# Remote Method Invocation (RMI)

# Introduction

RMI est un ensemble de classes permettant de manipuler des objets sur des machines distantes (objets distants) de manière similaire aux objets sur la machine locale (objet locaux).

RMI apparaît avec Java 1.1 et est complètement intégré dans Java 1.1 donc est gratuit (≠ CORBA)

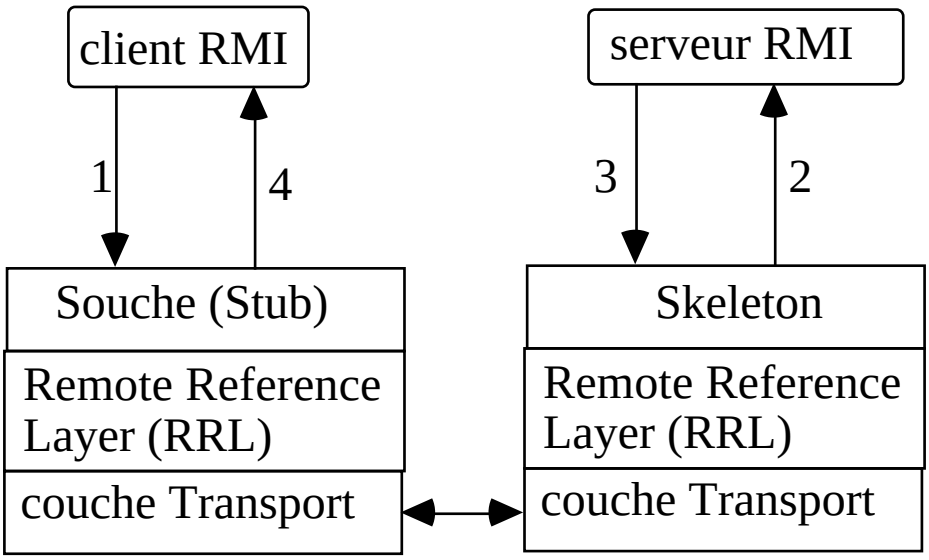
C'est un bon pas vers CORBA. C'est du CORBA allégé avec ses avantages (c'est plus simple) et ses inconvénients (tout doit être Java coté client comme coté serveur).
Si on veut être hétérogène coté client et coté serveur, il faut utiliser CORBA.

C'est un peu du "RPC orienté objet". Un objet local demande une fonctionnalité à un objet distant. Il y a donc une machine virtuelle Java des 2 cotés.

Ces manipulations sont "relativement" transparente. Pour cela RMI propose :
- un ramasse-miettes distribué.
- la gestion des représentants locaux d'objets distants et leur activation.
- la liaison avec les couches transport et l'ouverture de sockets appropriés.
- la syntaxe d'invocation, d'utilisation d'un objet distant est la même qu'un objet local.

# RMI en pratique

fonctionne dans les applications Java 1.1 et dans appletviewer.
pas dans IE 3.0.
Dans netscape 3.0 la classe `sunw.io.ObjioAppletHelper` doit être présente dans le répertoire repéré par `CODEBASE`) et l'URL doit être `htpp://` (et non `file://`).




la couche souche contient des représentants locaux de références d'objets distants. Une invocation d'une fonctionnalité sur cet objet distant est traité par la couche RRL (ouverture de socket, passage des arguments, marshalling, ...). Les objets distants sont repérés par des références gérées par la couche skeleton.

# Programmation avec RMI

Pour créer une application RMI, suivre les étapes suivantes.

1°) définir les interfaces pour les classes distantes.

2°) Créer et compiler les implémentations de ces classes.

3°) Créer les classes pour la souche et le skeleton à l'aide de la commande `rmic`.

4°) Créer et compiler une application serveur.

5°) lancer le `rmiregister` et lancer l'application serveur.

6°) Créer et compiler un programme client qui accède à des objets distants du serveur.

7°) lancer ce client.

# Un exemple classique : serveur bancaire

Un gestionnaire de comptes bancaires est capable de gérer plusieurs comptes. Il sera placé sur une machine distante, interrogé et manipulé par des clients.

exemple inspiré de [JDH] chapitre 12.

# 1°) définir les interfaces pour les classes distantes.

Les classes placées à distance sont spécifiées par des interfaces qui doivent implémenter `java.rmi.Remote` et dont les méthodes lèvent une `java.rmi.RemoteException` exception. On définit alors 2 interfaces `CreditCard` et `CreditManager`.

```
package credit;

import credit.*;
import java.rmi.*;

public interface CreditCard extends Remote {
    /** retourne le credit sur ce compte */
    public float getCreditLine() throws
RemoteException;

    /** permet de faire un achat de amount francs.
Leve une exception si la somme est supérieure
au credit bancaire.*/
    public void makePurchase(float amount, int
signature) throws RemoteException,
InvalidSignatureException,
CreditLineExceededException;

    /** positionne la signature de ce compte
bancaire. */
    public void setSignature(int pin) throws
RemoteException;
}
```

# 1°) définir les interfaces pour les classes distantes (suite)

```
package credit;

import credit.*;
import java.rmi.*;

public interface CreditManager extends
java.rmi.Remote {

    /** recherche le compte bancaire d'une
personne. S'il n'en a pas en crée un et le
retourne.*/
    public CreditCard findCreditAccount(String
Customer) throws DuplicateAccountException,
RemoteException;

    /** crée un nouveau compte bancaire pour la
personne newCustomer */
    public CreditCard newCreditAccount(String
newCustomer) throws RemoteException;
}
```

# 2°) Créer et compiler les implémentations de ces classes.

Il s'agit d'écrire 2 classes qui implémentent les interfaces ci dessus. En général ces classes ont pour nom *nomInterface*Impl. Ces classes doivent dériver de `java.rmi.server.UnicastRemoteObject`.

```
package credit;

import java.rmi.*;
import java.rmi.server.*;
import java.io.Serializable;

public class CreditCardImpl
    extends UnicastRemoteObject
    implements CreditCard, Serializable
 {
    private float creditLine = 5000f;
    private int signature = 0;
    private String accountName;

    public CreditCardImpl(String customer)
    throws RemoteException,
DuplicateAccountException {
        accountName = customer;
    }

    public float getCreditLine() throws
RemoteException {
        return creditLine;
    }
```

```
    public void setSignature(int pin) throws
RemoteException {
        signature = pin;
    }

    /** effectue un achat en s'assurant :
- de la bonne signature
- que l'achat est inférieure au credit.*/

    public void makePurchase(float amount, int
signature)
    throws RemoteException,
InvalidSignatureException,
CreditLineExceededException {
        if (signature != this.signature) {
            throw new InvalidSignatureException();
        }
        if (amount > creditLine) {
            throw new
CreditLineExceededException();
        } else {
            creditLine -= amount;
        }
    }
}
```

# 2°) Créer et compiler les implémentations de ces classes (suite)

```
package credit;

import java.rmi.*;
import java.rmi.server.*;
import java.util.Hashtable;

public class CreditManagerImpl extends
UnicastRemoteObject implements
CreditManager {
    private static transient Hashtable accounts =
new Hashtable();

   /** Le constructeur par défaut qui ne fait rien
d'autre qu'appeler super() */
    public CreditManagerImpl() throws
RemoteException { }

    /** Crée un nouveau compte. Entre autre ajoute
le nouveau client dans la hashtable accounts */
    public CreditCard newCreditAccount(String
customerName) throws RemoteException {
        CreditCardImpl newCard = null;
        try {
            newCard = new
CreditCardImpl(customerName);
        } catch (DuplicateAccountException e) {
            return null;
        }
        accounts.put(customerName, newCard);
        return newCard;
    }
```

```
    /** recherche un compte bancaire pour le client
customer. S'il n'en trouve pas, retourne en crée
un et le retourne */

    public CreditCard findCreditAccount(String
customer) throws DuplicateAccountException,
RemoteException {
        CreditCardImpl account =
(CreditCardImpl)accounts.get(customer);
        if (account != null) {
            return account;
        }

        account = new CreditCardImpl(customer);
        accounts.put(customer, account);
        return account;
    }
}
```

Pour compiler utiliser les commandes :

```
javac -d . CreditCard.java
javac -d . CreditCardImpl.java
javac -d . CreditManager.java
javac -d . CreditManagerImpl.java
```

Ces commandes vont créer les fichiers `.class` et les déposer dans `./credit`.

# 3°) Créer les classes pour la souche et le skeleton (commande rmic)

Il faut maintenant créer la souche et le skeleton pour accéder aux classes ci dessus. Java donne un outil rmic qui utilisent les fichiers .class.

la syntaxe de rmic est

```
rmic [options]
paquetage.sousPaquetage..nomBaseByteCo
de1
paquetage.sousPaquetage..nomBaseByteCo
de2 ...
```

On a donc ici :

```
rmic -d . credit.CreditCardImpl
credit.CreditManagerImpl
```

ce qui crée 4 classes dans le répertoire credit : CreditCardImpl_Skel.class, CreditCardImpl_Stub.class, CreditManagerImpl_Skel.class, CreditManagerImpl_Stub.class.

# 4°) Créer et compiler une application serveur

C'est le programme qui sera à l'écoute des demandes des clients. Ce programme crée un objet `CreditManagerImpl` qui attend alors les requêtes.

```
package credit;
import java.util.*;
import java.rmi.*;

public class CardBank {
    public static void main (String args[]) {
        //  Cree et installe un security manager.
        System.setSecurityManager(new
RMISecurityManager());

        try {
            // Cree un gestionnaire bancaire
            System.out.println("CardBank:
gestionnaire bancaire cree");
            CreditManagerImpl cmi = new
CreditManagerImpl();

            // enregistre ce gestionnaire bancaire
            // dans le registre des objets disponibles.
            System.out.println("CardBank: bind le
gestionnaire");
            Naming.rebind("cardManager", cmi);
        } catch (Exception e) {
            e.printStackTrace();
            System.out.println(e.getMessage());
        }
    }
}
```

# 4°) Créer et compiler une application serveur (suite)

Il faut pour un serveur d'objet installé un `SecurityManager` spécifique. Celui-ci est donné par Java et on l'installe par :

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

Pour rendre l'objet gestionnaire bancaire disponible, il faut l'enregistrer dans le "RMIregistry" (voir ci dessous). On lui donne pour cela un nom et on l'enregistre par la méthode statique

```
Naming.rebind("cardManager", cmi);
```

Par la suite cet objet serveur bancaire sera accessible par les autres machines en indiquant la machine sur laquelle est exécuté ce serveur et son nom (cf. ci dessous).

Le premier argument de `Naming.rebind()` doit être :
`rmi://`*`nomServeurRMI`*`:`*`port/nom`*.
Par défaut *`nomServeurRMI`* est la machine locale qui contient le serveur RMI et `port` est `1099`.
Cet "URL rmi" sera utilisé par les clients pour interroger le serveur grâce à l'appel `Naming.lookup()`.
On compile ce serveur par :

```
javac -d . CardBank.java
```

# 5°) lancer le rmiregister et lancer l'application serveur

Il faut d'abord lancer le rmiregister puis le serveur.

```
% rmiregister &
% java credit.CardBank
```

Ce qui rend maintenant le serveur disponible pour de futur clients.

Les sorties du serveur sont :

```
CardBank: gestionnaire bancaire cree
CardBank: bind le gestionnaire
```

# 6°) Créer et compiler un programme client qui accède au serveur

Le programme doit être lancé par :
`% java ClientBancaire serveurRMI nomClient`

```
package credit;

import java.rmi.*;

public class ClientBancaire{
    public static void main(String args[]) {
        CreditManager cm = null;
        CreditCard account = null;

        // verifie la ligne de commande
        if (args.length < 2) {
            System.err.println("Usage : ");
            System.err.println("java ClientBancaire
<server> <account name>");
            System.exit (1);
        }

        // Cree et installe un security manager.
        System.setSecurityManager(new
RMISecurityManager());
```

```
        // recupere une reference sur le
        // gestionnaire bancaire distant.

       try {
           String url = new String ("rmi://" + args[0] +
"/cardManager");
           System.out.println ("ClientBancaire
recherche le gestionnaire bancaire a l' url = " +
url);
           cm = (CreditManager)Naming.lookup(url);
        } catch (Exception e) {
           System.out.println("Erreur a l'acces du
gestionnaire bancaire" + e);
       }

        // recupere le compte bancaire (distant)
        // du client.
       try {
           account = cm.findCreditAccount(args[1]);
           System.out.println ("recherche du compte
de " + args[1]);
        } catch (Exception e) {
           System.out.println("Erreur a la recherche
du compte de " + args[1]);
       }
```

```
        // effectue des transactions.
        try {
            System.out.println("credit disponible : "
                        + account.getCreditLine());
            System.out.println("Changement de
signature");
            account.setSignature(1234);
            System.out.println("un achat de 100 F");
            account.makePurchase(100.00f, 1234);
            System.out.println("credit disponible : " +
account.getCreditLine());
          System.out.println("second achat de 160 F");
            account.makePurchase(160.00f, 1234);
            System.out.println("credit disponible :  "+
account.getCreditLine());
        } catch (Exception e) {
            System.out.println("Erreur dans les
transactions pour " + args[1]);
        }
        System.exit(0);
    }
}
```

remarque :
le code ci dessus manipule les objets distants comme s'ils étaient locaux.
Après avoir installé un security manager RMI, le client recherche le serveur bancaire distant par `Naming.lookup(url)` où `url` est une "URL rmi" de la forme `rmi://machineDuServeurRMI:port/nomGestionnaireBancaire`
Pour tester ce programme sur une seule machine utiliser `localhost` comme *machineDuServeurRMI*.
`Naming.lookup()` retourne une référence `Remote` qu'il faut nécessairement caster.

# 7°) lancer ce client

par
```
% java ClientBancaire serveurRMI
nomClient
```

# Bibliographie

[JDH] Java 1.1 developer's Handbook Philip Heller, Simon Roberts ed Sybex.

Java client-serveur Cédric Nicolas et al. collection fi-system ed Eyrolles

Remote Method Invocation Specification à : http://www.javasoft.com/products/jdk/1.1/docs/guide/rmi/spec/rmiTOC.doc.html