



# Programmation réseau avec Java

# 3/7 – RMI, un peu de sécurité et CORBA

Olivier Ricou

12 mai 2008



Java, comme CORBA avant, permet d'exécuter des taches à distances avec la RMI.

Cette partie aborde les points suivants :

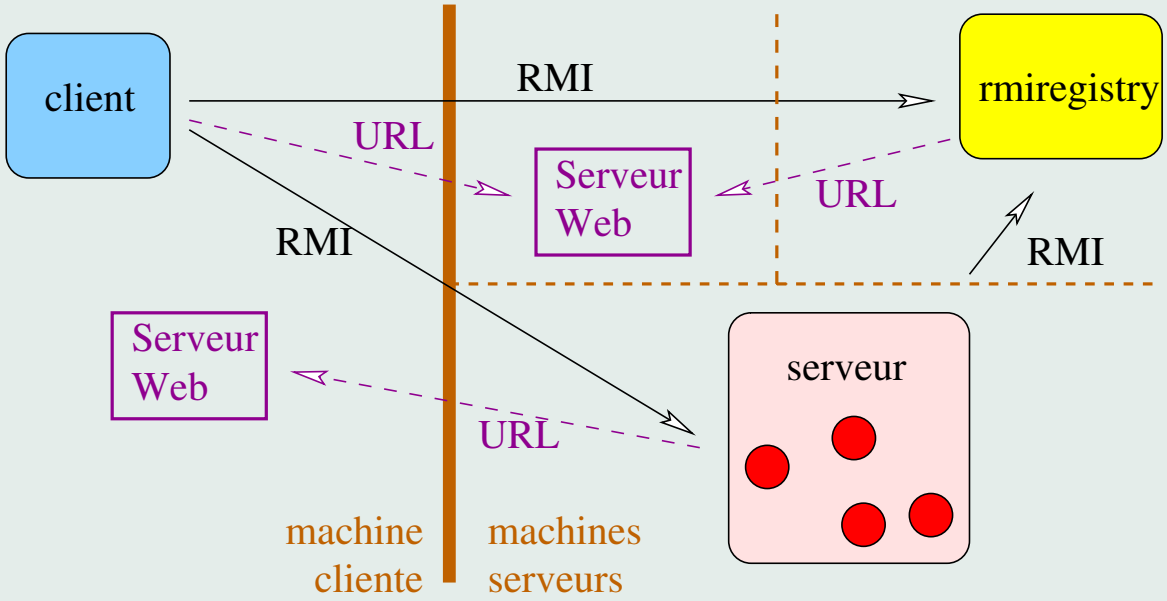
1. La Remote Method Invocation
2. La sécurité
3. Corba avec Java

Quelques URLs :

– http ://java.sun.com/docs/books/tutorial/rmi/index.html
– http ://java.sun.com/developer/onlineTraining/Programming/BasicJava1/rmi.html
– http ://java.sun.com/docs/books/tutorial/security1.2/index.html



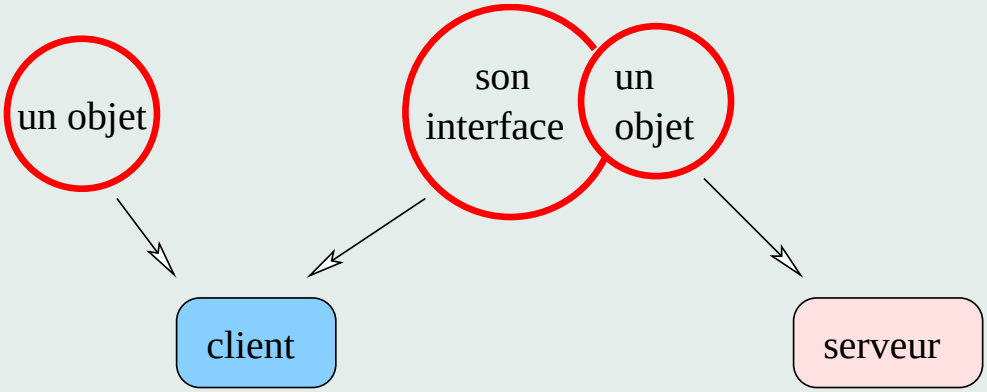
# 1. RMI




La RMI, Remote Method Invocation, permet d'utiliser des objets à distance.

Ci-dessus, le client peut demander au RMI registry de lui indiquer quel serveur pourrait exécuter la méthode d'un objet (en rouge) pour lui.






L'idée est qu'on peut très bien utiliser un objet dont on ne connait que l'interface.

```
Cube c = new Cube(3.);
Forme f = (Forme) quelque_chose_sur_le_serveur;

// l'objet Cube a un diametre comme toutes les Formes
System.out.println(c.diametre());

// f est un objet qui respecte l'interface Forme
// donc qui a aussi un diametre
System.out.println(f.diametre());
```



Le problème est que `f` est un objet sur le serveur et donc pour connaitre `f.diametre()` il faut demander au serveur.

A partir du moment où l'interface `Forme` hérite de la classe `Remote`, Java comprends qu'il doit utiliser une RMI pour calculer `f.diametre()`

Fibonacci.java

```
1  import java.rmi.Remote;
2  import java.rmi.RemoteException;
3  import java.math.BigInteger;
4
5  public interface Fibonacci extends Remote
6  {
7    public BigInteger getFibonacci(int n)
8                     throws RemoteException;
9
10   public final static String LOOKUPNAME="Fibonnaci";
11 }
```



Java a donc compris qu'il doit utiliser une RMI mais il ne sait pas quel est le serveur et où est l'objet du serveur dont il doit exécuter la méthode.

Le RMI registry est un serveur de nom d'objet qui permet de savoir quel serveur propose quel objet. La commande pour lancer manuellement le RMI registry est rmiregistry. Il écoute les requètes sur le port 1099 par défaut.

Il faut donc que le client indique quel RMI registry il contacte (nom de la machine) et quel objet il désire.


```
Object o = Naming.lookup("rmi://serveur.fr/fibonacci");
```

Ici le registry est sur `serveur.fr` et le nom de l'objet est `fibonacci`.



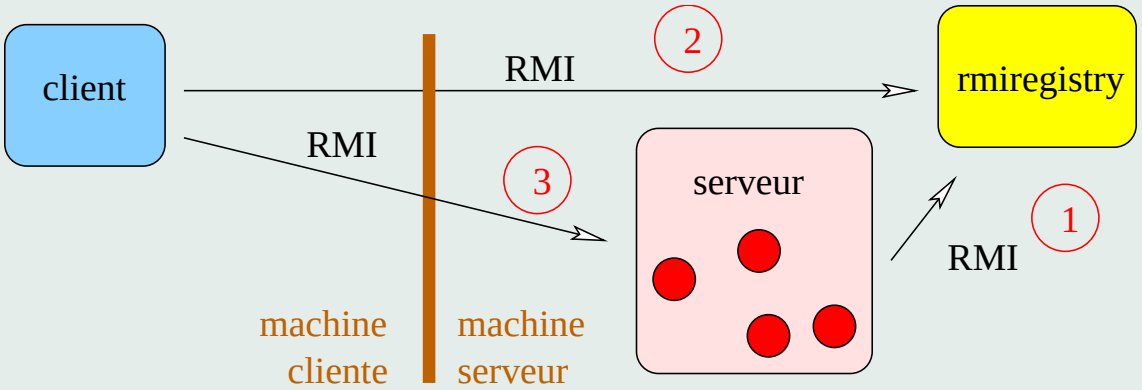
FibonacciClient.java

```
1  import java.rmi.registry.*;
2  import java.rmi.Naming;
3  import java.math.BigInteger;
4
5  public class FibonacciClient
6  {
7    public static void main(String args[])
8    {
9      try {
10       Object o = Naming.lookup(Fibonacci.LOOKUPNAME);
11       Fibonacci calculator = (Fibonacci) o;
12       BigInteger f = calculator.getFibonacci(666);
13
14       System.out.println("Fibonacci(666)="+f);
15     }
16     catch (Exception e) { System.err.println(e);}
17   }
18 }
```



Pour que le RMI registry puisse répondre au client, il faut qu'il ait été informé par un serveur que l'objet `fibonacci` est disponible.

Le serveur indique au RMI registry qu'il dipose d'un objet pouvant être exécuté via une RMI.




```
FibonacciServer fs = new FibonnaciServer();
Naming.rebind("fibonnaci",fs);
```

ou bien (on peut choisir un serveur en argument de `getRegistry`)

```
Registry registry = LocateRegistry.getRegistry();
registry.rebind("fibonnaci",fs);
```

FibonacciServer.java

```
1  import java.rmi.*;
2  import java.rmi.registry.*;
3  import java.rmi.server.UnicastRemoteObject;
4  import java.math.BigInteger;
5
6  public class FibonacciServer implements Fibonacci
7  {
8    public FibonacciServer() throws RemoteException
9    {
10     UnicastRemoteObject.exportObject(this);
11   }
12
13   public static void main(String[] args)
14   {
15     Registry registry;
16
17     try {
18       FibonacciServer f = new FibonacciServer();
```

```
19        try {
20          registry = LocateRegistry.getRegistry();
21          registry.rebind(Fibonacci.LOOKUPNAME, f);
22        }
23        catch (RemoteException re) {
24          System.out.println("Starting the registry");
25          registry = LocateRegistry.createRegistry(109
26          registry.rebind(Fibonacci.LOOKUPNAME, f);
27        }
28        System.out.println("Fibonacci Server ready.");
29      }
30      catch (Exception e) { System.out.println(e); }
31    }
32
33    public BigInteger getFibonacci(int n)
34                       throws RemoteException
35    {
36      System.out.println(n + "th Fibonacci number");
37      BigInteger zero = new BigInteger("0");
38      BigInteger one  = new BigInteger("1");
```



```
39        BigInteger a = zero, b = one;
40        int i=1;
41
42        if (n==0) return zero;
43        if (n==1) return one;
44        while (i<n) {
45          BigInteger temp = b;
46          b = b.add(a);
47          a = temp;
48          i++;
49        }
50        return b;
51      }
52    }
```

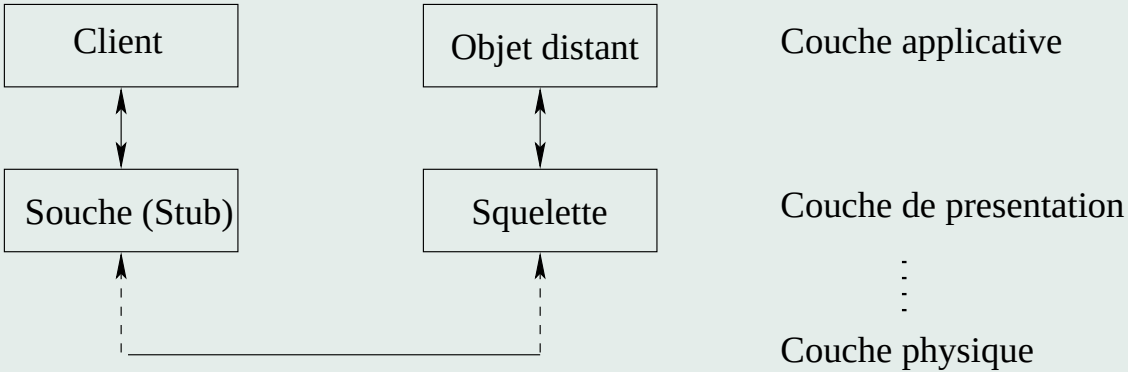


## 1.1. En pratique sur le client, `client.fr`

Il faut les fichiers :

- `FibonacciClient.java`, le programme client,
- `Fibonacci.java`, l'interface, utiliée dans le client,
- `FibonacciServer_Stub.class`, le *Stub* de l'objet qui sera utilisé.

La souche de l'objet distant est l'interface réseau permettant d'atteindre l'objet distant, de lui envoyer les arguments et de récupérer le résultat.





Le travail de la souche consiste donc à

1. initier le contact avec le serveur contenant l'objet distant,
2. empaqueter les paramêtres par sérialisation et les envoyer au serveur,
3. attendre le résultat,
4. dépaqueter le résultat ou les exceptions,
5. renvoyer le résultat au client.

La création de la souche se fait à partir du `.class` de l'objet distant (donc sur le serveur) :

```
rmic -v1.2 FibonacciServer
```

Sans l'option `-v1.2` on aurait aussi le squelette de l'objet qui au serveur ce que la souche est au client. Le squelette n'est plus utile depuis Java 1.2, le serveur le génère au vol par introspection.



## 1.2. En pratique sur le serveur, `serveur.fr`

Il faut les fichiers :

- `FibonacciServeur.java`, le programme serveur,
- `Fibonacci.java`, l'interface implémentée par le serveur,
- `FibonacciServer_Stub.class`, le *Stub* pour le transmettre au RMI registry,
- `FibonacciServer_Skeleton.class`, pour les versions de Java antérieures à 1.2.

Une fois ces fichiers crées et compilés, on peut lancer le RMI registry puis le serveur.

Dans une fenêtre (si ce n'est pas lancé par le programme serveur)

```
rmiregistry
```

Dans une autre fenêtre

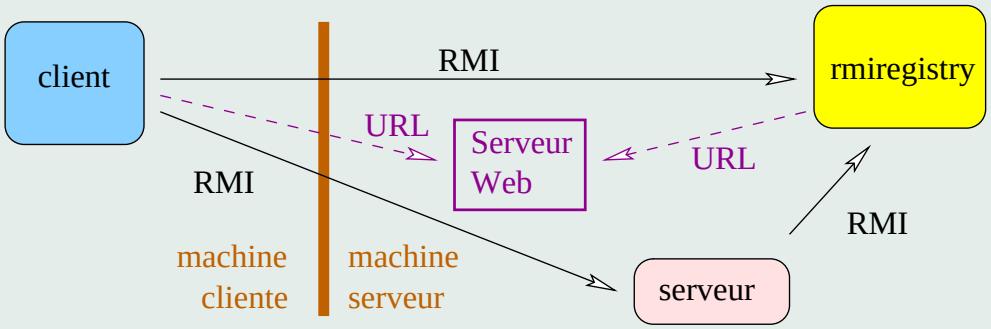
```
java FibonacciServer
```



## 1.3. Récupération de la souche sur la machine serveur

Il est dommage de devoir installer sur le client la souche créée sur le serveur.

Il est tentant de demander au serveur d'envoyer les souches des objets qu'il propose. Cela peut être fait avec un serveur Web (ou ftp).




Pour cela, il suffit d'indiquer au moment de l'exécution du client où il peut trouver les `.class` qui lui manquent :

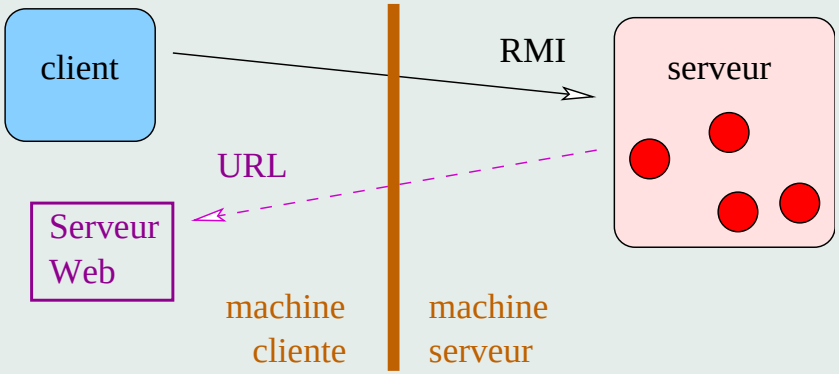
```
java -Djava.rmi.server.codebase=http://server.fr/ \
FibonacciClient
```



## 1.4. L'exécution à distance d'un objet local

Dans l'exemple précédent on a utilisé un objet du serveur pour effectuer le calcul du n-ieme élément de la suite de Fibonacci.

Imaginons que l'objet à exécuter soit sur le client et inconnu du serveur mais qu'on veut néanmoins que le serveur fasse le calcul. Il faut envoyer la classe complète de l'objet au serveur pour qu'il l'exécute.




L'objet distribué du serveur prend comme argument un Fonctor qu'il exécute pour en renvoyer le résultat. Ce fonctor est téléchargeable.



## 1.5. Un exemple complet et dangereux

Dans ce dernier cas, on a importé du bytecode pour l'exécuter localement. C'est potentiellement un trou de sécurité.

Voyons ce que cela donne dans avec un programme complet.

Fonctor.java

```
1  package ricou.rmi.test1;
2
3  public interface Fonctor
4  {
5    public String eval();
6  }
7
8
```

On ne peut pas faire plus générique, avec `Object`, car on devra faire passer le résultat via la RMI et un `Object` n'est pas sérialisable.

Le fonctor qu'on envoie au serveur est un objet qui permet d'exécuter des commandes systèmes :

Command.java

```
1  package ricou.rmi.test1;
2  import java.io.*;
3
4  public class Command implements Fonctor,
5                                   Serializable
6  {
7    String cmd;
8
9    public Command(String c) {cmd = c;}
10
11   public String eval()
12   {
13     String res="";
14     try {
15       Process proc = Runtime.getRuntime().exec(cmd);
```



```
16          InputStream inputstream=proc.getInputStream();
17          InputStreamReader inputstreamreader =
18                      new InputStreamReader(inputstream);
19          BufferedReader bufferedreader =
20                  new BufferedReader(inputstreamreader);
21
22          // read the ls output
23          String line;
24          while ((line=bufferedreader.readLine()) !=null
25            res = res +"n"+line;
26        }
27        catch(Exception e) {res = res + e;}
28        return res;
29      }
30  }
```



Le serveur évalue le fonctor qu'on lui envoie, son interface est donc un Evaluator. C'est la seule interface dont à besoin le client :

Evaluator.java

```
1  package ricou.rmi.test1;
2
3  import java.rmi.Remote;
4  import java.rmi.RemoteException;
5
6  public interface Evaluator extends Remote
7  {
8    public String eval(Fonctor f)
9                   throws RemoteException;
10 }
```



On a vu qu'il va avoir des fichiers en bytecode chargés à l'exécution. Java contrôle sévèrement ce genre d'agissement. Pour l'instant on court-circuite la sécurité en créant une classe de sécurité vide :

SecurityManagerPermissif.java

```
1  package ricou.rmi.test1;
2
3  import java.rmi.*;
4  import java.security.*;
5
6  public class SecurityManagerPermissif
7               extends RMISecurityManager
8  {
9    public void checkPermission(Permission p) { }
10 }
```

En utilisant cette classe, le client autorise l'importation de la souche et le serveur l'importation du fonctor (`Command.class`).



Client.java

```
 1  package ricou.rmi.test1;
 2
 3  import java.rmi.*;
 4
 5  public class Client
 6  {
 7   public static void main(String[] arg)
 8   {
 9    try
10    {
11     Evaluator server =
12      (Evaluator)Naming.lookup("rmi://server.fr/EvalSe
13     System.out.println(server.eval(new Command("ps"))
14    }
15    catch (Exception e) {System.out.println(e);}
16   }
17  }
```

EvalService.java

```
1  package ricou.rmi.test1;
2  import java.rmi.Naming;
3  import java.rmi.RemoteException;
4  import java.rmi.server.UnicastRemoteObject;
5
6  public class EvalService extends UnicastRemoteObject
7                           implements Evaluator
8  {
9    public EvalService() throws RemoteException
10   {
11     super(); setLog(System.out); //dialogue serveur
12   }                              //client sur stdout
13
14   public String eval(Fonctor f)
15   {
16     System.out.println("Eval of "+f.getClass().getNa
17     return f.eval();
18   }
```



```
20    public static void main(String[] arg)
21    {
22      if (System.getSecurityManager() == null)
23        System.setSecurityManager(
24                    new SecurityManagerPermissif());
25      try
26      {
27        EvalService es = new EvalService();
28
29        Naming.rebind("/EvalService",es);
30        System.out.println("EvalService bound in regis
31      }
32      catch (Exception e) {System.out.println(e); }
33    }
34  }
```



Et pour faire marcher l'ensemble, il faut :

- sur le serveur web du client : `Command.class`,
- sur le serveur web du serveur : `EvalService_Stub.class`,
- dans le jar du client : `Fonctor.class, Evaluator.class, Command.class, SecurityManagerPermissif.class, Client.class`
- dans le jar du serveur : `Fonctor.class, Evaluator.class, SecurityManagerPermissif.class, EvalService.class`

Comme on a utilisé des paquets, `package ricou.rmi.test1`, il faut stocker chaque fichier ou ensemble de fichiers dans des `.jar`.

Les commandes à faire sur le serveur sont :

```
rmiregistry
java -Djava.rmi.server.codebase=http://client.fr/ricou_rmi_test1.jar \
 ricou/rmi/test1/EvalService
```

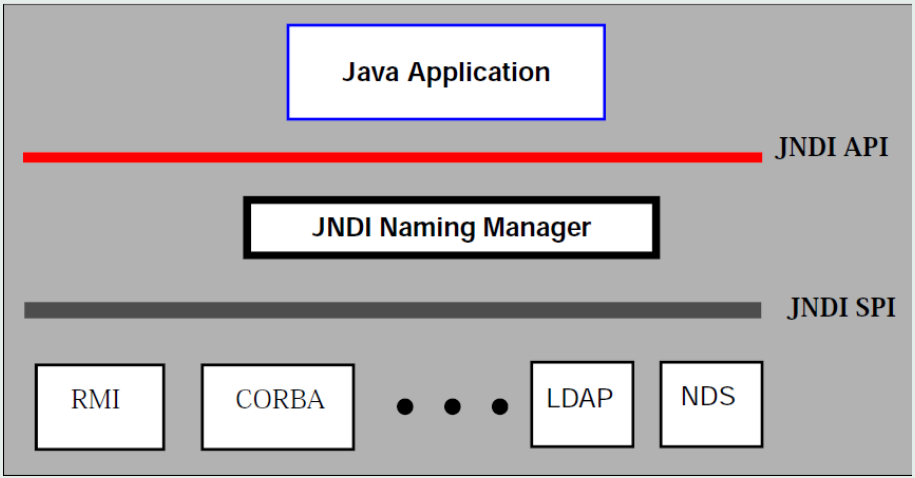
et sur la machine cliente

```
java -Djava.rmi.server.codebase=http://serveur.fr/ricou_
 ricou/rmi/test1/Client
```



## 1.6. JNDI

Java Naming and Directory Interface est l'interface de Java qui permet de se connecter simplement à des annuaires.




On peut aussi bien l'utiliser pour récupérer une adresse IP dans un annuaire DNS que pour récupérer un nom d'object comme avec le RMI registry. Dans l'exemple qui suit on stocke l'objet dans une base LDAP.



JNDIexample.java

```
1  import javax.naming.*;
2  import javax.naming.directory.*;
3  import java.util.Hashtable;
4  import java.rmi.*;
5
6  class JNDIexample
7  {
8    public static void main(String[] args)
9    {
10     // Set up env for creating initial context
11     Hashtable env = new Hashtable(11);
12     env.put(Context.INITIAL_CONTEXT_FACTORY,
13             "com.sun.jndi.ldap.LdapCtxFactory");
14     env.put(Context.PROVIDER_URL,
15             "ldap://tunnel:389/o=JNDIzone");
```



```
17      try
18      {
19        DirContext ctx = new InitialDirContext(env);
20
21        Fibonacci h = new FibonacciLight();
22
23        ctx.bind("cn="+Fibonacci.LOOKUPNAME, h);
24
25        Fibonacci h2 = (Fibonacci)
26             ctx.lookup("cn="+Fibonacci.LOOKUPNAME);
27        System.out.println(h2.getFibonacci(666));
28
29        ctx.close();
30      }
31      catch (NamingException e) { System.out.println("
32      catch (RemoteException e) { System.out.println("
33    }
34  }
```



## 2. La sécurité sous Java

À partir du moment où Java et son environnement permet de télécharger du code et de l'exécuter en direct, on risque que le code utilse à des fins malfaisantes des opérations comme

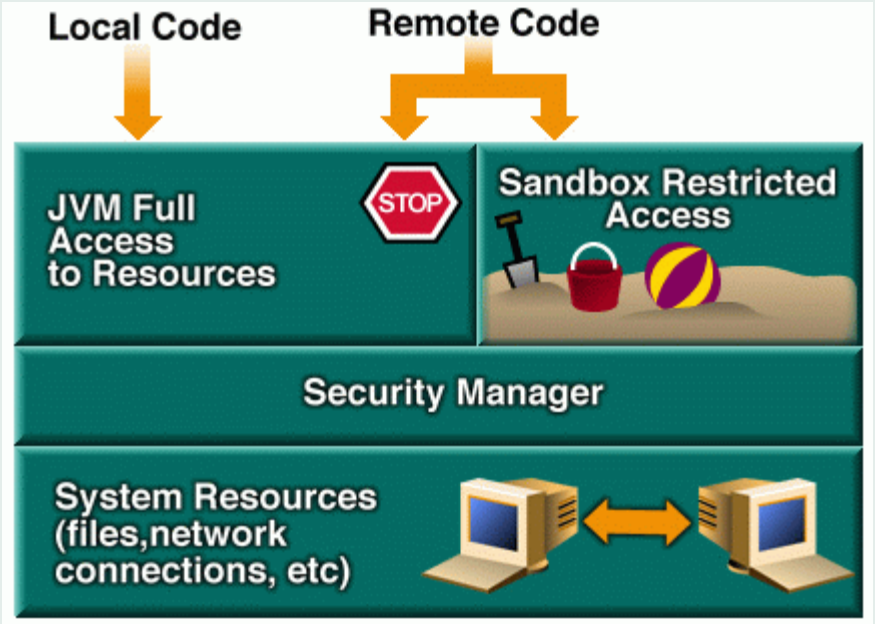
– l'acquisition d'information sur la machine ou sur l'utilisateur,
– l'écriture sur le disque dur,
– la connexion à des machines distantes,
– ...

Les Applets de Java étant directement concernées par ce problème, Java à introduit un système de sécurité pour lutter contre ces risques dès la première version.

Mais les applets ne sont pas les seules, on vient de voir que la RMI est aussi une source de risque.



## 2.1. Le modèle de sécurité de Java 1.0

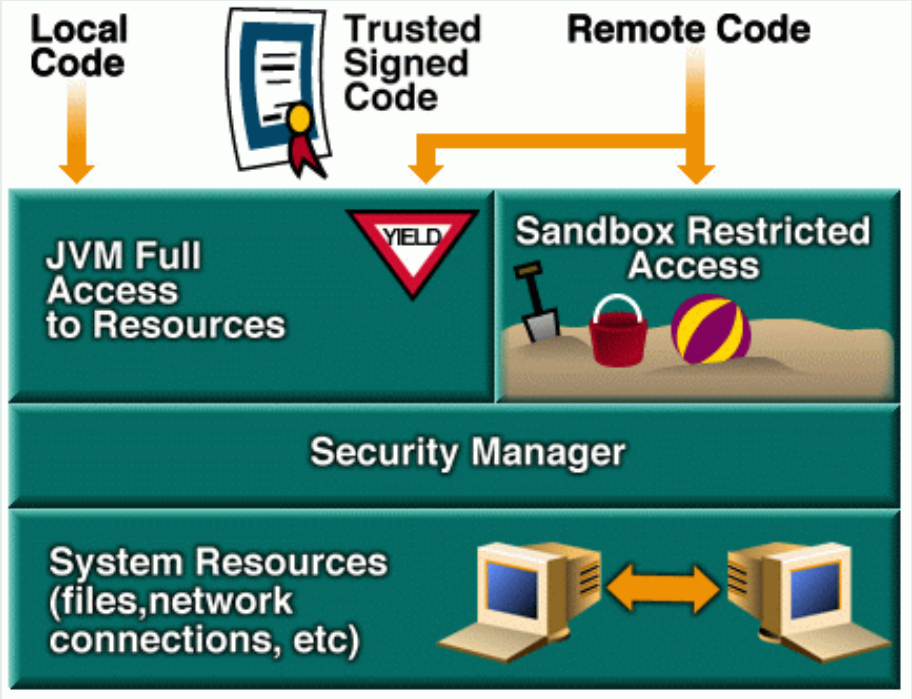



Le bac à sable est une zone depuis laquelle il n'est pas possible d'effectuer les opérations potentiellement dangereuses citées ci-dessus.

C'est efficace mais trop rigide en particulier si l'on a confiance dans le code téléchargé.



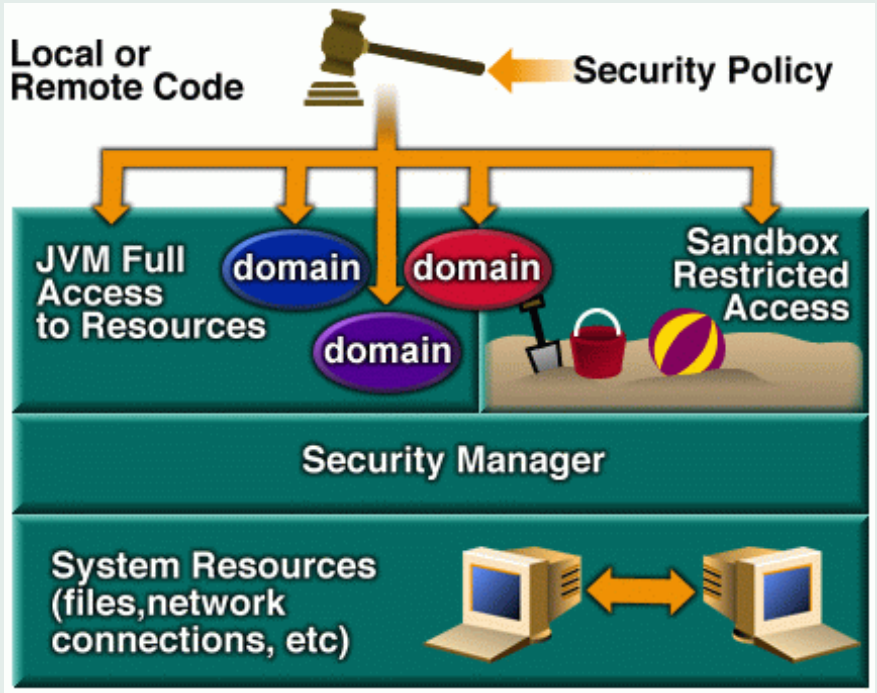
## 2.2. Le modèle de sécurité de Java 1.1




Java 1.1 introduit la notion de code signé. Si l'on a confiance dans la signature alors on accorde au code les mêmes droits qu'à du code local (i.e. tous les droits).



## 2.3. Le modèle de sécurité de Java 1.2




Java 1.2 affine le choix d'octroi de droits. Tout code est lié à une politique de sécurité qui définit ses droits de façon détaillée en fonction de paramètres choisis par l'utilisateur, l'administrateur ou le développeur.



## 2.4. Configuration de la politique de sécurité

Les règles de sécurité sont définies dans les fichiers référencés par le fichier général de la politique de sécurité :

```
>grep policy.u $JAVA_HOME/jre/lib/security/java.security
policy.url.1=file:${java.home}/lib/security/java.policy
policy.url.2=file:${user.home}/.java.policy
```

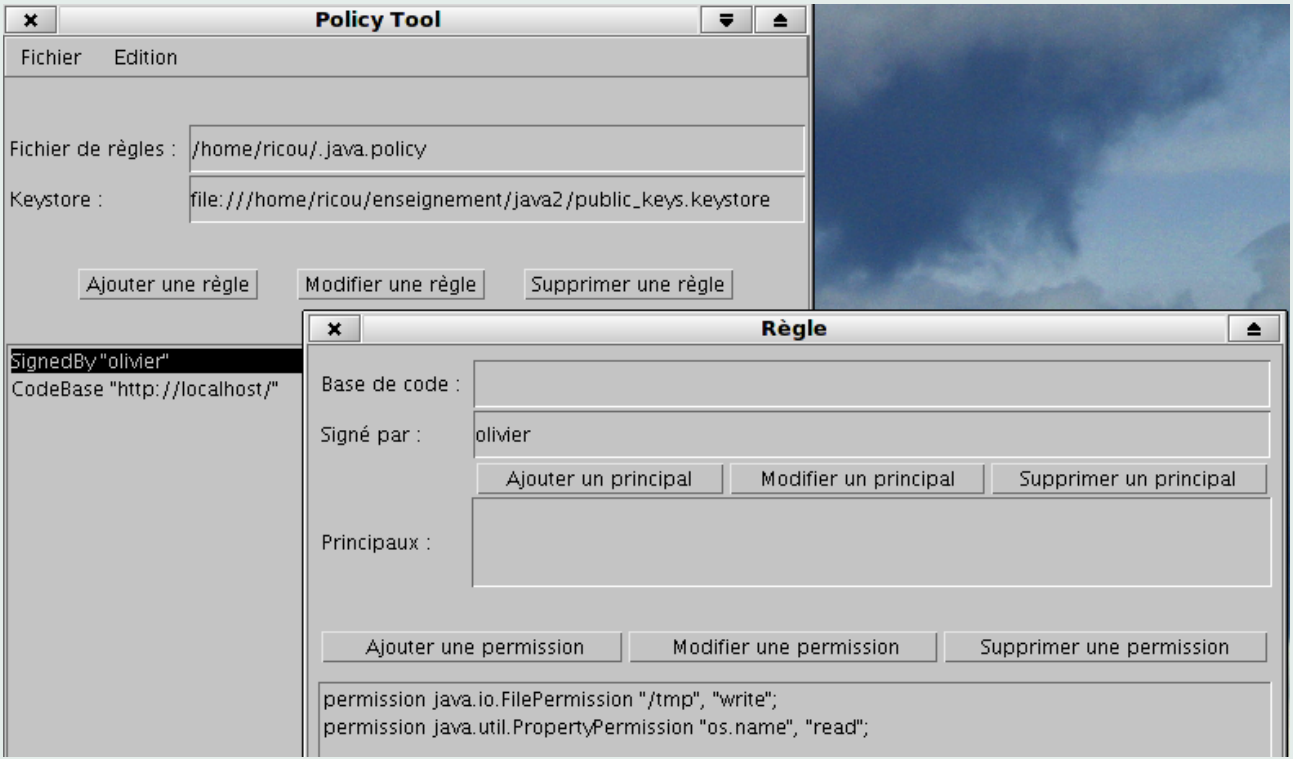
On voit que le 1er fichier à utiliser est celui de l'administrateur, défini dans `${java.home}`, le 2nd étant celui de l'utilisateur, défini dans son compte. Si ces fichiers n'existent pas, on utilise le bac à sable.

Ces règles suivent un formalise relativement simple

```
>head ~/.java.policy
keystore ``file:///home/ricou/java/public_keys.keystore''

grant signedBy ``olivier'' {
  permission java.io.FilePermission ``/tmp'', ``write'';
```



. . . formalisme surtout simplifié par l'usage du programme `policytool`[1].




[1] cf http ://java.sun.com/docs/books/tutorial/security1.2/tour1/wstep1.html



Chose surprenante il n'y a pas de méthode pour refuser explicitement des droits. Cela veut dire que l'administrateur ne peut pas indiquer que le code en provenance de `http://diane/` doit toujours être refusé, même si l'utilisateur l'autorise dans son fichier `~/.java.policy`.

Les fichiers d'autorisation s'additionnent les uns aux autres.

On peut aussi désigner explicitement le fichier de politique de sécurité avec l'option `-Djava.security.policy`. C'est la façon de restreindre les droits d'une application en indiquant qu'on utilise un gestionnaire de sécurité avec l'option `-Djava.security.manager` :

```
java -Djava.security.manager \
-Djava.security.policy=MonFichier.policy monApplication
```

On peut aussi activer un gestionnaire de sécurité dans le code :

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



## 2.5. Le gestionnaire de sécurité pour la RMI

Dans le cas du service RMI, EvalService étudié en première partie, on a créé un SecurityManagerPermissif pour contourner la sécurité qu'impose la RMI. Il est préférable d'utiliser un `RMISecurityManager` en y associant un fichier de politique de sécurité.

```
if (System.getSecurityManager() == null)
  System.setSecurityManager(new RMISecurityManager());
```

Avec par exemple le fichier de permissions suivant (pour tout acepter on aurait utilisé `grant` sans argument : "`grant {`") :

```
>cat ~/.java.policy
grant codeBase "file:///var/www/ricou_rmi_test1.jar" {
  permission java.security.AllPermission;
};
```

Pour débuger la vérification des droits d'accès, utiliser l'option `-Djava.security.debug=access,failure` à l'exécution.



## 2.6. Java et la signature de paquetages

Pour l'instant les règles de politique de sécurité se basaient sur l'origine du code, `codeBase`. Java 1.1 et Java 1.2 se basent aussi sur la signature chiffrée associée à du code importé pour décider ou non de l'exécuter.

Pour commencer on se crée une paire de clés privée/publique :

```
>keytool -genkey -keystore java.keystore \
-alias olivier -keyalg rsa
```

on répond aux questions

```
...
Quel est le code de pays a deux lettres pour cette unite
  [Unknown] :  FR
Est-ce CN=Olivier Ricou, OU=LRDE, O=Epita, L=Paris, ST=F
  [non] :  oui
```

et on a dans le fichier `java.store` ses clés. Attention à ne pas oublier le mot de passe qui protège ce fichier. Il sera demandé à chaque utilisation.



La clé privée permet de signer les paquetages `.jar` que l'on veut exporter :

```
jarsigner -keystore java.keystore \
ricou_rmi_test1.jar olivier
```

notez que d'autres peuvent aussi signer le paquetage, les signatures s'ajouteront.

Tout le problème est de transmettre sa clé publique à celui qui va importer le paquetage afin qu'il soit certain qu'il n'y a pas tromperie sur la signature.

Pour cela on fait un fichier certificat `OlivierRicou.cert` qui contient la clé publique :

```
>keytool -export -keystore java.keystore \
-alias olivier -file OlivierRicou.cert
```

Il ne reste plus qu'à transmettre de façon sûre ce fichier.



L'utilisateur qui importe le paquetage récupère la clé publique de l'émetteur et la range dans son trousseau de clé en lesquelles il a confiance. Il pourra ainsi ensuite vérifier le signature du paquetage.

Dans notre cas, il insert la clé publique contenue dans `OlivierRicou.cert` dans son fichier de clés publiques `public_keys.keystore` :

```
>keytool -import -alias olivier -file OlivierRicou.cert
-keystore public_keys.keystore

Tapez le mot de passe du Keystore :  xxxxxx
Propriétaire : CN=Olivier Ricou, OU=LRDE, O=Epita, L=Par
Émetteur : CN=Olivier Ricou, OU=LRDE, O=Epita, L=Paris,
Numéro de série : 3ef634d2
Valide du : Mon Jun 23 00:59:30 CEST 2003 au : Sun Sep 2
Empreintes de certificat :
  MD5 :  F7:0E:DF:ED:E6:02:B6:C3:16:51:FE:E0:23:19:32:8E
  SHA1: 4A:D6:93:E9:65:F1:28:16:F6:71:79:6E:E8:8F:31:55:
Faire confiance à ce certificat ? [non] :  oui
Certificat ajouté au Keystore
```



Celui qui importe le paquetage peut maintenant indiquer dans son fichier de politique de sécurité qu'il fait confiance au code en provenance d'olivier :

```
>cat ~/.java.policy
keystore "file:///home/pierre/public_keys.keystore";

grant signedBy "olivier" {
  permission java.security.AllPermission;
};
```

Le problème est qu'il a fallu récupérer la clé publique d'olivier via son fichier `OlivierRicou.cert` ce qui n'est pas toujours simple à faire de façon sûre.

Aussi il peut être bon de faire confiance à une Authorité de Certification pour signer la clé publique de l'émetteur du paquetage.



## 2.7. Les clés publiques des Authorité de Certification

Le point faible dans le système de clé publique, clé privée est la transmission de la clé publique. Chaque transfert de clé publique devrait être une opération lourde avec des procédures de vérification (comme la vérification via le condensat MD5).

Une fois qu'on a *une* clé publique sûre, on peut faire confiance à toutes les clés publiques certifiées i.e. signées par la clé privée correspondant à la clé publique dont on est sûr.

Le fait d'être la première clé sur lequel se bati notre confiance est un métier, il s'agit de l'Authorité de Certification.

C'est le problème de l'Authorité de Certification de transmettre sa clé publique de façon sure. Vérisign et Thawte, les AC (CA en anglais) les plus connues, ont leurs clés publiques dans les distributions de Java.

On peut aussi vérifier leurs clés sur leur site. Pour Vérisign,
cf http ://www.verisign.com/repository/root.html.



```
>keytool -list -v -keystore \
$JAVA_HOME/jre/lib/security/cacerts

Entrez le mot de passe du magasin de clés :  changeit

Type du magasin de clés : jks
Fournisseur du magasin de clés : IBMJCE
Votre magasin de clés contient 10 entrées

Nom d'alias : verisignclass2ca
Date de création : 29 juin 1998
Type d'entrée : trustedCertEntry

Propriétaire : OU=Class 2 Public Primary Certification A
Emetteur : OU=Class 2 Public Primary Certification Autho
Numéro de série : ba5ac94c053b92d6a7b6df4ed053920d
Valable du : Mon Jan 29 01:00:00 CET 1996 au : Thu Jan 0
Empreinte du certificat :
  MD5 :  EC:40:7D:2B:76:52:67:05:2C:EA:F2:3A:4F:65:F0:D8
  SHA1 : A5:EC:73:D4:8C:34:FC:BE:F1:00:5A:EB:85:84:35:24
```



## 2.8. Demander une certification de sa clé publique

Donc on a confiance en une AC, il ne reste plus qu'à lui transmettre de façon sûre sa propre clé publique pour certification.

Pour notre exemple on génère la demande de certification de la clé publique au format PEM :

```
>keytool -certreq -alias olivier -keystore java.keystore
Tapez le mot de passe du Keystore :  xxxxxx
-----BEGIN NEW CERTIFICATE REQUEST-----
MIIBpTCCAQ4CAQAwZTELMAkGA1UEBhMCRlIxDzANBgNVBAgTBkZyYW5j
aXMxDjAMBgNVBAoTBUVwaXRhMQ0wCwYDVQQLEwRMUkRFMRYwFAYDVQQD
MIGfMA0GCSqGSIb3DQEBAQUAA4GNADCBiQKBgQDDOUZp5mQET+gyTXCl
UH9cJ0stbH3ygY/kmLtNvCXGkarGp4rLerH/jQyeFAiQwqZXgR5bayxb
BjDPF60XYYhESCeEvzuc8vmsjB5M44SxvdNUP+Gm90TdSckK4BMpv0uZ
hvcNAQEEBQADgYEAUKcSsm5LNM/yqV9W50+LPw0Mi/IlL1B27b90XLR7
wY7Zm6sRHgkRpOrKDighSbCSiSmR0mMCNTpe8E5+9TY1LJkNtlMZMjfS
kjSaFFLG9vOh4tDTIjpY0vZehED1v+7dS4o=
-----END NEW CERTIFICATE REQUEST-----
```



## 2.9. Être une Authorité de Certification

Etre une Authorité de Certification n'entre pas dans le rôle de Java. Aussi on utilisera un outil hors du monde Java : Open SSL.

L'idée est là encore de créer sa pair de clé et de signer les demandes de certificats avec sa clé privée (cf la doc).

Une fois tout en place, on récupère la demande de certificat et on la signe :

```
>openssl ca -config openssl.cnf -policy policy_anything
-in /tmp/olivier-req.pem -out /tmp/olivier.pem
```

Dans cet exemple, l'AC s'appelle "CA Freenix" (cf transparent suivant).

Le fichier résultat contient plus d'info que la clé signée au format PEM. Il faut donc extraire les dernières lignes pour avoir la clé. Le résultat est rangé dans `olivier_short.pem` dans cet exemple.



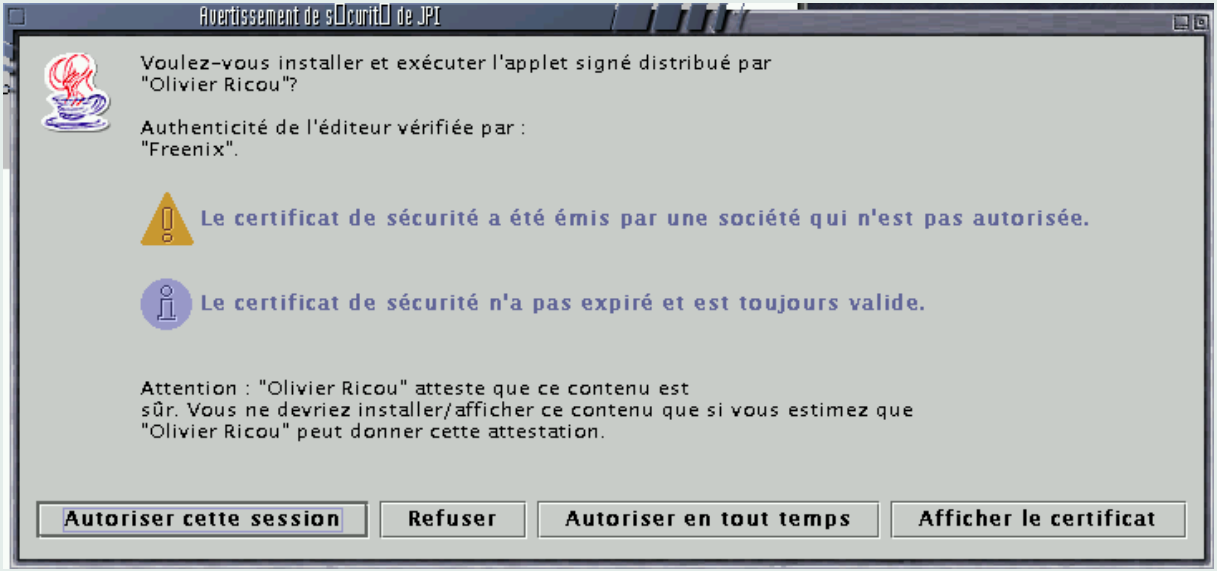
Maintenant que sa clé publique est certifiée, olivier l'intégre dans son trousseau personnel qui sert à signer les jar (la clé publique est inclue dans les jar signés).

```
>keytool -import -alias olivier -file olivier_short.pem\
-keystore java.keystore -trustcacerts

Tapez le mot de passe du Keystore :  xxxxxx
Réponse de certificat installée dans le Keystore

>keytool -list -v -keystore java.keystore
...
Certificat[1]:
Propriétaire : CN=Olivier Ricou, OU=LRDE, O=Epita, L=Par
Émetteur : EMAILADDRESS=postmaster@ns2.freenix.org,
           CN=CA Freenix, O=Freenix, ST=75, C=FR
...
Certificat[2]:
Propriétaire : EMAILADDRESS=postmaster@ns2.freenix.org,
Émetteur : EMAILADDRESS=postmaster@ns2.freenix.org, CN=C
```



## 2.10. Une applet signée certifiée

Une applet stockée dans un jar signé par la clé certifiée donne alors




En ajoutant l'AC Freenix dans le fichier `cacerts` du navigateur, on a comme premier message :

*"Le certificat a été transmis par une société autorisée"*



## 2.11. Fermer un paquetage .jar

Même si l'archiver jar est signée, rien ne garanti qu'un malfaisant ne va pas créer une classe déclarée étant membre de votre paquetage (`package toto.machin;`) et ainsi accéde aux données réservées aux classes de votre paquetage.

Pour éviter cela, on peut fermer un paquetage en ajoutant `Sealed` dans le fichier `manifest` :

```
Manifest-Version: 1.0
Sealed: true
```

Fermer le paquetage ainsi impose que toutes les classes se déclarant en faire parti soient dans le .jar.

On peut limiter la fermeture à des parties du paquetage.

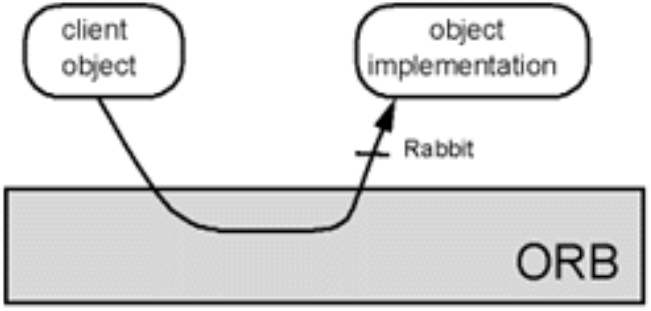
cf http ://java.sun.com/docs/books/tutorial/ext/security/sealing.html



# 3. CORBA et Java

CORBA, Common Object Request Broker Architecture, est une spécification définissant une architecture de programmation objet distribuée.

CORBA a été défini par l' Object Management Group (OMG).




L'Object Request Broker, ORB, est l'abstraction, ou le programme, qui transmet les demandes au serveur et les réponses au client.

On a le même schéma que pour la RMI de Java.



L'intéret de CORBA est

– son indépendance vis à vis des langages, le client et le serveur peuvent chacun utiliser leur langage, CORBA leur demande seulement de pouvoir s'interfacer avec l'interface de CORBA. Parmi les langages compatibles CORBA on a C, C++, Ada, Lisp, COBOL, Python.

– l'adressage abstrait, les clients et les objets (services) n'ont pas besoin de savoir l'adresse physique de l'autre,

– l'interopérabilité. Il existe de nombreuses implémentaions de CORBA faites par différents constructeurs sur différents matériels et OS. Elles communiquent toutes entres. Cela offre aussi la possibilité d'utiliser CORBA sur des réseaux hétérogènes.

On trouve parmi les implémentations de CORBA qui gère Java :

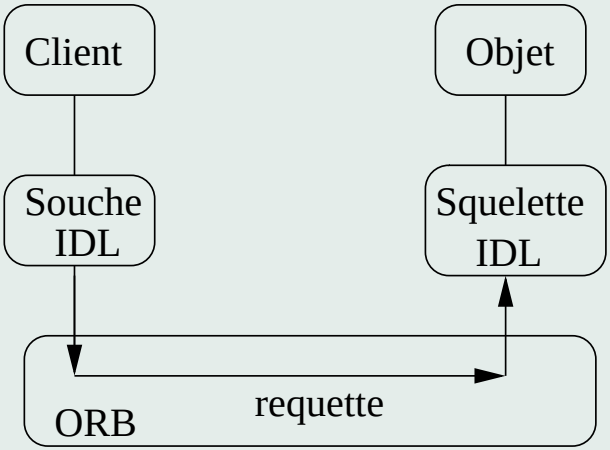
– l'ORB de Java 2 de Sun, ne couvre pas l'ensemble des caractéristiques,
– VisiBroker pour Java de Inprise Corporation, inclus dans Netscape,
– OrbixWeb d'Iona Technologies,
– WebSphere d'IBM.



## 3.1. L'interface

L'interfaçage entre les clients et les objets distants se fait en suivant l'Interface Definition Language, IDL.

A partir du moment où on dispose d'un logiciel effectuant la génération de la souche et du squelette IDL à partir d'un langage, ce langage est compatible CORBA.




```
interface Test {
 short method1(in short param1, in string param2);
 string method2(in short param1, in string param2);
};
```



Le langage de l'IDL est légèrement différent de celui et Java et de C++ :

| IDL | Java | C++ |
|---|---|---|
| module | package | namespace |
| interface | interface | abstract class |
| operation | method member | function |
| attribute | pair of methods | pair of functions |
| exception | exception | exception |

| IDL Type | Java Type |
|---|---|
| boolean | boolean |
| char / wchar | char |
| octet | byte |
| short / unsigned short | short |
| long / unsigned long | int |
| long long / unsigned long long | long |
| float | float |
| double | double |
| string / wstring | String |



On construit les fichiers dont on a besoin avec le programme fourni par Java `idlj` (java 1.4 a un nouveau mécanisme d'où le `-oldImplBase`) :

```
>idlj -fall -oldImplBase Test.idl
>ls
TestHelper.java  Test.idl          TestOperations.java
_TestStub.java   TestHolder.java   Test.java
_TestImplBase.class
```

avec

- TestOperation.java l'interface Java copie de l'interface IDL,
- Test.java l'interface de l'objet Test, elle hérite de l'interface ci-dessus et des différentes interfaces CORBA nécessaires,
- _TestStub.java la souche, elle implémente l'interface Test,
- _TestImplBase.java le squelette, il implémente l'interface Test,
- TestHelper.java founit des méthodes statiques utiles pour insérer ou retirer Test de l'environnement CORBA, pour gérer les flux (streams)...
- TestHolder.java est un conteneur à brancher sur un flux de Test.



## 3.2. Le serveur

On définit ce que fait l'objet :

TestObject.java

```
1  // Extrait du livre "Beginning Java Networking"
2  // chez Wrox
3
4  public class TestObject extends _TestImplBase {
5    public short method1(short param1, String param2)
6    {
7      return param1;
8    }
9
10   public String method2(short param1, String param2)
11   {
12     return param2;
13   }
14 }
```

puis on écrit le serveur qui attendra des connexions de la part de l'ORB :

TestServer.java

```
1  import org.omg.CORBA.*;                    // Step 1
2  import org.omg.CosNaming.*;
3
4  public class TestServer {
5    public static void main(String args[]) {
6      try {
7
8        // Initialize the ORB
9        ORB orb = ORB.init(args, null);     // Step 2
10
11       // Create the Test object
12       TestObject impl = new TestObject(); // Step 3
13
14
15       // Connect to the Naming Service
16       org.omg.CORBA.Object contextObj =   // Step 4
```

```
17        orb.resolve_initial_references("NameService"
18      NamingContext rootContext =
19    NamingContextHelper.narrow(contextObj);
20
21      // Insert the Test object reference in
22      // the Naming Service
23      NameComponent name =
24    new NameComponent("Test", "");        // Step 5
25
26      NameComponent path[] = { name };
27      // Show that the server is doing something
28      System.out.println("TestServer started;" +
29                         "waiting for client...");
30      rootContext.rebind(path, impl);
31
32
33      // Wait for incoming requests
34      java.lang.Object sync =
35        new java.lang.Object();    // Step 6
36      synchronized (sync) {
```



```
37            sync.wait();
38          }
39
40        } catch (Exception e) {
41          System.err.println("Exception : " + e);
42          e.printStackTrace(System.err);
43        }
44      }
45  }
```



Comme pour les RMI il faut commencer par lancer un serveur de nom :

```
# /usr/share/java/bin/tnameserv
Initial Naming Context:
IOR:000000000000002b49444c3a6f6d672e6f72672f436f734e616d
TransientNameServer: setting port for initial object
 references to: 900
Ready.
```

Le port par défaut étant 900, il est nécessaire d'être root pour lancer ce service ou alors il faut choisir un autre port avec l'option `-ORBInitialPort`

On lance ensuite le serveur :

```
>java TestServer
TestServer started;waiting for client...
```

S'il le port du serveur de nom ou s'il n'est pas lancé localement, on lancera le serveur avec les options ad hoc :

```
java TestServer -ORBInitialPort 9900 \
-ORBInitialHost machine
```

### 3.3. Le client

TestClient.java

```
1  import org.omg.CORBA.*;                    // Step 1
2  import org.omg.CosNaming.*;
3
4  public class TestClient {
5    public static void main(String args[]) {
6      try {
7        ORB orb = ORB.init(args, null);     // Step 2
8
9        org.omg.CORBA.Object contextObj =   // Step 3
10       orb.resolve_initial_references("NameService");
11       NamingContext rootContext =
12             NamingContextHelper.narrow(contextObj);
13
14
15       NameComponent name =
```



```
16                  new NameComponent("Test", "");
17          NameComponent path[] = { name };
18
19          org.omg.CORBA.Object obj =
20                  rootContext.resolve(path);   // Step 4
21          Test testObj = TestHelper.narrow(obj);
22
23          short result1 =                      // Step 5
24                  testObj.method1((short) 1, "Test");
25          String result2 =
26                  testObj.method2((short) 1, "Test");
27          System.out.println("Result 1 is " + result1);
28          System.out.println("Result 2 is " + result2);
29          testObj._release();
30        }
31        catch (Exception e) { System.err.println(e); }
32    }
33  }
```



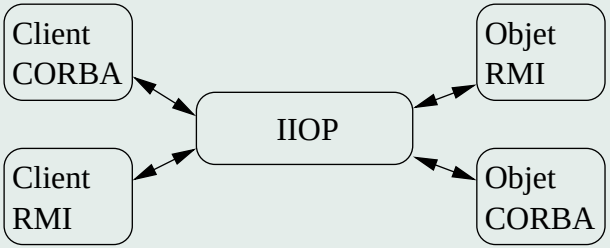
Ce qui donne lorsqu'on lance le client :

```
>java TestClient
Result 1 is 1
Result 2 is Test
```



# 4. RMI-IIOP et CORBA

On vient de voir comment programmer dans le monde des RMI ainsi que dans le monde de CORBA. En utilisant la Remote Method Invocation sur Internet Inter-ORB Protocol, RMI-IIOP, on peut bénéficier des 2 mondes en même temps.




Passer de la RMI à la RMI-IIOP ne demande que peu de modifications.