INP TOULOUSE ENSEEIHT

Ingénierie des Applications réparties

Appel de méthode à distance en Java : RMI (Remote Method Invocation)

Introduction

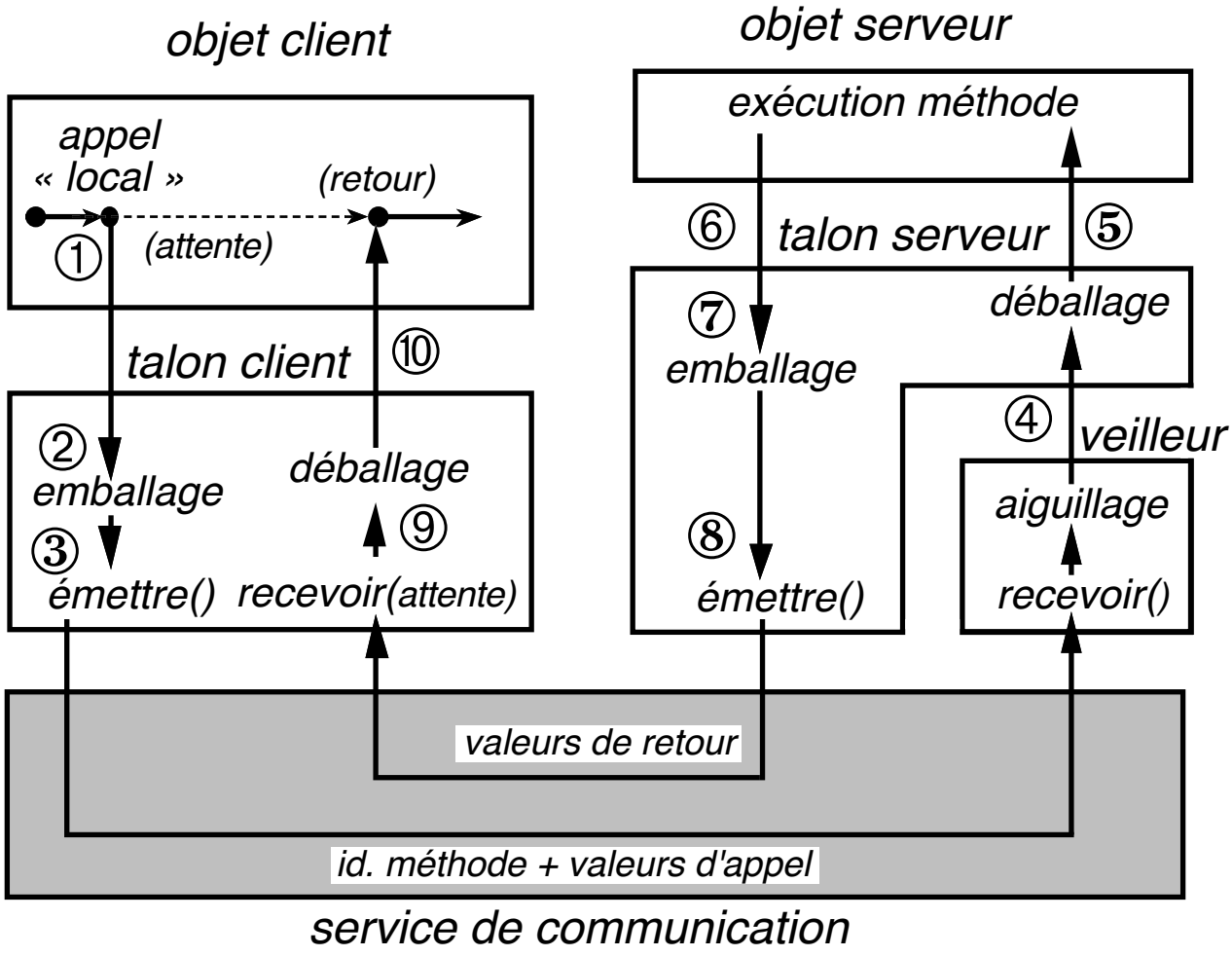
L'appel de méthode à distance a pour but d'offrir le plus de transparence possible vis-àvis de la répartition et de la localisation des composants d'une application répartie, et cela de manière intégrée à un langage de programmation (objet).

L'intégration au langage est obtenue en permettant d'exprimer les interactions de type client/serveur (requête/réponse) comme des appels de méthodes.

Pour assurer la transparence vis-à vis de la localisation, il faut fournir au client, à l'utilisateur du service, l'illusion d'un appel de méthode local, alors que la méthode est exécutée par un autre processus, éventuellement situé sur une autre machine. Pour cela, l'objet distant doit être représenté localement par un *proxy* : objet fournissant la même interface que l'objet distant et qui, en fait, prend en charge l'interaction avec cet objet distant. Du côté serveur, l'objet distant doit pouvoir être conçu comme s'il n'avait à traiter que des invocations locales, tout en ayant en réalité la possibilité de recevoir des invocations provenant de l'extérieur. A cette fin, on lui adjoint un *squelette,* qui prend en charge l'interaction avec les clients distants, et effectue pour le compte de ces derniers les appels à l'objet distant . Une configuration permettant au processus P_1 d'appeler une méthode sur l'objet accessible à distance A est alors la suivante :

Dans cette configuration, un appel se déroule ainsi :

1) le client appelle la méthode sur un objet local (le proxy, ou *talon* client, ou *stub*) qui représente l'objet distant.
2) le talon emballe l'identifiant de la méthode et les arguments (*marshalling*) ;
3) la requête est transmise en un ou plusieurs messages réseau ;
4) le squelette (ou talon serveur) reçoit et déballe les messages ;
5) la méthode demandée est appelée localement avec les arguments déballés ;
6) la méthode renvoie un résultat au squelette ;



7) le squelette emballe le résultat ;
8) la réponse est transmise en un ou plusieurs messages réseau ;
9) le talon client reçoit et déballe les messages ;
10) le talon client renvoie un résultat comme une méthode ordinaire.

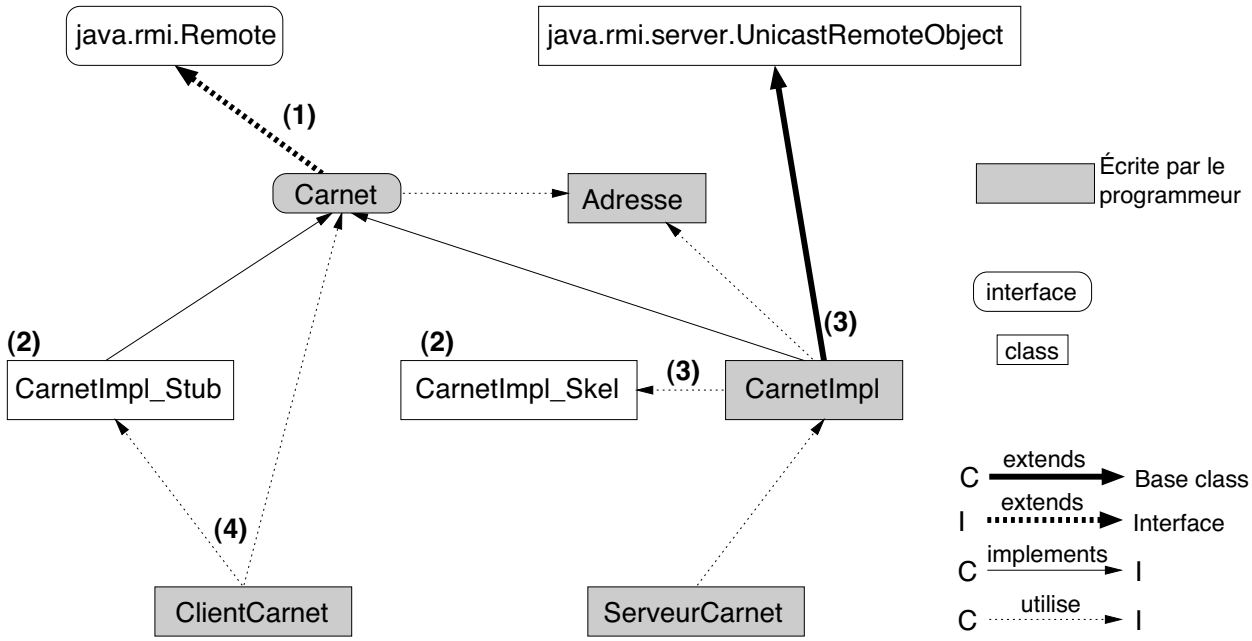
Ainsi, le client a l'illusion de faire un appel local, et l'objet serveur a l'illusion d'être appelé localement.

Déroulement du développement

La mise en œuvre d'objets accessibles à distance requiert plusieurs étapes :

1) la définition de l'interface d'appel aux objets ;
2) la programmation d'une implantation des méthodes ;
3) la génération automatique des talons (squelettes et proxys) ;
4) le développement d'un programme créant un ou plusieurs objets afin que ces derniers puissent être disponibles pour des clients distants éventuels ;
5) le développement de programmes utilisant ce(s) objet(s) par appel à distance.

Nous obtenons ainsi le graphe de classes suivant:



Les points importants sont :

1) l'interface visible par le client (ici, `Carnet`) dérive de `java.rmi.Remote` ;
2) le proxy et le squelette sont engendrés automatiquement ; le proxy `CarnetImpl_Stub` est effectivement une implantation de l'interface `Carnet` ;
3) l'implantation d'un objet accessible à distance `CarnetImpl` nécessite un squelette qui est constitué d'une partie spécifique à l'interface `CarnetImpl_Skel` et d'une partie générique provenant d'une classe système ;
4) un client utilise un objet proxy de type `CarnetImpl_Stub`, mais ne le manipule que sous son interface `Carnet`

Passage de paramètres

En Java « traditionnel », les paramètres sont passés par référence, sauf pour les types primitifs (booléens, entiers, flottants et caractères) qui sont passés par valeur (copie). Lors d'un appel à distance :

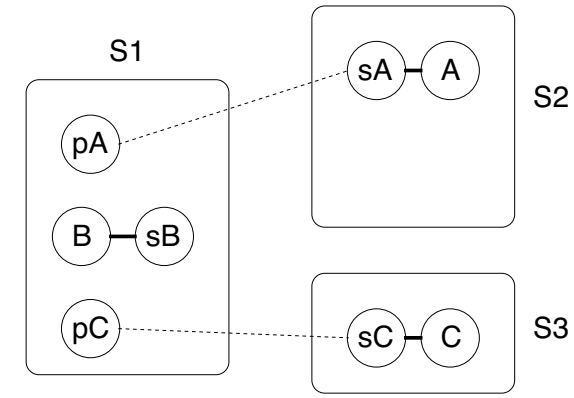
- les valeurs primitives sont passées par valeur (copie) ;
- les objets sérialisables sont sérialisés et passés par copie (différence notable avec un appel local de méthode) ;
- les objets accessibles à distance (implantant l'interface `java.rmi.Remote`) sont passés par référence (de fait leur stub est transmis);
- les objets qui ne sont ni sérialisables ni accessibles à distance provoquent une exception `java.rmi.MarshalException`

Il en est de même dans le cas d'une fonction accessible à distance qui renvoie un objet.

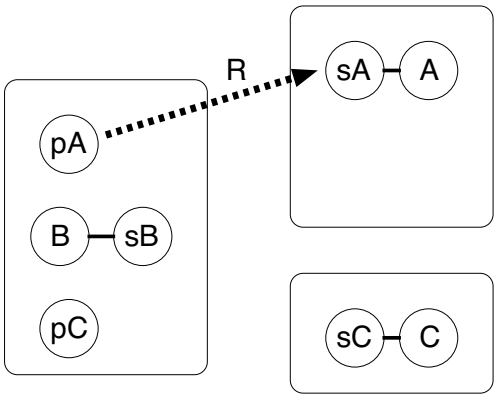
Obtenir un proxy

Pour qu'un client puisse appeler une méthode sur un objet distant, doit disposer d'un objet local, un proxy, qui contient le talon client de la méthode. Pour obtenir ce proxy, deux solutions sont possibles :

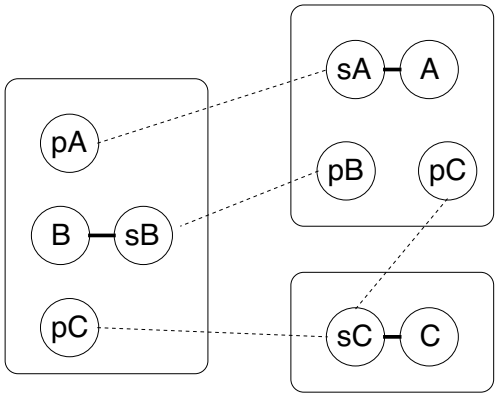
- Avoir appelé une méthode (à distance) qui transmet/renvoie la référence d'un (autre) objet accessible à distance. Par exemple, la figure suivante montre une situation où, sur le site S_1, un appel à distance vers un objet situé sur S_2 a lieu. Cet appel est effectué avec 3 paramètres : un objet accessible à distance de S_1 (B), un paramètre sérialisable (4) et un objet accessible à distance de S_3 (manipulé via son proxy pC). Sur S_2, des proxys pour B et C sont nécessaires et automatiquement créés pour pouvoir réaliser l'appel local, et S_2 a maintenant connaissance de 2 nouveaux objets accessibles à distance.
- Utiliser un service de nommage, qui conserve des associations entre objet accessible à distance et *nom externe* (une chaîne de caractères ou une URL) : le client demande au service de nommage de lui fournir un proxy correspondant à un nom externe donné.



Appel de pA.foo(B,4,pC)



Envoi de la requête (A, foo, B, 4, C)



Création du proxy pB

création du proxy pC

appel de A.foo(pB,4,pC)

Le service de nommage est lui-même un service accessible à distance, mais l'on dispose d'un mécanisme spécifique pour le trouver, et permettre d'amorcer la désignation (classes `Naming` et `LocateRegistry`}

Un exemple détaillé

Définition de l'interface d'appel à distance

La première étape consiste à définir l'interface d'accès aux objets appelables à distance, autrement dit à définir la liste des méthodes qui peuvent être appelées. Pour cela, la notion d'interface Java est utilisée. Tous les objets accessibles à distance doivent par ailleurs hériter de l'interface `java.rmi.Remote`. Par exemple, pour une classe d'objets accessibles à distance, on définit l'interface suivante :

```
import java.rmi.RemoteException;
public interface Carnet extends java.rmi.Remote {
 public void enregistrer (String Nom, Adresse a) throws RemoteException;
 public Adresse chercher (String Nom) throws RemoteException, AdresseInconnue;
 public void effacer (String Nom) throws RemoteException;
 public void lister () throws RemoteException;
 public void copier (String Nom, Carnet orig) throws RemoteException;
}

public class AdresseInconnue extends Exception { }

public class Adresse implements java.io.Serializable {
    private String ville;
    private String rue;
    private int    numero;
    public Adresse (String v, String r, int n) {
        ville = v; rue = r; numero = n;
    }
    public String toString () {
        return numero + " " + rue + " à " + ville;
    }
}
```

Par rapport à un appel local de méthode, un appel à distance peut rencontrer de nouveaux cas de défaillances (erreur interne de l'objet appelé, problème de transmission réseau…). Par conséquent, toute méthode accessible à distance peut provoquer l'exception `java.rmi.RemoteException`.

Programmation d'une implantation

Dans une deuxième étape, on fournit une implantation. L'appel de méthode nécessite un protocole point à point dédié à la mise en œuvre des appels. Pour cela, la classe hérite de `java.rmi.UnicastRemoteObject`. Un exemple d'implantation est :

```
import java.rmi.server.UnicastRemoteObject;
import java.rmi.RemoteException;
import java.util.Map;
import java.util.HashMap;
import java.util.Iterator;

public class CarnetImpl extends UnicastRemoteObject implements Carnet {
    private Map contenu = new HashMap();
    public CarnetImpl() throws RemoteException {}
    public void enregistrer (String nom, Adresse a) {
        contenu.put (nom, a);
    }
    public Adresse chercher (String nom) throws AdresseInconnue {
        if (contenu.containsKey (nom))
          return (Adresse) contenu.get (nom);
        else
          throw new AdresseInconnue();
    }
    public void effacer (String nom) {
        contenu.remove (nom);
    }
    public void lister () {
        System.out.println ("Contenu du carnet:");
        Iterator it = contenu.keySet().iterator();
        while (it.hasNext()) {
            String nom = (String) it.next();
         System.out.println (" "+ nom + " : " + (Adresse)contenu.get(nom));
        }
        System.out.println("----");
    }
    public void copier (String nom, Carnet orig) throws RemoteException {
        try {
            Adresse a = orig.chercher (nom);
            this.enregistrer (nom, a);
        } catch (AdresseInconnue e) {
            // pas grave => on ignore
        }
    }
}
```

Génération automatique des talons d'appel

La génération automatique des talons d'appel se fait

- à partir de Java 5 : de manière transparente pour le programmeur, à partir d'un talon d'appel générique, qui génèrera les appels dynamiquement (à l'exécution), en s'appuyant sur les API de réflexivité du langage Java.
- pour les Java antérieurs : grâce à la commande `rmic`. Pour le carnet, on exécute : `rmic CarnetImpl` qui engendre les deux fichiers contenant les talons : `CarnetImpl_Stub.class` et `CarnetImpl_Skel.class`. La commande `rmic` reste disponible à partir de Java 5, pour permettre l'interaction avec des clients produits avec des versions antérieures de Java.

Création d'objets accessibles à distance

On crée ensuite un objet accessible à distance (ou plusieurs) parl'intermédiaire d'un programme Java (hôte du ou des objets). L'objet est créé puis, dans le cas habituel, enregistré dans un serveur de noms (*registry*) apte à recevoir les requêtes de recherche de cet objet. Pour faciliter la gestion de la cohérence de l'annuaire géré par le serveur de noms, les inscriptions, modifications, ou suppressions de références dans l'annuaire doivent être demandées depuis la même machine que le serveur de noms. En revanche, les interrogations de l'annuaire peuvent être effectuées depuis n'importe quel site. Les éventuels processus clients de l'objet peuvent alors obtenir la référence de l'objet via ce serveur de noms.

Deux possibilités existent pour la création d'un serveur de noms :

- soit le serveur de noms est créé explicitement en tant que processus lourd Unix par appel d'une commande `rmiregistry &` ; par défaut, le port d'écoute du serveur est alors 1099 ;
- soit le serveur de noms est intégré au processus lourd gérant l'objet accessible à distance. Dans ce cas, le serveur de noms est implanté par des threads internes au processus lourd. La méthode à utiliser est alors :

```
java.rmi.registry.LocateRegistry.createRegistry(int Port)
```

Le programme suivant crée un carnet avec un serveur de noms intégré :

```
import java.rmi.Naming;
import java.rmi.registry.LocateRegistry;

/*Création d'un serveur de nom intégré et d'1 objet accessible à distance*/
public class ServeurCarnet {
    public static void main (String args[]) {
        try {
            //  Création de l'objet Carnet
            Carnet unCarnet = new CarnetImpl();
            //  Création du serveur de noms
            LocateRegistry.createRegistry(1099);
            //  Enregistrement du carnet dans le serveur local
            Naming.rebind("MonCarnet", unCarnet);
        } catch (Exception e) { System.err.println (e); }
        // Service prêt : attente d'appels
        System.out.println ("Le système est prêt.");
    }
}
```

Après compilation et activation de ce programme, un objet serveur, de nom externe `MonCarnet,` devient accessible, jusqu'à la terminaison du processus lourd le contenant.

Utilisation d'objets accessibles à distance

Pour accéder à un objet accessible à distance, un processus client utilise un objet proxy. Une phase de connexion est nécessaire pour obtenir un proxy via un serveur de noms (méthode `lookup`). L'exemple suivant décrit un client de l'objet de nom externe `MonCarnet` sur la machine dont le nom est fourni en argument du programme :

```
import java.rmi.Naming;
/* Exemple de client utilisant un objet Carnet accessible à distance*/
public class ClientCarnet {
    public static void main (String args[])
       throws java.rmi.NotBoundException, java.net.MalformedURLException,
              java.rmi.RemoteException {
        if (args.length != 2) {
            System.out.println ("java ClientCarnet <site serveur>"
                                + "{enregistrer|chercher|effacer|lister}");
            System.exit(1);
        }
        //  Connexion au serveur de noms (obtention d'un handle)
       Carnet carnet1 = (Carnet) Naming.lookup("//"+args[0]+"/MonCarnet");
        //  Exemples d'utilisation
        if (args[1].equals ("enregistrer")) {
            carnet1.enregistrer ("moi",
                    new Adresse ("Toulouse", "port Saint-Étienne", 32));
            carnet1.enregistrer ("007",
                    new Adresse ("Londres", "Trafalgar Square", 7));
            carnet1.enregistrer ("Han Solo",
                    new Adresse ("Voie Lactée", "Astéroïde à droite", 4));
            carnet1.enregistrer ("Deckard",
                  new Adresse ("Los Angeles", "Bradbury Building", 304));
            carnet1.lister();
        } else if (args[1].equals ("chercher")) {
            String nom = "007";
            try {
                Adresse a = carnet1.chercher (nom);
                System.out.println (nom + " habite au " + a);
            } catch (AdresseInconnue e) {
                System.out.println ("Pas d'adresse pour " + nom);
            }
        } else if (args[1].equals ("effacer")) {
            carnet1.effacer ("007");
        } else if (args[1].equals ("lister")) {
            carnet1.lister ();
        } else {
            System.err.println ("Commande inconnue.");
        }
    }
}
```

Il est à noter que l'obtention d'un proxy pour l'objet peut aussi être réalisée en utilisant explicitement un objet serveur de noms de la classe d'objets accessibles à distance `Registry`, fourni par la méthode `getRegistry` de la classe `LocateRegistry`.

Ainsi, la recherche :

```
Carnet carnet1 = (Carnet) Naming.lookup("//"+args[0]+"/MonCarnet");
```

peut aussi être programmée par la séquence :

```
// Accès_au_serveur_de_noms
Registry ns = LocateRegistry.getRegistry(args[0]);
Carnet carnet1 = (Carnet) ns.lookup("MonCarnet");
```

De même, côté serveur, il est possible de créer le serveur de noms et d'enregistrer l'objet par la séquence :

```
//  Creation_du_serveur_de_noms
Registry ns = LocateRegistry.createRegistry}();
// Enregistrement de l'objet dans le serveur de noms
ns.rebind("MonCarnet", unCarnet);
```

Les opérations sur la classe `Registry` sont les mêmes que les méthodes de la classe `Naming` (voir compléments). Les différences portent sur le fait que, dans le cas de la classe `Registry`, les méthodes s'appliquent à un objet de ce type et que la désignation de l'objet est locale (pas d'URL).

Exercices

1) Tester quelques cas d'erreurs : essai d'utilisation d'un objet non créé, création du serveur d'objet sans serveur de noms, création de plusieurs serveurs de noms sur le même site, etc. Bien noter les types d'exceptions.
2) Développer un couple de processus (P_A,P_B), P_A créant un objet A et P_B créant un objet B. Le processus P_A appelle uneméthode `B.m(…)` et au cours de son exécution, cette méthode rappelle une méthode de A. Envisager deux solutions pour que B connaisse A :
   - chaque processus crée un serveur de noms (attention aux problèmes d'écoute sur le même port) et enregistre son objet ;
   - soit l'objet A donne directement une référenceà l'objet B lors de l'appel de la méthode m.
3) Développer un serveur créant plusieurs objets carnets et utiliser la méthode `java.rmi.registry.Registry.list` pour vérifier le bon enregistrement des objets dans le serveur de noms.
4) Intégrer un objet de rappel aux clients, pour permettre au serveur de prévenir chaque client chaque fois
   - que 10 opérations ont été réalisées sur le carnet.
   - qu'un nouveeau client se connecte ou se déconnecte

Compléments

Distribution d'une application répartie à base de RMI

Les RMI offrent, à l'instar des applets Java, la possibilité de télécharger dynamiquement le code des classes constituant une application. En particulier, il est possible de télécharger le code des talons d'objets accessibles à distance.

L'utilisation de l'option `-Djava.rmi.server.codebase=URL,` au lancement de l'application permet d'indiquer le lieu (répertoire, archive, URL) où sont conservées les définitions des classes provenant de l'application (c.-à-d. utilisées comme paramètres des appels de méthodes), et qui pourront être téléchargées dynamiquement, au besoin, par les destinataires des appels de méthodes.

Il est possible d'étendre le chargement à d'autres sites en spécifiant un gestionnaire de sécurité. Un gestionnaire de sécurité par défaut `(RMISecurityManager)` est proposé pour les RMI, qui offre une politique restrictive (voisine de la politique des applets), mais permet de téléchargement des codes de talons. L'instruction à placer au début du `main` d'une application ayant à télécharger des objets est :

```
System.setSecurityManager(new RMISecurityManager());
```

Ce gestionnaire de sécurité étant restrictif, il sera souvent nécessaire d'ajouter explicitement des droits d'accès supplémentaires (opérations sur des ports réseau, accès aux fichiers). Ces droits d'accès supplémentaires sont définis dans un fichier `xxx.policy`, dont le chemin d'accès est fourni au lancement de la JVM, par l'option `-Djava.security.policy=chemin_d_acces`.

Le contenu du fichier `policy` énumère les droits accordés au processus lancé.

Exemples :

```
grant{
 permission.java.net.SocketPermission "*:1024-", "accept,connect"
 permission.java.io.FilePermission "${/}rep_serv${/}proxys${/}-", "read"
 };

grant codeBase "file:/home/gertrude/java/" {
 permission java.security.AllPermission;
};
```

La plateforme J2EE fournit un outil `policytool`, facilitant la saisie des fichiers `policy`.

Ramasse miettes réparti

La machine Java utilise un ramasse miettes réparti transparent pour le programmeur, basé sur le comptage de références. Pour résoudre les difficultés posées par les cycles de références, il utilise en outre une technique de « péremption » des références : une référence d'objet non utilisée durant plus de 10 minutes est considérée comme récupérable.

Classe java.rmi.Naming

Cette classe fournit l'interface d'accès standard à un serveur de noms. Les noms d'objets doivent obéir à la syntaxe générale des URL : `rmi://host:port/name`.

```
public class Naming extends Object {

  public static Remote lookup(String name)
    throws NotBoundException, MalformedURLException,
           UnknownHostException, RemoteException;

  public static void bind(String name, Remote obj)
    throws AlreadyBoundException, MalformedURLException,
           UnknownHostException, RemoteException;

  public static void rebind(String name, Remote obj)
    throws MalformedURLException, UnknownHostException,
           RemoteException;

  public static void unbind(String name)
    throws NotBoundException, MalformedURLException,
           UnknownHostException, RemoteException;

  public static String [] list(String name)
    throws MalformedURLException, UnknownHostException,
           RemoteException;

}
```

Localisation et accès au service de nommage

La classe `java.rmi.registry.LocateRegistry` offre un ensemble de méthodes pour créer ou obtenir la référence d'un serveur de noms local ou distant.

```
public final class LocateRegistry {

  public static Registry createRegistry(int port) throws RemoteException;

  public static Registry getRegistry() throws RemoteException;

  public static Registry getRegistry(int port) throws RemoteException;

  public static Registry getRegistry(String host)
    throws UnknownHostException, RemoteException;

  public static Registry getRegistry(String host, int port)
    throws UnknownHostException, RemoteException;

}
```