Ingénieurs sup Galilée

Ecole d'Ingénieurs SuP Galilée
Université Paris 13

Spécialité Informatique

Cours Interfaces Graphiques

(en Java avec Swing)

Catherine RECANATI
catherine.recanati@gmail.com

Cours Java (INHM)

MASTER 1

Catherine RECANATI
L.I.P.N. (Laboratoire d'Informatique de Paris Nord)
Université de Paris 13

http://www-lipn.univ-paris13.fr/~recanati

R M I

Remote **M**ethod **I**nvocation
= invocation de méthode distante

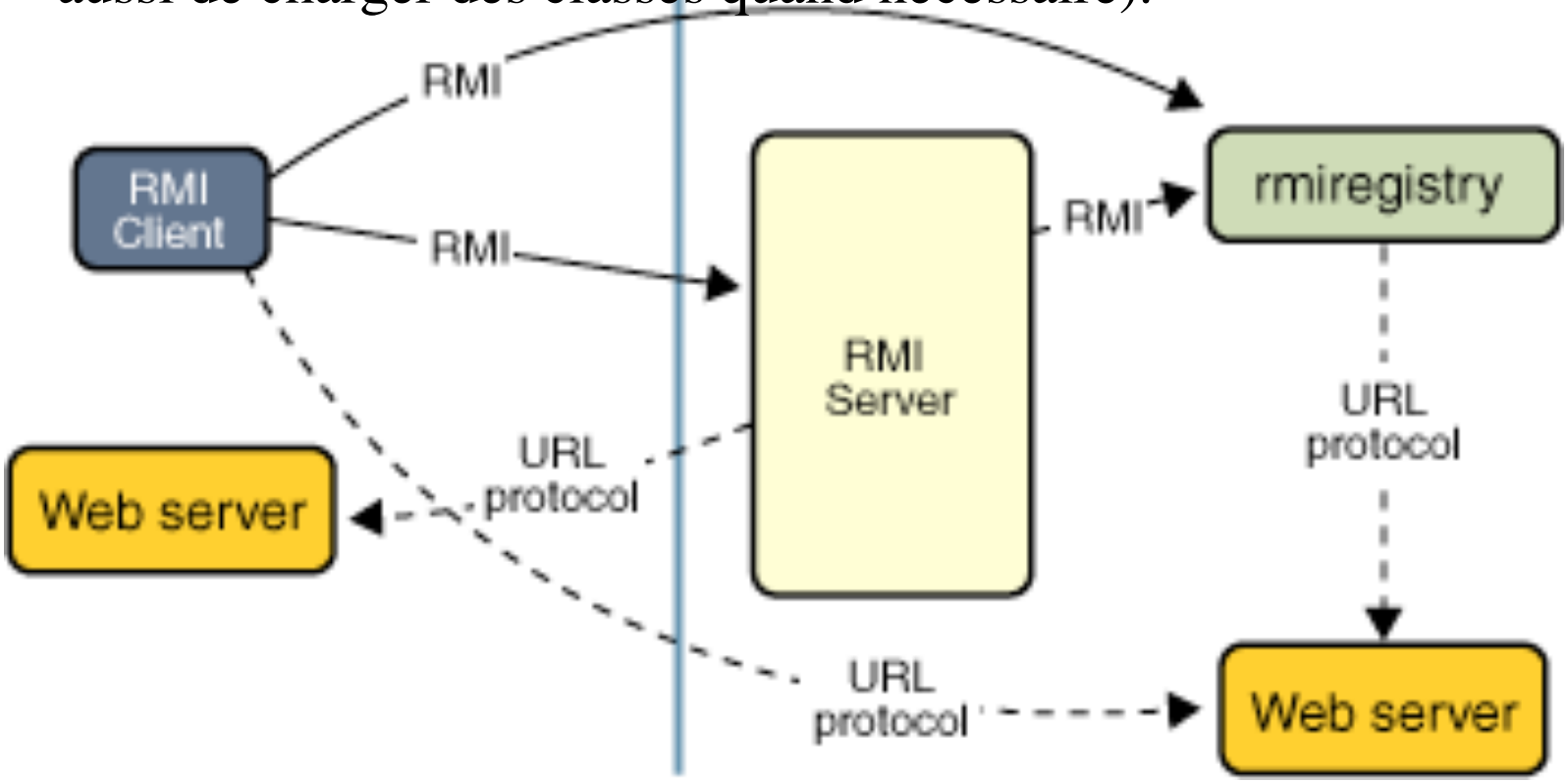
Applications RMI (à objets distribués)

- Usuellement 2 programmes : un client, un serveur
- Serveur = crée des objets distants (remote), les rend accessibles et attend que les clients invoquent des méthodes sur ces objets
- Le système RMI fournit les mécanismes de communication entre le serveur et les clients.
- On parle alors d'application ***à objets distribués***

Applications à objets distribués

- **Localiser les objets distants** : mécanismes divers pour obtenir des références, e.g. enregistrer des noms (RMI registry).
- **Communiquer avec les objets distants**. (Ressemble à des invocations de méthodes java).
- **Charger les définitions des classes des objets qui sont transférés**. Le système RMI fournit des mécanismes pour charger des définitions de classes et transmettre les données des objets.

ex: une application RMI utilisant le registre RMI pour obtenir une référence a un objet distant. Le serveur appelle le registre pour associer un nom à un objet distant. Le client recherche l'objet par son nom, et invoque une méthode sur lui. (Le Web server permet aussi de charger des classes quand nécessaire).



Avantage du chargement dynamique

Un des traits principaux et unique de RMI est la capacité à charger la définition d'une classe d'objet si elle n'est pas définie dans la machine virtuelle Java du receveur.

Ainsi, le comportement des objets est inchangé quand ils sont transférés et cette capacité permet d'introduire de nouveaux types et de nouveaux comportements qui étendent l'application à distance.

Interfaces, objets et méthodes distantes

Les objets avec méthodes invocables à distance dans une machine virtuelle java sont appelés objets distants (*remote objects*).

Un objet devient distant en implémentant une interface distante qui a les caractéristiques :

- `d'étendre l'interface java.rmi.Remote`
- `chaque méthode de l'interface déclare java.rmi.RemoteException dans sa clause throws (en sus des exceptions spécifiques)`

RMI traite différemment un objet distant et un objet non-distant quand l'objet est passé d'une machine virtuelle java à une autre machine virtuelle java. Plutôt que de faire une copie de l'implémentation de l'objet dans la mchine virtuelle java receveuse, RMI passe un morceau (stub) distant pour un objet distant.

Le *stub* agit comme le représentant local, ou procurateur (proxy) pour l'objet distant et est basiquement, pour le client, la référence distante. Le client invoque une méthode sur le stub local, qui est responsable de transporter l'invocation de méthode sur l'objet distant.

Un *stub* pour un objet distant implémente le même ensemble d'interfces distantes que l'objet distant implémente. Cette propriété permet à un stub d'être casté pour n'importe quelle interface que l'objet distant implémente. Cependant, seules les méthodes définies dans une interface distante sont disponibles pour être appelées à partir de la machine virtuelle java receveuse.

Créer des applications distribuées en utilisant le RMI

Cela nécessite 4 étapes :

1. modéliser et implanter les composants de l'application distribuée
2. compiler les sources
3. rendre le réseau de classes accessible
4. démarrer l'application

Modéliser et implanter les composants de l'application

d'abord déterminer l'architecture, les composants locaux, et ceux qui sont accessibles à distances. Cela inclut :

- la définition des interfaces distantes.
- l'implantation des objets distants.
- l'implantation des clients.

Exemple ici: construction d'un moteur de calcul générique

• le moteur de calcul est un objet distant sur le serveur qui effectue des tâches pour des clients (sur la machine où tourne le serveur), et retourne des résultats éventuels.

• les tâches effectuées n'ont pas besoin d'être définies au moment de l'écriture du moteur de calcul. La seule chose est que la classe de la tâche implémente une certaine interface. Le code lui-même peut être téléchargé dynamiquement par le système RMI dans le moteur de calcul.

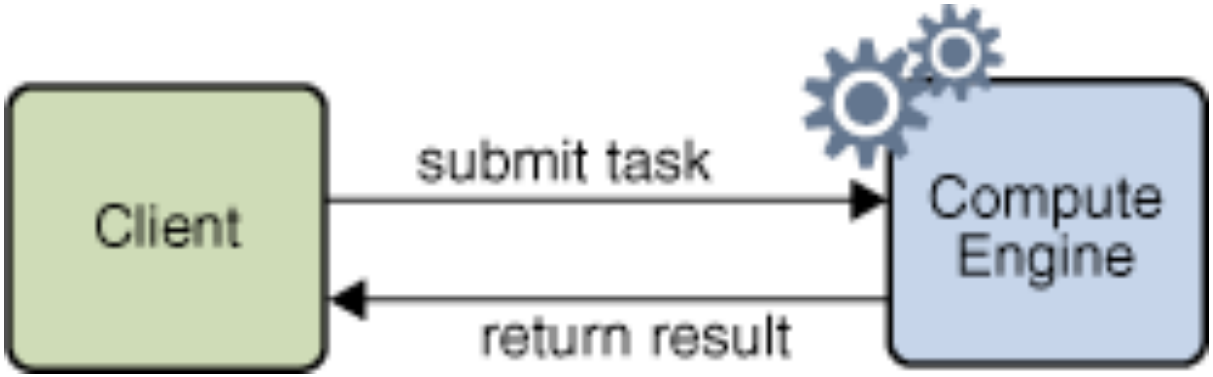
• Cette capacité à exécuter des tâches arbitraires est rendue possible par la nature dynamique de la plateforme java, étendue au réseau par le RMI.

Ecriture d'un serveur RMI

Le serveur du moteur de calcul accepte des tâches des clients, effectue ces tâches, et retourne les résultats. Le code du serveur consiste en une interface et une classe. L'interface définit les méthodes qui peuvent être invoquées par les clients. Principalement, l'interface définit la vue qu'ont les clients de l'objet distant. La classe fournit l'implémentation.

Définir une interface distante

Au coeur du moteur de calcul il y a un protocole qui permet aux tâches d'être soumises au moteur, d'y être exécutées, et que les résultats soient retournés au client. Ce protocole s'exprime dans les interfaces supportées par le moteur de calcul. La communication distante est illustrée par la figure suivante:



Définir une interface distante (suite)

Chaque interface contient une seule méthode. L'interface `Compute` du moteur permet aux tâches d'être soumises au moteur de calcul. L'interface client, `Task`, définit comment le moteur exécute la tâche soumise.

```
package compute;

import java.rmi.Remote;
import java.rmi.RemoteException;

public interface Compute extends Remote {
    <T> T executeTask (Task<T> t) throws
                              RemoteException;
}
```

Définir une interface distante (suite)

La seconde interface requise pour le moteur de calcul est l'interface `Task`, qui est le type du paramètre de la méthode `executeTask` dans l'interface `Compute`. L'interface `compute.Task` définit l'interface entre le moteur de calcul et le travail qu'il doit faire, en fournissant la manière de commencer le travail. Voici le code source pour l'interface `Task` :

```
package compute;

    public interface Task<T> {
        T execute();
    }
```

RMI utilise le mécanisme de sérialisation d'objets pour transporter les objets par valeur entre machine virtuelles java. Pour qu'un objet soit sérialisable, sa classe doit implémenter l'interface `java.io.Serializable`. Ainsi, les classes qui implémentent l'interface `Task` doivent également implémenter `Serializable`, comme le doivent les classes d'objets utilisés comme résultats des tâches.

Différentes sortes de tâches peuvent être lancées par un objet `Compute` tant qu'elles sont des implémentations du type `Task`. Les classes qui implémentent cette interface peuvent contenir d'autres données ou d'autres méthodes requises.

C’est ici que le RMI rend ce simple moteur de calcul générique possible.

Du fait que le RMI peut supposer que les objets `Task` sont écrits en langage java, les implémentations d’un objet `Task` qui sont précédemment inconnues du moteur de calcul sont téléchargées par RMI dans la machine virtuelle java du moteur de calcul quand c’est nécessaire.

Cette capacité permet aux clients du moteur de calcul de définir de nouvelles sorte de tâches pouvant être lancées sur la machine du serveur sans que le code ait explicitement été installé sur cette machine.

Le moteur de calcul, implémenté par la classe `ComputeEngine`, implémente l'interface `Compute`, qui permet à différentes tâches de lui être soumises par des appels à sa méthode `executeTask`.

Ces tâches sont lancées en utilisant l'implémentation de la tâche de la méthode `execute`, et les résultats sont retournés au client distant.

Implanter une interface distante

On va maintenant voir comment implanter une classe pour le moteur de calcul. En général, une classe qui implémente une interface distante doit au moins faire les choses suivantes :

- Déclarer les interfaces distantes qui sont implémentées
- Définir le constructeur de chaque objet distant
- Fournir une implémentation pour chaque méthode distante dans les interfaces distantes

Un programme serveur RMI doit créer les objets distants initiaux et les exporter au RMI à runtime, qui les rend disponible pour recevoir les invocations distantes qui arrivent. Cette procédure de setup peut soit être encapsulée dans une méthode de l'implémentation de classe distante elle-même, ou incluse entièrement dans une autre classe. La procédure de setup doit faire ce qui suit :

* Créer et installer un gestionnaire de sécurité (security manager)

* Créer et exporter un ou plusieurs objets distants

* Enregistrer au moins un objet distant avec le registre RMI (ou avec un autre service de noms, comme un service accessible à travers l'interface Java Naming and Directory) à des fins de bootstrapping

Voici l'implémentation complète du moteur de calcul. La classe `engine.ComputeEngine` implémente l'interface distante `Compute` et inclut également la méthode `main` pour installer le moteur de calcul :

```
package engine;

import java.rmi.RemoteException;
import java.rmi.registry.LocateRegistry;
import java.rmi.registry.Registry;
import java.rmi.server.UnicastRemoteObject;
import compute.Compute;
import compute.Task;
```

```
public class ComputeEngine implements Compute {

    public ComputeEngine() {
        super();
    }

    public <T> T executeTask(Task<T> t) {
        return t.execute();
    }

    public static void main(String[] args) {
        if (System.getSecurityManager() == null) {
            System.setSecurityManager(new
                                SecurityManager());
        }
```

```
        try {
            String name = "Compute";
            Compute engine = new ComputeEngine();
            Compute stub = (Compute)
UnicastRemoteObject.exportObject(engine, 0);
            Registry registry =
                    LocateRegistry.getRegistry();
            registry.rebind(name, stub);
            System.out.println("ComputeEngine
bound");
        } catch (Exception e) {
            System.err.println("ComputeEngine
exception:");
            e.printStackTrace();
        }
    }
}
```

```
public class ComputeEngine implements Compute {
```

Cette déclaration dit que la classe implémente l'interface distante `Compute` et par conséquent peut être utilisée par un objet distant.

La classe `ComputeEngine` définit une implémentation de classe d'objet distant qui implémente une seule interface distante et pas d'autres interfaces. La classe `ComputeEngine` contient également deux éléments de programme exécutables qui ne peuvent être invoqués que localement. Le premier de ces éléments est un constructeur pour les instances de `ComputeEngine`. Le second de ces éléments est une méthode `main` qui est utilisée pour créer une instance de `ComputeEngine` disponible aux clients.

```
public class ComputeEngine implements Compute {

    public ComputeEngine() {
        super();
    }
```

Ce constructeur de l'objet distant ne prend pas d'argument. Il invoque seulement le constructeur de la superclasse, qui est le constructeur sans argument de la classe Objet. Bien que le constructeur de la superclasse soit invoqué même s'il est omis du constructeur de `ComputeEngine`, il est ajouté ici pour la clarté.

La classe d'un objet distant doit fournir les implémentations de chaque méthode distante spécifiée dans les interfaces distantes. L'interface `Compute` contient une seule méthode distante, `executeTask`, qui est implémenté comme suit :

```
public <T> T executeTask(Task<T> t) {
    return t.execute();
}
```

Cette méthode implémente le protocole entre l'objet distant `ComputeEngine` et ses clients. Chaque client fournit à `ComputeEngine` un objet `Task` qui a une implémentation particulière de la méthode `execute` de l'interface `Task`. `ComputeEngine` exécute chaque tâche de client et retourne le résultat de la méthode `execute` de la tâche directement au client.

Passation d'objets en RMI

Les arguments et valeur de retour des méthodes distantes peuvent être de presque n'importe quel type, incluant des objets locaux, distants, ou des types de données primitifs. Plus précisément, il suffit que ces entités soient des instances de type primitifs, d'objets distants ou d'objets sérialisables, i.e. qui implémentent l'interface `java.io.Serializable`.

Certains types n'ont cependant pas ces propriétés et ne peuvent donc pas être passés à une méthode distante. La plupart de ces objets, comme les threads ou les descripteurs de fichiers, encapsulent des informations qui n'ont de sens qu'à l'intérieur de leur espace d'adressage.

Passation d'objets en RMI

Les règles qui déterminent comment les arguments et les valeurs de retour sont passées sont les suivantes :

* Les objets distants sont principalement passés par référence. Une référence d'objet distant est un bout de code (*stub*), qui est une procuration (*proxy*) côté client implémentant l'ensemble complet des interfaces distantes que l'objet distant implémente.

* Les objets locaux sont passés par copie, en utilisant la sérialisation d'objet. Par défaut, tous les champs sont copiés exceptés ceux qui sont `static` ou `transient`.

Passation d'objets en RMI

Passer un objet distant par référence signifie que tous les changements effectués sur l'état de l'objet par invocations de méthodes distantes seront reflétés dans l'objet distant d'origine. Lors du passage d'un objet distant, seules les interfaces qui sont des interfaces distantes sont disponibles au receveur. Toute méthode définie dans l'implémentation de la classe ou définie dans des interfaces non-distantes implémentées par la classe n'est pas disponible au receveur.

Par exemple, si vous voulez passer une référence à une instance de la classe `ComputeEngine`, le receveur n'aura accès qu'à la méthode `executeTask` du moteur de calcul.

Passation d'objets en RMI

Ce receveur ne verra pas le constructeur de `ComputeEngine`, ni sa méthode `main`, ni son implémentation d'aucune méthode de `java.lang.Object`.

Dans les paramètres et valeurs de retour d'invocations de méthodes distantes, les objets qui ne sont pas distants sont passés par valeur. Ainsi, une copie de l'objet est créée dans la machine virtuelle Java receveuse. Tout changement d'état de l'objet par le receveur n'est reflété que dans la copie du receveur, pas dans l'instance originale distante de l'envoyeur. Inversement, tout changement de l'état de l'objet par l'envoyeur n'est reflété que dans l'instance originale de l'envoyeur, et pas dans la copie du receveur.

Implémenter le `main` du serveur

La méthode la plus complexe de l'implémentation de `ComputeEngine` est sa méthode `main`. La méthode main est utilisée pour démarrer le moteur de calcul et par conséquent doit faire les initialisations et le ménage nécessaire pour préparer le serveur à accepter les appels des clients. Cette méthode n'est pas une méthode distante, ce qui signifie qu'elle ne peut pas être invoquée depuis une autre machine virtuelle java. Du fait qu'elle est déclarée `static`, cette méthode n'est pas associée à un objet mais plutôt avec la classe `ComputeEngine`.

```
 public static void main(String[] args) {
    if (System.getSecurityManager() == null) {
        System.setSecurityManager(new SecurityManager());
    }
    try {
        String name = "Compute";
        Compute engine = new ComputeEngine();
        Compute stub = (Compute)
            UnicastRemoteObject.exportObject(engine, 0);
        Registry registry = LocateRegistry.getRegistry();
        registry.rebind(name, stub);
        System.out.println("ComputeEngine bound");
    } catch (Exception e) {
        System.err.println("ComputeEngine exception:");
        e.printStackTrace();
    }
}
```

La première tâche de la méthode main est de créer et installer un gestionnaire de sécurité (*security manager*), qui protège les accès aux ressources système des intrusions de code téléchargé tournant sur la machine virtuelle Java. Un gestionnaire de sécurité détermine si le code téléchargé a accès au système de fichiers local ou s'il peut effectuer certaines opérations privilégiées.

Si un programme RMI n'installe pas de *security manager*, RMI ne télécharge pas de classes (autres que du chemin local de classes) pour des objets reçus en arguments ou valeurs de retour d'invocations de méthodes distantes. Cette restriction assure que les opérations effectuées par le code téléchargé soient sujettes à une politique de sécurité.

Ensuite, la méthode `main` crée une instance de `ComputeEngine` et l'exporte au runtime RMI par les instructions suivantes :

```
Compute engine = new ComputeEngine();
Compute stub = (Compute)
  UnicastRemoteObject.exportObject(engine, 0);
```

La méthode `static` `UnicastRemoteObject.exportObject` exporte l'objet distant fourni de sorte qu'il puisse recevoir les invocations de ses méthodes distantes provenant de clients distants. Le second argument, un `int`, spécifie quel port TCP utiliser pour écouter les requêtes d'invocations distantes entrantes pour l'objet. Il est commun d'utiliser la valeur zéro, qui spécifie l'utilisation d'un port anonyme. Le port réel sera alors choisi à l'exécution par RMI ou l'operating system.

Une fois que l'invocation `exportObject` termine avec succès, l'objet distant `ComputeEngine` est prêt à processer les invocations distantes entrantes.

La méthode `exportObject` retourne un « morceau », (`stub`) de l'objet distant exporté. Notez que le type de la variable `stub` doit être `Compute`, et pas `ComputeEngine`, parce que le bout (`stub`) d'un objet distant implémente seulement les interfaces distantes que l'objet distant exporté implémente.

La méthode `exportObject` déclare qu'elle peut lancer une `RemoteException`, qui est un type d'exception vérifiée.

La méthode `main` traite cette exception avec son bloc `try/catch`. Si l'exception n'était pas traitée de cette manière, `RemoteException` aurait du être déclarée dans la clause `throws` de la méthode `main`. Une tentative d'exporter un objet distant peut générer une `RemoteException` si les ressources nécessaires de communication ne sont pas disponibles, comme par exemple si le port requis est lié pour un autre usage.

Avant qu'un client puisse invoquer une méthode sur un objet distant, il doit d'abord obtenir une référence à cet objet. Elle s'obtient de la même manière que n'importe quelle référence à un autre objet est obtenue dans un programme, par exemple en récupérant la référence comme partie d'une valeur de retour de méthode ou de structure de donnée qui contienne cette référence.

Le système fournit un type particulier d'objet distant, le registre RMI (RMI registry), pour trouver des références à d'autres objets distants. Le registre RMI est un simple service de nommage d'objets distants qui permet aux clients d'obtenir une référence à un objet distant à partir d'un nom.

Le registre est typiquement utilisé uniquement pour localiser le premier objet distant qu'un client RMI a besoin d'utiliser. Ce premier objet distant peut alors fournir le support pour trouver d'autres objets.

L'interface distante `java.rmi.registry.Registry` est l'API pour lier (ou enregistrer) et rechercher des objets distants dans le registre. La classe `java.rmi.registry.LocateRegistry` founit des méthodes statiques pour synthétiser une référence distante à un registre, à une adresse réseau particulière (hôte et port). Ces méthodes créent l'objet référence distant contenant l'adresse de réseau spécifié sans effectuer aucune communication à distance.

LocateRegistry fournit aussi des méthodes statiques pour créer un nouveau registre dans la machine virtuelle Java courante, bien que l'exemple traité ici n'utilise pas ces méthodes. Une fois qu'un objet distant est enregistré sur un registre RMI sur l'hôte local, des clients de n'importe quel hôte peuvent rechercher cet objet distant par son nom, obtenir sa référence, et ensuite invoquer des méthodes distantes sur l'objet. Le registre peut être partagé par tous les serveurs tournant sur un hôte, ou un processus serveur individuel peut créer et utiliser son propre registre.

La classe ComputeEngine crée un nom pour l'objet avec l'instruction suivante :

```
String name = "Compute";
```

Le code ajoute ensuite le nom au registre RMI qui tourne sur le serveur. Cette étape est effectuée plus tard avec les instructions suivantes :

```
Registry registry =
            LocateRegistry.getRegistry();
registry.rebind(name, stub);
```

Cette invocation de rebind effectue un appel distant au registre RMI de l'hôte local. Comme tout appel distant, cet appel peut résulter en l'envoi d'une `RemoteException`, qui est is traitée par le `catch` à la fin de la méthode `main`.

3 points à noter à propos de l'appel à `Registry.rebind` :

1- La surcharge sans argument de `LocateRegistry.getRegistry` synthétise une référence à un registre de l'hôte local et sur le port de registre par défaut, 1099. Vous devez utiliser une surcharge qui a un paramètre `int` si le registre est créé sur un autre port.

2- Quand une invocation à distance est effectuée sur le registre, un *stub* pour l'objet est passé au lieu de la copie de l'objet distant lui-même. Les objets implémentés à distance, comme les instances de `ComputeEngine`, ne quittent jamais la machine java dans laquelle ils ont été créés. Ainsi, quand un client fait une recherche dans le registre d'objets distants d'un serveur, une copie du `stub` est retournée.

Les objets distant dans de tels cas sont donc effectivement passés par référence (distante) plutôt que par valeur.

3- Pour des raisons de sécurité, une application peut seulement lier, délier, ou lier à nouveau des références à des objets distants avec un registre tournant sur le même hôte qu'elle. Cette restriction prémunit un client distant contre les suppressions ou écrasements d'entrées dans le registre d'un serveur. Une recherche, cependant, peut être demandée à partir de n'importe quel hôte, local ou distant.

```
 public static void main(String[] args) {
    if (System.getSecurityManager() == null) {
        System.setSecurityManager(new SecurityManager());
    }
    try {
        String name = "Compute";
        Compute engine = new ComputeEngine();
        Compute stub = (Compute)
            UnicastRemoteObject.exportObject(engine, 0);
        Registry registry = LocateRegistry.getRegistry();
        registry.rebind(name, stub);
        System.out.println("ComputeEngine bound");
    } catch (Exception e) {
        System.err.println("ComputeEngine exception:");
        e.printStackTrace();
    }
}
```

Une fois que le serveur s'est enregistré sur le registre local RMI, il imprime un message. Ensuite, la méthode `main` termine. Mais il n'est pas nécessaire d'avoir un thread attendant et maintenant le serveur en vie. Tant qu'il y a une référence à l'objet `ComputeEngine` dans une autre machine virtuelle Java, locale ou distante, l'objet `ComputeEngine` ne sera pas arrêté, ni sa mémoire récupérée. Du fait que le programme a lié une référence au `ComputeEngine` dans le registre, il est atteignable par un client distant, le registre lui-même. Le système RMI laisse le processus `ComputeEngine` tourner. Il est disponible pour accepter des appels et ne sera pas récupéré avant que son lien ne soit retiré du registre et qu'aucun client n'ait une référence distante à l'objet `ComputeEngine`.

La fin du code de la méthode `ComputeEngine.main` traite toute exception qui peut arriver. La seule exception de type vérifié qui peut être lancée dans le code est `RemoteException`, soit par l'invocation de `UnicastRemoteObject.exportObject` invocation ou par l'invocation de `rebind` dans le registre. Dans les deux cas, le programme ne peut pas faire beaucoup mieux que exit après avoir imprimer un message d'erreur.

Dans certaines applications distribuées, repartir de l'erreur pour faire une invocation à distance est possible. Par exemple, l'application peut tenter de réessayer l'opération ou choisir un autre serveur pour continuer l'opération.

Créer un programme Client

Le moteur de calcul est un programme relativement simple: il tourne des tâches qui lui sont apportées. Les clients sont plus complexes; un client doit appeler le moteur de calcul et doit aussi définir la tâche à effectuer par le moteur de calcul.

Le client de notre exemple sera fait de deux classes séparées. La première, `ComputePi`, recherche et invoque un objet `Compute`. La seconde, `Pi`, implémente l'interface `Task` et définit le travail à faire par le moteur de calcul. Le job de la classe `Pi` est de calculer la valeur d'un certain nombre de décimales de π.

L'interface non-distante `Task` est définie comme suit :

```
package compute;

    public interface Task<T> {
        T execute();
    }
```

Le code qui invoque des méthodes d'un objet `Compute` doit obtenir une référence à cet objet, créer un objet `Task`, et ensuite demander que la tâche soit exécutée. La définition de la classe tâche `Pi` est montrée plus loin. Un objet `Pi` est construit avec un seul argument, la précision désirée du résultat. Le résultat de l'exécution de la tâche est un `java.math.BigDecimal` représentant π calculé à la précision spécifiée.

// Code de la classe client.ComputePi

```
package client;

import java.rmi.registry.LocateRegistry;
import java.rmi.registry.Registry;
import java.math.BigDecimal;
import compute.Compute;

public class ComputePi {
    public static void main(String args[]) {
        if (System.getSecurityManager() == null) {
            System.setSecurityManager(new
                                SecurityManager());
        }
        try {
```

Comme le serveur `ComputeEngine`, le client commence par installer un *security manager*. Cette étape est nécessaire parce que le processus de recevoir le morceau d'objet (stub) du serveur distant peut nécessiter de télécharger des définitions de classes à partir du serveur. Pour que RMI télécharge des classes, un *security manager* doit être présent.

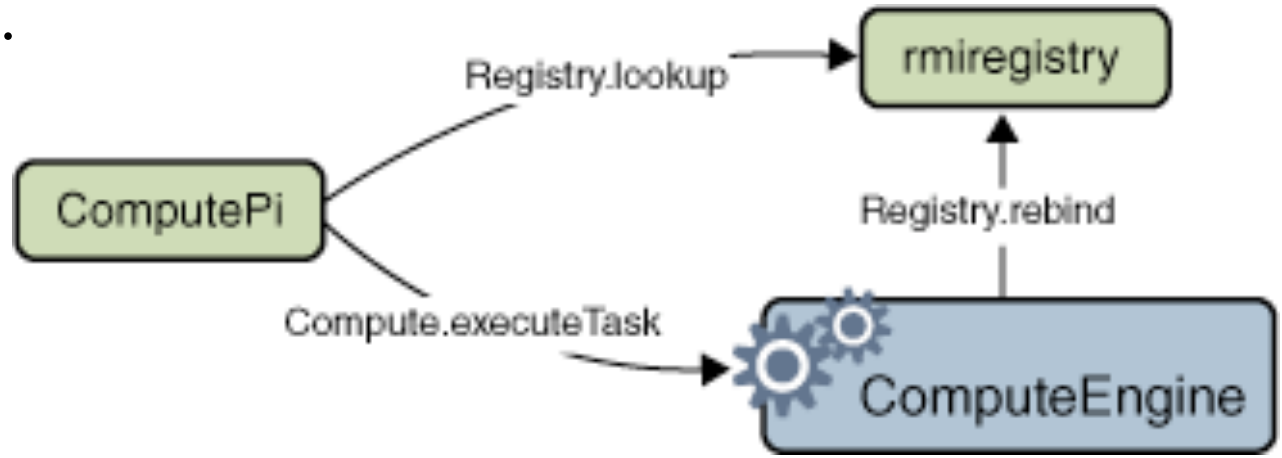
Après l'installation du *security manager*, le client crée un nom à utiliser pour rechercher un objet `Compute` distant, en utilisant le même nom que celui utilisé par `ComputeEngine` pour lier cet objet distant. Le client utilise l'API `LocateRegistry.getRegistry` pour synthétiser une référence distante au registre de l'hôte du serveur.

```
        try {
             String name = "Compute";
             Registry registry =
                LocateRegistry.getRegistry(args[0]);
             Compute comp = (Compute)
                          registry.lookup(name);
             Pi task = new Pi
                     (Integer.parseInt(args[1]));
             BigDecimal pi = comp.executeTask(task);
             System.out.println(pi);
         } catch (Exception e) {
             System.err.println("ComputePi
                                  exception:");
             e.printStackTrace();
         }
    }
}
```

La valeur du premier argument de la ligne de commande, `args[0]`, est le nom de l'hôte distant sur lequel l'objet `Compute` tourne. Le client invoque alors la méthode de recherche dans le registre pour rechercher l'objet distant par nom dans le registre de l'hôte du serveur. La surcharge particulière de `LocateRegistry.getRegistry` utilisée, qui prend un unique paramètre `String`, retourne une référence à un registre de l'hôte nommé, sur le port de registre par défaut 1099.

Il faut utiliser une autre surcharge avec paramètre `int` si le registre est créé sur un port autre que le port 1099.

Ensuite, le client crée un nouvel objet `Pi`, en passant au constructeur `Pi` la valeur du second argument de la ligne de commande, `args[1],` parsé comme entier. Cet argument indique le nombre décimales à utiliser dans le calcul. Finalement, le client invoque la méthode `executeTask` de l'objet distant `Compute`. L'objet passé à l'invocation de `executeTask` retourne un objet de type `BigDecimal`, que le programme stocke dans la variable `pi` résultat.



La classe Pi implémente l'interface Task et calcule la valeur de π jusqu'à un certain nombre spécifié de décimales. L'algorithme ici n'est pas important pour l'exemple. Ce qui compte est qu'il soit coûteux, et qu'on préfère donc qu'il soit exécuté sur un serveur performant.

```
package client;

    import compute.Task;
    import java.io.Serializable;
    import java.math.BigDecimal;

    public class Pi implements Task<BigDecimal>,
                                        Serializable {

  private static final long serialVersionUID = 227L;
```

Notez que toutes les classes sérialisables doivent déclarer un champ `private static final` nommé `serialVersionUID` pour grantir la compatibilité de sérialisations entre versions. Si aucune version antérieure n'a été dispensée, alors la valeur de ce champ peut être n'importe quel `long`, comme le `227L` utilisé ici par `Pi`, tant que cette valeur est utilisée de manière consistante dans les versions futures.

```
/** constants used in pi computation */
        private static final BigDecimal FOUR =
            BigDecimal.valueOf(4);

/** rounding mode to use during pi computation */
        private static final int roundingMode =
            BigDecimal.ROUND_HALF_EVEN;

/** digits of precision after the decimal point */
        private final int digits;
/**
 * Construct a task to calculate pi to the specified
 * precision */
        public Pi(int digits) {
            this.digits = digits;
        }
```

```
/**
  * Calculate pi.
  */
        public BigDecimal execute() {
            return computePi(digits);
        }
 /**
     * Compute the value of pi to the specified number of
     * digits after the decimal point.  The value is
     * computed using Machin's formula:
     *
     *          pi/4 = 4*arctan(1/5) - arctan(1/239)
     *
     * and a power series expansion of arctan(x) to
     * sufficient precision.
     */
```

```
public static BigDecimal computePi(int digits) {
        int scale = digits + 5;
        BigDecimal arctan1_5 = arctan(5, scale);
        BigDecimal arctan1_239 = arctan(239, scale);
        BigDecimal pi = arctan1_5.multiply(FOUR).subtract(
                                    arctan1_239).multiply(FOUR);
        return pi.setScale(digits,
                           BigDecimal.ROUND_HALF_UP);
    }
/**
     * Compute the value, in radians, of the arctangent of
     * the inverse of the supplied integer to the specified
     * number of digits after the decimal point.  The value
     * is computed using the power series expansion for the
     * arc tangent:
     *
     * arctan(x) = x - (x^3)/3 + (x^5)/5 - (x^7)/7 +
     *     (x^9)/9 ...
     */
```

```
public static BigDecimal arctan(int inverseX,
                                int scale)
   {
       BigDecimal result, numer, term;
       BigDecimal invX = BigDecimal.valueOf(inverseX);
       BigDecimal invX2 =
           BigDecimal.valueOf(inverseX * inverseX);

       numer = BigDecimal.ONE.divide(invX,
                                     scale, roundingMode);

       result = numer;
       int i = 1;
       do {
           numer =
               numer.divide(invX2, scale, roundingMode);
           int denom = 2 * i + 1;
           term =
               numer.divide(BigDecimal.valueOf(denom),
                            scale, roundingMode);
```

```
            if ((i % 2) != 0) {
                result = result.subtract(term);
            } else {
                result = result.add(term);
            }
            i++;
        } while (term.compareTo(BigDecimal.ZERO) != 0);
        return result;
    }
}
```

Le plus intéressant dans cet exemple est que l'implémentation de l'objet `Compute` n'a jamais besoin de la définition de la classe `Pi` jusqu'à ce qu'un objet `Pi` soit passé en tant qu'argument d'une méthode `executeTask`. A ce moment, le code de la classe est chargé par RMI dans la machine virtuelle Java de l'objet `Compute`, la méthode `execute` est invoquée, et le code de la tâche est exécuté. Le résultat, qui dans le cas de la tâche `Pi` est un objet `BigDecimal`, est retourné au client appelant, dans lequel il est utilisé pour imprimer le résultat du calcul.

Le fait que l'objet `Task` donné calcule la valeur de `Pi` n'est pas pertinent pour l'objet `ComputeEngine`. On pouvait aussi bien implémenter une tâche qui, par exemple, génèrerait un nombre premier aléatoire en utilisant un algorithme probabiliste. Cette tâche serait aussi coûteuse sur un plan calculatoire et donc un bon candidat pour être passé au moteur de calcul `ComputeEngine`, mais il aurait eu un code très différent. Ce code aurait aussi pu être téléchargé au moment où l'objet `Task` est passé à un objet `Compute`.

De la même manière que l'algorithme pour calculer est apporté quand il est nécessaire, le code qui génère un nombre premier aléatoire serait apporté quand nécessaire. L'objet `Compute` sait simplement que chaque objet qu'il reçoit implémente la méthode execute. L'objet `Compute` ne sait pas, et n'a pas besoin de savoir, ce que l'implémentation fait.

Compiler les programmes de l'exemple

Cet exemple a séparé l'implémentation des interfaces et de l'objet distant, et le code client en trois packages:

* `compute` – interfaces `Compute` et `Task`
* `engine` – classe `ComputeEngine`
* `client` – code client `ComputePi` et implémentation de la tâche `Pi`

Créer le fichier jar des interfaces

En supposant que les sources soient dans le répertoire de l'utilisateur cathy /home/cathy/src, on créé un fichier jar avec :

```
cd /home/cathy/src
javac compute/Compute.java compute/Task.java
jar cvf compute.jar compute/*.class
```

On peut alors distribuer le fichier `compute.jar` aux développeurs de serveurs ou d'applications client pour qu'ils puissent utiliser les interfaces.

Après avoir construit les classes côté client ou serveur avec le compilateur javac, si ces dernières peuvent avoir besoin d'être téléchargées dynamiquement par d'autre machine virtuelles java, ils faut s'assurer que leurs fichiers class soient placés dans un endroit accessible par le réseau.

Ici, pour Solaris OS or Linux cet emplacement est `/home/user/public_html/classes` parce que beaucoup de serveurs web autorisent l'accès d'un répertoire d'utilisateur public_html à travers une HTTP URL construite par http://host/~user/.

Construire les classes du serveur

Le package `engine` ne contient qu'une classe implémentée côté serveur, `ComputeEngine`, implémentant l'interface distante `Compute`. Si l'utilisateur Anne a les sources dans `/home/anne/src/engine`, et qu'elle a rendu accessible le fichier `compute.jar` dans `/home/anne/public_html/classes`, pour compiler elle fera

```
cd /home/ann/src
javac -cp
  /home/anne/public_html/classes/compute.jar
     engine/ComputeEngine.java
```

Construire les classes du client

Si l'utilisateur Jean a les sources de `ComputePi` et `Pi` dans `/home/jean/src/client`, et qu'il a rendu accessible le fichier `compute.jar` dans le répertoire `/home/jean/public_html/classes`, il les compilera par :

```
cd /home/jones/src
javac -cp
   /home/jean/public_html/classes/compute.jar
   client/ComputePi.java client/Pi.java
mkdir /home/jean/public_html/classes/client
cp client/Pi.class
     /home/jean/public_html/classes/client
```

Faire tourner les programmes

A propos de la sécurité

Les programmes serveur et client tournent avec un *security manager*. Au lancement de ces programmes, vous devez spécifier un fichier de politique de sécurité de sorte que le code soit en accord avec les permissions nécessaires pour tourner. Voici un exemple de fichier de politique de sécurité pour le programme serveur :

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

Ici, un exemple de fichier de politique à utiliser avec le programme client :

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

Pour ces deux fichiers, toutes les permissions sont accordées pour les classes provenant du chemin des sources (elles sont sûres), mais aucune pour un autre accès.

Dans la suite, on supposera que ces fichiers s'intitule respectivement `server.policy` et `client.policy`.

Lancer le serveur

Avant de lancer le serveur, il faut lancer le registre RMI à l'aide de la commande `rmiregistry` :

```
rmiregistry &
```

Prendre garde à ce que le shell dans lequel on le lance n'ait pas de variable d'environnement CLASSPATH d'affectée (ou que ce chemin ne contienne pas le chemin aux classes qu'on souhaiterait télécharger depuis des objets distants).

Par défaut, le registre tourne sur le port 1099. Pour le lancer sur un port différent, spécifier le numéro de port sur la ligne de commande :

```
rmiregistry 2001 &
```

Une fois que le registre est lancé, vous pouvez démarrer le serveur. Il faut alors s'assurer qu'à la fois le fichier `compute.jar` et la classe d'implémentation de l'objet distant soient dans votre chemin d'accès aux classes. Quand vous démarrez le moteur de calcul, vous devez spécifier, au moyen de la propriété `java.rmi.server.codebase`, où les classes du serveur sont accessibles sur le réseau.

Dans cet exemple, les classes côté serveur à rendre accessibles pour téléchargement sont les interfaces `Compute` et `Task`, qui sont disponibles dans le fichier `compute.jar` du répertoire `public_html/classes` de l'utilisateur anne. Le serveur du moteur de calcul est lancé sur l'hôte `zu`, le même hôte sur lequel le registre a été lancé :

```
java -cp
      /home/anne/src:/home/anne/public_html/classes/compute.jar
 -Djava.rmi.server.codebase=http://zu/~anne/classes/compute.jar
     -Djava.rmi.server.hostname=zu.east.sun.com
     -Djava.security.policy=server.policy
        engine.ComputeEngine
```

La commande java précédente définit les propriétés système suivantes :

1- La propriété `java.rmi.server.codebase` spécifie l'emplacement, un codebase URL, à partir duquel les définitions des classes provenant de ce serveur peuvent être téléchargées. Si le codebase spécifie une hiérarchie de répertoires (par opposition à un fichier JAR), il faut inclure un slash séparateur à la fin du codebase URL.

2- La propriété `java.rmi.server.hostname` spécifie le nom de l'hôte ou l'adresse où mettre les morceaux (stubs) pour les objets distants exportés dans cette machine virtuelle. Cette valeur est le nom de l'hôte ou l'adresse utilisée par les clients quand ils font des invocations de méthodes distantes.

Par défaut, l'implémentation RMI utilise l'adresse IP du serveur comme l'indique l'API `java.net.InetAddress.getLocalHost`. Cependant, parfois, cette adresse n'est pas appropriée pour tous les clients et un nom d'hôte entièrement qualifié sera plus effectif. Pour assurer que RMI utilise un nom d'hôte (ou adresse IP) pour le serveur qui soit un itinéraire pour tous les clients potentiels, il faut affecter la propriété java.rmi.server.hostname property.

3- La propriété `java.security.policy` est utilisée pour spécifier le fichier de politique qui contient les permissions que vous entendez accorder.

Lancer le client

Une fois le registre et le moteur de calcul lancé, on peut lancer le client en spécifiant :

* L'emplacement où le client sert ses classes (la classe `Pi`) en utilisant la propriété `java.rmi.server.codebase`

* La propriété `java.security.policy`, qui est utilisée pour spécifier le fichier de politique de sécurité contenant les autorisations données aux différents morceaux de code

* En arguments de ligne de commande, le nom de l'hôte du serveur (pour que le client sache où localiser l'objet distant Compute) et le nombre de décimales à utiliser dans le calcul

On lance le client sur un autre hôte (par exemple ford):

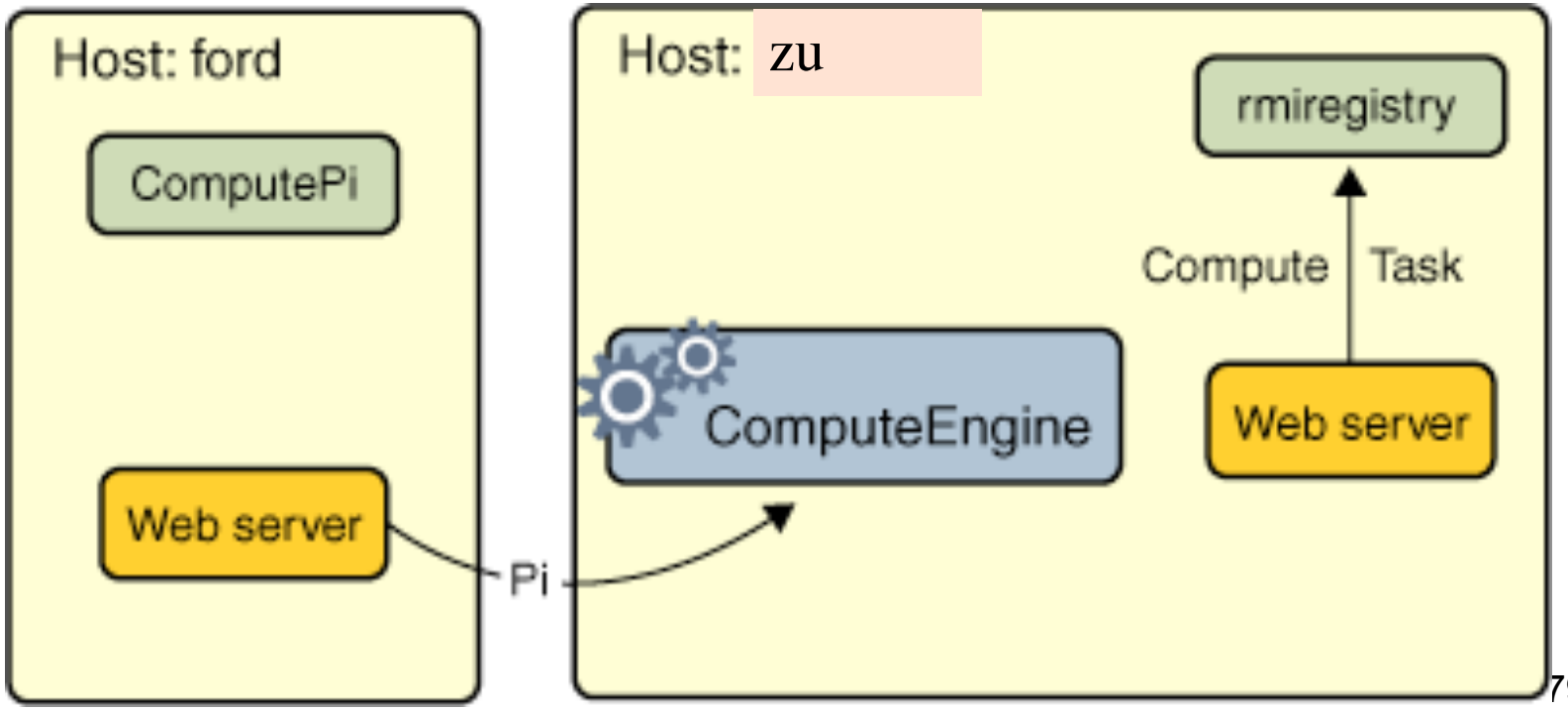
```
java -cp
/home/jean/src:/home/jean/public_html/classes/compute.jar
     -Djava.rmi.server.codebase=http://ford/~jean/classes/
     -Djava.security.policy=client.policy
        client.ComputePi zu.east.sun.com 45
```

Notez que le chemin d'accès aux classes est donné sur la ligne de commande. Notez aussi que la valeur de la propriété java.rmi.server.codebase, qui spécifie une hiérarchie de répertoires, finit par un slash.

Après avoir lancé le client, la sortie suivante est affichée:

3.14159265358979323846264338327950288419716939937

La figure suivante illustre où le registre RMI `rmiregistry`, le serveur `ComputeEngine`, et le client `ComputePi` obtiennent les classes durant l'exécution des programmes.



Quand le serveur `ComputeEngine` lie sa référence d'objet distant dans le registre, le registre télécharge les interfaces `Compute` et `Task` interfaces dont la classe stub dépend. Ces classes sont téléchargées à partir du serveur web du serveur `ComputeEngine` ou à partir du système de fichier, file system, selon le type de codebase URL utilisé au lancement du serveur.

Du fait que le client `ComputePi` a à la fois les interfaces `Compute` et `Task` disponibles dans son chemin d'accès aux classes, il charge leurs définitions à partir de ce chemin, et pas à partir du codebase du serveur.

Finalement, la classe `Pi` est chargée dans la machine virtuelle java du serveur `ComputeEngine` quand l'objet `Pi` est passé dans l'appel distant à `executeTask` de l'objet `ComputeEngine`. La classe `Pi` class est chargée par le serveur à partir soit du serveur web des clients ou du système de fichier, selon le type de codebase URL utilisé quand on démarre le client.