

RMI

contenu de la section

RMI.....1

CONTENU DE LA SECTION.....2

LES ORBS : GÉNÉRALITÉS (1).....4

les objectifs et les principes4

le modèle.....4

LES ORBS : GÉNÉRALITÉS (2).....5

le design pattern « ORB ».....5

les éléments essentiels d'un ORB.....5

LES ORBS : ÉLÉMENTS D'INGÉNIÉRIE.....6

la mise en oeuvre d'une invocation de méthode.....6

LES ORBS : ENVIRONNEMENT DE DÉVELOPPEMENT (1).....7

l'objectif.....7

RMI : GÉNÉRALITÉS.....8

un peu de terminologie RMI.....8

les packages et les classes.....8

la construction d'une application.....8

UN EXEMPLE SIMPLE (1).....9

la définition de l'interface distante.....9

la définition de l'implémentation de l'interface (servant).....9

UN EXEMPLE SIMPLE (2).....10

la définition du serveur.....10

la définition du client.....10

LE SERVICE DE NOMMAGE RMI.....11

l'interface générique du service.....11

une implémentation par défaut.....11

la création-obtention d'un registry.....11

L'INSTANCIATION DES STUBS ET SKELETON (1).....12

le mécanisme de base.....12

L'INSTANCIATION DES STUBS ET SKELETON (2).....13

le chargement des stubs à travers le réseau13

QUELQUES COMPLÉMENTS SUR RMI (1).....14

le passage des objets en argument ou par valeur de retour.....14

l'appel du client via une interface de callbacks.....14

la notification de non référencement.....14

QUELQUES COMPLÉMENTS SUR RMI : LE FRAMEWORK D'ACTIVATION (1).....15

le modèle.....15

la programmation.....15

java avancé	chapitre 07
QUELQUES COMPLÉMENTS SUR RMI : LE FRAMEWORK D'ACTIVATION (2).....	16
<i>la définition de l'interface distante.....</i>	<i>16</i>
<i>la définition de l'implémentation de l'interface (servant).....</i>	<i>16</i>
QUELQUES COMPLÉMENTS SUR RMI : LE FRAMEWORK D'ACTIVATION (3).....	17
<i>la définition du serveur.....</i>	<i>17</i>
QUELQUES COMPLÉMENTS SUR RMI : LE FRAMEWORK D'ACTIVATION (4).....	18
<i>quelques classes.....</i>	<i>18</i>

Les ORBs : généralités (1)

les objectifs et les principes ...

- structurer le code selon la logique applicative
- externaliser le code *technique* de communication dans des objets spécialisés (les stubs,)

le modèle

- objets interagissant via des invocations de méthodes
- interface = ensemble d'attributs et opérations d'un objet de service

```
lObject1 objRef1= annuaire.getReference(objName1);  
lObject1 objRef2= annuaire.getReference(objName2);
```

```
try {  
    x=objRef1.meth2(.....);  
    x=x+objRef2.meth2(.....);  
    .....  
} catch(XXException e) { .... }
```

```
interface lObject1 {  
    cTyp1 ch1;  
    .....  
    cTypm chm;  
  
    public rTyp1 meth1(aTyp11,aTyp12,...);  
    public rTyp2 meth2(aTyp21,aTyp22,...);  
    public rTypn methn(aTypn1,aTypn2,...);  
};
```

```
Object1 obj1=new Object1(.....);  
Object1 obj2=new Object1(.....);  
annuaire.register(obj1,objName1);  
annuaire.register(obj2,objName2);
```

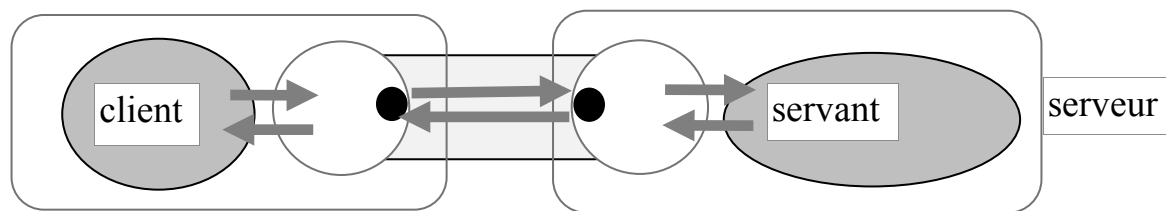
classe Object1 implémente l'interface lObject1

- **RMI** (Remote Method Invocation) ⇔ hypothèse "tout JAVA" (langage, JVM et API)
- **CORBA** (Common Object Request Broker Architecture) ⇔ pas d'hypothèse sur langage de programmation, sur infrastructure

Les ORBs : généralités (2)

le design pattern ORB

- introduction de représentants locaux (proxies, stubs, skeletons) chargés de transformer l'invocation de méthode sur l'objet en un mécanisme de communication effectif.



- stubs, skeletons
- marshalling, unmarshalling
- servants vs serveurs

les éléments essentiels d'un ORB

- un ensemble de messages échangés (nature et structure)
- un format d'encodage des informations dans les messages
- un protocole de transport
- un mapping des messages sur des protocoles de transport
- un mapping entre les messages et les objets
- un service d'annuaire

exemples : GIOP

exemples : sérialisation, CDR, XDR

exemples : TCP

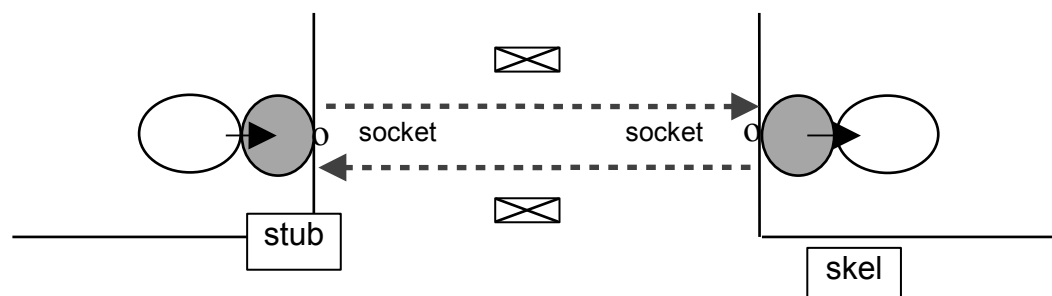
exemples : JRMP, (IIOP CORBA)

stubs/skel

RMI Registry (URL), CORBA NameService, ...

Les ORBs : éléments d'ingénierie

la mise en oeuvre d'une invocation de méthode



partie client (stub) :

marshalling et envoi de message

- *allocation du buffer*
- codage du message de requête dans le buffer :
objet destinataire + méthode + arguments
- émission du message de requête
- *libération du buffer*

reception du message de réponse et unmarshalling

- *allocation du buffer de réponse*
- réception du message de réponse
- récupération des valeurs et retour
 - si normal : *libération buffer* + retour normal de la méthode
 - si exception : *libération buffer* + génération de l'exception

partie serveur (skeleton) :

reception du message de réponse et unmarshalling

- *allocation du buffer de réception de la requête*
- réception du message de requête
- décodage du message de requête
objet destinataire + méthode + arguments
- *libération du buffer de réception de la requête*
- invocation de la méthode

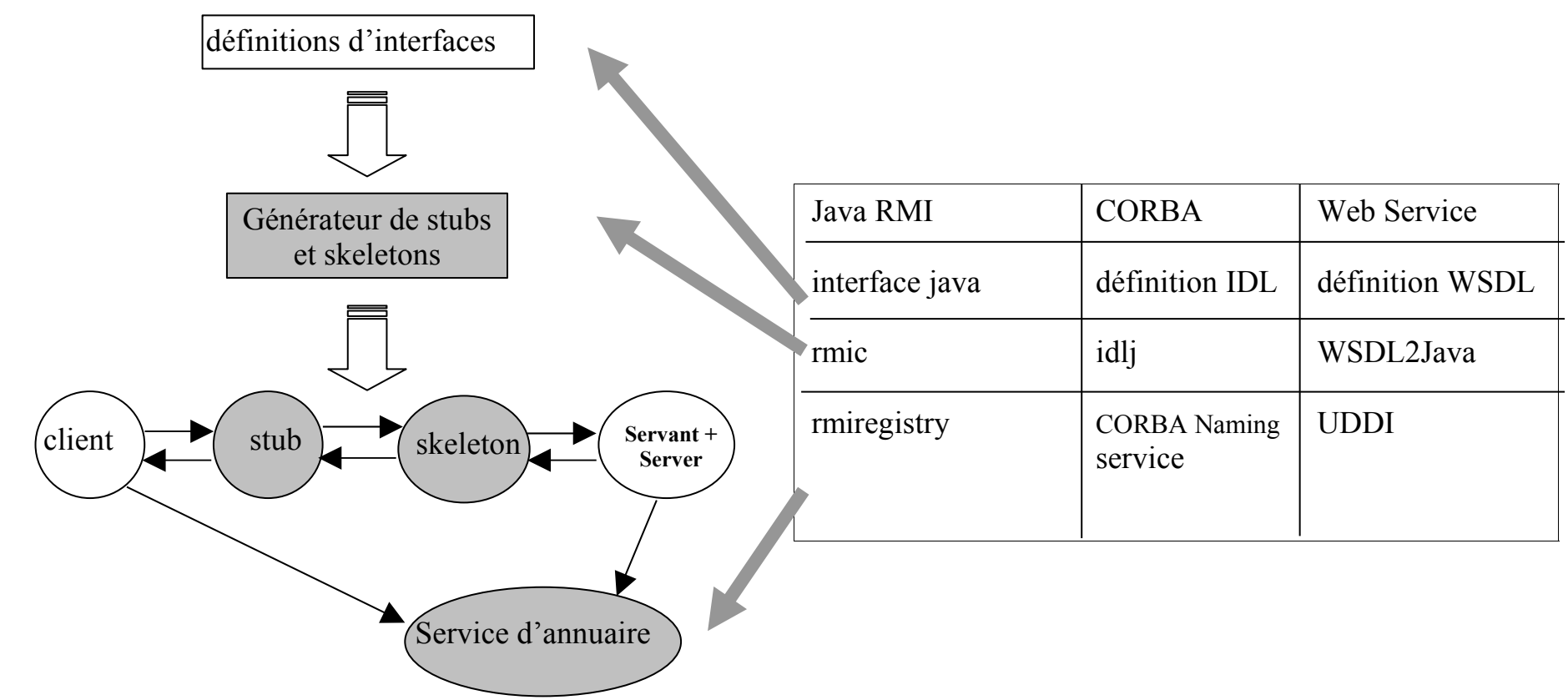
marshalling et envoi de message

- réception de la réponse (retour de la méthode)
- *allocation du buffer d'émission de la réponse*
- copie des valeurs de retour dans le buffer
 - si exception => copie de l'exception
- émission du message de retour
- *libération du buffer d'émission de la réponse*

les ORBs : environnement de développement (1)

l’objectif

- offrir un générateur de classes de stubs et skeletons + classes utilitaires (compilateurs d’interfaces, assistants,)
- offrir les services de base : annuaires,
- offrir des outils de déploiement



RMI : généralités

un peu de terminologie RMI

- interface java.rmi.Remote interface de tag autorisant l'accès via RMI : surtype des interfaces RMI et des stubs client
- service de nom ⇔ registry (rmiregistry)
- enregistrement ⇔ Naming.bind(IdString,servantRef)
- obtention d'une référence ⇔ Naming.lookup(IdString)

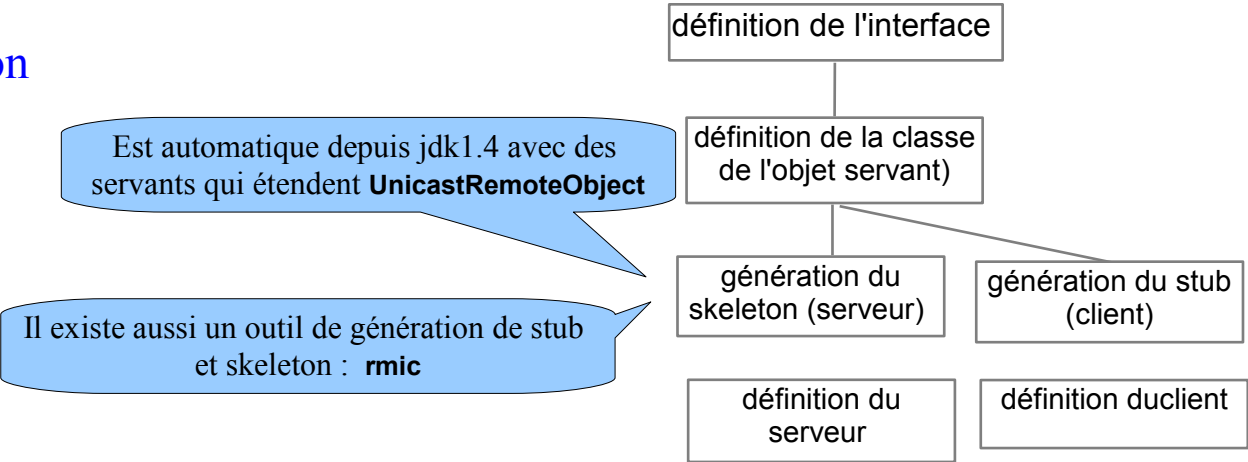
les packages et les classes

- packages : java.rmi, java.rmi.server, java.rmi.registry, ...
- classes d'exception : java.rmi.RemoteException

la construction d'une application

- écrire l'interface de service
- écrire le servant
- écrire le serveur (et les clients)

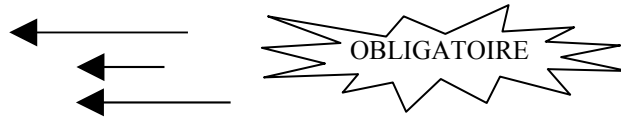
- dans le site du serveur
 - lancer le registry
 - lancer le serveur
- dans le/les sites clients
 - lancer le/les clients



Un exemple simple (1)

la définition de l'interface distante

```
import java.rmi.*;
public interface IEcho extends Remote {
    public String echo(String s) throws RemoteException;
    public String echoLast( ) throws RemoteException;
}
```

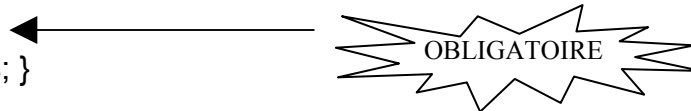


la définition de l'implémentation de l'interface (servant)

```
import java.rmi.*;
import java.rmi.server.*;
import java.net.*;
```

UnicastRemoteObject instancie dans son constructeur sans argument un stub et skeleton. Il peut générer une **RemoteException**

```
public class Echo extends UnicastRemoteObject implements IEcho {
    private String last = "";
    public Echo( ) throws RemoteException { }
    public String echo (String s ) { last = s ; return s ; }
    public String echoLast( ) { return last ; }
}
```



Un exemple simple (2)

la définition du serveur

import

```
public class EchoServer {
    public static void main(String[] args) {
        try {
            LocateRegistry.createRegistry(1099);
            Echo h1=new Echo( );
            Naming.rebind("rmi://Winnt/echo1",h1);
        } catch (RemoteException e) { System.out.println("Pb in EchoServer : " + e); }
        catch (Exception e) { System.out.println("Pb in EchoServer : "+e); }
    }
}
```

création d'un registry écoutant sur le port 1099

Naming.bind/rebind → lancement du thread
dans le skeleton associé au servant

la définition du client

import

```
public class EchoClient{
    public static void main(String[] args) {
        try {
            IEcho h1=(IEcho) Naming.lookup("rmi://Winnt/echo1");
            h1.echo("bonjour les amis !!");
            System.out.println("Echo Client : "+ h1.echoLast());
        } catch (RemoteException e) { System.out.println("Pb in Echo Client : "+e);
        } catch (MalformedURLException e) { System.out.println("Pb in Echo Client : "+e);
        } catch (NotBoundException e) { System.out.println("Pb in Echo Client : "+e); }
    }
}
```

ne pas oublier d'utiliser l'interface de service !!!

Le service de nommage RMI

l'interface générique du service

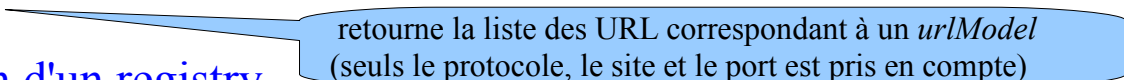
- l'interface `java.rmi.registry.Registry`
- déclaration des méthodes : `lookup()`, `bind()`, `rebind()`, `unbind()`, `list()`

une implémentation par défaut

- La classe `java.rmi.Naming` (ne dérive pas de l'interface `java.rmi.Registry` !!!)
- classe *final* implémentant un serveur de noms sous forme d'URL

```
static Remote lookup(String Name) throws RemoteException,NotBoundException, MalformedURLException,UnknownHostException;  
static void bind(String name,Remote obj) throws RemoteException,AlreadyBoundException, MalformedURLException, .....;  
static void rebind(String name,Remote obj) throws RemoteException,MalformedURLException,UnknownHostException;  
static void unbind(String name) throws RemoteException,NotBoundException,MalformedURLException,UnknownHostException;
```

```
static String[ ] list(String urlModel) throws RemoteException,NotBoundException,MalformedURLException,UnknownHostException;
```



retourne la liste des URL correspondant à un *urlModel*
(seuls le protocole, le site et le port est pris en compte)

la création-obtention d'un registry

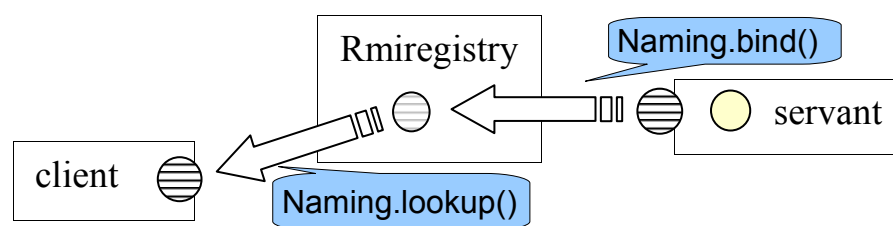
- La classe `LocateRegistry`

```
static Registry getRegistry( ) throws RemoteException;  
static Registry getRegistry(int port ) throws RemoteException;  
.....  
static Registry createRegistry(int port) throws RemoteException;  
.....
```

L'instanciation des stubs et skeleton (1)

le mécanisme de base

- `UnicastRemoteObject.exportObject(Remote r)` → instanciation du skeleton et du stub
- `Naming.bind()` → écriture du stub sérialisé dans le registry (`writeObject()`)
- `Naming.lookup()` → lecture du stub sérialisé (`readObject()`)
- `UnicastRemoteObject.exportObject` impose que :
 - le .class du skeleton soit dans le CLASSPATH du serveur
 - le .class du stub soit dans le CLASSPATH du serveur
- `Naming.bind` impose que :
 - le .class du stub soit dans le CLASSPATH du rmiregistry et du serveur et du client
- `Naming.lookup` impose que :
 - le .class du stub soit dans le CLASSPATH du client.
- possibilité de chargement de classes (en particulier le [.class d'un stub](#)) à partir d'un site distant (`RMIClassLoader`) :
le chargement de classes impose l'existence d'un serveur http sur le site des contenant le .class du stub
indication du serveur http via la propriété : `java.rmi.server.codebase` ou manuellement



L'instanciation des stubs et skeleton (2)

le chargement du .class des stubs à travers le réseau

- il impose l'installation d'un Security Manager (et donc d'un fichier policy)
- le fichier policy est indiqué par la propriété : ***-Djava.security.policy="nom fichier sécurité"***
- la forme du fichier policy :

```
/* all permissions */  
grant {  
    permission java.security.AllPermission;  
};
```

ou bien

```
grant {  
    permission java.net.SocketPermission "":1024-65535,"connect, accept";  
    permission java.net.SocketPermission "":80,"connect";  
};
```



numéro de port du serveur http

Quelques compléments sur RMI (1)

le passage des objets en argument ou par valeur de retour

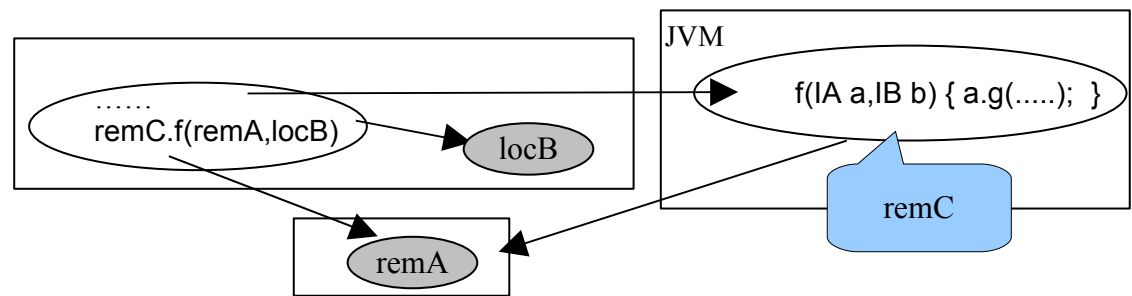
- les arguments d'un appel RMI sont sérialisés :
 - la référence d'un objet local est transmise par sérialisation de l'objet local
 - la référence d'un objet distant est transmise par sérialisation du stub (obtenu par Naming.lookup)

un objet local non sérializable ne peut être passé en argument d'un appel rmi

l'appel du client via une interface de callbacks

- le client doit implémenter une interface `java.rmi.Remote`
- le client crée un stub et un skeleton :

```
Remote UnicastRemoteObject.exportObject(this, 0);
RemoteStub UnicastRemoteObject.exportObject(this);
```
- le client enregistre le stub auprès du servant
- le servant doit pour cela disposer d'une méthode permettant au client de s'enregistrer (par exemple `hRef.register(UnicastRemoteObject.exportObject(this, 0));`);



Seule la 1ere méthode génère automatiquement la classe de stub.
Le 2ème argument est un numéro de port (généralement inutilisé)

Les 2 méthodes retournent une référence sur le stub

la notification de non référencement

- notification des servants qui implémentent l'interface `java.rmi.server.Unreferenced` lorsque l'infrastructure estime qu'un servant n'est plus référencé par des clients

méthode `void unreferenced()`

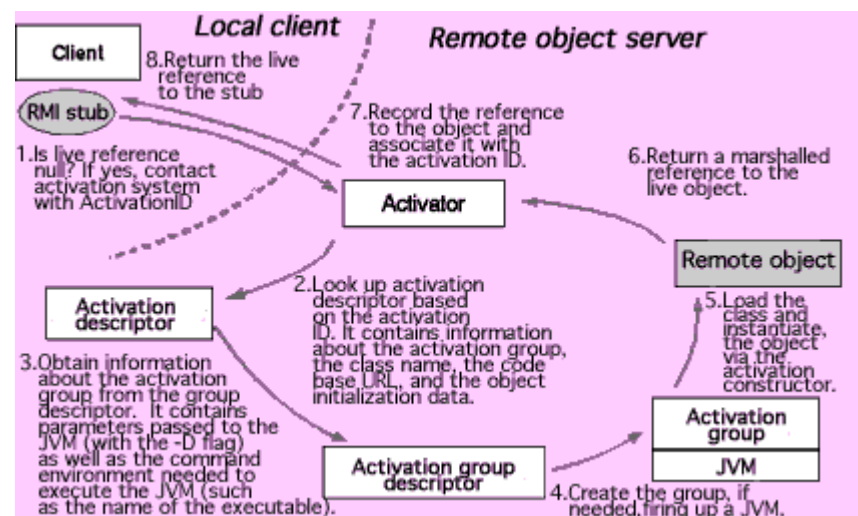
Quelques compléments sur RMI : le framework d'activation (1)

le modèle

- un descripteur de servant est enregistré auprès du service d'activation : un dameon Activator (rmid)
- le service d'activation déclenche l'instanciation du servant lorsque cela est nécessaire.
- 2 types de descripteur
 - ActivationGroupDesc :: les caractéristiques de l'environnement d'exécution (paramètres spécifiques de la JVM)
 - ActivationDesc :: les caractéristiques du servant (classe, données spécifiques)
- spécifier au service d'activation :
 - la JVM à utiliser et l'objet à exécuter
 - enregistrer l'ID de l'objet à exécuter
 - exporter l'objet

la programmation

- le servant :
 - implémente l'interface `java.rmi.Remote`
 - étend `java.rmi.server.Activatable` et non `java.rmi.server.UnicastRemoteObject`
 - définit un ctor à 2 arguments (voir exemple)
- le serveur :
 - installe un `SecurityManager`
 - crée un `ActivationGroupDesc` et un `ActivationDesc`
 - enregistre les informations dans l'Activator et le `rmiregistry`



extrait de l'article "Activatable Jini Services" de F. Sommers
(disponible sur le site javaworld à l'adresse
<http://www.javaworld.com/javaworld>)

Quelques compléments sur RMI : le framework d'activation (2)

la définition de l'interface distante

```
package tps.rmi;
```

```
import java.rmi.*;  
public interface Counter extends Remote {  
    public void increment( ) throws RemoteException;  
    public long value( ) throws RemoteException;  
}
```



la définition de l'implémentation de l'interface (servant)

```
package tps.rmi;
```

```
import java.rmi.*;  
import java.rmi.activation.*;  
import java.net.*;
```

```
public class CounterI extends Activatable implements Counter {  
    private long val=0;  
    public CounterI( ActivationID id, MarshalledObject obj) throws RemoteException { super(id, obj); }  
    public void increment( ) { val++; }  
    public long value( ) { return val; }  
}
```



Il s'agit essentiellement d'un conteneur
stockant les informations nécessaires au
constructeur de l'implémentation

Quelques compléments sur RMI : le framework d'activation (3)

la définition du serveur

```
package tps.rmi;
```

```
import java.rmi.*;  
import java.rmi.activation.*;  
.....
```

```
public class CounterServer {  
    public static void main(String[] args) {  
        try {  
            Properties props = new Properties();  
            props.put("java.security.policy", "/home/fho/activation/policy");  
            ActivationGroupDesc.CommandEnvironment.ace = null;  
            ActivationGroupDesc grp = new ActivationGroupDesc(props, ace);  
            ActivationGroupDesc agi = ActivationGroupDesc.getSystem().registerGroup(grp);  
  
            String location = "file:///home/fho/activation/bin/";  
            MarshalledObject obj = null;  
            ActivationDesc desc = new ActivationDesc(agi, "tps.rmi.CounterI", location, obj);  
  
            ICounter h1 = (ICounter ) Activatable.register(desc);  
            Naming.rebind("rmi ://Winnt/counter1",h1);  
  
        } catch (RemoteException e) { System.out.println("Pb in CounterServer : "+e); }  
        catch (AlreadyBoundException e) { System.out.println("Pb in CounterServer : "+e); }  
    }  
}
```

on indication du répertoire dans lequel se trouve le fichier policy définissant la politique sécuritaire

on utilise la JVM par défaut avec les options par défaut

ne pas oublier le / terminal

le servant n'a besoin d'aucune information pour s'instancier

Quelques compléments sur RMI : le framework d'activation (4)

quelques classes

- la classe `java.rmi.MarshalledObject` contient la forme sérialisée d'un autre objet
- son API est :
`MarshalledObject(Object obj);`
`Object get();`
- la classe `java.rmi.activation.ActivationDesc` contient les informations nécessaires à l'activation d'un objet : le `groupID`, la classe de l'objet, la localisation de la classe, les données d'initialisation

```
ActivationDesc(ActivationGroupID groupID, String className, String location, MarshalledObject data);  
ActivationDesc(ActivationGroupID groupID, String className, String location, MarshalledObject data, boolean restart);  
ActivationDesc(String className, String location, MarshalledObject data);  
ActivationDesc(String className, String location, MarshalledObject data, boolean restart);
```

- la classe `java.rmi.activation.ActivationGroupDesc` contient les informations : le nom du groupe, la localisation du code, les données d'initialisation

```
ActivationGroupDesc(Properties overrides, ActivationGroupDesc.CommandEnvironment cmd);  
ActivationGroupDesc(String className, String location, MarshalledObject data, Properties overrides,  
ActivationGroupDesc.CommandEnvironment cmd);
```

- la classe `java.rmi.activation.ActivationGroupDesc.CommandEnvironment`

```
ActivationGroupDesc.CommandEnvironment(String cmdpath, String[] argv);
```