



[[Viewing Hints](#)] [[Revision History](#)] [[Report an Error](#)]
[[1st Edition](#)] [[Free Newsletter](#)]
[[Seminars](#)] [[Seminars on CD ROM](#)] [[Consulting](#)]

Thinking in Java, 2nd edition, Revision 9

©2000 by Bruce Eckel

[[Previous Chapter](#)] [[Short TOC](#)] [[Table of Contents](#)] [[Index](#)] [[Next Chapter](#)]

15: Informatique distribuée

...

RMI (Remote Method Invocation) : Invocation de méthodes distantes

Les approches traditionnelles pour exécuter des instructions à travers un réseau sur d'autres ordinateurs étaient aussi confuses qu'ennuyeuses et sujettes aux erreurs. La meilleure manière d'aborder ce problème est de considérer qu'en fait un objet donné est présent sur une autre machine, on lui envoie un message et l'on obtient le résultat comme si l'objet était instancié sur votre machine locale. Cette simplification est exactement celle que Java 1.1 *Remote Method Invocation* (RMI - Invocation de méthodes distantes) permet de faire. Cette section vous accompagne à travers les étapes nécessaires pour créer vos propres objets RMI.

Interfaces Remote

RMI utilise beaucoup d'interfaces. Lorsque l'on souhaite créer un objet distant, l'implémentation sous-jacente est masquée en passant par une interface. Ainsi, lorsque qu'un client obtient une référence vers un objet distant, ce qu'il possède réellement est une référence intermédiaire, qui renvoie à un bout de code local capable de communiquer à travers le réseau. Mais nul besoin de se soucier de cela, il suffit de juste d'envoyer des messages par le biais de cette référence intermédiaire.

La création d'une interface distante doit respecter ces directives :

1. L'interface distante doit être **public** (il ne peut y avoir accès `package` , autrement dit d'accès friendly). Sinon, le client recevra une erreur lorsqu'il tentera d'obtenir un objet distant qui implémente l'interface distante.
2. L'interface distante doit hériter de l'interface **java.rmi.Remote**.
3. Chaque méthode de l'interface distante doit déclarer **java.rmi.RemoteException** dans sa clause **throws** en plus des autres exceptions spécifiques à l'application.
4. Un objet distant passé en argument ou en valeur de retour (soit directement ou inclus dans un objet local), doit être déclaré en tant qu'interface distante et non comme la classe d'implémentation.

Voici une interface distante simple qui représente un service d'heure exacte :

```

//: c15:rmi:PerfectTimeI.java
// L'interface distante PerfectTime.
package c15.rmi;
import java.rmi.*;

interface PerfectTimeI extends Remote {
    long getPerfectTime() throws RemoteException;
} ///:~

```

Cela ressemble à n'importe quelle autre interface mis à part qu'elle hérite de **Remote** et que toutes ses méthodes émettent **RemoteException**. Rappelez vous qu'une **interface** et toutes ses méthodes sont automatiquement **publiques**.

Implémenter l'interface distante

Le serveur doit contenir une classe qui hérite de **UnicastRemoteObject** et qui implémente l'interface distante. Cette classe peut avoir aussi des méthodes supplémentaires, mais bien sûr seules les méthodes appartenant à l'interface distante seront disponibles pour le client puisque celui-ci n'obtiendra qu'une référence vers l'interface, et non vers la classe qui l'implémente.

Vous devez explicitement définir le constructeur de cet objet distant même si vous définissez seulement un constructeur par défaut qui appelle le constructeur de base. Vous devez l'écrire parce qu'il doit émettre **RemoteException**.

Voici l'implémentation de l'interface distante **PerfectTimeI**:

```

//: c15:rmi:PerfectTime.java
// L'implémentation de
// l'objet distant PerfectTime.
package c15.rmi;
import java.rmi.*;
import java.rmi.server.*;
import java.rmi.registry.*;
import java.net.*;

public class PerfectTime
    extends UnicastRemoteObject
    implements PerfectTimeI {
    // Implémentation de l'interface:
    public long getPerfectTime()
        throws RemoteException {
        return System.currentTimeMillis();
    }
    // Doit implémenter le constructeur
    // pour émettre RemoteException:
    public PerfectTime() throws RemoteException {
        // super(); // Appelé implicitement
    }
    // Inscription auprès du service RMI :
    public static void main(String[] args) {
        System.setSecurityManager(
            new RMISecurityManager());
        try {

```

```

        PerfectTime pt = new PerfectTime();
        Naming.bind(
            "//peppy:2005/PerfectTime", pt);
        System.out.println("Ready to do time");
    } catch (Exception e) {
        e.printStackTrace();
    }
}
} ///:~

```

Ici, **main()** se charge de tous les détails de mise en place du serveur. Une application qui met en service des objets RMI doit à un moment :

1. Créer et installer un gestionnaire de sécurité qui supporte RMI. Le seul disponible pour RMI fourni dans la distribution Java est **RMISecurityManager**.
2. Créer une ou plusieurs instances de l'objet distant. Ici, vous pouvez voir la création de l'objet **PerfectTime**.
3. Enregistrer au moins un des objets distants grâce au registre d'objets distants RMI pour des raisons d'amorçage. Un objet distant peut avoir des méthodes qui retournent des références vers les autres objets distants. En le mettant en place, le client ne doit s'adresser au registre qu'une seule fois pour obtenir ce premier objet distant.

Mise en place du registre

Ici, vous pouvez voir un appel à la méthode **statique Naming.bind()**. Toutefois, cet appel nécessite que le registre fonctionne dans un autre processus de l'ordinateur. Le nom du registre serveur est **rmiregistry**, et sous Windows 32-bit vous utiliserez :

```
start rmiregistry
```

pour démarrer celui-ci en fond. Sous Unix, ce sera :

```
rmiregistry &
```

Comme beaucoup d'applications réseau, le **rmiregistry** est localisé à l'adresse IP de la machine qui l'a démarré, mais il doit aussi écouter un port. Si vous invoquez le **rmiregistry** comme ci-dessus, sans argument, le port écouté par le registre sera par défaut 1099. Si vous souhaitez que ce soit un autre port, vous devez ajouter un argument à la ligne de commande pour le préciser. Dans cet exemple, le port sera 2005, ainsi le **rmiregistry** devra être démarré de cette manière sous Windows 32-bit :

```
start rmiregistry 2005
```

ou pour Unix:

```
rmiregistry 2005 &
```

L'information concernant le port doit aussi être fourni à la commande **bind()**, ainsi que l'adresse IP de la machine où se trouve le registre. Mais il faut mentionner que cela peut être un problème frustrant en cas de tests de programmes RMI en local (de la même manière que les programmes réseau testés plus loin dans ce chapitre).

En effet dans le JDK version 1.1.1, il y a quelques problèmes : [\[69\]](#)

1. **localhost** ne fonctionne pas avec RMI. Aussi, pour faire une expérience avec RMI sur une seule machine, vous devez fournir le nom de la machine. Pour découvrir le nom de votre machine sous Windows 32-bit, allez dans le Panneau de configuration et sélectionnez Réseau. Sélectionnez l'onglet Identification , vous verrez apparaître le nom de l'ordinateur. Dans mon cas, j'ai appelé mon ordinateur Peppy . Il semble que la différence entre majuscules et minuscules soit ignorée.
2. RMI ne fonctionnera pas à moins que votre ordinateur ait une connexion TCP/IP active, même si tous vos composants discutent entre eux sur la machine locale. Cela signifie que vous devez vous connecter à Internet pour essayer de faire fonctionner le programme ou vous obtiendrez quelques messages d'exception obscurs.

En ayant tout cela à l'esprit, la commande **bind()** devient :

```
Naming.bind( "//peppy:2005/PerfectTime", pt );
```

Si vous utilisez le port par défaut 1099, nul besoin de préciser le port, donc vous pouvez dire :

```
Naming.bind( "//peppy/PerfectTime", pt );
```

Vous devriez être capable de réaliser un test local en omettant l'adresse IP et en utilisant simplement l'identifiant :

```
Naming.bind( "PerfectTime", pt );
```

Le nom pour le service est arbitraire ; il se trouve que c'est PerfectTime ici, comme le nom de la classe, mais un tout autre nom conviendrait. L'important est que ce nom soit unique dans le registre, connu par le client pour qu'il puisse se procurer l'objet distant. Si le nom est déjà utilisé dans le registre, celui renvoie une **AlreadyBoundException**. Pour éviter cela, il faut passer à chaque fois par un appel à **rebind()** à la place de **bind()**, en effet **rebind()** soit ajoute une nouvelle entrée, soit remplace celle existant déjà.

Durant l'exécution de **main()**, l'objet a été créé et enregistré, il reste ainsi en activité dans le registre, attendant qu'un client arrive et fasse appel à lui. Tant que **rmiregistry** fonctionne et que **Naming.unbind()** n'est pas appelé avec le nom choisi, l'objet sera présent. C'est pour cela que lors de la conception du code, il faut redémarrer **rmiregistry** après chaque compilation d'une nouvelle version de l'objet distant.

rmiregistry n'est pas forcément démarré en tant que processus externe. S'il est sûr que l'application est la seule qui utilise le registre, celui peut être mis en fonction à l'intérieur même du programme grâce à cette ligne :

```
LocateRegistry.createRegistry(2005);
```

Comme auparavant, nous utilisons le numéro de port 2005 dans cet exemple. C'est équivalent au fait de lancer **rmiregistry 2005** depuis la ligne de commande, mais c'est souvent plus pratique lorsque l'on développe son code RMI puisque cela élimine les étapes supplémentaires qui consistent à arrêter et redémarrer le registre. Une fois cette instruction exécutée, la méthode **bind()** de la classe **Naming** peut être utilisée comme précédemment.

Création des stubs et des skeletons

Si l'on compile et exécute **PerfectTime.java**, cela ne fonctionnera pas même si **rmiregistry** a été correctement mis en route. Cela parce que la machinerie pour RMI n'est pas complète. Il faut d'abord créer les stubs et skeletons qui assurent les opérations de connexions réseaux et permettent de faire comme si l'objet distant était juste un objet local sur de la machine.

Ce qui se passe derrière la scène est complexe. Tous les objets envoyés à un objet distant ou reçus de celui-ci doivent **implémenter Serializable** (pour passer des références distantes plutôt que les objets complets, les arguments peuvent **implémenter Remote**), ainsi on peut considérer que les stubs et les skeletons assurent automatiquement la sérialisation et la désérialisation tout en gérant l'acheminement des arguments et du résultat au travers du réseau. Par chance, vous n'avez rien à connaître de tout cela, mais vous *devez* avoir créé les stubs et les skeletons. C'est une procédure simple : on invoque l'outil **rmic** sur le code compilé, et il crée les fichiers nécessaires. La seule chose obligatoire est donc d'ajouter cette étape à la procédure de compilation.

L'outil **rmic** est particulier en ce qui concerne les packages et les classpaths. **PerfectTime.java** est dans le **package c15.rmi**, et même **rmic** est invoqué dans le même répertoire que celui où se trouve **PerfectTime.class**, **rmic** ne trouvera pas le fichier, puisqu'il se repère grâce au classpath. Vous devez ainsi préciser la localisation du classpath, comme ceci :

```
rmic c15.rmi.PerfectTime
```

La commande ne nécessite pas d'être exécutée à partir du répertoire contenant **PerfectTime.class**, mais les résultats seront placés dans le répertoire courant.

Lorsque **rmic** a été exécuté avec succès, deux nouvelles classes sont obtenues dans le répertoire :

```
PerfectTime_Stub.class  
PerfectTime_Skel.class
```

correspondant au stub et au skeleton. Dès lors, vous êtes prêt à faire communiquer le serveur et le client.

Utilisation de l'objet distant

Le but de RMI est de simplifier l'utilisation d'objets distants. La seule chose supplémentaire qui doit être réalisée dans le programme client est de rechercher et de rapatrier depuis le serveur l'interface distante. Après quoi, c'est simplement de la programmation Java classique : envoyer des messages aux objets. Voici le programme qui utilise **PerfectTime**:

```
//: c15:rmi:DisplayPerfectTime.java  
// Utilise l'objet distant PerfectTime.  
package c15.rmi;  
import java.rmi.*;  
import java.rmi.registry.*;  
  
public class DisplayPerfectTime {  
    public static void main(String[] args) {  
        System.setSecurityManager(  
            new RMISecurityManager());  
        try {  
            PerfectTimeI t =  
                (PerfectTimeI)Naming.lookup(  
                    "//peppy:2005/PerfectTime");  
            for(int i = 0; i < 10; i++)
```

```

        System.out.println("Perfect time = " +
            t.getPerfectTime());
    } catch (Exception e) {
        e.printStackTrace();
    }
}
} ///:~

```

L'identifiant alpha-numérique est le même que celui utilisé pour enregistrer l'objet avec **Naming**, et la première partie représente l'adresse et le numéro du port. Cette URL permet par exemple de désigner une machine sur Internet.

Ce qui est retourné par **Naming.lookup()** doit être transtypé vers l'interface distante, *pas* vers la classe. Si l'utilisation de la classe à la place renverrait une exception.

Vous pouvez observer dans l'appel de la méthode

```
t.getPerfectTime()
```

qu'une fois la référence vers l'objet distant obtenue, la programmation avec celle-ci ou avec un objet distant est identique (avec une différence : les méthodes distantes émettent **RemoteException**).

Introduction à CORBA

Dans le cadre d'importantes applications distribuées, vos besoins risquent de ne pas être satisfaits par les approches précédentes : comme par exemple, l'intégration de bases existantes, ou l'accès aux services d'un objet serveur sans se soucier de sa localisation physique. Ces situations nécessitent une certaine forme de Remote Procedure Call (RPC), et peut-être une indépendance par rapport au langage. Là, CORBA peut vous aider.

CORBA n'est pas une fonctionnalité du langage ; c'est une technologie d'intégration. C'est une spécification que les fabricants peuvent suivre pour implémenter des produits supportant une intégration CORBA. CORBA fait partie du travail réalisé par OMG afin de définir une structure standard pour l'interopérabilité d'objets distribués et indépendants du langage.

CORBA fournit la possibilité de faire des appels à des procédures distantes dans des objets Java et des objets non-Java, et d'interfacer des systèmes existants sans se soucier de leur emplacement. Java ajoute un support réseau et un langage orienté-objet agréable pour construire des applications graphiques ou non. Le modèle objet de l'OMG et celui de Java vont bien ensemble ; par exemple, Java et CORBA mettent tout deux en oeuvre le concept d'interface et le modèle de référence objet.

Principes de base de CORBA

La spécification de l'interopérabilité objet est généralement désignée comme l'Object Manager Architecture (OMA). L'OMA définit deux composants : le Core Object Model et l'OMA Reference Architecture. Le Core Object Model met en place les concepts de base d'objet, d'interface, d'opération, et ainsi de suite (CORBA est un raffinement de Core Object Model). L'OMA Reference Architecture définit l'infrastructure sous-jacente des services et des mécanismes qui permettent aux objets d'inter-opérer. L'OMA Reference Architecture contient l'Object Request Broker (ORB), les Object Services (désignés aussi comme les services CORBA) et les outils communs.

L'ORB est le bus de communication par lequel les objets peuvent réclamer des services auprès des autres objets,

sans rien connaître de leur emplacement physique. Cela signifie que ce qui ressemble à un appel de méthode dans le code client est vraiment une opération complexe. D'abord, une connexion avec l'objet servant doit exister et pour créer cette connexion, l'ORB doit savoir où se trouve le code implémentant cette partie serveur. Une fois que la connexion est établie, les arguments de la méthode doivent être arrangés (marshaled), c'est à dire convertis en un flux binaire pour être envoyés à travers le réseau. Les autres informations qui doivent être envoyées sont le nom de la machine serveur, le processus serveur et l'identité de l'objet servant au sein de ce processus. Finalement, l'information est envoyée par l'intermédiaire d'un protocole bas-niveau, elle est décodée du côté du serveur, l'appel est exécuté. L'ORB masque tout de cette complexité au programmeur et rend l'opération presque aussi simple que l'appel d'une méthode d'un objet local.

Cette spécification n'a pas pour but d'expliquer comment le coeur de l'ORB devrait être implémenté, mais elle permet une compatibilité fondamentale entre les différents ORBs des fournisseurs, l'OMG définit un ensemble de services qui sont accessibles par l'intermédiaire d'interfaces standards.

CORBA Interface Definition Language (IDL - Langage de Définition d'Interface)

CORBA a été mis au point pour être transparent vis-à-vis du langage : un objet client peut appeler des méthodes d'un objet serveur d'une classe différente, sans se préoccuper du langage avec lequel elles sont implémentées. Bien sûr, l'objet client doit connaître le nom et les prototypes des méthodes que l'objet servant met à disposition. C'est là que l'IDL intervient. L'IDL de CORBA est un moyen indépendant du langage qui permet de préciser les types de données, les attributs, les opérations, les interfaces, et plus encore. La syntaxe de l'IDL est similaire à celles du C++ ou de Java. La table qui suit montre la correspondance entre quelques uns des concepts communs au trois langages qui peuvent être spécifiés à travers l'IDL de CORBA :

CORBA IDL	Java	C++
Module	Package	Namespace
Interface	Interface	Pure abstract class
Method	Method	Member function

Le concept d'héritage est également supporté, en utilisant le séparateur deux-points comme en C++. Le programmeur écrit en IDL la description des attributs, des méthodes et des interfaces qui seront implémentés et utilisés par le serveur et les clients. L'IDL est ensuite compilé par un compilateur IDL/Java propre au fournisseur, qui lit le source IDL et génère le code Java.

Le compilateur IDL est un outil extrêmement utile : il ne génère pas juste le source Java équivalent à l'IDL, il génère aussi le code qui sera utilisé pour réunir les arguments des méthodes et pour réaliser les appels distants. Ce code, appelé le code stub et le code skeleton, est organisé en plusieurs fichiers source Java et fait habituellement partie d'un même package Java.

Le service de nommage (naming service)

Le service de nommage est l'un des services fondamentaux de CORBA. L'objet CORBA est accessible par l'intermédiaire d'une référence, une information qui n'a pas de sens pour un lecteur humain. Mais les références peuvent être des chaînes de caractères définies par le programmeur. Cette opération est désignée comme chaînifier la référence, et l'un des composants de l'OMA, le service de nommage, est dévoué à la conversion nom-vers-objet et objet-vers-nom et gère ces correspondances. Puisque le service de nommage joue le rôle d'un annuaire téléphonique que les serveurs et les clients peuvent consulter et manipuler, il fonctionne dans un processus séparé. Créer une correspondance objet-vers-nom est appelé *lier (binding) un objet*, et supprimer cette

correspondance est dit *délier* (*unbinding*). Obtenir l'objet référence en passant la chaîne de caractères est appelé *résoudre le nom*.

Par exemple, au démarrage, une application serveur peut créer un objet servant, enregistrer l'objet auprès du service de nommage, et attendre que des clients fassent des requêtes. Un client obtient d'abord une référence vers cet objet servant en résolvant le nom et ensuite peut faire des appels auprès du serveur en utilisant cette référence.

A nouveau, la spécification du service de nommage fait partie de CORBA, mais l'application qui l'implémente est mise à disposition par le fournisseur de l'ORB. Le moyen d'accéder à ce service de nommage peut varier d'un fournisseur à l'autre.

Un exemple

Le code montré ici ne sera pas très élaboré car les différents ORBs ont des moyens d'accéder aux services CORBA qui divergent, ainsi les exemples dépendent du fournisseur. L'exemple qui suit utilise JavaIDL, un produit gratuit de Sun, qui fournit un ORB basique, un service de nommage, et un compilateur IDL-vers-Java. De plus, comme Java est encore jeune et en constante évolution, les différents produits CORBA pour Java n'incluent forcément toutes les fonctionnalités de CORBA.

Nous voulons implémenter un serveur, fonctionnant sur une machine donnée, qui soit capable de retourner l'heure exacte. Nous voulons aussi implémenter un client qui demande l'heure exacte. Dans notre cas, nous allons réaliser les deux programmes en Java, mais nous pourrions faire de même avec deux langages différents (ce qui se produit souvent dans la réalité).

Écrire le source IDL

La première étape consiste à écrire une description en IDL des services proposés. Ceci est généralement réalisé par le programmeur du serveur, qui est ensuite libre d'implémenter le serveur dans n'importe quel langage pour lequel un compilateur CORBA IDL existe. Le fichier IDL est communiqué au programme de la partie cliente et devient le pont entre les langages.

L'exemple qui suit montre la description IDL de notre serveur **ExactTime** :

```
///  
// # c15:corba:ExactTime.idl  
// # Vous devez installer idltojava.exe de  
// # java.sun.com et ajuster le paramétrage pour utiliser  
// # votre préprocesseur C local pour compiler  
// # ce fichier. Voyez la documentation sur java.sun.com.  
module remotetime {  
    interface ExactTime {  
        string getTime();  
    };  
}; ///  
// ~
```

C'est donc la déclaration de l'interface **ExactTime** au sein de l'espace de nommage **remotetime**. L'interface est composée d'une seule méthode qui retourne l'heure actuelle dans une **chaîne de caractères**.

Création des stubs et des skeletons

La deuxième étape consiste à compiler l'IDL pour créer le code Java du stub et du skeleton que nous utiliserons pour implémenter le client et le serveur. L'outil fourni par le produit JavaIDL est **idltojava** :


```
idltojava remotetime.idl
```

Cela générera automatiquement à la fois le code pour le stub et celui pour le skeleton. **Idltojava** génère un **package** Java nommé selon le module IDL **remotetime** et les fichiers Java générés sont déposés dans ce sous-répertoire **remotetime**. **_ExactTimeImplBase.java** est le skeleton que nous allons utiliser pour implémenter l'objet servant et **_ExactTimeStub.java** sera utilisé pour le client. Il y a des représentations Java de l'interface IDL dans **ExactTime.java** et toute une série d'autres fichiers de support utilisés, par exemple, pour faciliter l'accès aux fonctions du service de nommage.

Implémentation du serveur et du client

Ci-dessous, vous pouvez voir le code pour la partie serveur. L'implémentation de l'objet servant est dans la classe **ExactTimeServer**. **RemoteTimeServer** est l'application qui crée l'objet servant, l'enregistre auprès de l'ORB, donne un nom à la référence vers l'objet, et qui ensuite s'assoit tranquillement en attendant les requêtes des clients.

```

//: c15:corba:RemoteTimeServer.java
import remotetime.*;
import org.omg.CosNaming.*;
import org.omg.CosNaming.NamingContextPackage.*;
import org.omg.CORBA.*;
import java.util.*;
import java.text.*;

// Implémentation de l'objet servant
class ExactTimeServer extends _ExactTimeImplBase {
    public String getTime(){
        return DateFormat.
            getTimeInstance(DateFormat.FULL).
            format(new Date(
                System.currentTimeMillis()));
    }
}

// Implémentation de l'application distante
public class RemoteTimeServer {
    public static void main(String[] args) {
        try {
            // Crée l'ORB et l'initialise:
            ORB orb = ORB.init(args, null);
            // Crée l'objet servant et l'enregistre :
            ExactTimeServer timeServerObjRef =
                new ExactTimeServer();
            orb.connect(timeServerObjRef);
            // Obtient la racine du contexte de nommage :
            org.omg.CORBA.Object objRef =
                orb.resolve_initial_references(
                    "NameService");
            NamingContext ncRef =
                NamingContextHelper.narrow(objRef);
            // Associe un nom
            // à la référence de l'objet (binding):
            NameComponent nc =

```

```

        new NameComponent("ExactTime", "");
NameComponent[] path = { nc };
ncRef.rebind(path, timeServerObjRef);
// Attend les requêtes des clients :
java.lang.Object sync =
    new java.lang.Object();
synchronized(sync){
    sync.wait();
}
}
catch (Exception e) {
    System.out.println(
        "Remote Time server error: " + e);
    e.printStackTrace(System.out);
}
}
} ///:~

```

Comme vous pouvez le constater, implémenter l'objet servant est simple ; c'est une classe Java classique qui hérite du code du skeleton généré par le compilateur IDL. Les choses se complexifient un peu lorsqu'on en vient à interagir avec l'ORB et les autres services CORBA.

Quelques services CORBA

Voici une courte description de ce qui réalise le code relevant de JavaIDL (en évitant la partie de code CORBA qui dépend du fournisseur). La première ligne dans le **main()** démarre l'ORB parce que bien sûr notre objet servant aura besoin d'interagir avec celui-ci. Juste après l'initialisation de l'ORB, l'objet servant est créé. En réalité, le terme correct serait un *objet servant temporaire* (*transient*) : un objet qui reçoit les requêtes provenant des clients, et dont la durée de vie est limitée à celle du processus qui l'a créé. Une fois l'objet servant *temporaire* créé, il est enregistré auprès de l'ORB, ce qui signifie alors que l'ORB connaît son existence et peut rediriger les requêtes vers lui.

A partir de là, tout ce dont nous disposons est **timeServerObjRef**, une référence vers un objet qui n'est connu qu'à l'intérieur du processeur serveur actuel. L'étape suivante va consister à associer un nom alphanumérique à cet objet servant ; les clients utiliseront ce nom pour localiser l'objet servant. Cette opération sera réalisée grâce à l'aide du service de nommage. Tout d'abord, nous avons besoin d'une référence vers le service de nommage ; la méthode **resolve_initial_references()** utilise la référence objet « chainifiée » du service de nommage qui s'appelle **NameService** dans JavaIDL, et retourne la référence vers l'objet. Celle-ci est transformée en une référence spécifique à **NamingContext** au moyen de la méthode **narrow()**. Nous pouvons maintenant utiliser les services de nommage.

Pour associer l'objet servant avec une référence objet « chainifiée », nous créons d'abord un objet **NameComponent**, initialisé avec la chaîne de caractères qui sera associée : « ExactTime ». Ensuite, en utilisant la méthode **rebind()**, la référence alphanumérique est associée à la référence vers l'objet. Nous utilisons **rebind()** pour mettre en place une référence, même si celle-ci existe déjà. Un nom est composé dans CORBA d'une séquence de NameComponents (voilà pourquoi nous utilisons un tableau pour associer le nom à la référence).

L'objet est enfin prêt à être utilisé par des clients. A ce moment-là, le serveur entre dans un état d'attente. Encore une fois, ceci est nécessaire puisqu'il s'agit d'un serveur temporaire : sa durée de vie dépend du processus serveur. JavaIDL ne supporte pas actuellement les objets persistants, qui survivent après la fin de l'exécution du processus qui les a créés.

Maintenant que nous avons une idée de ce que fait le code du serveur, regardons le code du client :

```

//: c15:corba:RemoteTimeClient.java
import remotetime.*;
import org.omg.CosNaming.*;
import org.omg.CORBA.*;

public class RemoteTimeClient {
    public static void main(String[] args) {
        try {
            // Création et initialisation de l'ORB :
            ORB orb = ORB.init(args, null);
            // Obtient la racine du contexte de nommage :
            org.omg.CORBA.Object objRef =
                orb.resolve_initial_references(
                    "NameService");
            NamingContext ncRef =
                NamingContextHelper.narrow(objRef);
            // Résout la référence alphanumérique
            // du serveur d'heure :
            NameComponent nc =
                new NameComponent("ExactTime", "");
            NameComponent[] path = { nc };
            ExactTime timeObjRef =
                ExactTimeHelper.narrow(
                    ncRef.resolve(path));
            // Effectue une requête auprès de l'objet servant :
            String exactTime = timeObjRef.getTime();
            System.out.println(exactTime);
        } catch (Exception e) {
            System.out.println(
                "Remote Time server error: " + e);
            e.printStackTrace(System.out);
        }
    }
} //::~~

```

Les premières lignes font la même chose que ce qu'elles font dans le processus serveur : l'ORB est initialisé et la référence vers le service de nommage est résolue. Ensuite, nous avons besoin d'une référence objet pour l'objet servant, dont nous passons la référence alphanumérique à la méthode **resolve()**, et transformons le résultat en une interface **ExactTime** en utilisant la méthode **narrow()**. Enfin, nous appelons **getTime()**.

Activation du processus du service de nommage

Nous avons enfin un serveur et un client prêts à interopérer. Nous avons pu voir que tous deux ont besoin du service de nommage pour associer et résoudre les références alphanumériques. Le processus du service de nommage doit être démarré avant de faire fonctionner aussi bien le serveur que le client. Dans JavaIDL, le service de nommage est une application Java fournie avec le produit, mais qui peut être différente dans d'autres produits. Le service de nommage de JavaIDL fonctionne dans une JVM et écoute par défaut le port réseau 900.

Activation du serveur et du client

Vos applications serveur et client sont prêtes à démarrer (dans cet ordre, puisque votre serveur est transient (temporaire)). Si tout est bien en place, vous obtiendrez une simple ligne de sortie dans la fenêtre console de

vos client, indiquant l'heure courante. Bien sûr, cela ne semble pas très excitant en soit, mais vous devriez prendre en compte une chose : même si ils sont physiquement sur la même machine, les applications client et serveur fonctionnent dans des machines virtuelles différents et peuvent communiquer à travers une couche de l'intégration sous-jacente, l'ORB et le service de nommage.

Ceci constitue un exemple simple, destiné à fonctionner sans réseau, mais un ORB est généralement configuré pour qu'il rende les localisations transparentes. Lorsque le serveur et le client sont sur des machines différentes, l'ORB peut résoudre des références alphanumériques en utilisant un composant désigné *Implementation Repository*. Bien que le Implementation Repository fasse partie de CORBA, il n'y a quasiment pas de spécification, et donc il est différent d'un fabricant à l'autre.

Ainsi que vous vous en doutez, CORBA représente davantage que ce qui est montré ici, mais ainsi vous aurez compris l'idée de base. Si vous souhaitez plus d'informations à propos de CORBA, le premier endroit à visiter est le site Web de l'OMG : <http://www.omg.org>. Vous y trouverez la documentation, les white papers et les références vers les autres sources et produits pour CORBA.

Les Applets Java et CORBA

Les applets Java peuvent se comporter comme des clients CORBA. Par ce biais, une applet peut accéder à des informations et des services distants publiés sous la forme d'objets CORBA. Mais une applet ne peut se connecter qu'au serveur depuis lequel elle a été téléchargée, aussi tous les objets CORBA avec lesquels l'applet interagit doivent être situés sur ce serveur. C'est l'opposé de ce que CORBA essaye de faire : vous offrir une totale transparence de localisation.

C'est une question de sécurité réseau. Si vous êtes sur un intranet, la solution est d'éliminer les restrictions de sécurité du navigateur. Ou, de mettre en place une politique de firewall pour la connexion vers des serveurs externes.

Certains produits proposant des ORBs Java offrent des solutions propriétaires à ce problème. Par exemple, certains implémentent ce qui s'appelle un HTTP Tunneling, alors que d'autres ont des fonctionnalités firewall spécifiques.

C'est un point trop complexe pour être exposé dans une annexe, mais c'est quelque chose que vous devriez approfondir.

CORBA face à RMI

Vous avez vu que la principale fonctionnalité de CORBA est le support de RPC, qui permet à vos objets locaux d'appeler des méthodes d'objets distants. Bien sûr, il y a déjà une fonctionnalité native à Java qui réalise exactement la même chose : RMI (voir Chapitre 15). Là où RMI rend possible RPC entre des objets Java, CORBA rend possible RPC entre des objets implémentés dans n'importe quel langage.

Toutefois, RMI peut être utilisé pour appeler des services auprès de code distant non-Java. Tout ce dont vous avez besoin est d'une sorte d'objet Java adaptateur autour du code non-Java sur la partie serveur. L'objet adaptateur est connecté à l'extérieur aux objets clients par RMI, et en interne il se connecte au code non-Java en utilisant l'une des techniques vues précédemment, comme JNI ou J/Direct.

Cette approche nécessite que vous écriviez une sorte de couche d'intégration, exactement ce que CORBA fait pour vous, mais vous n'avez pas besoin ici d'un ORB venu de l'extérieur.

Enterprise Java Beans

[70] A partir de maintenant, vous avez été présenté à CORBA et à RMI. Mais pourriez-vous imaginer de tenter le développement d'une application à grande échelle en utilisant CORBA et/ou RMI ? Votre patron vous a demandé de développer une application multi-tiers pour consulter et mettre à jour des enregistrements dans une base de données, le tout à travers une interface Web. Vous vous assoyez et pensez à ce que cela veut vraiment dire. Sûr, vous pouvez écrire une application qui utilise JDBC, une interface Web qui utilise JSP et des Servlets, et un système distribué qui utilise CORBA/RMI. Mais quelles autres considérations devez-vous prendre en compte lorsque vous développez un système basé sur des objets distribués plutôt qu'avec des APIs classiques ? Voici les problèmes :

Performance : Les nouveaux objets distribués que vous avez créé devront être performants puisque ils devront potentiellement répondre à plusieurs clients en même temps. Donc vous allez vouloir mettre en place des techniques d'optimisation comme l'utilisation de cache, la mise en commun des ressources (comme les connexions à des bases de données par JDBC). Vous devrez aussi gérer le cycle de vie de votre objet distribué.

Adaptation à la charge : Les objets distribués doivent aussi s'adapter à l'augmentation de la charge. La scalabilité dans une application distribuée signifie que le nombre d'instances de votre objet distribué peut augmenter et passer sur une autre machine sans la modification du moindre code. Prenez par exemple un système que vous développez en interne comme une petite recherche de clients dans votre organisation depuis une base de données. L'application fonctionne bien quand vous l'utilisez, mais votre patron l'a vue et a dit : "Robert, c'est un excellent système, mettez ça sur notre site Web public maintenant !!!". Est-ce que mon objet distribué est capable de supporter la charge d'une demande potentiellement illimitée.

Sécurité : Mon objet distribué contrôle-t-il l'accès des clients ? Puis-je ajouter de nouveaux utilisateurs et des rôles sans tout recompiler ?

Transactions distribuées : Mon objet distribué peut-il utiliser de manière transparente des transactions distribuées ? Puis-je mettre à jour ma base de données Oracle et Sybase simultanément au sein de la même transaction et les annuler ensemble si un certain critère n'est pas respecté ?

Réutilisation : Ai-je créé mon objet distribué de telle sorte que je puisse le passer d'une application serveur d'un fournisseur à une autre ? Puis-je revendre mon objet distribué (composant) à quelqu'un d'autre ? Puis-je acheter un composant de quelqu'un d'autre et l'utiliser sans avoir à recompiler et à tailler dans son code ?

Disponibilité : Si l'une des machines de mon système tombe en panne, mes clients sont-ils capables de basculer des copies de sauvegarde de mes objets fonctionnant sur d'autres machines ?

Comme vous avez pu le voir ci-dessus, les considérations qu'un développeur doit prendre en compte lorsqu'il développe un système distribué sont complexes, et nous n'avons même pas parlé de la solution du problème qui nous essayons de résoudre à l'origine !

Donc maintenant vous avez une liste de problèmes supplémentaires que vous devez résoudre. Alors comme allez-vous vous y prendre ? Quelqu'un l'a certainement déjà fait ? Ne pourrais-je pas utiliser des modèles de conception bien connus pour m'aider à résoudre ces problèmes ? Soudain, une idée vous vient à l'esprit... « Je pourrais créer un framework qui prend en charge toutes ces contraintes et écrire mes composants en m'appuyant sur ce framework ! »... C'est là que les Entreprise JavaBeans rentrent en jeu.

Sun, avec d'autres fournisseurs d'objets distribués a compris que tôt ou tard toute équipe de développement serait en train de réinventer la roue. Ils ont donc créé la spécification des Entreprise JavaBeans (EJB). Les EJB sont une spécification d'un modèle de composant côté serveur qui prend en compte toutes les considérations mentionnées plus haut utilisant une approche standard et définie qui permet aux développeurs de créer des composants (qui sont appelés des Entreprise JavaBeans), qui sont isolés du code de la 'machinerie' bas-niveau et

focalisés seulement sur la mise en place de la logique métier. Et puisque les EJBs sont définis sur un standard, ils peuvent être utilisés sans être dépendant d'un fournisseur.

JavaBeans contre EJBs

La similitude entre les deux noms amène souvent une confusion entre le modèle de composants JavaBeans et la spécification des Enterprise JavaBeans. Bien que les spécifications des JavaBeans et des Enterprise JavaBeans partagent toutes deux les mêmes objectifs (comme la mise en avant de la réutilisation et la portabilité du code Java entre développements, comme aussi des outils de déploiement qui utilisent des modèles de conception standards), les motivations derrière chaque spécification ont pour but de résoudre des problèmes différents.

Les standards définis dans le modèle de composants JavaBeans sont conçus pour créer des composants réutilisables qui sont typiquement utilisés dans des outils de développement IDE et sont communément, mais pas exclusivement, des composants visuels.

La spécification des Enterprise JavaBeans définit un modèle de composants pour développer du code Java côté serveur. Comme les EJBs peuvent potentiellement fonctionner sur différentes plate-formes serveurs (incluant des systèmes qui n'ont pas d'affichage visuel), un EJB ne peut pas utiliser de bibliothèques graphiques tels que AWT ou Swing.

Que définit la spécification des EJBs ?

La spécification des EJBs, actuellement la version 1.1 (public release 2) définit un modèle de composant côté serveur. Elle définit six rôles qui sont utilisés pour réaliser les tâches de développement et de déploiement ainsi que la définition des composants du système.

Les rôles

La spécification des EJBs définit des rôles qui sont utilisés durant le développement, le déploiement et le fonctionnement d'un système distribué. Les fabricants, les administrateurs et les développeurs jouent des rôles variés. Ils permettent la séparation du savoir-faire technique, de celui propre au domaine. Ceci permet au fabricant de proposer un framework technique et aux développeurs de créer des composants spécifiques au domaine comme par exemple un composant de compte bancaire. Un même intervenant peut jouer un ou plusieurs rôles. Les rôles définis dans la spécification des EJBs sont résumés dans la table suivante :

Rôle	Responsabilité
Fournisseur d'Enterprise Bean	Le développeur qui est responsable de la création des composants EJB réutilisables, regroupés dans un fichier jar spécial (fichier ejb-jar).
Assembleur d'Application	Crée et assemble des applications à partir d'une collection de fichiers ejb-jar. C'est l'assembleur qui met en oeuvre la collection d'EJB (comme des Servlets, JSP, etc.).
Déployeur	Le rôle du déployeur est de prendre la collection de fichiers ejb-jar de l'Assembleur et de déployer ceux-ci dans un environnement d'exécution (un ou plusieurs Conteneurs).
Fournisseur de Conteneurs/Serveurs d'EJB	Fournit un environnement d'exécution et les outils qui sont utilisés pour déployer et faire fonctionner les composants EJB.
Administrateur Système	Par dessus tout, le rôle le plus important de l'ensemble du système : mettre en place la gestion d'une application distribuée consiste à ce que les composants et les serveurs interagissent ensemble correctement.

Composants EJB

Les composants EJB sont de la logique métier réutilisable. Les composants EJB suivent strictement les standards et les modèles de conception définis dans la spécification des EJBs. Cela permet à ces composants d'être portables et aussi à tous les autres services tels que la sécurité, la mise en cache et les transactions distribuées, d'être mis en oeuvre sur ces composants eux-mêmes. Un fournisseur d'Entreprise Bean est responsable du développement des composants EJB. Les entrailles d'un composant EJB sont traités dans

Qu'est-ce qui compose un composant EJB ?

Conteneur d'EJB

Le conteneur d'EJB est un environnement d'exécution qui contient et fait fonctionner les composants EJB tout en leur fournissant un ensemble de services. Les responsabilités des conteneurs d'EJB sont définies précisément par la spécification pour permettre une neutralité vis-à-vis du fournisseur. Les conteneurs d'EJB fournissent la machinerie bas-niveau des EJBs, incluant les transactions distribuées, la sécurité, la gestion du cycle de vie des beans, la mise en cache, la gestion de la concurrence et des sessions. Le fournisseur de conteneur d'EJB est responsable de la mise à disposition d'un conteneur d'EJB.

Serveur EJB

Un serveur d'EJB est défini comme un Serveur d'Applications et comporte un ou plusieurs conteneurs d'EJBs. Le fournisseur de serveur EJB est responsable de la mise à disposition d'un serveur EJB. Vous pouvez généralement considérer que le conteneur d'EJB et le serveur EJB sont une seule et même chose.

Java Naming and Directory Interface (JNDI)

Java Naming and Directory Interface (JNDI) est utilisé dans les Enterprise JavaBeans comme le service de nommage pour les composants EJB sur le réseau et pour les autres services du conteneur comme les transactions. JNDI ressemble fort aux autres standards de nommage et de répertoires tels que CORBA CosNaming et peut être implémenté comme un adaptateur de celui-ci.

Java Transaction API / Java Transaction Service (JTA/JTS)

JTA/JTS est utilisé dans les Enterprise JavaBeans comme API transactionnelle. Un fournisseur d'Entreprise Bean peut utiliser le JTS pour créer un code transactionnel bien que le conteneur d'EJB implémente généralement les transactions dans les EJBs sur les composants EJBs eux-mêmes. Le déploreur peut définir les attributs transactionnels d'un composant EJB au moment du déploiement. Le conteneur d'EJB est responsable de la prise en charge de la transaction qu'elle soit locale ou distribuée. La spécification du JTS est la version Java de CORBA OTS (Object Transaction Service).

CORBA et RMI/IIOP

La spécification des EJBs définit **l'interopérabilité** avec CORBA. La spécification 1.1 précise que *L'architecture des Enterprise JavaBeans sera compatible avec les protocoles CORBA*. **L'interopérabilité** avec CORBA passe par l'adaptation des services EJB comme JTS et JNDI aux services CORBA et l'implémentation de RMI à travers le protocole CORBA IIOP.

L'utilisation de CORBA et de RMI/IIOP dans les Enterprise JavaBeans est implémentée dans le conteneur EJB et est sous la responsabilité du fournisseur du conteneur d'EJB. L'utilisation de CORBA et de RMI/IIOP dans le conteneur EJB est invisible pour le composant EJB lui-même. Cela signifie que le fournisseur d'Entreprise Bean peut écrire ses composants EJBs et les déployer dans un conteneur EJB sans s'inquiéter du protocole de

communication qui est utilisé.

Qu'est-ce qui compose un composant EJB ?

Un EJB se décompose en un ensemble de pièce, dont le Bean lui-même, l'implémentation d'interfaces et un fichier d'informations. Le tout est rassemblé dans un fichier jar spécial.

Enterprise Bean

L'Enterprise Bean est une classe Java que le fournisseur d'Enterprise Bean développe. Elle implémente une interface `EnterpriseBean` (pour plus de détails, voir la section qui suit) et fournit l'implémentation des méthodes métier que le composant supporte. La classe n'implémente aucun mécanisme d'autorisation ou d'authentification, de concurrence ou transactionnel.

Interface Home

Chaque Enterprise Bean créé doit être associé à une interface Home. L'interface Home est utilisée comme une Factory de votre EJB. Les clients utilisent l'interface Home pour trouver une instance de votre EJB ou pour créer une nouvelle instance de votre EJB.

Interface Remote

L'interface Remote est l'interface Java qui reflète les méthodes de l'Enterprise Bean que l'on souhaite rendre disponible au monde extérieur. L'interface Remote joue un rôle similaire à celui de l'interface IDL de CORBA.

Descripteur de Déploiement

Le descripteur de Déploiement est un fichier XML qui contient les informations relatives à l'EJB. L'utilisation d'XML permet au Déployeur de facilement changer les attributs propres à l'EJB. Les attributs configurables définis dans le descripteur de déploiement incluent :

- Les noms des interfaces Home et Remote que nécessite l'EJB ;
- Le nom avec lequel sera publiée dans JNDI l'interface Home de l'EJB ;
- Les attributs transactionnels pour chaque méthode de l'EJB ;
- Les listes de contrôle d'accès (Access Control Lists) pour l'authentification.

Fichier EJB-Jar

Le fichier EJB-Jar est un fichier jar Java normal qui contient un EJB, les interface Home et Remote ainsi que le descripteur de déploiement.

Comment travaille un EJB ?

Maintenant que l'on a un fichier EJB-Jar contenant un Bean, les interfaces Home et Remote, et un descripteur de déploiement, voyons un peu comment toutes ces pièces vont ensemble, pourquoi les interfaces Home et Remote sont nécessaires et comment le conteneur d'EJB les utilise.

Le conteneur d'EJB implémente les interfaces Home et Remote qui sont dans le fichier EJB-Jar. Comme mentionné précédemment, l'interface Home met à disposition les méthodes pour créer et trouver votre EJB. Cela signifie que le conteneur d'EJB est responsable de la gestion du cycle de vie de votre EJB. Ce niveau d'abstraction permet aux optimisations d'intervenir. Par exemple, cinq clients peuvent demander simultanément

la création d'un EJB à travers l'interface Home, le conteneur d'EJB pourrait n'en créer qu'un seule et partager cet EJB entre les cinq clients. Ceci est réalisé à travers l'interface Remote, qui est aussi implémentée par le conteneur d'EJB. L'objet implémentant Remote joue le rôle d'objet proxy vers l'EJB.

Tous les appels de l'EJB sont redirigés à travers le conteneur d'EJB grâce aux interfaces Home et Remote. Cette abstraction explique aussi pourquoi le conteneur d'EJB peut contrôler la sécurité et le comportement transactionnel.

Types d'EJBs

Il doit y avoir une question dans votre tête depuis le dernier paragraphe : partager le même EJB entre les clients peut certainement augmenter les performances, mais qu'en est-il lorsque je souhaite conserver l'état sur le serveur ?

La spécification des Enterprise JavaBeans définissent différents types d'EJBs qui peuvent avoir différentes caractéristiques et adopter un comportement différent. Deux catégories d'EJBs ont été définies dans cette spécification : les Session Beans et les Entity Beans, et chacune de ces catégories a des variantes.

Session Beans

Les Session Beans sont utilisés pour représenter les cas d'utilisations ou des traitements spécifiques du client. Ils représentent les opérations sur les données persistantes, mais non les données persistantes elles-mêmes. Il y a deux types de Session Beans : non-persistant (Stateless) and persistant (Stateful). Tous les Session Beans doivent implémenter l'interface `javax.ejb.SessionBean`. Le conteneur d'EJB contrôle la vie d'un Session Bean.

Les Session Beans non-persistants

Les Session Beans non-persistants sont le type d'EJB le plus simple à implémenter. Ils ne conservent aucun état de leur conversation avec les clients entre les invocations de méthodes donc ils sont facilement réutilisables dans la partie serveur et puisqu'ils peuvent être mis en cache, ils supportent bien les variations de la demande. Lors de l'utilisation de Session Beans non-persistants, tous les états doivent être stockés à l'extérieur de l'EJB.

Les Session Beans persistants

Les Session Beans persistants conservent un état entre l'invocation de leurs méthodes (comme vous l'aviez probablement compris). Ils ont une association 1-1 avec un client et sont capables de conserver leurs états eux-mêmes. Le conteneur d'EJBs a en charge le partage et la mise en cache des Session Beans persistants, ceux-ci passe par la Passivation et l'Activation.

Entity Beans

Les Entity Beans sont des composants qui représentent une donnée persistante et le comportement de cette donnée. Les Entity Beans peuvent être partagés par plusieurs clients, comme une donnée d'une base. Le conteneur d'EJBs a en charge de mettre en cache les Entity Beans et de maintenir leur intégrité. La vie d'un Entity Bean est supérieur à celle du conteneur d'EJBs, donc si un conteneur tombe en panne, l'Entity Bean est censé être encore disponible lors que le conteneur le devient à nouveau.

Il y a deux types d'Entity Beans, ceux dont la persistance est assurée par le Bean lui-même et ceux dont la persistance est assurée par le conteneur.

Gestion de la persistance par le conteneur (CMP - Container Managed Persistence)

Un CMP Entity Bean a sa persistance assurée par le conteneur d'EJBs. A travers les attributs spécifiés dans le descripteur de déploiement, le conteneur d'EJBs fera correspondre les attributs de l'Entity Bean avec un stockage persistant (habituellement, mais pas toujours, une base de données). La gestion de la persistance par le conteneur réduit le temps de développement et réduit considérablement le code nécessaire pour l'EJB.

Gestion de la persistance par le Bean (BMP - Bean Managed Persistence)

Un BMP Entity Bean a sa persistance implémentée par le fournisseur de l'Entreprise Bean. Le fournisseur d'Entity Bean a en charge d'implémenter la logique nécessaire pour créer un nouvel EJB, mettre à jour certains attributs des EJBs, supprimer un EJB et trouver un EJB dans le stockage persistance. Cela nécessite habituellement d'écrire du code JDBC pour interagir avec une base de données ou un autre stockage persistant. Avec la gestion de persistance par le Bean (BMP), le développeur a le contrôle total de la manière dont la persistance de l'Entity Bean est réalisée.

Le principe de BMP apporte aussi de la flexibilité là où l'implémentation en CMP n'est pas possible, par exemple si vous souhaitez créer un EJB qui encapsule du code d'un système mainframe existant, vous pouvez écrire votre persistance en utilisant CORBA.

Développer un Enterprise Java Bean

Nous allons maintenant implémenter l'exemple de Perfect Time de la précédente section à propos de RMI sous forme d'un composant Enterprise JavaBean. Cet exemple est un simple Session Bean non-persistant. Les composants Enterprise JavaBean représenteront toujours au moins à une classe et deux interfaces.

La première interface définie est l'interface Remote de notre composant Enterprise JavaBean. Lorsque vous créez votre interface Remote de votre EJB, vous devez suivre ces règles :

1. L'interface Remote doit être **public**.
2. L'interface Remote doit hériter de l'interface **javax.ejb.EJBObject**.
3. Chaque méthode de l'interface Remote doit déclarer **java.rmi.RemoteException** dans sa section **throws** en addition des exceptions spécifiques à l'application.
4. Chaque objet passé en argument ou retourné par valeur (soit directement soit encapsulé dans un objet local) doit être un type de donnée valide pour RMI-IIOP (ce qui inclut les autres objets EJB).

Voici l'interface Remote plutôt simple de notre EJB PerfectTime :

```

//: c15:ejb:PerfectTime.java
//# Vous devez installer le J2EE Java Enterprise
//# Edition de java.sun.com et ajouter j2ee.jar
//# à votre CLASSPATH pour pouvoir compiler
//# ce fichier. Pour plus de détails,
//# reportez vous au site java.sun.com.
//# Interface Remote de PerfectTimeBean
import java.rmi.*;
import javax.ejb.*;

public interface PerfectTime extends EJBObject {
    public long getPerfectTime()
        throws RemoteException;
} ///:~

```

La seconde interface définie est l'interface Home de notre composant Enterprise JavaBeans. L'interface Home est la Factory du composant que vous allez créer. L'interface Home peut définir des méthodes de création ou de recherche. Les méthodes de création créent les instances des EJBs, les méthodes de recherche localisent les EJBs existants et sont utilisés pour les Entity Beans seulement. Lorsque vous créez votre interface Home d'un EJB, vous devez respecter ces quelques règles :

1. L'interface Home doit être **public**.
2. L'interface Home doit hériter de l'interface **javax.ejb.EJBHome**.
3. Chaque méthode de l'interface Home doit déclarer **java.rmi.RemoteException** dans sa section **throws** de même que **javax.ejb.CreateException**
4. La valeur retournée par une méthode de création doit être une interface Remote.
5. La valeur retournée par une méthode de recherche (pour les Entity Beans uniquement) doit être une interface Remote, **java.util Enumeration** ou **java.util.Collection**.
6. Chaque objet passé en argument ou retourné par valeur (soit directement ou encapsulé dans un objet local) doit être un type de donnée valide pour RMI-IIOP (ce qui inclut les autres objets EJB).

La convention standard de nommage des interfaces Home consiste à prendre le nom de l'interface Remote et d'y ajouter à la fin `Home`. Voici l'interface Home de notre EJB PerfectTime :

```

//: c15:ejb:PerfectTimeHome.java
// Interface Home de PerfectTimeBean.
import java.rmi.*;
import javax.ejb.*;

public interface PerfectTimeHome extends EJBHome {
    public PerfectTime create()
        throws CreateException, RemoteException;
} ///:~

```

Maintenant que nous avons défini les interfaces de notre composant, nous pouvons implémenter la logique métier qu'il y a derrière. Lorsque vous créez la classe d'implémentation de votre EJB, vous devez suivre ces règles (notez que vous pourrez trouver dans la spécification des EJBs la liste complète des règles de développement des Enterprise JavaBeans) :

1. La classe doit être **public**.
2. La classe doit implémenter une interface (soit **javax.ejb.SessionBean**, soit **javax.ejb.EntityBean**).
3. La classe doit définir les méthodes correspondant aux méthodes de l'interface Remote. Notez que la classe n'implémente pas l'interface Remote, c'est le miroir des méthodes de l'interface Remote mais elle n'émet pas **java.rmi.RemoteException**.
4. Définir une ou plusieurs méthodes **ejbCreate()** qui initialisent votre EJB.
5. La valeur retournée et les arguments de toutes les méthodes doivent être des types de données compatibles avec RMI-IIOP.

```

//: c15:ejb:PerfectTimeBean.java
// Un Session Bean non-persistant
// qui retourne l'heure système courante.
import java.rmi.*;
import javax.ejb.*;

public class PerfectTimeBean
    implements SessionBean {
    private SessionContext sessionContext;
    // retourne l'heure courante

```

```

public long getPerfectTime() {
    return System.currentTimeMillis();
}
// méthodes EJB
public void
ejbCreate() throws CreateException {}
public void ejbRemove() {}
public void ejbActivate() {}
public void ejbPassivate() {}
public void
setSessionContext(SessionContext ctx) {
    sessionContext = ctx;
}
} ///:~

```

Notez que les méthodes EJB (**ejbCreate()**, **ejbRemove()**, **ejbActivate()**, **ejbPassivate()**) sont toutes vides. Ces méthodes sont appelées par le conteneur d'EJBs et sont utilisées pour contrôler l'état de votre composant. Comme c'est un exemple simple, nous pouvons les laisser vides. La méthode **setSessionContext()** transmet un objet `javax.ejb.SessionContext` qui contient les informations concernant le contexte dans lequel se trouve le composant, telles que la transaction courante et des informations de sécurité.

Après que nous ayons créé notre Enterprise JavaBean, nous avons maintenant besoin de créer le descripteur de déploiement. Dans les EJBs 1.1, le descripteur de déploiement est un fichier XML qui décrit le composant EJB. Le descripteur de déploiement doit être stocké dans un fichier appelé **ejb-jar.xml**.

```

<?xml version="1.0" encoding="Cp1252"?>
<!DOCTYPE ejb-jar PUBLIC "-//Sun Microsystems, Inc.//DTD Enterprise Jav

<ejb-jar>
  <description>Exemple pour le chapitre 15</description>
  <display-name></display-name>
  <small-icon></small-icon>
  <large-icon></large-icon>
  <enterprise-beans>
    <session>
      <ejb-name>PerfectTime</ejb-name>
      <home>PerfectTimeHome</home>
      <remote>PerfectTime</remote>
      <ejb-class>PerfectTimeBean</ejb-class>
      <session-type>Stateless</session-type>
      <transaction-type>Container</transaction-type>
    </session>
  </enterprise-beans>
  <ejb-client-jar></ejb-client-jar>
</ejb-jar>

```

Dans la balise **<session>** de votre descripteur de déploiement, vous pouvez voir que le composant, les interfaces `Remote` et `Home` sont définis en partie. Les descripteurs de déploiement peuvent facilement être générés automatiquement grâce à des outils tels que `JBuilder`.

Par l'intermédiaire de ce descripteur de déploiement standard **ejb-jar.xml**, la spécification des EJBs 1.1 institue que toutes balises spécifiques aux fournisseurs doivent être stockées dans un fichier séparé. Ceci a pour but d'assurer une compatibilité complète entre les composants et les conteneurs d'EJBs de différents marques.

Maintenant que nous avons créé notre composant et défini sa composition dans le descripteur de déploiement, nous devons alors archiver les fichiers dans un fichier archive Java (JAR). Le descripteur de déploiement doit être placé dans le sous répertoire **META-INF** du fichier Jar.

Une fois que nous avons défini notre composant EJB dans le descripteur de déploiement, le déploreur doit maintenant déployer le composant EJB dans le conteneur d'EJB. A ce moment-là du développement, le processus est plutôt orienté IHM et spécifique à chaque conteneur d'EJBs. Nous avons donc décidé de ne pas documenter entièrement le processus de déploiement dans cette présentation. Chaque conteneur d'EJBs propose cependant à un processus détaillé pour déployer un EJB.

Puisqu'un composant EJB est un objet distribué, le processus de déploiement doit créer certains stubs clients pour appeler le composant EJB. Ces classes seront placées dans le classpath de l'application cliente. Puisque les composants EJB sont implémentés par dessus RMI-IIOP (CORBA) ou RMI-JRMP, les stubs générés peuvent varier entre les conteneurs d'EJBs, néanmoins ce sont des classes générées.

Lorsqu'un programme client désire invoquer un EJB, il doit rechercher le composant EJB dans JNDI et obtenir une référence vers l'interface Home du composant EJB. L'interface HOME peut alors être invoquée pour créer une instance de l'EJB, qui peut à son tour être invoquée.

Dans cet exemple le programme client est un simple programme Java, mais vous devez garder en mémoire qu'il pourrait s'agir aussi bien d'un Servlet, d'un JSP que d'un objet distribué CORBA ou RMI.

Le code de PerfectTimeClient code est le suivant.

```

//: c15:ejb:PerfectTimeClient.java
// Programme Client pour PerfectTimeBean

public class PerfectTimeClient {
public static void
main(String[] args) throws Exception {
    // Obtient un context JNDI utilisant le
    // service de nommage JNDI :
    javax.naming.Context context =
        new javax.naming.InitialContext();
    // Recherche l'interface Home dans le
    // service de nommage JNDI :
    Object ref = context.lookup("perfectTime");
    // Transforme l'objet distant en une interface Home :
    PerfectTimeHome home = (PerfectTimeHome)
        javax.rmi.PortableRemoteObject.narrow(
            ref, PerfectTimeHome.class);
    // Crée un objet distant depuis l'interface Home :
    PerfectTime pt = home.create();
    // Invoque getPerfectTime()
    System.out.println(
        "Perfect Time EJB invoked, time is: " +
        pt.getPerfectTime() );
    }
} //::~~

```

Le déroulement de cet exemple est expliqué dans les commentaires. Notez l'utilisation de la méthode narrow() pour réaliser une sorte de transtypage de l'objet avant qu'un transtypage Java soit fait. Ceci est très similaire à ce qui se passe en CORBA. Notez aussi que l'objet Home devient une Factory pour les objets PerfectTimes.

En résumé

La spécification des Enterprise JavaBeans est un pas important vers la standardisation et la simplification de l'informatique distribuée. C'est une pièce majeure de Java 2, Enterprise Edition Platform et reçoit en plus le concours la communauté travaillant sur les objets distribués. De nombreux outils sont actuellement disponibles ou le seront dans un futur proche pour accélérer le développement de composants EJBs.

Cette présentation avait pour but de donner un bref aperçu de ce que sont les EJBs. Pour plus d'informations à propos de la spécification des Enterprise JavaBeans, vous pouvez vous reporter à la page officielle des Enterprise JavaBeans à l'adresse <http://java.sun.com/products/ejb/>. Vous pourrez y télécharger la dernière spécification ainsi que Java 2, Enterprise Edition Reference Implementation, qui vous permettra de développer et de déployer vos propres composants EJBs.

Jini : services distribués

Cette section [71] donne un aperçu de la technologie Jini de Sun Microsystem. Elle décrit quelques spécificités Jini et montre comment l'architecture Jini aide à augmenter le niveau d'abstraction dans la programmation de systèmes distribués, transformant réellement la programmation réseau en programmation orientée-objets.

Contexte de Jini

Traditionnellement, les systèmes d'exploitation sont conçus dans l'hypothèse qu'un ordinateur aura un processeur, un peu de mémoire et un disque. Lorsque vous démarrez votre ordinateur, la première chose qu'il fait est de chercher un disque. S'il ne le trouve pas, il ne peut assurer sa fonction d'ordinateur. Cependant de plus en plus les ordinateurs apparaissent sous diverses formes : comme des systèmes embarqués qui possèdent un processeur, un peu de mémoire et une connexion réseau mais pas de disque. La première chose que fait, par exemple, un téléphone cellulaire lorsqu'il s'allume est de rechercher le réseau téléphonique. S'il ne le trouve pas, il ne peut assurer sa fonction de téléphone. Cette nouvelle mode dans l'environnement matériel, le passage d'un système centré sur un disque à un système centré sur un réseau, va affecter la manière d'organiser notre logiciel. C'est là qu'intervient Jini.

Jini est une manière de repenser l'architecture de l'ordinateur, étant donné l'importance croissante des réseaux et la prolifération des processeurs dans des systèmes qui n'ont pas de disque dur. Ces systèmes, qui proviennent de nombreux fabricants différents, vont avoir besoin d'interagir à travers le réseau. Le réseau lui-même sera très dynamique : les systèmes et les services seront ajoutés et retirés régulièrement. Jini apporte les mécanismes permettant facilement l'ajout, la suppression et la recherche de systèmes et de services sur le réseau. De plus, Jini propose un modèle de programmation qui rend tout cela plus facile pour les programmeurs qui souhaitent voir leurs systèmes discuter entre eux.

S'appuyant sur Java, la sérialisation objet et RMI (qui permet aux objets de bouger à travers le réseau en passant d'une machine virtuelle à une autre), Jini permet d'étendre les bénéfices de la programmation orientée-objet au réseau. Au lieu de nécessiter un accord entre les différents fabricants sur un protocole réseau à travers lequel les systèmes peuvent interagir, Jini permet à ces systèmes de discuter ensemble par l'intermédiaire d'interfaces vers des objets.

Qu'est-ce que Jini ?

Jini est un ensemble d'APIs et de protocoles réseaux qui peuvent vous aider à construire et déployer des systèmes distribués qui sont organisés sous la forme de *fédérations de services*. Un *service* peut être n'importe quoi qui se trouve sur le réseau et qui est prêt à réaliser une fonction utile. Des composants matériels, logiciels, des canaux de communications, les utilisateurs eux-mêmes peuvent être des services. Une imprimante

compatible Jini pourra offrir un service d'impression. Une fédération de services est un ensemble de services, actuellement disponibles sur le réseau, que le client (ce qui signifie programme, service ou utilisateur) peut combiner pour s'aider à atteindre à son but.

Pour réaliser une tâche, un client enchaîne les possibilités des services. Par exemple, un programme client peut charger des photographies d'un système de stockage d'image d'un appareil numérique, envoyer les photos vers un service de stockage persistant offert par un disque dur, et transmettre une page contenant les vignettes de chaque image à un service d'impression d'une imprimante couleur. Dans cet exemple, le programme client construit un système distribué constitué de lui-même, le service de stockage d'images, le service de stockage persistant et le service d'impression couleur. Le client et ces services de ce système distribué collaborent pour réaliser une tâche : télécharger et stocker les images d'un appareil numérique et imprimer une page de vignettes.

L'idée derrière le mot *fédération* est que la vision Jini d'un réseau n'instaure pas d'autorité de contrôle centrale. Puisque aucun service n'est responsable, l'ensemble de tous les services disponible sur le réseau forme une fédération, un groupe composé de membres égaux. Au lieu d'une autorité centrale, l'infrastructure d'exécution de Jini propose un moyen pour les clients et les services de se trouver mutuellement (à travers un service de recherche, qui stocke la liste des services disponibles à moment donné). Après que les services se sont trouvés, ils sont indépendants. Le client et ces services mis à contribution réalisent leurs tâches indépendamment de l'infrastructure d'exécution de Jini. Si le service de recherche Jini tombe, tous les systèmes distribués mis en place par le service de recherche, avant qu'il ne plante, peuvent continuer les travaux. Jini incorpore même un protocole réseau qui permet aux clients de trouver les services en l'absence d'un service de nommage.

Comment fonctionne Jini

Jini définit une *infrastructure d'exécution* qui réside sur le réseau et met à disposition des mécanismes qui vous permettent d'ajouter, d'enlever, de localiser et d'accéder aux services. L'infrastructure d'exécution se situe à trois endroits : dans les services de recherche qui sont sur le réseau, au niveau des fournisseurs de service (tels que les systèmes supportant Jini), et dans les clients. *Les services de recherche* forment le mécanisme centralisé d'organisation des systèmes basés sur Jini. Lorsque des services deviennent disponibles sur le réseau, ils s'enregistrent eux-même grâce à un service de recherche. Lorsque des clients souhaitent localiser un service pour être assistés dans leur travail, ils consultent le service de recherche.

L'infrastructure d'exécution utilise un protocole au niveau réseau, appelé *discovery* (*découverte*), et deux protocoles au niveau objet appelés *join* (*joindre*) et *lookup* (*recherche*). Discovery permet aux clients et aux services de trouver les services de recherche. Join permet au service de s'enregistrer lui-même auprès du service de recherche. Lookup permet à un client de rechercher des services qui peuvent l'aider à atteindre ses objectifs.

Le processus de découverte

1. Le processus de découverte travaille ainsi : imaginez un disque supportant Jini et offrant un service de stockage persistant. Dès que le disque est connecté au réseau, il diffuse une *annonce de présence* en envoyant un paquet multicast sur un port déterminé. Dans l'annonce de présence, sont inclus une adresse IP et un numéro de port où le disque peut être contacté par le service de recherche.

Les services de recherche scrutent sur le port déterminé les paquets d'annonce de présence. Lorsqu'un service de recherche reçoit une annonce de présence, il l'ouvre et inspecte le paquet. Le paquet contient les informations qui permet au service de recherche de déterminer s'il doit ou non contacter l'expéditeur de ce paquet. Si tel est le cas, il contacte directement l'expéditeur en établissant une connexion TCP à l'adresse IP et sur le numéro de port extraits du paquet. En utilisant RMI, le service de recherche envoie à l'initiateur du paquet un objet appelé un *enregistreur de service* (*service registrar*). L'objectif de cet enregistreur de service est de faciliter la communication future avec le service de recherche. Dans le cas d'un disque dur, le service de recherche établirait une connexion TCP vers le disque dur et lui enverrait un *enregistreur de service*, grâce auquel le

disque dur pourra faire enregistrer son service de stockage persistant par le processus de jonction.

Le processus de jonction

Dès lors qu'un fournisseur de service possède un *enregistreur de service*, le produit final du processus de découverte, il est prêt à entreprendre une jonction pour intégrer la fédération des services qui sont enregistrés auprès du service de recherche. Pour réaliser une jonction, le fournisseur de service fait appel à la méthode **register()** de l'*enregistreur de service*, passant en argument un objet appelé élément de service (service item), un ensemble d'objets qui décrit le service. La méthode **register()** envoie une copie de cet élément de service au service de recherche, où celui-ci sera stocké. Lorsque ceci est achevé, le fournisseur de service a fini le processus de jonction : son service est maintenant enregistré auprès du service de recherche.

L'élément de service contient plusieurs objets, parmi lesquels un objet appelé un *objet service*, que les clients utilisent pour interagir avec le service. L'élément de service peut aussi inclure un certain nombre d'*attributs*, qui peuvent être n'importe quel objet. Certains de ces attributs sont des icônes, des classes qui fournissent des interfaces graphiques pour le service et des objets apportant plus de détails sur le service.

Les objets service implémentent généralement une ou plusieurs interfaces à travers lesquelles les clients interagissent avec le service. Par exemple, le service de recherche est un service Jini, et son objet service est un service de registre. La méthode **register()** appelée par les fournisseurs de service durant la jonction est déclarée dans l'interface **ServiceRegistrar** (un membre du package **net.jini.core.lookup**) que tous les services de registre implémentent. Les clients et les fournisseurs de registre discutent avec le service de recherche à travers l'objet de service de registre en invoquant les méthodes déclarées dans l'interface **ServiceRegistrar**. De la même manière, le disque dur fournit un objet service qui implémente l'une des interfaces connues de service de stockage. Les clients peuvent rechercher le disque dur et interagir avec celui-ci par cette interface de service de stockage.

Le processus de recherche

Une fois qu'un service a été enregistré par un service de recherche grâce au processus de jonction, ce service est utilisable par les clients qui le demandent au service de recherche. Pour construire un système distribué de services qui collaborent pour réaliser une tâche, un client doit localiser ses services et s'aider de chacun d'eux. Pour trouver un service, les clients formulent des requêtes auprès des services de recherche par l'intermédiaire d'un processus appelé *recherche*.

Pour réaliser une recherche, un client fait appel à la méthode **lookup()** d'un service de registre (comme un fournisseur de service, un client obtient un service de registre grâce au processus de découverte décrit précédemment). Le client passe en argument un modèle de service à **lookup()**, un objet utilisé comme critère de recherche. Le modèle de service peut inclure une référence à un tableau d'objets **Class**. Ces objets **Class** indiquent au service de recherche le type Java (ou les types) de l'objet service voulu par le client. Le modèle de service peut aussi inclure un *service ID*, qui identifie de manière unique le service, ainsi que des attributs, qui doivent correspondre exactement aux attributs fournis par le fournisseur de service dans l'*élément de service*. Le modèle de service peut aussi contenir des critères génériques pour n'importe quel attribut. Par exemple, un critère générique dans le champ service ID correspondra à n'importe quel service ID. La méthode **lookup()** envoie le modèle de service au service de recherche, qui exécute la requête et renvoie s'il y en a les objets services correspondants. Le client récupère une référence vers ces objets services comme résultat de la méthode **lookup()**.

En général, un client recherche un service selon le type Java, souvent une interface. Par exemple, si un client avait besoin d'utiliser une imprimante, il pourrait créer un modèle de service qui comprend un objet **Class** d'une interface connue de services d'impression. Tous les services d'impression implémenteraient cette interface

connue. Le service de recherche retournerait un ou plusieurs objets services qui implémentent cette interface. Les attributs peuvent être inclus dans le modèle de service pour réduire le nombre de correspondances de ce genre de recherche par type. Le client pourrait utiliser le service d'impression en invoquant sur l'objet service les méthodes définies dans l'interface.

Séparation de l'interface et de l'implémentation

L'architecture Jini met en place pour le réseau une programmation orientée-objet en permettant aux services du réseau de tirer parti de l'un des fondements des objets : la séparation de l'interface et l'implémentation. Par exemple, un objet service peut permettre aux clients d'accéder au service de différentes manières. L'objet peut réellement représenter le service entier, qui sera donc téléchargé par le client lors de la recherche et exécuté localement ensuite. Autrement, l'objet service peut n'être qu'un proxy vers un serveur distant. Lorsqu'un client invoque des méthodes de l'objet service, il envoie les requêtes au serveur à travers le réseau, qui fait réellement le travail. Une troisième option consiste à partager le travail entre l'objet service local et le serveur distant.

Une conséquence importante de l'architecture Jini est que le protocole réseau utilisé pour communiquer entre l'objet service proxy et le serveur distant n'a pas besoin d'être connu du client. Comme le montre la figure suivante, le protocole réseau est une partie de l'implémentation du service. Ce protocole est une question privée prise en compte par le développeur du service. Le client peut communiquer avec l'implémentation du service à travers un protocole privée car le service injecte un peu de son propre code (l'objet service) au sein de l'espace d'adressage du client. L'objet service ainsi injecté peut communiquer avec le service à travers RMI, CORBA, DCOM, un protocole fait maison construit sur des sockets et des flux ou n'importe quoi d'autre. Le client ne porte simplement aucune attention quant aux protocoles réseau, puisqu'il ne fait que communiquer avec l'interface publique que le service implémente. L'objet service prend en charge toutes les communications nécessaires sur le réseau.



Le client communique avec le service à travers une interface publique

Différentes implémentations de la même interface d'un service peuvent utiliser des approches et des protocoles réseau totalement différents. Un service peut utiliser un matériel spécialisé pour répondre aux requêtes clientes, ou il peut tout réaliser de manière logicielle. En fait, le choix d'implémentation d'un même service peut évoluer dans le temps. Le client peut être sûr qu'il possède l'objet service qui comprend l'implémentation actuelle de ce service, puisque le client reçoit l'objet service (grâce au service de recherche) du fournisseur du service lui-même. Du point de vue du client, le service ressemble à une interface publique, sans qu'il ait à se soucier de l'implémentation du service.

Abstraction des systèmes distribués

Jini tente d'élever passer le niveau d'abstraction de la programmation de systèmes distribués, passant du niveau du protocole réseau à celui de l'interface objet. Dans l'optique d'une prolifération des systèmes embarqués connectés au réseau, beaucoup de pièces d'un système distribué pourront venir de fournisseurs différents. Jini évite aux fournisseurs de devoir se mettre d'accord sur les protocoles réseau qui permettent à leurs systèmes d'interagir. A la place, les fournisseurs doivent se mettre d'accord sur les interfaces Java à travers lesquelles leurs systèmes peuvent interagir. Les processus de découverte, de jonction et de recherche, fournis par l'infrastructure d'exécution de Jini, permettront aux systèmes de se localiser les uns les autres sur le réseau. Une fois qu'ils se sont localisés, les systèmes sont capables de communiquer entre eux à travers des interfaces Java.

Résumé

Avec Jini pour des réseaux de systèmes locaux, ce chapitre vous a présenté une partie (une partie seulement) des composants que Sun regroupe sous le terme de J2EE : the Java 2 Enterprise Edition. Le but de J2EE est de construire un ensemble d'outils qui permettent au développeur Java de construire des applications serveurs beaucoup plus rapidement et indépendamment de la plate-forme. Construire de telles applications n'est pas seulement difficile et coûteux en temps, mais il est particulièrement dur de les construire en faisant en sorte qu'elles puissent être facilement portées sur une autre plate-forme, et aussi que la logique métier soit isolée des détails relevant de l'implémentation. J2EE met à disposition une structure pour assister la création d'applications serveurs ; ces applications sont très demandées en ce moment, et cette demande semble grandir.

Exercices

1. Compiler et exécuter les programmes **JabberServer** et **JabberClient** de ce chapitre. Maintenant éditer les fichiers pour supprimer toute bufferisation sur l'entrée et la sortie, ensuite les compiler et les lancer à nouveau pour observer le résultat.
2. Créer un serveur qui demande un mot de passe, puis qui ouvre un fichier et qui en envoie le contenu à travers la connexion réseau. Créer le client qui se connecte à ce serveur, qui donne le mot de passe approprié et qui capture et sauve le fichier. Tester ces deux programmes sur votre machine en utilisant **localhost** (l'adresse IP de la boucle locale **127.0.0.1** obtenue en appelant **InetAddress.getByName(null)**).
3. Modifier le serveur de l'exercice 2 pour qu'il utilise plusieurs threads pour répondre à plusieurs clients.
4. Modifier **JabberClient** pour que le **flushing** de la sortie ne se fasse pas et observer l'effet.
5. Modifier **MultiJabberServer** pour qu'il utilise un *pool de threads*. Au lieu d'abandonner un thread à chaque fois qu'un client se déconnecte, le thread doit se poser lui-même dans un pool de threads disponibles. Lorsqu'un nouveau client veut se connecter, le serveur se tournera vers ce pool pour répondre à la requête, et s'il n'y plus de thread disponible, il en créera un nouveau. De cette manière, le nombre de threads nécessaires grossira naturellement jusqu'à atteindre la quantité requise. L'avantage du pool de threads est qu'il n'est pas nécessaire de créer et de détruire un thread pour chaque client.
6. Construire à partir de **ShowHTML.java** une applet qui soit un pont protégé par un mot de passe vers une portion particulière de votre site Web.
7. (Plus difficile) Créer une paire de programmes client/serveur qui utilisent des datagrammes pour transmettre un fichier d'une machine à l'autre (Regardez la description à la fin de la section consacrée au datagramme dans ce chapitre).
8. (Plus difficile) Prendre le programme **VLookup.java** et le modifier pour que lorsque vous cliquez sur le nom obtenu, il prenne automatiquement ce nom et le copie dans le presse-papier (ainsi vous pouvez simplement le copier dans votre courrier électronique). Vous aurez besoin de retourner au chapitre sur les flux d'entrée/sortie pour vous rappeler comment utiliser le presse-papier de Java 1.1.

[68] Ce qui représente un peu plus de 4 milliards de valeurs, ce qui sera vite épuisé. Le nouveau standard pour les adresses IP utilisera des nombres de 128 octets, qui devraient produire assez d'adresses IP pour le futur proche.

[69] Beaucoup de neurones sont morts après une atroce agonie pour découvrir cette information.

[70] Cette section a été réalisée par Robert Castaneda, avec l'aide de Dave Bartlett.

[71] Cette section a été réalisée par Bill Venners (www.artima.com).

[[Previous Chapter](#)] [[Short TOC](#)] [[Table of Contents](#)] [[Index](#)] [[Next Chapter](#)]

Last Update:03/13/2000