (SQLException x) { } // Ignore
    }
}
public static void setLink(String uri)

```

```

throws java.net.MalformedURLException, javax.naming.NamingException
{
String user="";
String pass="";
if (!uri.startsWith("jdbc:"))
    throw new java.net.MalformedURLException();
int prot=5;
if (uri.startsWith("jdbc://"))
    prot=7;
int idx=uri.indexOf('@',prot);
if (idx!=-1)
{
    String userpass=uri.substring(prot,idx);
    int idx2=userpass.indexOf(':');
    if (idx2!=-1)
    {
        user=userpass.substring(0,idx2);
        pass=userpass.substring(idx2+1);
    }
    else
        user=userpass;
    uri="jdbc:"+uri.substring(idx+1);
}
// Check if use jndi
String path=uri.substring(5);
if (path.startsWith("jndi:"))
{
    dataSource_=
        (DataSource)new javax.naming.InitialContext()
            .lookup(path.substring(5));
}
else
    jdbcUri_=uri;
    jdbcUser_=user;
    jdbcPassword_=pass;
}
public int size()
{
    PreparedStatement stm=null;
    ResultSet rs=null;
    try
    {
        Connection cnx=getConnection();
        cnx.setReadOnly(true);
        stm=cnx.prepareStatement(size_);
        rs=stm.executeQuery();
        rs.next();
        return rs.getInt(1);
    }
    catch (SQLException x)
    {
        throw new DistributedMapException(x);
    }
    finally
    {
        try
        {
            if (rs!=null) rs.close();
        }
        catch (SQLException x) { } // Ignore
        try
        {
            if (stm!=null) stm.close();
        }
        catch (SQLException x) { } // Ignore
    }
}
public Collection values()
{
    PreparedStatement stm=null;
    ResultSet rs=null;
    try
    {
        Collection set=new ArrayList();
        Connection cnx=getConnection();
        cnx.setReadOnly(true);
        stm=cnx.prepareStatement(values_);

```

```

        for (rs=stm.executeQuery();rs.next();)
        {
            set.add(new ObjectInputStream(
                rs.getBinaryStream(1)).readObject());
        }
        return set;
    }
    catch (SQLException x)
    {
        throw new DistributedMapException(x);
    }
    catch (ClassNotFoundException x)
    {
        throw new DistributedMapException(x);
    }
    catch (IOException x)
    {
        throw new DistributedMapException(x);
    }
    finally
    {
        try
        {
            if (rs!=null) rs.close();
        }
        catch (SQLException x) { } // Ignore
        try
        {
            if (stm!=null) stm.close();
        }
        catch (SQLException x) { } // Ignore
    }
}
}
}

```

#### Source 7

Pour initialiser cette version, il faut invoquer la méthode statique `setLink()` en lui fournissant une URL de type `jdbc`. Cette URL a été étendu pour pouvoir recevoir le nom de l'utilisateur et son mot de passe. La syntaxe respecte la syntaxe des URL HTTP. Il faut indiquer le nom de l'utilisateur suivi du caractère deux-points, puis son mot de passe, et enfin un caractère arabase.

```
jdbc:user[:password]@//driver/database
```

Par exemple, la connexion peut être initialisé ainsi :

```
JDBCDistributedMap.setLink("jdbc:sa:pass@oracle:thin:@localhost:madb");
```

Dans ce cas, l'approche `DriverManager` est utilisée.

Pour bénéficier du pool de connexion, il faut indiquer un driver JNDI et une clef.

```
JDBCDistributedMap.setLink("jdbc:sa:pass@jndi://jdbc/DistributedMap");
varcluster=new JDBCDistributedMap("varcluster");
```

Ainsi, le driver JNDI par défaut est utilisé et la clef `/jdbc/DistributedMap` doit posséder une instance sérialisée d'un `DriverManager`.

Nous venons de forger la deuxième lame de notre couteau suisse de variable cluster. Nous verrons plus tard comment regrouper les lames dans le même couteau. Les différentes versions utiliseront toute une approche similaire pour l'initialisation. Une simple URL permettra d'associer une technologie et ces paramètres.

### 7.3 Version LDAP

Nous avons proposé deux versions pour partager des variables entres différents serveurs : utiliser un répertoire partagé ou exploiter une base de données relationnelles. Nous allons maintenant nous intéresser aux bases de données LDAP.

J2EE propose l'API JNDI pour manipuler différents annuaires. Des drivers permettent de s'interfacer avec un serveur de nom CORBA, la base de registre RMI, un disque dur, un serveur DNS ou un serveur LDAP ([java.sun.com/jndi](http://java.sun.com/jndi)). Les annuaires permettent d'organiser des informations dans une structure arborescente et d'associer des attributs à chaque objet. Les drivers offrent plus moins toutes les fonctionnalités JNDI (Tableau 1).

Tableau 1

Annuaire	Limitation
CORBA	N'accepte de référencer que des objets CORBA.
RMI	N'accepte de référencer que des objets RMI, et n'accepte pas les répertoires.
Disque dur	Ne peut référencer que des objets <code>java.io.File</code> ou des références.

DNS	Ne fonctionne qu'en lecture.
LDAP	Utilise des noms complexes pour organiser les informations.

Une base de donnée LDAP est capable de mémoriser de nombreuses informations structurées et de les retrouver rapidement. Vous pouvez télécharger OpenLDAP, une implémentation Open Source de ce protocole ([www.openldap.org](http://www.openldap.org)). Une version compilée pour Windows est disponible sur le site de FiveSight ([www.fivesight.com](http://www.fivesight.com))

Une base LDAP est organisée en couple clef-valeur. Chaque branche possède un nom et une valeur. Pour naviguer dans l'arborescence, il faut indiquer les différentes clefs et leurs valeurs dans l'ordre inverse de la navigation. Par exemple, l'expression « `cn=Manager,dc=mon-site,dc=org` » indique que le couple `cn=Manager` est présent dans le sous-répertoire `dc=mon-site`, lui-même présent le répertoire `dc=org` (Figure 15).

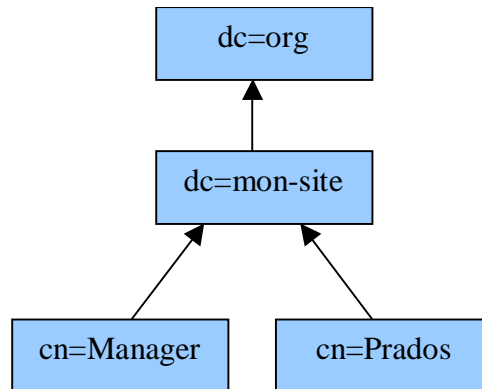


Figure 15

Nous allons proposer une version de l'interface `java.util.Map` s'interfacant à un serveur LDAP. Nous allons utiliser un répertoire LDAP `shared` pour y construire des sous-répertoires et y sauver des objets java. Pour que cela fonctionne, il faut ajouter le schéma java à la base LDAP. Cela s'effectue en incluant le fichier `java.schema` dans le fichier de paramétrage. Le schéma java permet de sérialiser des objets java dans l'annuaire. Il faut également ajouter un droit d'accès au répertoire LDAP qui sera manipulé par notre application. Le fichier de paramétrage `slapd.conf` ressemble à ceci :

```

include schema/core.schema
include schema/java.schema

pidfile slapd.pid
argsfile slapd.args
access to dn=shared by * write
access to *
    by self write
    by users read
    by anonymous auth
database ldbm
suffix "dc=mon-site,dc=org"
rootdn "cn=Manager,dc=mon-site,dc=org"
rootpw secret
directory openldap-ldbm
index objectClass eq
  
```

Ce fichier indique que l'administrateur possède le nom « `cn=Manager,dc=mon-site,dc=org` » et que son mot de passe est « `secret` ». Le schéma standard et le schéma java sont présent. Tout le monde peut accéder en écriture dans le répertoire `shared`.

Il faut ensuite initialiser la base de donnée en important un premier enregistrement. Pour cela, construisez un fichier `manager.ldif` comme ceci :

```

dn: dc=mon-site,dc=org
objectclass: dcObject
dc: mon-domain

dn: cn=Manager,dc=mon-site,dc=org
objectclass: organizationalRole
cn: Manager
  
```

Attention, il ne faut pas ajouter d'espace supplémentaire ou de ligne vide. Importez ensuite le fichier dans la base de données. Utilisez l'utilitaire `slapadd`.

```
slapadd -f slapd.conf -l manager.ldif
```

Vous pouvez alors lancer le démon LDAP en invoquant le programme `slapd`. Auparavant, nous allons créer le répertoire `shared` qui nous servira à partager les données entre tous les serveurs. Le fichier `shared.ldif` suivant permet la création de ce répertoire.

```
dn: dc=shared,dc=mon-site,dc=org
objectclass: dcObject
dc: shared
```

Ajoutons ce répertoire dans la base LDAP.

```
slapadd -f slapd.conf -l shared.ldif
```

Nous démarrons alors le démon LDAP.

```
slapd
```

Nous allons pouvoir utiliser cet annuaire ou cette base de donnée pour servir de référentiel à notre groupe de serveurs. La classe `LdapDistributedMap` s'occupe de proposer un accès à des objets, par l'intermédiaire d'un serveur LDAP. Nous allons abondamment utiliser l'API JNDI pour faire cela.

Il faut obtenir un premier contexte vers la base LDAP. Pour cela, nous allons ouvrir un contexte initial et demander un accès à une racine particulière. Celle-ci est exprimée à l'aide d'une URL.

```
ldap://localhost/dc=shared,dc=mon-site,dc=org
```

Elle indique un désir de connexion à un serveur LDAP, présent sur le poste local (`localhost`), utilisant le port par défaut (`389`). Nous désirons commencer à travailler à partir du nœud `dc=shared,dc=mon-site,dc=org`. La méthode `LdapDistributedMap.setLink()` est là pour obtenir le contexte de travail initial. Elle reçoit une URL et ouvre la connexion.

À partir du contexte JNDI, nous allons pouvoir créer des sous-contextes pour chaque MAP partagée. Dans chaque sous-contexte, nous placerons les objets java sérialisés (Figure 16).

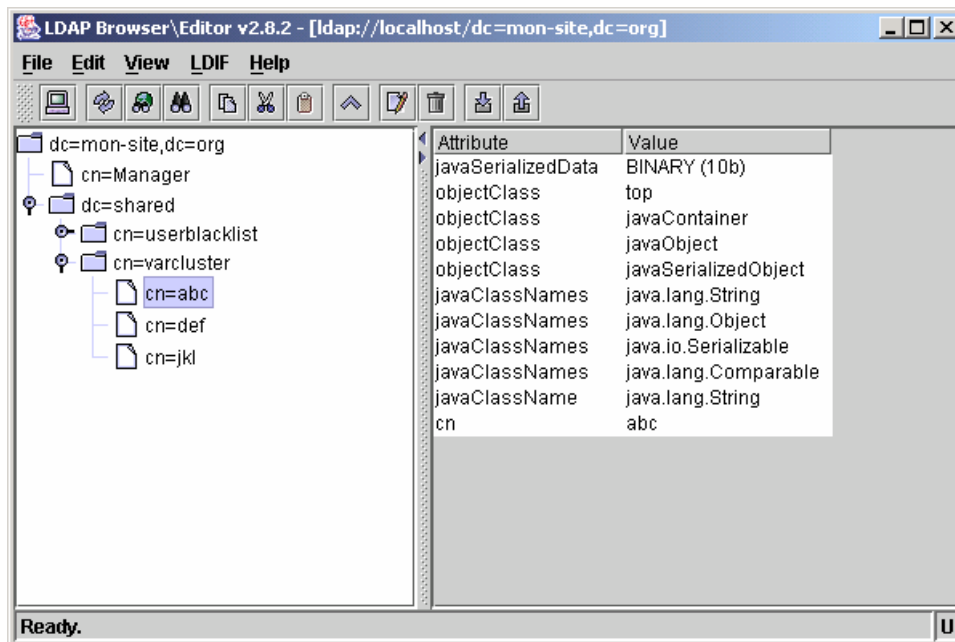


Figure 16

La méthode `rebind()` d'un contexte JNDI permet de sauver un objet java sérialisable dans l'annuaire. Pour un annuaire LDAP, la clef doit posséder le préfixe « `cn=` ». Cela limite l'universalité de l'API JNDI. En effet, l'application doit savoir qu'elle manipule un annuaire LDAP et non un annuaire CORBA par exemple. Nous ajoutons, quand cela est nécessaire, « `cn=` » avant d'invoquer l'API JNDI (Source 8). La méthode `lookup()` permet de retrouver un objet à partir de sa clef.

```
import java.util.*;
import javax.naming.*;

final public class LDAPDistributedMap implements Map
{
    private static Context initCtx_;
    private Context ctx_;
    public LDAPDistributedMap(String name)
    {
        try
        {
            String folder=initCtx_.composeName("cn="+name,"");
            try
            {
```

```

        ctx_=initCtx_.createSubcontext(folder);
    }
    catch (NameAlreadyBoundException x)
    {
        ctx_=(Context)initCtx_.lookup(folder);
    }
}
catch (NamingException x)
{
    throw new DistributedMapException(x);
}
}
public void clear()
{
    try
    {
        Vector names=new Vector();
        for (NamingEnumeration e=ctx_.list("");e.hasMore());
        {
            names.add(e.next());
        }
        for (int i=names.size()-1;i>=0;--i)
            ctx_.unbind(((NameClassPair)names.get(i)).getName());
    }
    catch (NamingException x)
    {
        throw new DistributedMapException(x);
    }
}
public boolean containsKey(Object key)
{
    try
    {
        ctx_.lookup("cn="+key);
        return true;
    }
    catch (NameNotFoundException x)
    {
        return false;
    }
    catch (NamingException x)
    {
        throw new DistributedMapException(x);
    }
}
public boolean containsValue(Object value)
{
    try
    {
        Collection result=new Vector();
        for (NamingEnumeration e=ctx_.listBindings("");e.hasMore());
        {
            Binding binding=(Binding)e.next();
            Object v=binding.getObject();
            if (value==null ? v==null : value.equals(v)) return true;
        }
        return false;
    }
    catch (NamingException x)
    {
        throw new DistributedMapException(x);
    }
}
public java.util.Set entrySet()
{
    class Entry implements Map.Entry
    {
        private Object key_;
        private Object value_;
        public Object getKey()
        {
            return key_;
        }
        public Object getValue()
        {
            return value_;
        }
    }
}

```

```

public Object setValue(Object value)
{
    Object old=put(key_,value);
    value_=value;
    return old;
}
public boolean equals(Object o)
{
    if (!(o instanceof Map.Entry))
        return false;
    Map.Entry m=(Map.Entry)o;
    return (getKey()==null ? m.getKey()==null
            : getKey().equals(m.getKey())) &&
           (getValue()==null ? m.getValue()==null
            : getValue().equals(m.getValue()));
}
public int hashCode()
{
    return (getKey()==null ? 0 : getKey().hashCode()) ^
           (getValue()==null ? 0 : getValue().hashCode());
}
public String toString()
{
    return "["+key_+":"+value_+"];
}
};
try
{
    Set result=new HashSet();
    for (NamingEnumeration e=ctx_.listBindings("");e.hasMore();)
    {
        Binding binding=(Binding)e.next();
        Entry entry=new Entry();
        entry.key_=binding.getName().substring(3);
        entry.value_=binding.getObject();
        result.add(entry);
    }
    return result;
}
catch (NamingException x)
{
    throw new DistributedMapException(x);
}
}
public Object get(Object key)
{
    try
    {
        return ctx_.lookup("cn="+key);
    }
    catch (NameNotFoundException x)
    {
        return null;
    }
    catch (NamingException x)
    {
        throw new DistributedMapException(x);
    }
}
public boolean isEmpty()
{
    return size()==0;
}
public java.util.Set keySet()
{
    try
    {
        Set result=new HashSet();
        for (NamingEnumeration e=ctx_.listBindings("");e.hasMore();)
        {
            Binding binding=(Binding)e.next();
            result.add(binding.getName().substring(3));
        }
        return result;
    }
    catch (NamingException x)
    {

```

```

        throw new DistributedMapException(x);
    }
}
public Object put(Object k, Object value)
{
    try
    {
        Object old=get(k);
        ctx_.rebind("cn="+k,value);
        return old;
    }
    catch (NamingException x)
    {
        throw new DistributedMapException(x);
    }
}
public void putAll(java.util.Map t)
{
    Set set=t.keySet();
    for (Iterator i=set.iterator();i.hasNext();i)
    {
        String key=(String)i.next();
        Object data=t.get(key);
        put(key,data);
    }
}
public Object remove(Object key)
{
    try
    {
        Object old=get(key);
        ctx_.unbind("cn="+key);
        return old;
    }
    catch (NamingException x)
    {
        throw new DistributedMapException(x);
    }
}
static public void setLink(String uri)
    throws javax.naming.NamingException
{
    setLink(new InitialContext(),uri);
}
static public void setLink(Context ctx,String uri)
    throws javax.naming.NamingException
{
    try
    {
        initCtx_=(Context)ctx.lookup(uri);
    }
    catch (javax.naming.NameNotFoundException x)
    {
        initCtx_=ctx.createSubcontext(uri);
    }
}
public int size()
{
    try
    {
        int s=0;
        for (NamingEnumeration e=ctx_.list("");e.hasMore();e.next(),++s);
        return s;
    }
    catch (NamingException x)
    {
        throw new DistributedMapException(x);
    }
}
public Collection values()
{
    try
    {
        Collection result=new Vector();
        for (NamingEnumeration e=ctx_.listBindings("");e.hasMore();i)
        {
            Binding binding=(Binding)e.next();

```

```

        result.add(binding.getObject());
    }
    return result;
}
catch (NamingException x)
{
    throw new DistributedMapException(x);
}
}
}

```

#### Source 8

Il faut fournir une URL pour déclarer la localisation du serveur LDAP et le contexte initial à utiliser.

```

LdapDistributedMap.setLink(
    "ldap://localhost/dc=shared,dc=mon-site,dc=org");

```

S'il est nécessaire de s'identifier pour accéder à la base LDAP, il faut valoriser différentes variables dans l'environnement.

```

java -Djava.naming.security.authentication=simple
-Djava.naming.security.principal=cn=Manager,dc=mon-site,dc=org
-Djava.naming.security.credentials=secret ...

```

Comme nous l'avons vu dans la version JDBC de notre API, il est possible de valoriser ces paramètres lors de l'invocation de la machine virtuelle, dans le fichier `jndi.properties`, ou en construisant une `Hashtable` et en la donnant en paramètre du constructeur de l'`InitialContext`.

```

Hashtable env=new Hashtable();
env.put(Context.SECURITY_AUTHENTICATION, "simple");
env.put(Context.SECURITY_PRINCIPAL, dn);
env.put(Context.SECURITY_CREDENTIALS, passwd);
...
LdapDistributedMap.setLink(new InitialContext(env),
    "ldap://localhost/dc=shared,dc=mon-site,dc=org");
varcluster=new LdapDistributedMap("varcluster");

```

Voici notre troisième lame de notre couteau suisse permettant de gérer des variables java entre serveurs. Nous avons une lame à base de répertoire partagé, une autre utilisant une base de données relationnelles et une troisième permettant d'invoquer une base LDAP.

### 7.4 Version RMI

Nous souhaitons partager des variables entre plusieurs serveurs. Pour cela, nous avons sélectionné l'interface `java.util.Map` afin de les manipuler. Des Maps nommées permettent de regrouper des variables visibles par tous les serveurs.

Les versions que nous avons proposées pour le moment ne permettent pas de maintenir un cache en lecture dans chaque serveur. Nous allons maintenant aborder une technologie plus complexe, permettant l'invocation à distance d'objets java. Java propose deux technologies pour manipuler des objets à distance : RMI et CORBA. Nous allons étudier une version RMI avant de la faire évoluer vers une technologie CORBA.

Un objet RMI est un objet qui est manipulable à partir d'une autre machine virtuelle ou d'un autre serveur. Un démon est en écoute sur un port, et attend des requêtes pour invoquer des méthodes sur des objets. Un objet serveur doit s'enregistrer auprès d'un référentiel, associé à un nom. Cela permet au client de retrouver le premier objet et d'invoquer des traitements serveur (Figure 17).

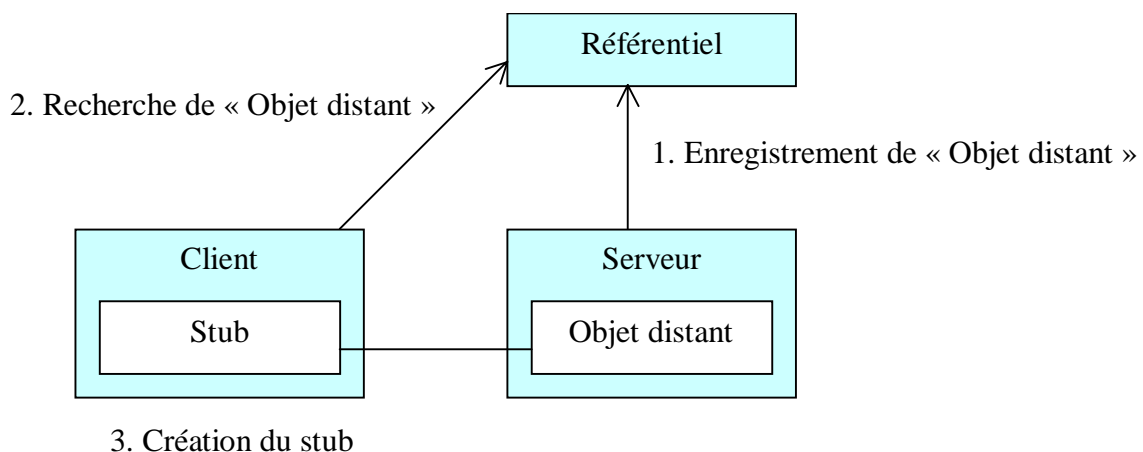


Figure 17

Tous les paramètres et les objets de retour des méthodes sont sérialisés lors des invocations. Pour effectuer ce travail, une instance `stub` s'occupe de simuler l'instance du serveur, vue par le client. Le `stub` et le serveur partagent la même interface qui définit le protocole d'appel des objets distants (Figure 18).

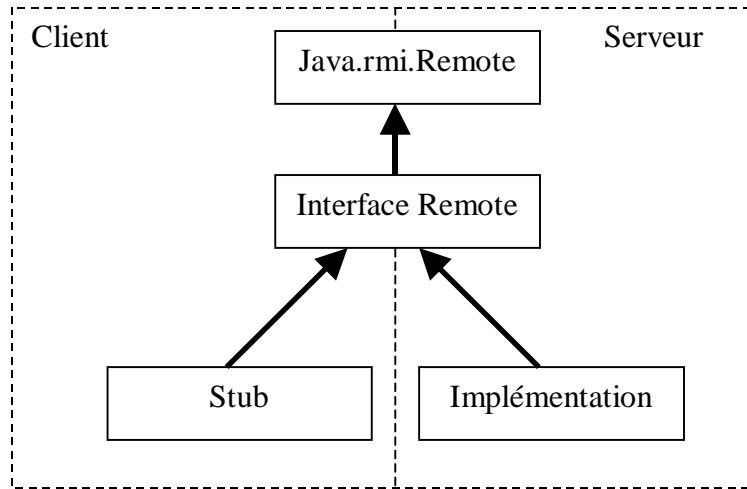


Figure 18

Les méthodes du `stub` empaquettent les requêtes pour les envoyer vers le serveur par le réseau. Sur le serveur, un démon analyse le flux, extrait la référence vers l'instance du serveur, récupère les paramètres et le nom de la méthode à invoquer. Il invoque l'instance du serveur avant de sérialiser la réponse et de la retourner au `stub`. Celui-ci déplie la réponse ou l'exception et propage l'information au client comme si le traitement avait été exécuté en local.

Le `stub` est généré par l'utilitaire `rmic` du JDK. Lors de la sérialisation d'un paramètre ou d'un objet de retour, si celui-ci implémente l'interface `java.rmi.Remote` l'instance est remplacée par un stub référençant l'objet distant. Ainsi, il est possible à un serveur de retourner une référence vers une autre instance du serveur. La figure 19 symbolise la génération d'un flux RMI lors de l'invocation d'une méthode dont l'un des paramètres est `Remote`.

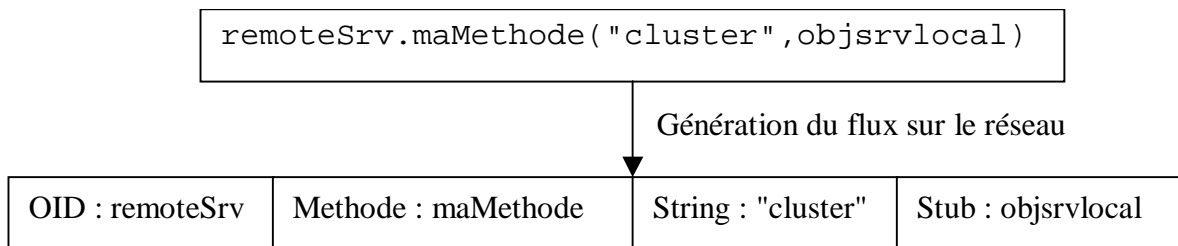


Figure 19

Le protocole RMI permet le téléchargement dynamique des classes nécessaires au client ou au serveur. Lors de la dé-sérialisation d'une instance, la classe correspondante n'est pas forcément présente sur le client. Celui-ci peut alors demander au serveur de lui fournir le fichier `.class` correspondant.

Pour gérer la durée de vie des différentes instances RMI dans une grappe de serveur, un ramasse miette répartie maintient des compteurs de références vers les instances publiées. Cela permet de libérer la mémoire d'un serveur lorsqu'un client n'existe plus.

RMI utilise un protocole qui lui est propre, pas forcément accepté par les pare-feu. Pour contourner cette difficulté, l'API est capable de s'appuyer sur HTTP. Il faut alors ajouter un programme CGI sur le serveur HTTP afin de faire transiter les requêtes vers le serveur RMI, présent à l'intérieur du pare-feu.

Nous allons utiliser RMI pour garder sur le serveur les instances des Maps. Pour utiliser cette technologie, il faut définir des interfaces ayant quelques contraintes : elles doivent étendre l'interface `java.rmi.Remote` et chaque méthode doit émettre une exception `java.rmi.RemoteException`. Tous les attributs des méthodes seront manipulés par valeurs. Ils doivent alors implémenter l'interface `java.io.Serializable`.

```
public interface RemoteMaps extends java.rmi.Remote
{
    public RemoteMap getMap(String name)
        throws java.rmi.RemoteException;
}
```

Les paramètres ou les objets retournés sont envoyés du client vers le serveur ou du serveur vers le client par valeur. Les instances sont sérialisées, envoyés sur le réseau et dé-sérialisés à l'arrivée. Si ces objets implémentent l'interface `java.rmi.Remote`, ils restent à leurs places. Un stub est envoyé à la place, permettant de résoudre l'interface et de déléguer tous les traitements vers l'objet distant.

Une fois l'interface définie, nous pouvons proposer une classe qui l'implémente (Source 9). Elle doit hériter de `UnicastRemoteObject` afin de s'enregistrer automatiquement auprès du serveur RMI.

```
public class RemoteMapsImpl extends UnicastRemoteObject
    implements RemoteMaps
{
    private java.util.Map maps_=new java.util.HashMap();
    public RemoteMapsImpl() throws RemoteException
    {
    }
    public synchronized RemoteMap getMap(String name)
        throws java.rmi.RemoteException
    {
        RemoteMap map=(RemoteMap)maps_.get(name);
        if (map==null)
        {
            map=new ConcurrentMapImpl();
            maps_.put(name,map);
        }
        return map;
    }
}
```

#### Source 9

Cette classe s'occupe de maintenir les différentes Map, associés à leurs noms. La méthode `getMap()` retourne une Map distante.

Il faut ensuite générer la classe stub qui implémente la même interface, mais qui s'exécute sur le client. Celle-ci délègue toutes les méthodes vers le serveur. L'utilitaire `rmic` du JDK permet la génération de celle-ci.

```
rmic -v1.2 RemoteMapsImpl
```

La classe `RemoteMapsImpl_Stub` produite doit alors être ajoutée aux archives clientes.

Nous souhaitons trois interfaces pour communiquer entre le client et le serveur. La première, comme nous l'avons vu, permet de retrouver les Maps par leurs noms. La deuxième reprend tous les services de l'interface `java.util.Map` et ajoute pour chaque méthode la diffusion de l'exception `RemoteException`. La troisième interface est plus subtile. Nous désirons maintenir un cache local. Nous devons alors recevoir des événements pour les modifications de la map distante. Nous déclarons une interface `RemoteMapListener` pour cela. Cette interface est utilisée à l'envers. C'est le client qui diffuse un objet pour le serveur. L'instance distante de la Map signale à tous les clients les modifications de son état.

L'architecture complète est indiquée figure 20.

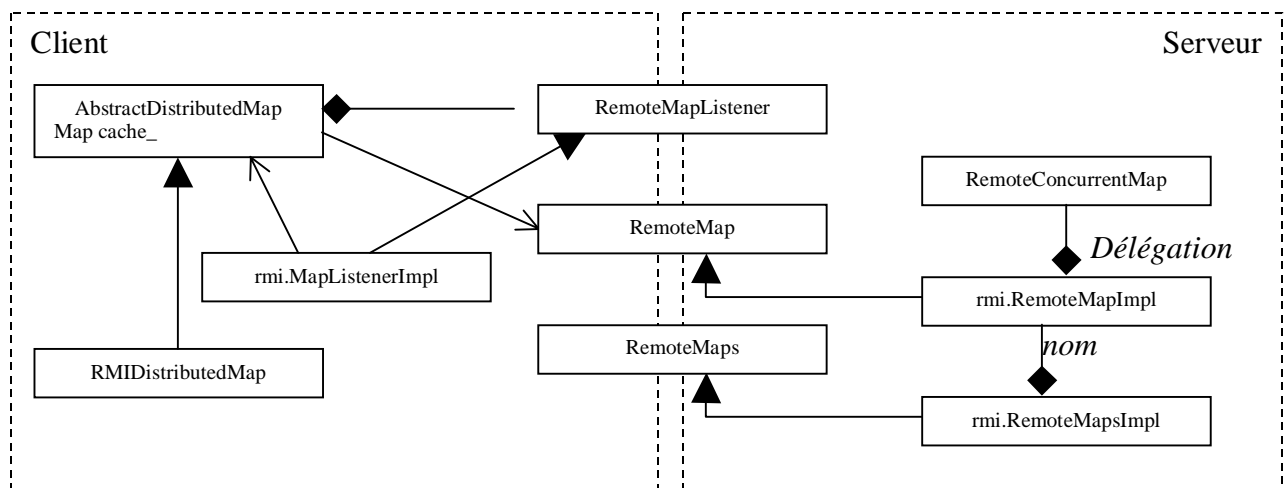


Figure 20

La classe `AbstractDistributedMap` s'occupe de présenter au client une Map classique. Elle maintient un cache local et communique avec la Map sur le serveur RMI. La classe `RMIDistributedMap` spécialise cette classe pour une utilisation de RMI. Une implémentation de `RemoteMapListener` est construite en locale pour recevoir les événements du serveur.

Une instance `RemoteMapImpl`, dérivée de `RemoteMap`, est construite au premier accès et représente la Map distante. Elle délègue tous les traitements vers une instance `RemoteConcurrentMap`. Pourquoi avoir deux classes pour maintenir la Map distante ? Parce-que les instances RMI doivent hériter de la classe `UnicastRemoteObject` (non représenté sur le schéma). Ce n'est pas le cas avec CORBA. Pour pouvoir factoriser les traitements de cette instance en utilisation RMI et CORBA, nous utilisons une approche par délégation.

Une instance `RemoteMapsImpl` implémente l'interface `RemoteMaps`. Il s'agit de l'interface racine de l'application. L'instance s'occupe de retrouver ou de construire à la demande des Maps distantes.

La classe `RemoteConcurrentMap` ne mémorise pas des objets mais des tableaux de bytes. En effet, il n'est pas nécessaire de posséder les fichiers `.class` de l'application dans le référentiel. En transformant les données en `byte[]`, il n'est pas nécessaire d'adapter le CLASSPATH du référentiel. Par contre, il faut sérialiser et dé-sérialiser les objets lorsque cela est nécessaire.

Une instance `RemoteMapsImpl` doit être enregistrée dans la base de registre RMI afin de pouvoir être retrouvé par les clients lors du démarrage. L'approche classique consiste à utiliser la classe `java.rmi.Naming`. Pour travailler avec une approche plus moderne, nous allons utiliser le driver JNDI RMI (`rmiregistry.jar`), téléchargeable à l'adresse <http://java.sun.com/jndi>. Cela correspond à une approche équivalente aux versions de notre Map pour JDBC ou LDAP.

Avant de lancer le serveur, il faut démarrer le programme `rmiregistry`. Ce programme publie un objet RMI standard qui peut être retrouvé à l'aide d'API standard. Il sert de référentiel aux autres objets RMI publiés sur le réseau. Il est nécessaire d'ajouter les classes stub dans le CLASSPATH. Pourquoi cela ? Parce que le programme `rmiregistry` est un programme RMI classique. Lors de l'enregistrement d'une instance serveur, c'est le stub qui est mémorisé dans la base de registre. La classe correspondante doit donc être disponible.

La méthode `main()` de la classe `RMIDistributedMap` construit l'instance du serveur et l'enregistre dans la base de registre RMI.

```
static public void main(String[] args)
{
    ...
    String uri="rmi://localhost/DistributedMap";
    ...
    Context ctx=new InitialContext();
    ctx.rebind(uri, new RemoteMapsImpl());
}
```

Le premier paramètre de la ligne de commande permet d'indiquer la localisation de la base d'enregistrement RMI et le nom sous lequel sera enregistrée l'instance `RemoteMapsImpl`. Cette même URL sera utilisé par le client pour retrouver l'instance serveur.

La classe `RMIDistributedMap` récupère un lien vers l'instance du serveur, puis s'enregistre pour récupérer les événements de modifications de la Map distante. Elle récupère en même temps l'état courant de celle-ci. La méthode `addMapListener()` enregistre l'instance en écoute et retourne simultanément l'état de la Map actuel. Pourquoi combiner ces deux traitements ? Pour plusieurs raisons. Tous d'abord, parce que cela permet de n'avoir qu'un seul appel distant. Cela améliore les performances. D'autre part, si les traitements étaient distincts, entre le premier et le deuxième, il est possible qu'un événement parte vers le client, avant que celui-ci n'ait récupéré l'état initial de la Map distante. N'oubliez pas que la Map distante est utilisé par plusieurs serveurs simultanément.

Il y a une subtilité dans le mécanisme de notification. En effet, lorsqu'un `Client` ajoute une nouvelle donnée dans la Map distante, cela invoque le serveur. Celui-ci envoie un événement à tous les clients pour leurs signaler de cette information. `Client` reçoit l'évènement en retour, à l'aide de son instance `MapListener`. Celle-ci désire modifier la Map locale pour adapter le cache. Dans cette situation, il n'est pas possible d'obtenir un accès `synchronized` à l'instance, car elle est en train d'attendre le retour de l'invocation sur le serveur. Dans un environnement classique – sans RMI – la réception de l'évènement s'effectue sur la même tâche que celle ayant permis la modification de la Map. Avec RMI, la situation est différente. La réception de l'évènement est dans une autre tâche ! Rappelez-vous que le client publie également un objet RMI. Un démon RMI est alors lancé automatiquement dans une autre tâche sur le client. Nous avons alors une situation de « d'étreinte mortelle » (Figure 21).

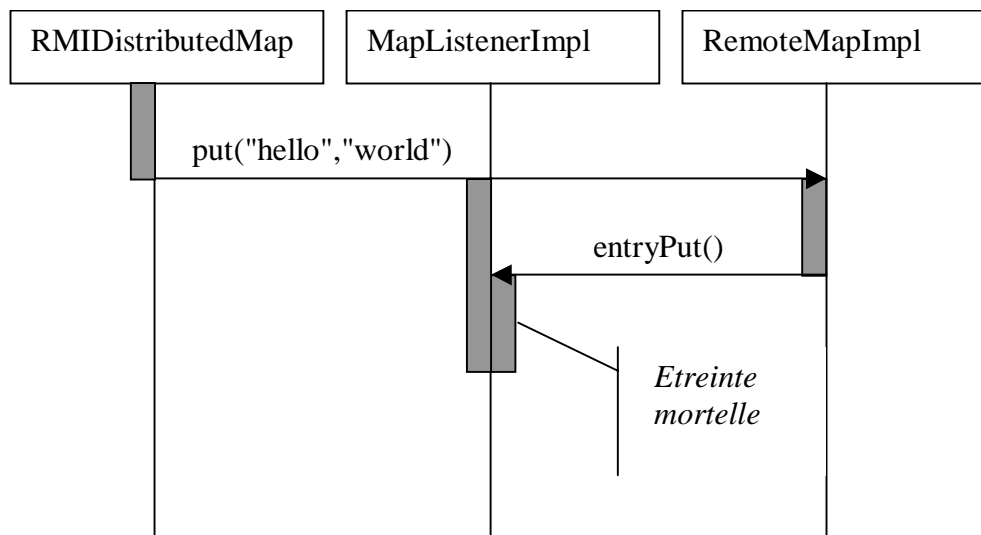


Figure 21

Pour remédier à cette situation, nous modifions la classe du serveur pour que celui-ci émette les évènements de façon asynchrone. Ainsi, il n'est plus possible d'avoir un blocage lors de l'évolution d'une Map (Figure 22).

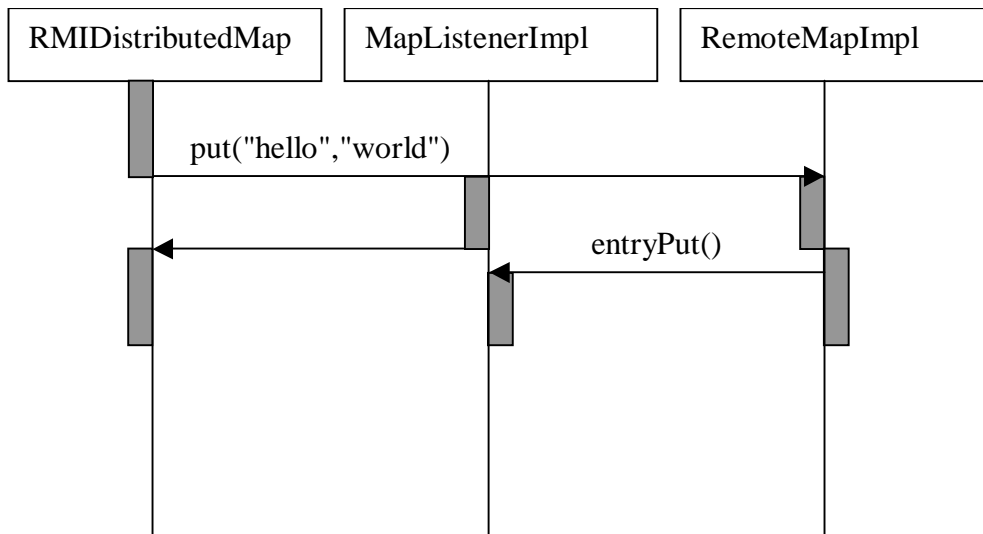


Figure 22

Avant de pouvoir utiliser la version cliente, il faut sûrement modifier quelques paramètres de sécurité de la machine virtuelle. Il faut en effet autoriser la consultation des propriétés de la branche `java.naming`, et permettre la manipulation des sockets de port supérieur à 1024.

Plusieurs solutions sont possibles :

- Modifier le fichier `java.policy` du JDK ;
- Créer et valoriser un fichier `{user.home}\.java.policy`
- Indiquer dans la ligne de commande le fichier de sécurité à utiliser pour l'application (`-Djava.security.policy=URL`)

Ces fichiers permettent d'indiquer des privilèges à des classes, suivant leurs localisations ou leurs signatures. Par exemple, si toutes les classes du projet se trouvent dans le répertoire `test`, indiquez :

```

grant codeBase "file:/test/*" {
    permission java.net.SocketPermission "localhost:1024-", "accept, connect, listen";
    permission java.util.PropertyPermission "java.naming.*", "read, write";
};
  
```

Source 10 vous trouverez le détail de la classe `AbstractDistributedMap`. Cette classe maintient un cache local et l'adapte sur réception d'évènements du serveur. Cela permet d'améliorer notablement les performances. Les sources complètes sont disponibles sur mon site ([www.philippe.prados.net](http://www.philippe.prados.net)).

```

import java.io.*;
import java.util.*;
import java.rmi.*;
import javax.naming.*;

public abstract class AbstractDistributedMap
    implements Map, RemoteMapListener
{
    private RemoteMapListener listener_;
    protected Map cache_;
    protected RemoteMap map_;
    public AbstractDistributedMap(RemoteMap map, String name)
        throws java.rmi.RemoteException
    {
        super();
        map_=map;
    }
    final public void clear()
    {
        try
        {
            map_.clear();
        }
        catch (java.rmi.RemoteException x)
        {
            throw new RuntimeException(x.getLocalizedMessage());
        }
    }
}
  
```

```

}
final public boolean containsKey(Object key)
{
    return cache_.containsKey(key);
}
final public boolean containsValue(Object value)
{
    return cache_.containsValue(value);
}
final public void entryClear(MapEvent event)
{
    cache_.clear();
}
final public void entryDeleted(MapEvent event)
{
    cache_.remove(event.getKey());
}
final public void entryPut(MapEvent event)
{
    try
    {
        Object obj=new ObjectInputStream(
            new ByteArrayInputStream(event.getValue())).readObject();
        cache_.put(event.getKey(),obj);
    }
    catch (IOException x)
    {
        // Ignore
    }
    catch (ClassNotFoundException x)
    {
        throw new RuntimeException(x.getLocalizedMessage());
    }
}
final public Set entrySet()
{
    class Entry implements Map.Entry
    {
        private Object key_;
        private Object value_;
        public Object getKey()
        {
            return key_;
        }
        public Object getValue()
        {
            return value_;
        }
        public Object setValue(Object value)
        {
            Object old=put(key_,value);
            value_=value;
            return old;
        }
        public boolean equals(Object o)
        {
            if (!(o instanceof Map.Entry))
                return false;
            Map.Entry m=(Map.Entry)o;
            return (getKey()==null ? m.getKey()==null
                : getKey().equals(m.getKey())) &&
                (getValue()==null ? m.getValue()==null
                : getValue().equals(m.getValue()));
        }
        public int hashCode()
        {
            return (getKey()==null ? 0 : getKey().hashCode()) ^
                (getValue()==null ? 0 : getValue().hashCode());
        }
        public String toString()
        {
            return "["+key_+"':'"+value_+"']";
        }
    };
    Set result=new HashSet();
    Set list=cache_.entrySet();

```

```

    for (Iterator i=list.iterator();i.hasNext();)
    {
        Map.Entry cacheEntry=(Map.Entry)i.next();
        Entry entry=new Entry();
        entry.key_=cacheEntry.getKey();
        entry.value_=cacheEntry.getValue();
        result.add(entry);
    }
    return result;
}
final public Object get(Object key)
{
    return cache_.get(key);
}
final public boolean isEmpty()
{
    return cache_.isEmpty();
}
final public Set keySet()
{
    return cache_.keySet();
}
final public Object put(Object key, Object value)
{
    try
    {
        ByteArrayOutputStream buf=new ByteArrayOutputStream();
        new ObjectOutputStream(buf).writeObject(value);
        cache_.put(key,value);
        return map_.put(key,buf.toByteArray());
    }
    catch (java.rmi.RemoteException x)
    {
        throw new RuntimeException(x.getLocalizedMessage());
    }
    catch (IOException x)
    {
        throw new RuntimeException(x.getLocalizedMessage());
    }
}
final public void putAll(Map t)
{
    try
    {
        cache_.putAll(t);
        map_.putAll(toMapByteArray(t));
    }
    catch (java.rmi.RemoteException x)
    {
        throw new RuntimeException(x.getLocalizedMessage());
    }
}
final public Object remove(Object key)
{
    try
    {
        cache_.remove(key);
        return map_.remove(key);
    }
    catch (java.rmi.RemoteException x)
    {
        throw new RuntimeException(x.getLocalizedMessage());
    }
}
final protected void setListener(RemoteMapListener listener)
{
    listener_=listener;
}
final public int size()
{
    return cache_.size();
}
final protected Map toMapByteArray(Map map)
{
    Map rc=new HashMap(map.size());
    ByteArrayOutputStream buf=new ByteArrayOutputStream();
    for (Iterator i=map.keySet().iterator();i.hasNext();)

```

```

    {
        Object key;
        Object value=map.get(key=i.next());
        buf.reset();
        try
        {
            new ObjectOutputStream(buf).writeObject(value);
        }
        catch (IOException x)
        {
            // Ignore
        }
        rc.put(key,buf.toByteArray());
    }
    return rc;
}
final protected Map toObjectMap(Map map)
{
    Map rc=new HashMap(map.size());
    ByteArrayOutputStream buf=new ByteArrayOutputStream();
    for (Iterator i=map.keySet().iterator();i.hasNext();)
    {
        Object key;
        byte[] value=(byte[])map.get(key=i.next());
        buf.reset();
        try
        {
            rc.put(key,new ObjectInputStream(
                new ByteArrayInputStream(value)).readObject());
        }
        catch (IOException x)
        {
            // Ignore
        }
        catch (ClassNotFoundException x)
        {
            throw new RuntimeException(x.getLocalizedMessage());
        }
    }
    return rc;
}
final public Collection values()
{
    return cache_.values();
}
}

```

**Source 10**

Voilà, la version RMI est fin prête. Pour démarrer le serveur, il faut lancer la base d'enregistrement RMI puis l'application `org.eactivity.cluster.RMIDistributedMap`. N'oubliez pas d'adapter le CLASSPATH avant de démarrer ces programmes.

```

set CLASSPATH=...
start rmiregistry
start java org.eactivity.cluster.RMIDistributedMap

```

Le référentiel RMI est démarré. Les différents clients peuvent le contacter pour partager des informations. Il faut auparavant indiquer où localiser le référentiel RMI et le nom utilisé pour obtenir le premier objet.

```

RMIDistributedMap.setLink("rmi://localhost/DistributedMap");
Map varcluster=new RMIDistributedMap("varcluster");

```

Voici une nouvelle lame à notre couteau suisse de variable cluster. Nous en avons quatre pour le moment. Lorsque nous réunirons toutes les lames dans un beau manche rouge avec une croix blanche, nous réunirons les différentes techniques d'initialisation de ces technologies pour simplifier l'utilisation des différentes lames.

Nous allons enrichir cette architecture pour accepter une communication CORBA avec le protocole IIOP.

**7.5 Version CORBA**

La version CORBA est très similaire à la version RMI. CORBA utilise le protocole IIOP qui est un standard permettant la communication entre objets distants, quels que soient leurs langages de développement. Une instance C++ peut invoquer une instance Java qui elle-même invoque une instance Eiffel ou Smalltalk. Java offre un Object Request Broker (ORB) pour pouvoir facilement publier des instances Java en respectant le protocole IIOP ou pour contacter des objets IIOP distants.

L'architecture d'un objet CORBA est décrit figure 23.

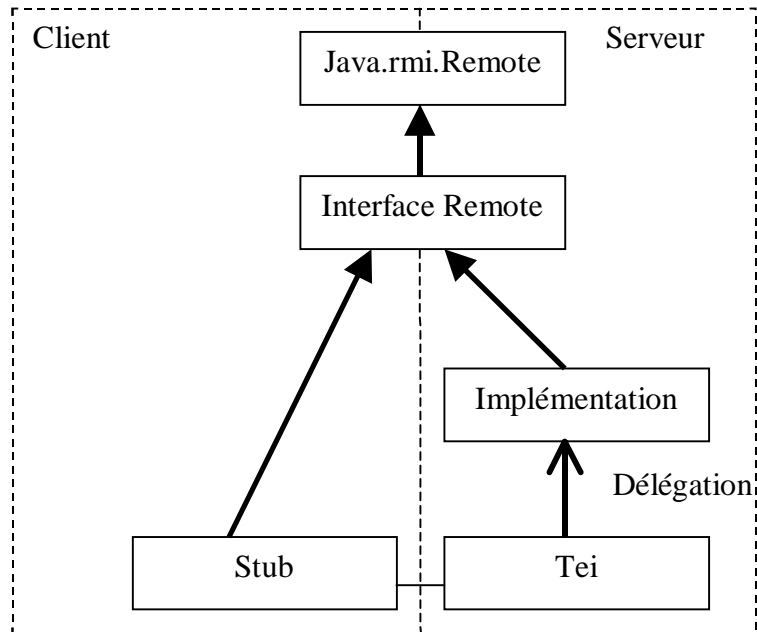


Figure 23

L'instance TEI s'occupe d'interpréter les demandes IIOP pour les convertir en invocation java.

Nous avons repris l'architecture RMI. Dans un package `iiop`, nous déclarons à nouveaux les classes `RemoteMapsImpl`, `RemoteMapImpl` et `MapListenerImpl`. Ces classes sont équivalentes à leurs sœurs du package `rmi`, mais héritent de `PortableRemoteObject` pour pouvoir s'enregistrer automatiquement auprès du serveur CORBA.

Il faut ensuite générer les classes `Tei` et `Stub` nécessaires. Un paramètre de l'utilitaire `rmic` permet cela.

```
rmic -iiop RemoteMapsImpl
rmic -iiop RemoteMapImpl
rmic -iiop MapListenerImpl
```

Une instance `RemoteMapsImpl` doit être enregistré dans la base de registre IIOP afin de pouvoir être retrouvée par les clients lors du démarrage. Il est possible d'enregistrer cette instance dans un « Common Object Services Naming Server », un serveur CORBA de nom ou une base LDAP. Avant de lancer le serveur, il faut démarrer la base de registre. Le programme `orbd` est là pour cela. Ce programme publie un objet CORBA normalisé qui peut être retrouvé à l'aide d'API standard. Il sert de référentiel aux autres objets CORBA publié sur le réseau.

La méthode `main()` de la classe `IIOPDistributedMap` construit l'instance du serveur et l'enregistre dans la base de registre RMI.

```
static public void main(String[] args)
{
    ...
    String uri="iiop://localhost/DistributedMap";
    ...
    Context ctx=new InitialContext();
    ctx.rebind(uri, new RemoteMapsImpl());
}
```

Nous pouvons alors lancer la base d'enregistrement CORBA et le serveur de Map CORBA.

```
start orbd
start java org.eactivity.cluster.IIOPDistributedMap
```

Les différents clients peuvent contacter le démon pour partager des informations. Il faut auparavant indiquer où localiser le référentiel RMI et le nom utilisé pour obtenir le premier objet.

```
IIOPDistributedMap.setLink("iiop://localhost/DistributedMap");
Map varcluster=new IIOPDistributedMap("varcluster");
```

Nous venons de forger la dernière lame de notre couteau suisse. Il faut maintenant les réunir toutes dans un manche rouge avec une belle croix blanche.

## 7.6 Synthèse des différentes versions

À partir d'un besoin simple à exprimer « partager des variables entres différents clusters » et d'une interface normalisée `java.util.Map`, nous avons proposé différentes approches utilisant des technologies très diverses. Chacune de ces technologies doit être initialisées pour indiquer où et comment accéder au référentiel. Nous avons fait le choix d'utiliser systématiquement une URL. Cela nous offre l'opportunité de regrouper toutes ces technologies dans un cadre uniforme.

La classe `FactoryDistributedMap` permet d'obtenir une usine à fabriquer des Maps partagées, suivant différentes technologies (Source 11). La méthode statique `getFactoryMap()` attend une URL en paramètre, l'analyse et retourne une usine adaptée à la technologie utilisée. La variable d'environnement `org.eactivity.cluster` permet de proposer une URL par défaut.

```
import java.net.*;
import javax.naming.*;

public abstract class FactoryDistributedMap
{
    public static FactoryDistributedMap getFactoryMap()
        throws MalformedURLException, NamingException
    {
        return getFactoryMap(System.getProperty("org.eactivity.cluster"));
    }
    public static FactoryDistributedMap getFactoryMap(String uri)
        throws java.net.MalformedURLException, NamingException
    {
        int idx=uri.indexOf(':');
        String protocol=uri.substring(0,idx);
        if (protocol.equals("file"))
        {
            FileDistributedMap.setLink(uri);
            return new FactoryDistributedMap()
            {
                public java.util.Map getMap(String name)
                {
                    return new FileDistributedMap(name);
                }
            };
        }
        else if (protocol.equals("jdbc"))
        {
            JBDCDistributedMap.setLink(uri);
            return new FactoryDistributedMap()
            {
                public java.util.Map getMap(String name)
                {
                    return new JBDCDistributedMap(name);
                }
            };
        }
        else if (protocol.equals("iiop") || protocol.equals("iiopname"))
        {
            IIOPDistributedMap.setLink(uri);
            return new FactoryDistributedMap()
            {
                public java.util.Map getMap(String name)
                {
                    try
                    {
                        return new IIOPDistributedMap(name);
                    }
                    catch (java.rmi.RemoteException x)
                    {
                        throw new DistributedMapException(x);
                    }
                }
            };
        }
        else if (protocol.equals("rmi"))
        {
            RMIDistributedMap.setLink(uri);
            return new FactoryDistributedMap()
            {
                public java.util.Map getMap(String name)
                {
                    try
                    {
                        return new RMIDistributedMap(name);
                    }
                    catch (java.rmi.RemoteException x)
                    {
                        throw new DistributedMapException(x);
                    }
                }
            };
        }
    }
}
```

```

else if (protocol.equals("ldap"))
{
    LDAPDistributedMap.setLink(uri);
    return new FactoryDistributedMap()
    {
        public java.util.Map getMap(String name)
        {
            return new LDAPDistributedMap(name);
        }
    };
}
else if (protocol.equals("coherence"))
{
    return new FactoryDistributedMap()
    {
        public java.util.Map getMap(String name)
        {
            return new CoherenceDistributedMap(name);
        }
    };
}
}
throw new java.net.MalformedURLException(uri);
}
public abstract java.util.Map getMap(String name)
    throws DistributedMapException;
}

```

**Source 11**

Pour exploiter cette API, il faut obtenir une usine, puis demander un accès une Map particulière.

```

factory=FactoryDistributedMap.getFactoryMap("file://master/shared");
Map varcluster=factory.getMap("varcluster");

```

Il est alors facile de manipuler les différentes technologies à l'aide d'une simple URL (Tableau 2).

**Tableau 2**

<code>file://master/shared</code>	Partage de fichier. Utilise le répertoire <code>shared</code> présent sur la machine <code>master</code> .
<code>jdbc:oracle:thin:@localhost:madb</code>	Accès JDBC. Ouvre une connexion directe avec la base de donnée, en utilisant l'utilisateur <code>sa</code> et le mot de passe <code>pass</code> .
<code>jdbc:sa:pass@jndi://jdbc/DistributedMap</code>	Accès JDBC via JNDI. Utilise le driver JNDI standard pour retrouver la Data source de nom <code>DistributedMap</code> .
<code>jdbc:sa:@jndi:iiop://localhost:900/jdbc/DistributedMap</code>	Accès JDBC via JNDI. Utilise le driver JNDI de type IIOP, localisé sur <code>localhost</code> , sur le port <code>900</code> . Recherche alors la Data-source de nom <code>DistributedMap</code> .
<code>ldap://localhost/dc=shared,dc=mon-site,dc=org</code>	Accès LDAP. Utilise une base LDAP sur <code>localhost</code> , et manipule le nœud <code>shared</code> .
<code>rmi://localhost/DistributedMap</code>	Accès RMI. Utilise la base d'enregistrement RMI sur <code>localhost</code> et recherche l'objet <code>DistributedMap</code> .
<code>iiop://localhost/DistributedMap</code>	Accès CORBA. Utilise la base d'enregistrement CORBA sur <code>localhost</code> et recherche l'objet <code>DistributedMap</code> .

Cela nous un beau manche avec ces cinq lames. L'utilisateur peut ouvrir la lame qu'il désire à l'aide d'une simple URL.

Pour que ces différentes versions fonctionnent, il faut bien entendu démarrer les différentes technologies serveurs correspondantes (partager un répertoire, lancer la base de donnée, démarrer la base LDAP, le serveur RMI ou IIOP).

Nous avons marié de nombreuses technologies. Il est alors aisé de partager des informations entre plusieurs applications ou entre plusieurs serveurs d'un groupe de cluster. Sélectionnez une technologie pour répartir les requêtes des clients vers les différents serveurs.

Les situations étant très variables, cela permet d'effectuer le choix technologique le plus tard possible, après un test de performance pertinent dans l'environnement d'exploitation. C'est le défi que nous avons relevé.

Notre couteau suisse est déjà très sympathique. Il est possible de l'enrichir de nouvelles lames comme SOAP, mais au vue du trafic réseau engendré, cela s'apparente plus à du gadget.

J'espère que ces introductions aux technologies java vous ouvriront de nouveaux horizons. Nous avons manipulé le JDK 1.4, HTTP, URL, JDBC, SQL, JNDI, LDAP, JRMP, RMI, CORBA et IIOP. Quelles sont les lettres de l'alphabet qui nous manquent ;-) ?