3.4 Méthode type de développement

En général, on suit toujours les étapes suivantes:

3.4.1 Interfaces distantes

Choisir les informations que l'on doit accéder à distance

Les structurer en objet: interface sans accès direct à des données (uniquement des routines)

Définir l'interface avec les méthodes et les exceptions (RemoteException + éventuellement d'autres)

3.4.2 Classes implémentant les Interfaces Distantes: serveur(s)

Définir des classes qui implémentent :

- Interfaces Distantes
- UnicastRemoteObject (en général)

Enregistrement d'au moins un objet dans le rmiregistry

3.4.3 Clients utilisant les objets distants

Déclaration de références (attribues, etc.) vers des Interfaces Java Distantes (Remote)

Récupération d'au moins un objet par le

rmiregistry

Appels de méthodes classiques, avec récupération des exceptions

3.4.4 Compiler les sources

Clients et serveurs: javac

3.4.5 Précompilation: création Stubs/ Skeletons

Utilisation de rmic pour générer les Stub/Skeletons pour toutes les classes qui implémentent Remote

Il faut, auparavant, que les classes “Remote“ compilent (javac).

3.4.6 Lancer le rmiregistry

Choisir un port (défaut: 1099)

(Voir plus loin: unset CLASSPATH)

Lancer le rmiregistry

3.4.7 Installer le serveur

Lancer un ou plusieurs serveurs

(Voir plus loin:

java.rmi.server.codebase **property**

security policy)

3.4.8 Lancer le ou les clients

Lancer un ou plusieurs clients

On indiquera en général au client, en paramètres de la ligne de commande :

- où (host), et
- sur quel port

le serveur écoute

3.5 Exemple simple: Remote Hello

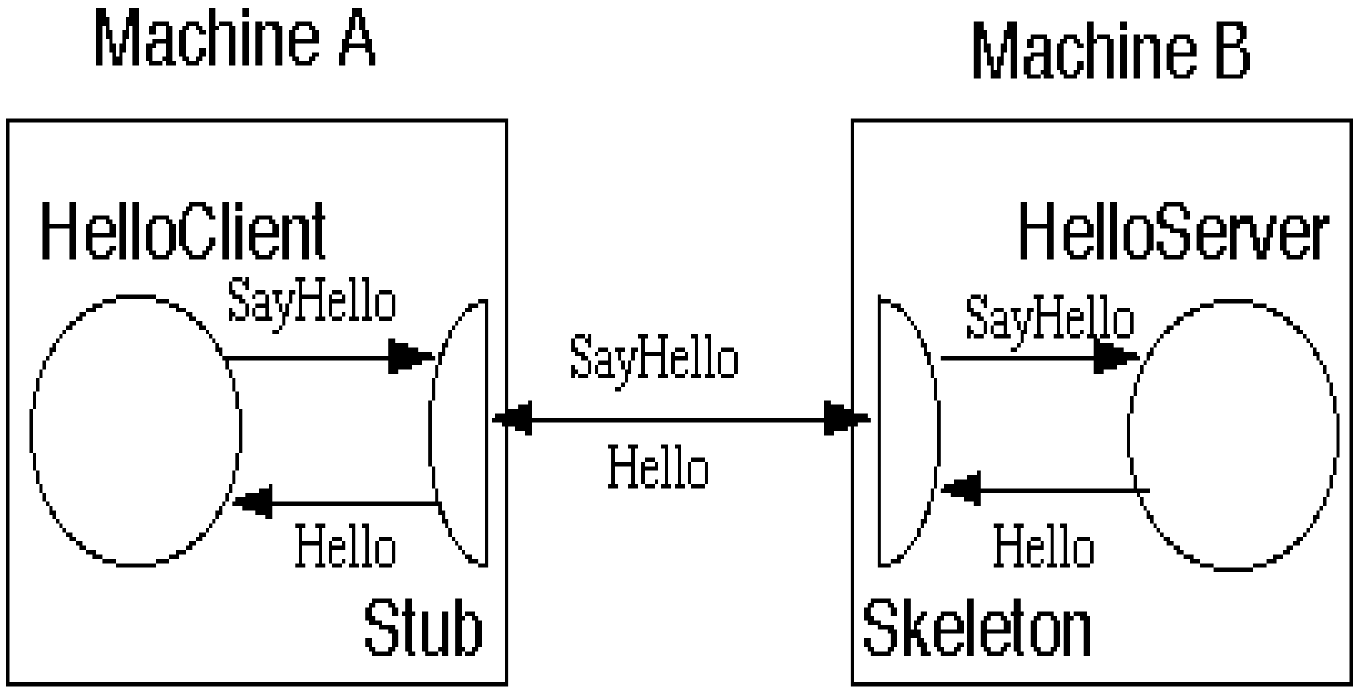

FIGURE 15 Remote Hello world

3.5.1 Remote Interface

```
public interface Hello extends java.rmi.Remote
  {
  String sayHello() throws java.rmi.RemoteException;
  }
```

3.5.2 Serveur HelloWorld

```
import java.net.InetAddress;
import java.rmi.Naming;
import java.rmi.RemoteException;
import java.rmi.RMISecurityManager;
import java.rmi.server.UnicastRemoteObject;

public class HelloServer
    extends UnicastRemoteObject
    implements Hello
  {
  public HelloServer() throws RemoteException {  }

  public String sayHello()
    {
    return  "Hello World from " + getHostName();
    }

  protected static String getHostName()
    {
    try
      {
      return InetAddress.getLocalHost().getHostName();
      }
    catch (java.net.UnknownHostException who)
      {
      return "Unknown";
      }
    }
```

3.5.3 Enregistrement dans le rmiregistry

Création de l'objet accessible à distance, et enregistrement dans le rmiregistry (HelloServerMain.java)

```
public static void main(String args[])
    {
    // Create and install a security manager
    System.setSecurityManager(new RMISecurityManager());

    try
      {
      // Create and register the server object
      HelloServer serverObject = new HelloServer();
          Naming.rebind("rmi://clio.unice.fr/HelloServer",    serverObject );
                                            // Objet Remote ...
                                            // not the remote interface: Hello
                                            // not the stub
                                            // How does it work ???
      // Signal successful registration
      System.out.println("HelloServer bound in registry");
      }
    catch (Exception e)
      {
      System.out.println("HelloServer err: ");
      e.printStackTrace();
      } } }
```

Naming :

```
Naming.rebind("rmi://clio.unice.fr/HelloServer", serverObject );
```

La syntaxe complète pour Naming :

rebind et lookup

est la suivante:

"rmi://host:port/ServerLabel"

“rmi:“, machine, et “port" (si 1099) sont optionnels dans le cas d’un bind et rebind :

```
Naming.rebind( "rmi://clio.unice.fr:1099/HelloServer", serverObject);
        ou
Naming.rebind( "//clio.unice.fr/serverObject);
        ou
Naming.rebind( "HelloServer", serverObject);
```

les méthodes retournent void et throw:

MalformedURLException (not an URL)

RemoteException (registry contact failed)

AccessException (non-local host)

Par contre on indiquera toujours au moins la machine pour un lookup:

```
Hello remote = (Hello) Naming.lookup( "//clio.unice.fr:1099/HelloServer");
Hello remote = (Hello) Naming.lookup( "//clio.unice.fr/HelloServer");
```

sachant que si le port n’est pas spécifié, on utilise celui par défaut : 1099

On ne peut bind ou unbind sur un rmiregistry, que depuis une VM sur le même host (sécurité).

Bien sur: lookup depuis n'importe quel host.

3.5.4 Un client de HelloWorld

```
import java.rmi.*;
import java.net.MalformedURLException;
public class HelloClient
  {
  public static void main(String args[])
    {
    try {
      Hello remote = (Hello) Naming.lookup(
                    "rmi://clio.unice.fr/HelloServer");
      String message = remote.sayHello();
      System.out.println( message );
      }
    catch ( NotBoundException error)
      {
      error.printStackTrace();
      }
    catch ( MalformedURLException error)
      {
      error.printStackTrace();
      }
    catch ( UnknownHostException error)
      {
      error.printStackTrace();
      }
    catch ( RemoteException error)
      {
      error.printStackTrace();
      }
    }
  }
```

Les catch multiples permettent de détecter quelle exception est levée.

3.5.5 Compilation, génération, registry

a) Côté Serveur

Pour Java 2, placer le fichier suivant dans votre home directory:

.java.policy

```
grant {
    // Allow everything for now
    permission java.security.AllPermission;
};
// Ne pas utiliser ce fichier sytématiquement !!
```

Compiler les sources:

javac Hello.java HelloServer.java

** *Hello.java* **

lien, connaissance statique Client/serveur*

Générer les Stubs et les Skeletons

rmic HelloServer

Produit:

HelloServer_Skel.class

HelloServer_Stub.class

Démarrer le rmiregistry

rmiregistry 2001 &

Démarrer le serveur:

java HelloServer

b) Côté Client

Compiler les sources:

javac Hello.java HelloClient.java

On a besoin de l'interface remote (Hello.java)

** *Hello.java* **

lien, connaissance statique Client/serveur*

Rendre la classe HelloServer_Stub.class disponible du côté du client (solution ad hoc):

Copier le fichier HelloServer_Stub.class *du serveur à la machine client (ou NFS),*

Lancer le client:

java HelloClient

La place du serveur est dans le code, mais normalement, il faut utiliser un argument de la ligne de commande.

Notez bien:

Avec Java RMI, on peut créer en local des objets accessibles à distance,

mais on ne peut pas directement:

créer directement à distance des objets accessibles à distance !

On ne peut pas du tout:

Créer directement à distance une JVM

sur une machine pour laquelle on n'a pas

deja une JVM avec un objet remote accessible:

- enregistré dans le registry,
- ou transitivement connu d'un autre objet remote, ... lui même connu.

3.6 Implémentation

Version 1.1.x

3.6.1 Stub

```
// Stub class generated by rmic, do not edit.
// Contents subject to change without notice.

public final class HelloServer_Stub
   extends java.rmi.server.RemoteStub
   implements Hello, java.rmi.Remote
{
   private static java.rmi.server.Operation[] operations = {
      new java.rmi.server.Operation("java.lang.String sayHello()")
   };

   private static final long interfaceHash = 6486744599627128933L;

   // Constructors
   public HelloServer_Stub() {
      super();
   }
   public HelloServer_Stub(java.rmi.server.RemoteRef rep) {
      super(rep);
   }
   // Methods from remote interfaces

   // Implementation of sayHello
   public java.lang.String sayHello() throws java.rmi.RemoteException {
      int opnum = 0;
      java.rmi.server.RemoteRef sub = ref;
      java.rmi.server.RemoteCall call =  / REIFICATION  de l'appel :
sub.newCall((java.rmi.server.RemoteObject)this, operations, opnum, interfaceHash);
      try { // Appel lui-même
         sub.invoke(call);
      } catch (java.rmi.RemoteException ex) {
         throw ex;
```

```
        } catch (java.lang.Exception ex) {
            throw new java.rmi.UnexpectedException("Unexpected exception",
ex);
        };
        java.lang.String $result;
        try {
            java.io.ObjectInput in = call.getInputStream();
            $result = (java.lang.String)in.readObject();
        } catch (java.io.IOException ex) {
            throw new java.rmi.UnmarshalException("Error unmarshaling return",
ex);
        } catch (java.lang.ClassNotFoundException ex) {
            throw new java.rmi.UnmarshalException("Return value class not
found", ex);
        } catch (Exception ex) {
            throw new java.rmi.UnexpectedException("Unexpected exception",
ex);
        } finally {
            sub.done(call);
        }
        return $result;
}
}
```

3.6.2 Skeleton

```
// Skeleton class generated by rmic, do not edit.
// Contents subject to change without notice.
public final class HelloServer_Skel
    extends java.lang.Object
    implements java.rmi.server.Skeleton
{
    private static java.rmi.server.Operation[] operations = {
        new java.rmi.server.Operation("java.lang.String sayHello()")
    };

    private static final long interfaceHash = 6486744599627128933L;

    public java.rmi.server.Operation[] getOperations() {
        return operations;
    }
            // obj: objet cible à appelé, call: appel réifié, avec adres. retour resultat
            // opnum: opération à effectuer
    public void dispatch(java.rmi.Remote obj, java.rmi.server.RemoteCall call, int
opnum, long hash) throws java.rmi.RemoteExcepti
on, Exception {
        // Exceptions pass through, to be caught, identified and marshalled

        if (hash != interfaceHash)
            throw new java.rmi.server.SkeletonMismatchException("Hash
mismatch");
        HelloServer server = (HelloServer) obj;
        switch (opnum) {
        case 0: { // sayHello
            call.releaseInputStream();
            java.lang.String $result = server.sayHello(); // Appel effectif
            try {
                java.io.ObjectOutput out = call.getResultStream(true); // There is a result
                out.writeObject($result);          // Send back the result: serialization first
            } catch (java.io.IOException ex) {
                throw new java.rmi.MarshalException("Error marshaling return", ex);
            };
            break;
            }
        default:
            throw new java.rmi.RemoteException("Method number out of range");
        }}}
```

Version sans skeleton:

Reflexion Java :

public final class Class

```
public final class Class {
  public static Class forName( String className );
  public Object newInstance();
  public String getName();
  public Class getSuperclass();
  public Class[] getInterfaces();
  public int getModifiers();
  public Class[] getDeclaredClasses();
  public Field[] getDeclaredFields();
  public Method[] getDeclaredMethods();
  public Constructor[] getDeclaredConstructors();
  public Field getDeclaredField( String name );
  public Method getDeclaredMethod( String name, Class parameterTypes[]);
  public Constructor getDeclaredConstructor( Class parameterTypes[] );...
}
```

classes Array, Constructor, Field, Method, Modifier

```
public final class Method {
```

getName(), getParameterTypes(), getReturn-Type(), hashCode()

```
  Object invoke(Object obj, Object[] args),
...
}
```

Exemples:

```
Class thisClass = this.getClass(); // method getClass of Object
Class thisClass2 = Class.forName("Triangle");

Method[] methodsOfThisClass = thisClass.getDeclaredMethods();

concatMethod = c.getMethod("concat", parameterTypes);
result = (String) concatMethod.invoke(firstWord, arguments);// Effective Call
                                              // Target Object    Effective Arguments
```

Utilisation pour RMI :

1. On encode dans Call (stub) le nom ou Code de la méthode à appeller.
2. Dans le Runtime RMI du coté du serveur, à la place du skeleton, on trouve:

3.

--> • Récupérer le Code de la méthode

--> • Récupérer la méthode (objet de type Method)

--> • Exécuter la méthode:

```
result = toBeCalledMethod.invoke(onObj, withArguments);
```

Remplace le code généré:

```
...

switch (opnum) {
	case 0: { // sayHello
	    call.releaseInputStream();
	    java.lang.String $result = server.sayHello(); // Appel effectif
	    try {
...
```

Pourquoi a-t-on toujours besoin de Stub client ?

- Stub :

Typage, polymorphisme (interface d'accès) entre objets locaux et objets distants,

+ Serialization , Encodage

- Skeleton : le problème est différent, simplement un problème technique d'appel de la méthode

En résumé :

Skeleton spécifique:

v = obf.foo (param);

Pour se passer de skeleton :

Dans le stub :

Method m = ... getMethod ("foo");

Dans le runtime RMI ("skeleton générique"):

r = m.invoke (obj, p);

CHAPITRE 4 Techniques et utilisations avancées de RMI

4.1 Sécurité et chargement dynamique de code

4.1.1 Rappels et cadre

Codebase:

Un codebase est un endroit depuis lequel charger du code dans une VM.

CLASSPATH est un codebase local,
autre exemple de codebase:

http://myhost/~myusrname/myclasses/

Le codebase d'une applet est toujours relatif à l'URL depuis laquelle elle a été chargée.

Security Manager

Toute JVM qui doit downloader du code a besoin (Java 2) d'un Security Manager (SM).

Principe d'un Security Manager:

si aucun security manager n'est installé, alors seules les classes accessibles depuis le CLASSPATH peuvent être chargées

un security manager vérifie différents critères sur les classes au chargement.

Une façon de charger un SecurityManager uniquement s'il n'y en a pas déjà un :

```
if (System.getSecurityManager() == null) {
  System.setSecurityManager(
        newRMISecurityManager());
}
```

4.1.2 Security Manager

Un SecurityManager doit être utilisé avec RMI, sinon les clients et les serveurs RMI ne peuvent pas charger de code autre que directement par le CLASSPATH.

Dans le cas d'une application générale (ni NFS, ni une applet), il faut un SecurityManager dans le client et le serveur.

Les clients RMI download du code: les stub

Dans le cas d'une applet:

> on peut se contenter d'un SM dans le serveur car le client dispose déjà d'un SM: celui du browser qui a chargé l'applet

4.1.3 Security file et RMI

Pour obtenir les sécurités de base nécessaire à RMI

.java.policy

```
grant
  {
     permission java.net.SocketPermission "*:1024-65535",
            "connect,accept,resolve";
     permission java.net.SocketPermission "*:1-1023",
            "connect,resolve";
  };
```

Deuxième parti pour les ports réservés

Ou toutes les permissions:

```
grant {
     // Allow everything for now
     permission java.security.AllPermission;
};
```

4.1.4 Applets signés et RMI

Par défaut, une applet (voir dessin en 4.2.1, Figure 19) ne peut communiquer (tcp, donc RMI) qu'avec le host qui l'a chargée

Pour obtenir son nom: getCodeBase().getHost()

Si l'on veut faire une communication vers une objet RMI se trouvant sur un autre host, il faut faire une applet signée

Voir cours sécurité DESS Télécom

Une applet ne peut pas non-plus par défaut communiquer par RMI avec le host qui l'a chargée (celui sur lequel elle est en train de s'exécuter).

4.1.5 codebase et RMI

Avec RMI, propriété Java spéciale :

java.rmi.server.codebase

Au lancement du client (mais aussi du seveur si nécessaire, voir plus loin) on va spécifier le java rmi server codebase, de sorte que l'on puisse télécharger du code (byte code de classes Java) si nécessaire.

Par exemple: le code d'implémentation du stub.

codebase est une propriété qui représente une ou plusieurs URL depuis lesquelles des classes peuvent être téléchargées.

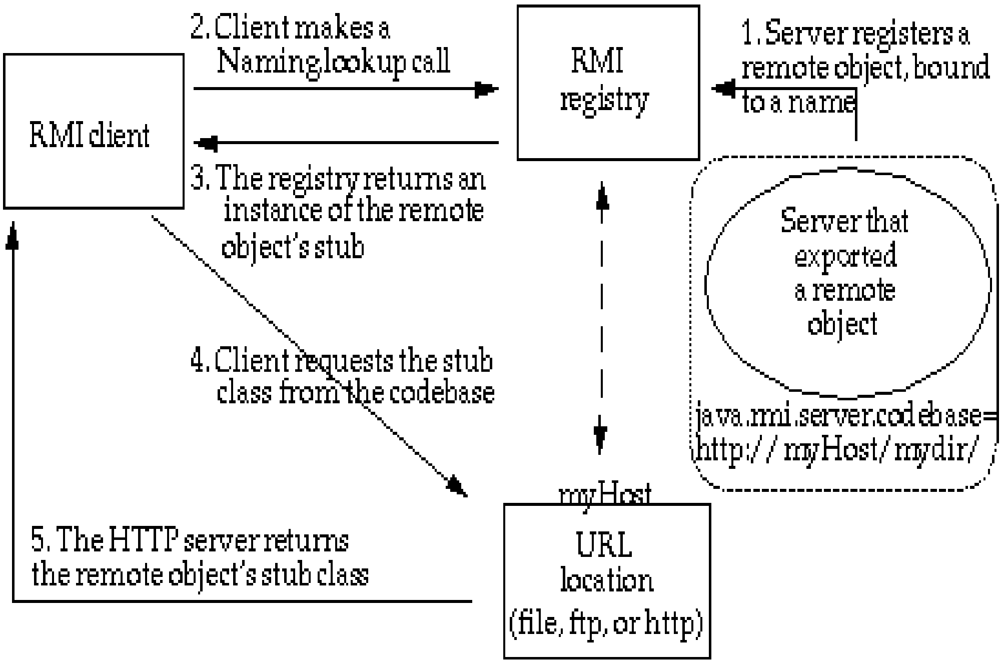

FIGURE 16 Téléchargement de stub en RMI

1. bis: si configuree, le RMIregistry charge le code depuis le HTTP server

6. RMI client peut (enfin) faire son appel distant!

Codebase = label (étiquette attachée au code)

--> Dessin avec n JVM au Tableau

Exemple d'utilisation:

java -Djava.rmi.server.codebase=http://myhost/~myusrname/myclasses/ examples.hello.HelloImpl

Ou plusieurs séparés par des blanc et entre “ ” :

-Djava.rmi.server.codebase="http://webfront/myStuff.jar http://webwave/myOtherStuff.jar http://webwave/dir/"

Il faut un / si c'est un répertoire et non un fichier.