Introduction à JAVA RMI

Applications réparties

Chapitre 3

Polytech Marseille

Département Informatique

Formation par alternance / HUGo

5ème année

Plan

- Introduction
- Architecture logique
- Principe de programmation
- Un exemple
- Fabrique d'objets
- Appels en retour

Bibliographie

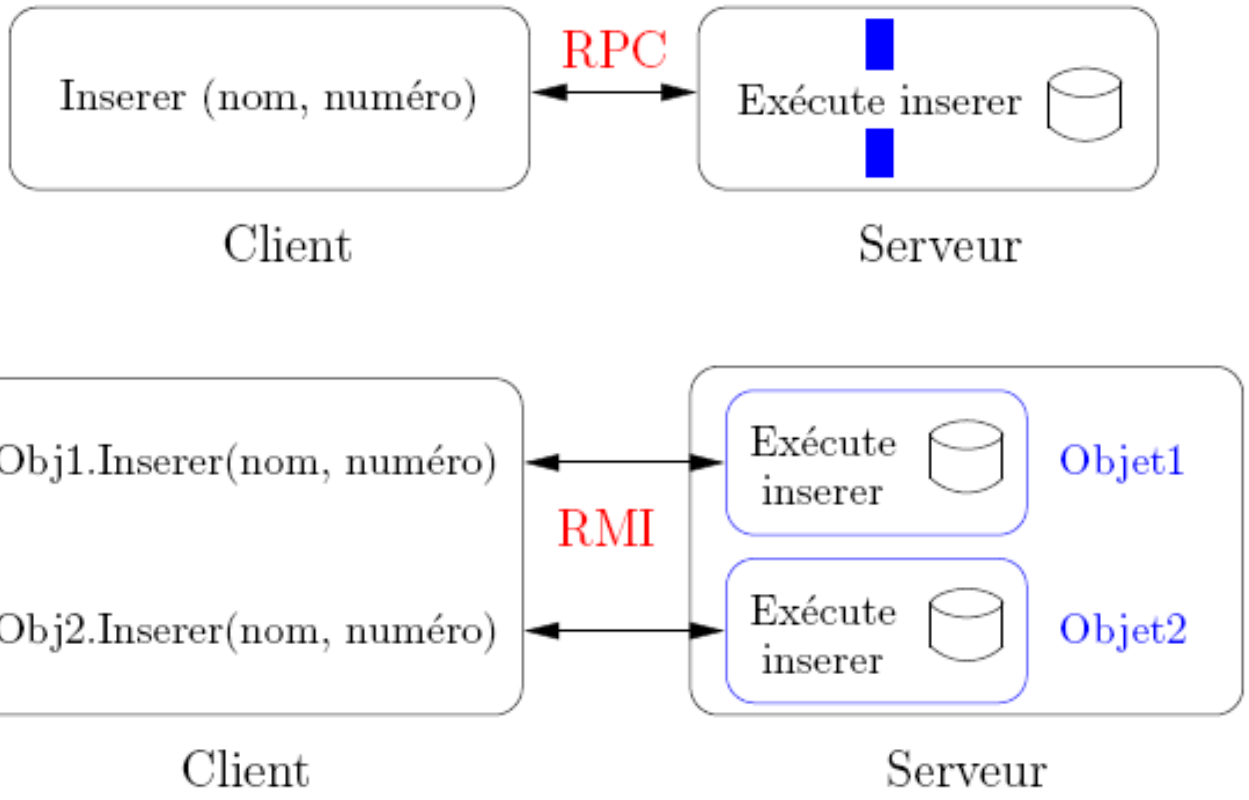
- Documentation officielle en ligne (Oracle)
- A. Fron. "*Architectures réparties en JAVA*", Dunod 2007.
- G. Roussel, E. Duris, …, "*Java et Internet : concepts et programmation*", Vuibert.
  - Tome 1 : côté client, 2002, ISBN 2-7117-8689-7.
  - Tome 2 : côté serveur, 2007, ISBN 2-7117-8690-0.

Des RPC aux RMI

- Les RPC
  - Avantage principal : abstraction (masquage des communications).
  - Limitations :
    - Structure d'application statique, Schéma synchrone
    - Relativement difficile à mettre en œuvre.
- Programmation par objet des applications reparties
  - Avantages :
    - Encapsulation : interface bien définie ; état interne masqué.
    - Classes et instances : génération d'exemplaires selon un modèle.
    - Héritage : spécialisation → récupération et réutilisation de code
    - Polymorphisme : objets d'interface compatible interchangeables
    - Facilite l'évolution et l'adaptation des applications.

Des RPC aux RMI

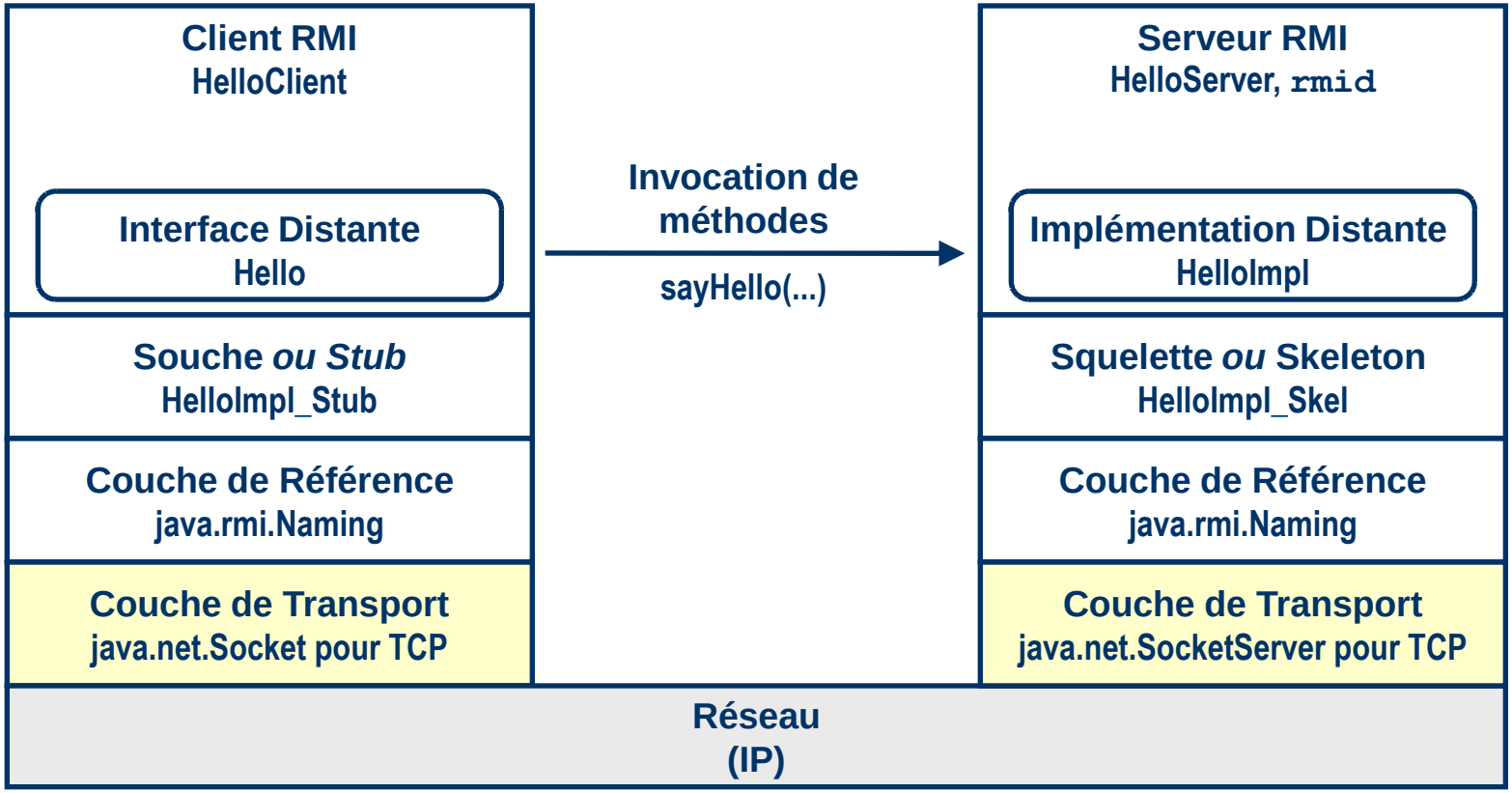
- RPC vs. RMI



Java RMI (Remote Method Invocation)

- Principe :
  - Le développeur fournit :
    - Les interfaces (en Java),
    - Les *implémentations* des interfaces (classes des futurs servants),
    - Le programme du serveur, *instanciant* les objets servants,
    - Le programme du client, *invoquant* ces objets servants.
  - L'environnement Java fournit :
    - Un générateur de talons nommé **rmic**,
    - Un service de noms (`Object Registry`),
    - Un ***middleware*** pour l'invocation à distance (ensemble de classes),
    - La faculté d'exécuter du code généré ailleurs.

Java RMI
Architecture logique (runtime RMI)



Principes de programmation

- Grandes lignes :
  - L'**interface** avant le programme.
  - Réalisation de la classe qui implémente l'interface (**classe des futurs objets serveurs / servants**).
  - Réalisation du programme qui crée le servant (main **serveur**).
  - Réalisation du programme qui invoque à distance des méthodes du servant (main **client**).

Principes de programmation

- 1) Interface d'un objet accessible à distance
  - Doit être publique
  - Doit étendre l'interface `java.rmi.Remote` (elle est vide)
  - Chaque méthode émet l'exception `java.rmi.RemoteException`
- **Passage d'objets** en paramètre / en retour
  - Locaux, passage par valeur
    - `Serializable` (ou `Externalizable`) (sinon erreur)
  - Distants, passage par référence (distante)

  Pour les **types simples** (int, float, …), passage par valeur

Principes de programmation

- 2) Réalisation d'une classe distante (classe du servant)
  - Doit implémenter une interface distante (`Remote`).
  - Doit étendre la classe `java.rmi.server.UnicastRemoteObject`
    - Principal intérêt : le constructeur retourne une instance du talon (utilisant le résultat de *rmic*)
    - En fait, `UnicastRemoteObject` est l'un des possibles parmi les descendants de `java.rmi.server.RemoteObject`
  - Peut avoir des méthodes locales (absentes de l'interface `Remote`) et aussi des attributs.

Principes de programmation

- 3) Le main Serveur crée le servant
  - Créer et installer le gestionnaire de sécurité.
  - Créer au moins une instance de la classe du servant.
  - Enregistrer au moins une instance dans le serveur de noms.
- Le serveur de noms (***registre***)
  - Gère les associations entre noms symboliques et références d'objets.
  - Implémente `java.rmi.registry.Registry`
  - Méthodes : `bind`, `rebind`, `unbind`, `lookup`, `list`
  - Accès aux objets avec une syntaxe URL : `rmi://ipmachine:port/NomObjet`
- 4) Le main Client utilise le servant

Exemple
Hello World

- Définition d'interface (<base>Interface.java)

```
import java.rmi.* ;

public interface HelloInterface extends Remote {
/* méthode retournant un message prédéfini dans
   l'objet appelé */
public String sayHello() throws RemoteException;
}
```

Exemple
Hello World

- Classe réalisant l'interface (<base>Impl.java)

```
import java.rmi.* ;
import java.rmi.server.* ;
public class HelloImpl extends UnicastRemoteObject
   implements HelloInterface {
 private String message;
 /* le constructeur */
 public HelloImpl (String s) throws RemoteException {
  message = s; }
 /* implémentation de la méthode */
 public String sayHello () throws RemoteException {
  return message ;
 }
}
```

Exemple Hello World

- Programme du serveur

```
import java.rmi.* ;
public class HelloServer {
 public static void main(String[] args) {
  try { /* créer une instance de la classe Hello et
  l'enregistrer dans le serveur de noms */
   Naming.rebind("Hello1", new HelloImpl("Hello world"));
  System.out.println("Serveur prêt") ;
  } catch (Exception e) {
  System.out.println("Erreur serveur : " + e) ;
  }
 }
}
```

Exemple Hello World

- Programme du client

```
import java.rmi.* ;
public class HelloClient {
 public static void main(String[] args){//args= serveur port
  try { /* cherche référence objet distant */
  HelloInterface hello =
      (HelloInterface)Naming.lookup("rmi://"+args[0]+":"+args
   [1]+"/Hello1");
   /* appel de méthode à distance */
   System.out.println(hello.sayHello());
  } catch (Exception e) {
     System.out.println ("Erreur client : " + e);
  }
 }
}
```

Compilation et déploiement

- Compilation
  - Il faut bien sûr **compiler toutes les classes**
    - **javac** HelloInterface.java HelloImpl.java HelloServer.java HelloClient.java
  - **Créer les talons client et serveur (mais pas depuis JDK 1.5)**
    - **rmic** HelloImpl
    - Génère **HelloImpl_Stub.class** (talon client) et **HelloImpl_Skel.class** (talon serveur)
- Déploiement des classes (si pas de téléchargement dynamique, voir annexes)

Sur le client :
- HelloInterface
- HelloClient
- *HelloImpl_Stub*

Sur le serveur :
- HelloInterface
- HelloImpl
- HelloServer
- *HelloImpl_Stub*
- *HelloImpl_Skel*

Exécution (sans chargement dynamique)

- Coté serveur
  - Lancer le serveur de noms (port **1099** par défaut)
    - **rmiregistry**
    - On peut ajouter -*J-Dsun.rmi.loader.logLevel=BRIEF* (ou *VERBOSE*), : cela rend le *registry* bavard pour trouver les problèmes
  - Lancer le serveur
    - java -Djava.security.policy=/cheminAindiquer/security.policy -Djava.rmi.server.codebase=file:/cheminAindiquer/ HelloServer
- Coté client
  - Lancer le client
    - java -Djava.security.policy=… -Djava.rmi….=… HelloClient

Sécurité

- Motivation
  - Accepter des connexions réseau de machines distantes est dangereux
  - Exécuter du code téléchargé peut être dangereux
- Politique de sécurité : spécification des actions autorisées
- -Djava.security.policy=<nom du fichier>
- ***Sans fichier "policy", les connexions externes sont refusées***
- Exemples de "security.policy" :

Seules utilisations autorisées

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

Dangereux ! (ok en TP)

```
grant {
permission java.security.AllPermission ;
} ;
```

Service de nommage (Registre RMI)

- Service de nommage, service de noms, registre
- Lancement de façon autonome dans un terminal, commande **rmiregistry**
- Lancement à partir du programme, appel à

  java.rmi.registry.LocateRegistry.**createRegistry**(int port)
- Accès direct via méthodes java.RMI.Naming, ***bind, rebind, unbind, list, lookup***
- Accès depuis programme :
  - récupération de la référence du registre

    java.rmi.registry.LocateRegistry.**getRegistry**(String host, int port)
  - utilisation des méthodes, bind, …
- Méthodes bind/rebind/unbind ne **disponibles que** dans la JVM locale au registry par mesure de sécurité.

Concurrence

- Objet serveur : susceptible d'être accéder par plusieurs clients simultanément
- Agit comme un serveur multi-threadé

Un thread pour répondre à l'appel d'une méthode distante

- Attention aux accès concurrents, ...

→A la charge du concepteur/développeur !

Méthodes distantes doivent être "thread-safe"

Fabrique d'objets (Factory)

- Motivation
  - Comment créer des objets `C` à distance ?

    `new` valable seulement en local.

    → Appel d'un objet distant `FabriqueC` réalisant `new(C)` sur le serveur.
- Exemple
  - Un mécanisme d'annuaire (répertoire téléphonique).

Fabrique d'objets
Exemple : Annuaire

```
public interface AnnuaireInterface extends Remote  {
public String titre ;
public boolean inserer(String nom, Info info) throws RemoteException, ExisteDeja ;
public boolean supprimer(String nom) throws RemoteException, PasTrouve ;
public Info rechercher(String nom) throws RemoteException, PasTrouve ;        }
```

```
public class Info implements Serializable  {
public String adresse ;
public int numtel ;   }
public class ExisteDeja extends Exception{} ;
public class PasTrouve extends Exception{} ;
```

```
public interface FabAnnuaireInterface extends Remote  {
public AnnuaireInterface newAnnuaire(String titre) throws RemoteException ;   }
```

Fabrique d'objets
Implémentation de la fabrique

```
public class Annuaire extends UnicastRemoteObject implements AnnuaireInterface {
 private String letitre ;
 public Annuaire(String titre) throws RemoteException{ this.letitre=titre; }
 public String titre( ) { return letitre; }
 public boolean inserer(String nom, Info info) throws RemoteException, ExisteDeja {}
 public boolean supprimer(String nom) throws RemoteException, PasTrouve { }
 public Info rechercher(String nom) throws RemoteException, PasTrouve { }
}
```

```
public class FabAnnuaire extends UnicastRemoteObject
            implements FabAnnuaireInterface {
 public FabAnnuaire() throws RemoteException { }
 public AnnuaireInterface newAnnuaire(String titre) throws RemoteException {
      return  new Annuaire(titre);
   }
}
```

Fabrique d'objets
Mise en œuvre de la fabrique

- Serveur

```
import java.rmi.*;
public class Server {
 public static void main (String[] argv) {
  System.setSecurityManager(...);
  try {
   Naming.rebind("Fabrique", new FabAnnuaire());
   System.out.println ("Serveur prêt.");
  } catch (Exception e) {
     System.out.println("Erreur serveur : " + e);
  }
 }
}
```

Fabrique d'objets
Mise en œuvre de la fabrique

- Client

```
import java.rmi.* ;
public class Client {
 public static void main (String args []) {
  System.setSecurityManager (...);
  try {  /* trouver une référence vers la fabrique */
  FabAnnuaireInterface fabrique = (FabAnnuaireInterface)
   Naming.lookup("rmi://"+args[0]+":"+args[1]+"/Fabrique");
   /* créer et utiliser des annuaires */
   AnnuaireInterface annuaireESIL =
   (AnnuaireInterface) fabrique.newAnnuaire("ESIL");
   annuaireESIL.inserer(..., ...) ;
} catch (Exception e) {
System.out.println("Erreur client : " + e) ;
```

Appels en retour (Callback)

- Problème : Appels RMI sont synchrones (client bloqué en attente).
- Pourquoi :
  - Informations complémentaires serveur→client lors de l'exécution.
  - Évite le scrutation explicite.
  - Exécution du service nécessite le client.

Appels en retour (Callback)

- Une solution : les appels en retour (**callback**).
- Permettre au serveur d'invoquer une méthode du client
  - Appel client→serveur avec retour immédiat (demande service).
  - Rappel serveur→client en cours d'exécution du service.
- Comment
  - Client implémente lui-même `Remote`.
  - Objet de la JVM cliente (références mutuelles avec client).
- Attention : accès concurrents aux données ; *deadlocks*.

Appels en retour (Callback) Exemple : les interfaces

```
public interface IServer extends Remote {
public void callMeBack(int time, String param, ICallback callback) throws
RemoteException ;
} /* serveur classique */
```

```
public interface ICallback extends Remote {
public void doCallback(String message) throws RemoteException ;
} /* s'exécute sur le client (sur demande du serveur) et affiche une chaîne */
```

On passe au serveur une référence d'un objet local.
Le serveur l'utilise comme un objet distant normal.

Appels en retour (Callback) Exemple : le serveur

```
import java.rmi.* ;
import java.rmi.server.* ;
public class Server extends UnicastRemoteObject
                implements IServer {
 public Server() throws RemoteException {
  super();   }
 public static void main(String[ ] args) throws Exception {
  Naming.rebind("Server", new Server()) ;
  System.out.println("Serveur pret");   }
 public void callMeBack(int time, String param, ICallback
   callback) throws RemoteException {
  ThreadServ thserv = new ThreadServ(time, param, callback);
  thserv.start();   }
 }
```

Appels en retour (Callback) Exemple : le client

```
import java.rmi.* ;
public class Client {
 public static void main(String[] args) throws Exception {
  Callback callback = new Callback();
  IServer serveur=(IServer)Naming.lookup("Server");
  System.out.println("démarrage de l'appel");
  serveur.callMeBack(5, "coucou", callback);
  for (int i=0 ; i<=5 ; i++) {
   System.out.println(i);
   try {
    Thread.sleep(2000);
   } catch (InterruptedException e) { }
  }
  System.out.println("fin du main");
} }
```

Appels en retour (Callback) Exemple : le servant

```
public class ThreadServ extends Thread {
  private int time;
  private String param;
  private ICallback callback;
 public ThreadServ(int time, String param, ICallback callback) {
   this.time = time;
   this.param = param;
   this.callback = callback; }
 public void run() {
   try { /* Action du serveur */
         Thread.sleep(1000*time);
   } catch(InterruptedException e) { }
   try {
         callback.doCallback(param);
   } catch(RemoteException e) { System.err.println("Echec: "+e); }
   callback = null ; /* nettoyage */
   System.gc();
} }
```

- Il s'exécute sur le serveur et appelle le client en retour.

Appels en retour (Callback) Exemple : le callback lui-même

```
import java.rmi.* ;
import java.rmi.server.* ;
public class Callback extends UnicastRemoteObject
            implements ICallback
{
 public Callback() throws RemoteException {
  super() ;
 }
 public void doCallback(String message) throws RemoteException
   {
  System.out.println(message) ;
 }
}
```

- Où a lieu le println (quelle JVM) ?

Conclusion

- Java RMI, intergiciel relativement basique, 3 services :
  - Service de nommage (rmiregistry),
  - Service d'activation d'objet à la demande (voir annexes),
  - Service de "garbage collector" (gestion mémoire).
- Ce qu'il manque à Java RMI :
  - Construction modulaire (évolution et ouverture),
  - Services communs (ne pas "réinventer la roue"),
  - Outils de développement (écriture, assemblage),
  - Outils de déploiement (mise en place des éléments),
  - Outils d'administration (observation, reconfiguration).
- Composants (comme EJB) visent à fournir ces compléments.

ANNEXES

Divers

Il est aussi possible de :

- Personnaliser la couche de transport des RMI.
- D'utiliser les RMI over SSL.
- Télécharger de code distant.
- Utiliser des objets activables côté serveur.

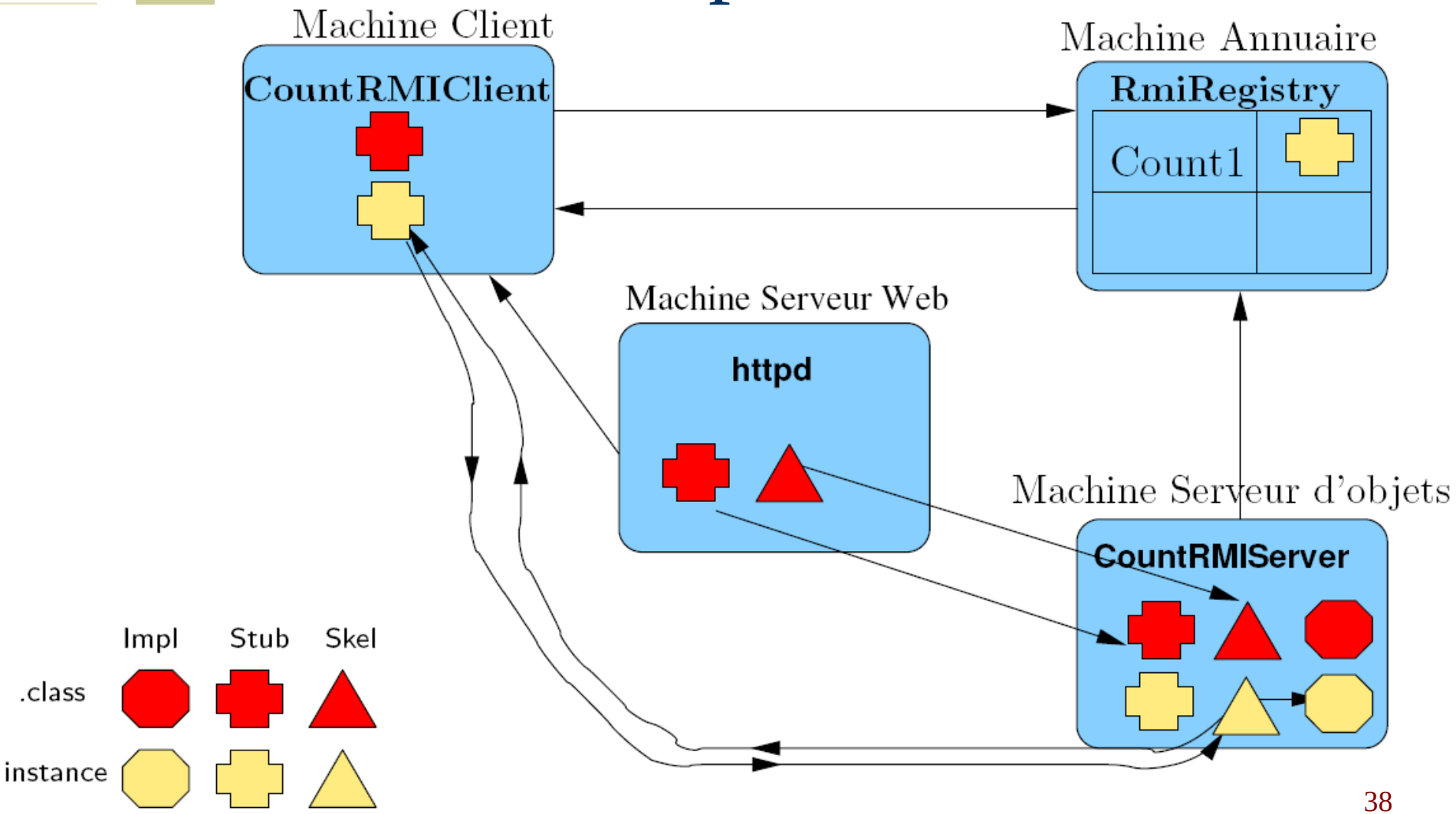
Téléchargement de code distant

- Motivation : simplification du déploiement du code
  - Classes d'implémentation centralisées sur un site web.
  - Téléchargement automatique sur un ensemble de machines.
- Il y a téléchargement quand :
  - Client obtient une souche dont la classe n'est pas dans *CLASSPATH* (récupération depuis l'annuaire, par exemple).
  - Serveur obtient une référence d'objet dont la classe est *inconnue* (passage de paramètre, par exemple).
- Désignation de l'endroit contenant les classes : **codebase**
  - Liste d'URL desquelles on autorise le téléchargement de code
    - -Djava.rmi.server.codebase=http://toto.loria.fr/truc.jar http://sun.com/JavaDir/
  - Le CLASSPATH est prioritaire sur codebase !

Téléchargement de code distant

- Cas des applets
  - Seule source possible : serveur web, même pas local. Sauf applet signée.
- Limitation : le client doit connaître le serveur.
- Solution : JINI
  - Lookup sur une interface
  - Broadcast pour localiser un serveur local
  - Lease : mandataire à durée de vie finie et pré-établie

Téléchargement de code distant Exemple de RMI



Activation automatique de serveurs

- Objectif et motivation
  - Libérer les ressources quand le service n'est pas utilisé
  - Pérennité des talons malgré arrêts serveur (volontaires ou non)

→ objets persistants (stockés sur disque) et activés au besoin

- Réalisation
  - paquetage java.rmi.Activation, classe Activatable
  - démon *rmid* recharge les objets, voire relance la VM

Les objets activables

- Présentation
  - Objets créés (ou recréés) lors d'accès par le client.

    Transparence pour le client : comme si l'objet existait avant
- Mise en œuvre
  - Servant implémente Activatable (et non UnicastRemoteObject)
  - constructeur(ActivationId id, MarshalledObject data), appellé par le système pour [ré]activer data (objet sérialisé)
  - Classe Setup, pas Serveur (prépare activation sans créer l'objet)
  - Crée groupe d'activation, l'enregistre dans rmid et rmiregistry
- (Client inchangé)

Enregistrement des objets activables

- Notion de descripteur d'activation
  - Décrit toutes les informations nécessaires à la création de l'objet distant au moment de son activation
  - ID du groupe d'activation de l'objet (une JVM par groupe)
  - nom de classe
  - URL pour récupérer le code de la classe
- Utilisation
  - Enregistrement du descripteur d'objet dans rmid
  - Cela retourne un talon pouvant être placé dans rmiregistry

Objets activables : mise en œuvre

```
import java.rmi.* ;
import java.rmi.activation.* ;
import java.util.Properties ;
public class Setup {
public static void main(String[ ] args) throws Exception {
System.setSecurityManager(new RMISecurityManager()) ;
// Cree un groupe d'activation
Properties props = new Properties() ;
props.put("java.security.policy", "/home/moi/activation/policy") ;
ActivationGroupDesc.CommandEnvironment ace=null ;
ActivationGroupDesc exampleGroup = new ActivationGroupDesc(props, ace) ;
ActivationGroupID agi =
ActivationGroup.getSystem().registerGroup(exampleGroup) ;
ActivationGroup.createGroup(agi,exampleGroup,0) ;
```

Objets activables : mise en œuvre

```
// Cree une activation (nom,location,data)
ActivationDesc desc = new ActivationDesc
("ActivServer","file:/tmp/activ/",null) ;
// Informe rmid
MyRemoteInterface mri = (MyRemoteInterface)Activatable.register(desc) ;
System.out.println("Talon recu") ;
// Informe registry
Naming.rebind("ActivServer", mri) ;
System.out.println("Servant exporte") ;
} }
```

Objets activables : mise en œuvre

```
import java.rmi.* ;
import java.rmi.activation.* ;
public class ActivServer extends Activatable implements MyRemoteInterface
{
// Constructeur pour reactivation
public ActivServer(ActivationID id, MarshalledObject data) throws
RemoteException {
// Enregistre l'objet au systeme d'activation
super(id, 0) ;
}
public Object callMeRemotely() throws RemoteException {
return "Success" ;
} }
```

- Marche à suivre
  - Compilation des classes
  - rmic
  - rmiregistry, rmid
  - Exécuter le programme setup
  - Exécuter le client.