Programmation d'Applications Réparties
PAR

Martin Quinson <martin.quinson@loria.fr>

École Supérieure d'Informatique et Applications de Lorraine – 3ième année

2007-2008
(version du 25 janvier 2009)

Motivation

La plupart des applications informatiques sont réparties

Répartition intrinsèque d'applications ou choix structurel

- Travail collaboratif (télétravail)
- Développement des réseaux (performances et prix)
- Mise en commun de ressources
- Intégration d'applications distinctes et distantes (mondialisation des entreprises)
- Parallélisme : puissance de traitement ; grandes masses de données

Présentation du module

Objectifs du cours

- Fondements de programmation d'applications réparties
  - Modèles de programmation
  - Architecture logicielle des applications et du *middleware*
- Maîtriser les principales solutions techniques existantes
  - Schémas de conception (*design pattern*)
  - Pratique par la mise en œuvre des différentes plates-formes

Prérequis

- Programmation Java

Évaluation

- Un examen sur table
- Un TP noté
- (dates et modalités à préciser)

Plan de cette partie du cours

Bibliographie succincte

Autres cours disponibles sur Internet

- Systèmes et Applications Répartis (Sacha Krakowiak, Grenoble).
  `http://sardes.inrialpes.fr/~krakowia/Enseignement/M2P-GI/`
- Programmation Répartie et Architecture N 1/3 (Denis Caromel, Nice).
  `http://www-sop.inria.fr/oasis/Denis/ProgRpt/`
- École d'été sur les intergiciels et la construction d'applications réparties
  `http://sardes.inrialpes.fr/ecole/2003/support.html`

Sites d'information

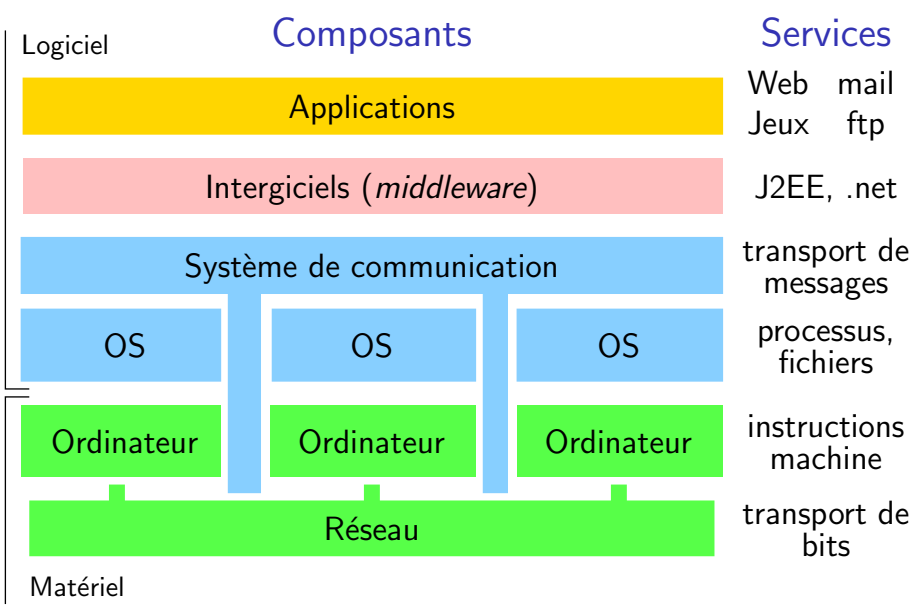
- `http://www.cetus-links.org/`
  Index de cours et tutoriels sur la programmation répartie (en anglais).
- `http://rangiroa.essi.fr/cours/`
  Pointeurs pour l'enseignement de l'informatique.

`http://www.loria.fr/~quinson/teach-PAR.html`

Premier chapitre

Introduction et principes de base

Systèmes informatique modernes



Introduction

Définition : Application répartie

- Coopération = communication + synchronisation
- Diverses fonctions :
  - Traitement : processeurs
  - Stockage : disques
  - Relation avec l'extérieur : capteur, interface
- Éléments non indépendants, mais tâche commune

Problèmes fondamentaux

- ⇒ Architectures applicatives ; Modèle client-serveur
- ⇒ Schémas de conception
- ⇒ Intergiciels (ou *middleware*, cf. chap II)

Premier chapitre

Introduction et principes de base

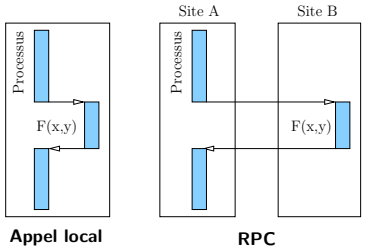
Comment découper l'application ?

La procédure comme brique de base

- Procédure = abstraction importante en impératif :
  Encapsulation : boîte noire

Appel de procedure à distance (RPC – Remote Procedure Call)

-



Avantages

- Formes et effets identiques à local
  - Simplicité conceptuelle et mise au point
- Abstraction
  - Vis-à-vis protocole de communication
  - Distribution masquée

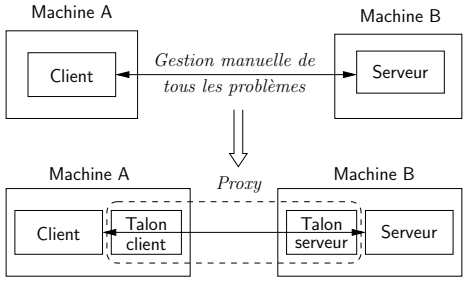
Historique des RPC

- Idée en 1984 (par Birel et Nelson)
- Implémentation par SUN en 1988
- Utilisé dans le protocole NFS (entre autres)

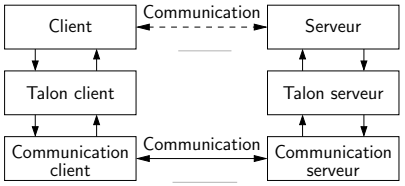
Les SUN RPC (1988)

Idée : génération automatique du code commun à tous les RPC

Application du *pattern proxy*



Vision en couches



Talon client (mandataire du serveur)

1. Reçoit l'appel local
2. Emballe les paramètres
3. Crée un identificateur
4. Exécute l'appel distant
5. Place le client en attente

10. Reçoit et déballe résultats
11. Les « retourne » au client

Talon serveur (mandataire du client)

6. Reçoit le message d'appel
7. Déballe les paramètres
8. Fait exécuter la procédure
9. Emballe, transmet résultat

Quelle découpe de l'application ?

Découpe RPC un peu fine

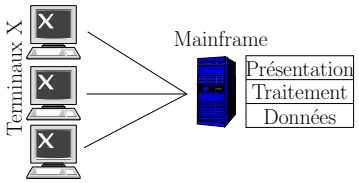
- ie, beaucoup d'appels successifs pour peu de chose chacun
  ⇒ Temps de communications et charge réseau «inutiles»
- Comment grossir le grain ?
  - Augmenter ratio calcul/communications

Une découpe naturelle d'une application (d'entreprise)

- Présentation
  - Interface homme-machine
- Logique applicative
  - Traitements
  - Applications « métiers »
- Gestion des données
  - Stockage
  - Sauvegarde
  - Base de données

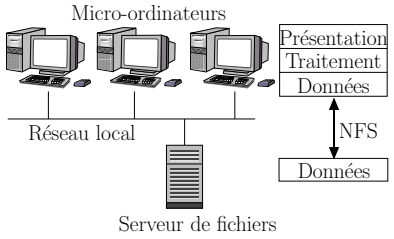
Architectures applicatives : un niveau

Applications sur site central



- Modèle utilisé dans le passé
- De nos jours : puissance peu chère

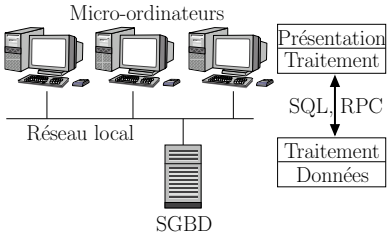
Distribution d'applications autonomes



- Modèle couramment utilisé :
- Problèmes de maintenance : mise à jour

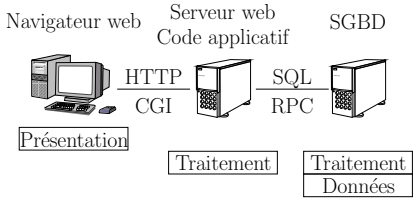
Architectures applicatives : deux, trois niveaux

Deux niveaux (ou « client lourd »)



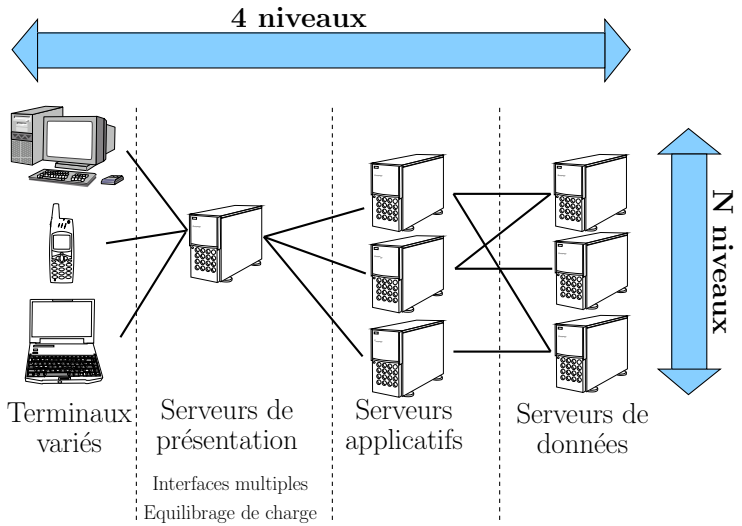
- Problèmes de maintenance : programmes duppliqués partout
- Potentiellement bon pour les clients avec
- Traitement distant : algorithmique distribuée

Trois niveaux (ou « client léger », trois tiers)



- Exemple : page web php
- Bien adapté aux

Architectures applicatives : Quatre, N niveaux



Premier chapitre

Introduction et principes de base

Programmer des applications réparties

Certains problèmes sont récurrents

Problèmes classiques des applications réparties

- Architecture logicielle, sémantique.
  - Unités d'organisation, relations ;
- Désignation et liaison
  - Découverte et nommage des machines, processus et services.
- Sécurité
  - Authentification ; Intégrité ; Confidentialité ; Non-répudiation.
- Gestion des erreurs ; Tolérance aux pannes.
  - Panne ou congestion du réseau ; Panne du serveur, du client.
- Qualité de service
  - En particulier performances, passage à l'échelle.
- Transactions ; Communications de groupe ; Anonymat/facturation ; Administration.

Programmer des applications réparties

Il existe des solutions éprouvées

Principe directeur : séparation des préoccupations

- Isoler les aspects indépendants et les traiter séparément
- Examiner un problème à la fois
- Éliminer les interférences
- Évolutions indépendantes des solutions à chaque aspect

Mise en œuvre

- Encapsulation : séparer interface et réalisation (contrat commun)
- Abstraction : décomposition en niveaux, en boîtes noires
- Schémas (*patterns*) : _____

Schémas de conceptions (design patterns)

Définition :

- Règles répondant à une classe de besoins dans un environnement donné
- (définitions d'éléments, principes de composition et d'usage)

Propriétés

- Élaboré à partir de l'expérience passée ;
- Capture des éléments de solution communs à des classes de problèmes
- Définit des principes de conception, non des implémentations
- Introduit une terminologie structurante

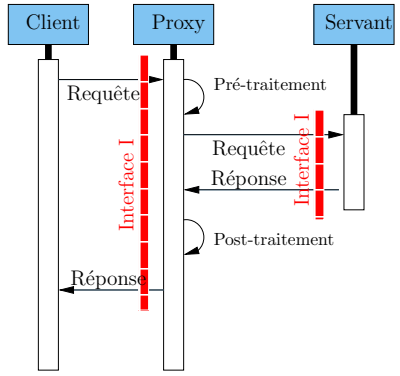
Étymologie

- Terme introduit par des architectes (en 1977)
- "Rond-point" = solution à toute une catégorie de problèmes

Schéma Proxy (Mandataire)

Contexte

- Applications constituées d'objets répartis
- Client accède à des services distants



Problème

- Masquer au client : localisation du service ; protocole de communication
- Propriétés souhaitables : Accès efficace et sûr ; Masquer la complexité au client.
- Contraintes : Environnement réparti (espace d'adressage aussi)

Solution

- _____
- _____

Schéma Fabrique (Factory)

Contexte
Application = ensemble d'objets

Problème
- Créer à distance et dynamiquement des instances d'une classe d'objets
- Propriétés souhaitables :
  Instances paramétrables ; évolution facile (rien « en dur »)

Solutions
- Factory :
- Factory factory : création de créateurs paramétrés
- Abstract factory : interface et organisation génériques de création d'objets
  Création effective déléguée à des fabriques concrètes dérivées

Pattern Pool ou Allocator

Idée
- Réduire le coût de la gestion de ressources par «recyclage des déchets»

Solution
-
-

Mise en œuvre

détruire()
Si le pool est plein
  détruire l'élément
Sinon ajouter l'élément au pool

créer()
Si le pool est vide
  créer un élément
Sinon prendre un élément du pool
  initialiser/nettoyer l'élément

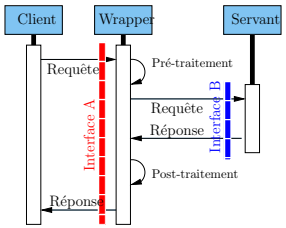
Applications
- Mémoire (malloc) et objets en général ; Threads ; Connexion réseau ...

Wrapper (ou Adapter)

Contexte
Services fournis par servants, utilisés par clients et définis par interfaces

Problème
- Modifier l'interface d'un servant existant
- Propriétés souhaitables :
  Efficacité ; adaptativité ; généricité.

Solution
-
-



Intercepteur (Interceptor)

Contexte : Cadre général de la fourniture de services
Client-serveur, P2P, hiérarchique ; Uni/Bi directionnel, Synchrone ou non

Problème : Transformer le service
- Ajouter de nouvelles fonctions, modifier les existantes
- Changer la destination de l'appel

Contraintes
- Pas de modification des clients serveurs
- Ajout/suppression dynamique de service

Solution
-
-
- peuvent rediriger l'appel vers une cible différente
- peuvent utiliser des appels en retour (callbacks) sur le client

Comparaison des schémas de base

Wrapper vs. Proxy

Wrapper et *Proxy* ont une structure similaire

- *Proxy* préserve l'interface ; *Wrapper* transforme l'interface
- *Proxy* accès souvent distant ; *Wrapper* accès souvent local

Wrapper vs. Interceptor

Wrapper et *Interceptor* ont une fonction similaire

- *Wrapper* transforme l'interface
- *Interceptor* transforme la fonction (voire la cible)

Proxy vs. Interceptor

Proxy est une forme simplifiée d'*Interceptor*

Ajouter un *Interceptor* à un *proxy* $\Rightarrow$ *smart proxy*

Résumé du premier chapitre

Architectures d'applications

- Modèle client/serveur :
- 3-tiers :
- RPC :
  - Avantages :
  - Talons, définition :
    Intérêt :

Schémas de conception

- Proxy :
- Factory :
- Pool :
- Wrapper :
- Interceptor :

Deuxième chapitre

Intergiciels

- Quelques uns des défis de l'informatique répartie
  - Désignation et liaison
  - Transmission des données
    - Représentation des données
    - Modes de passage
  - Gestion des pannes
- Intergiciels
  - Caractérisation
  - Historique
- Conclusion du chapitre

Quelques défis à relever pour l'informatique répartie

Problèmes théoriques

- Les difficultés classiques de l'informatique concurrente
  Conditions de compétition, interblocages, famines (*Cf.* cours RS 2A)
- Difficultés de l'algorithmique distribuée
  Connaissances partielles, communications asynchrones et consensus

Problèmes plus pratiques

- Découverte/nommage du serveur par le client
- Transmission des paramètres et résultats
  - Passage par valeur, par référence
  - Structures complexes
  - Emballage/déballage (représentation sur le réseau)
  - Hétérogénéité (représentation sur les hôtes)
- Traitement d'erreurs :
  - Détection des pannes
  - Sémantique d'appel en cas d'erreur
- Gestion à l'exécution : déploiement, lancement, arrêt, reconfiguration des serveurs
- Gestion mémoire : ramasse-miette distribué

Désignation et liaison

Définitions

désignation

liaison

Principes de la désignation

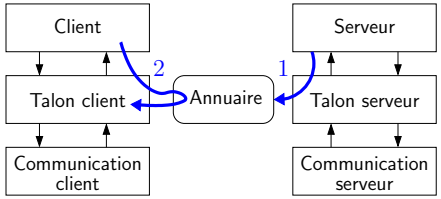
- Objets à désigner :
  - Procédure appelée
  - Site d'exécution
- Propriété souhaitée : désignation indépendante de localisation
  - ⇒ Reconfiguration (panne, équilibrage de charge)

Types de liaison

- Liaison précoce
- Liaison tardive
  - ⇒ désignation symbolique des services ⇒ sélection à l'exécution
  Différents types : liaison au premier appel, à chaque appel

Recherche de serveur par annuaire

Principe



1. Enregistrement du serveur dans l'annuaire
2. Recherche du serveur par le client

L'appel normal peut avoir lieu

Critères et mode de recherche

- Nom de machine (DNS), de fonction ou de service
- Charge des serveurs (CPU, mémoire, disque) ou autres propriétés

Deuxième chapitre

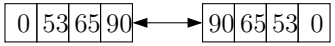
Intergiciels

Représentation des données : en mémoire

Client et serveur sur machines différentes
Il faut transmettre données et résultats

Différences de représentation en mémoire

- Little-endian (x86, alpha) vs. Big-endian (ppc, sparc, réseau) : (dépend processeur)



- Représentation des flottants (fixé par IEEE 754)
- Alignement des données dans structure (dépend compilateur)
- Tailles (32 ou 64 bits ; 4 ou 8 octets pour un registre)

type	char	short	int	long	long long	*	float	double
sizeof	1	2	4	4/8	8	4/8	4	8

Représentation des données : sur le réseau

eXternal Data Representation (XDR, utilisé par SUN RPC)

- Représentation commune sur le réseau (format pivot)
- Bibliothèques de conversion {représentation locale} ↔ {ce format}

Scalaires + Types complexes (tableaux, struct, union, enum) + Pointeurs

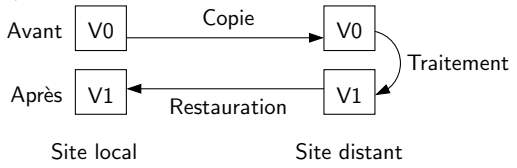
Autres solutions

- `htonl, ntohl, etc` : conversions manuelles
- ASN.1 : solution normalisée, mais lourde
- XML : SOAP/Web Services ; JXTA. Interopérable mais inefficace
- CDR/IIOP : CORBA ; J2EE
- NDR : le récepteur décode (*Native Data Representation*)

Modes de passage

- Passage par valeur : `toto(43)`
  Pas de problème
- Passage par référence : `toto(&x)`

- Passage par copie/restauration :



Palliatif au passage par référence, mais imparfait (*cf.* transparent suivant)

Problème d'aliasing

Définition

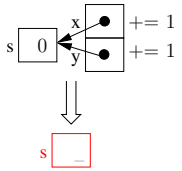
```
procedure inc2(x,y)
  x += 1
  y += 1
```

Invocation

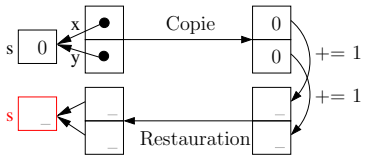
```
s = 0
inc2(s,s)
```

Que se passe-t-il ?

Passage par référence



Passage par copie/restauration



Deuxième chapitre

Intergiciels

Différents types de défaillances

Réseau

- Congestion ⇒ message retardé (mais pas perdu)
- Perte de message
- Modification/réordonnement de messages

Serveur

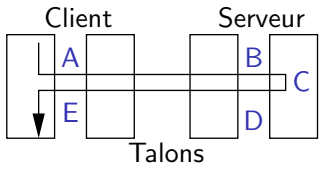
- Avant, pendant ou après l'exécution de la procédure
- Panne définitive ou reprise possible

Client

- Pendant l'exécution de la procédure
- Panne définitive ou retour ultérieur

Traitement des défaillances

Détection : Délais de garde (*timeouts*) à divers points du RPC



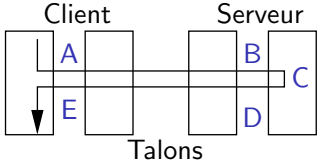
A Émission de la requête
B Réception de la requête
C Exécution du calcul
D Émission du résultat
E Réception du résultat

Sémantiques du RPC en présence de défaillances

- Au moins une fois :
- Au plus une fois :
- Exactement une fois :

Congestion du réseau ou du serveur

- Panne transitoire
- Détection : expiration des délais de garde A ou D
- Reprise en A :
  - Réémission par le talon client (même ID)
  - Lorsque le talon serveur détecte une réémission :
    Appel en cours ⇒ aucun effet
    Retour déjà effectué ⇒ réémission du résultat
- Reprise en D : réémission du résultat
- Sémantique assurée :
  - Si défaillance transitoire : « exactement une fois »
  - Si défaillance permanente : détection, notification de l'application



Panne du serveur

- Détection : expiration du délai de garde A
- Le client ne connait pas l'avancement avant la défaillance.
  Traitement pas commencé, en cours, fini.
- Reprise : client réémet dès le retour du serveur
- Sémantique : « au moins une fois »
  « Exactement une fois » possible avec service transactionnel
  (mémorisation des actions avant réalisation)

Panne du client

- Appel traité, servant déclaré « orphelin »
- Détection : expiration du délai de garde D (n réémissions)
- Reprise avant retour client :
  - Élimination des orphelins
- Reprise après retour client :
  - Client réémet l'appel (ID différent ⇒ réémission non détectée)
- Sémantique : « au moins une fois »

On peut faire mieux, si le client a un service de transactions

Deuxième chapitre

Intergiciels

Introduction aux intergiciels (middleware)

Motivations

- Interface OS trop complexe pour programmation d'applis distribuées
  - Hétérogénéité
  - Complexité des mécanismes (bas niveau)
  - Nécessité de gérer (voire masquer) la répartition
- Envie de partager l'infrastructure entre applications

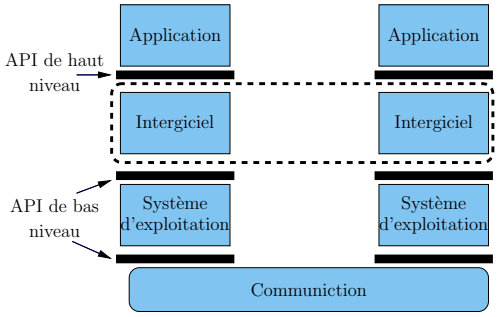
Solution : l'intergiciel

- Couche de logiciel intermédiaire entre bas niveau (OS) et applications
  - Des services disponibles
  - Des facilités pour développer de nouveaux services
  - Un support de communication entre services

$\Rightarrow$ « super-système d'exploitation » pour système réparti

Introduction aux intergiciels (middleware)

« couche du milieu »



Fonctions

- Fournir une API adaptée
- Masquer l'hétérogénéité
- Masquer la répartition
- Portabilité
- Interopérabilité

Mot relativement nouveau (1990), idée bien plus ancienne

Divers intergiciels selon la plate-forme cible

Propriétés de communication de la plate-forme cible

- Topologie : fixe (traditionnelle) ou variable (réseaux mobiles)
- Propriétés garanties ou non (bande passante, latence, pertes)
  - Système **synchrone** :
    Exemple : téléphone
  - Système **asynchrone** :
    Exemple : email, TCP

	Propriétés non garanties	Propriétés garanties
Fixe	Internet (classique)	Temps réel «dur»
Variable	Réseaux mobiles	?

Architectures et interfaces des intergiciels

Type des entités manipulées (et réparties sur la plate-forme)

- Objets : *cf.* POO
- Composants : type d'objets facilement assemblables (*cf.* cours F. Charoy)
- Agents : programmes sur chaque site (aussi appelés acteurs)

Structure du service

- Rôle spécifique des entités :
  - Client (demandeur de service), Serveur (fournisseur de service)
    Multi-niveaux (*multi-tiers*) : serveur lui-même client d'autres serveurs
  - Rédacteur (fournisseur d'information), Abonné (consommateur d'information)
- Système dit pair-à-pair : pas de rôle spécifique, les entités alternent

Interface du service

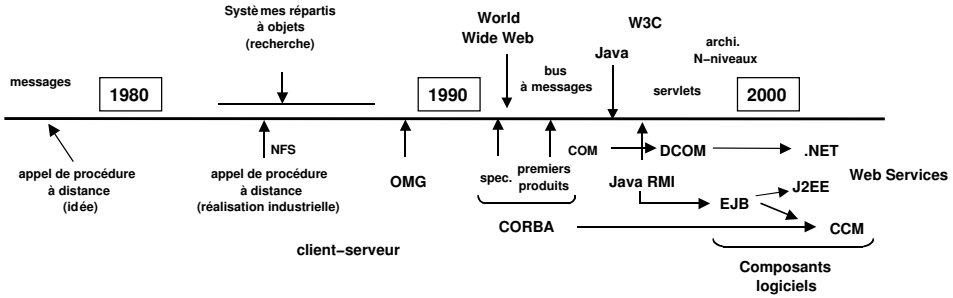
- Synchrone : _____
  Asynchrone : _____
- Push : _____
  Pull : _____

Classes d'intergiciels

- Objets répartis
  - Java RMI, CORBA, DCOM
- Composants répartis
  - Java Beans, Enterprise Java Beans, .NET, CCM
- Message-Oriented Middleware (MOM)
  - Message Queues, Publish-Subscribe
- Intégration d'applications
  - Web Services
- Accès aux données, persistance
- Support d'applications mobiles

Ne pas confondre catégorie d'intergiciel et architecture des applications

Historique des intergiciels



D'après Sacha Krakoviak

Résumé du deuxième chapitre

Problèmes à résoudre :

-
-
  -
  -
  -
-
  -
  -

Intergiciel

- Définition :
- Avantages :

Troisième chapitre

Invocations de méthodes à distance (Java RMI)

- POO répartie
- Présentation de Java RMI
- Principes de programmation, mise en œuvre et exemple
- Téléchargement de code et polymorphisme
- Création d'objets à distance : fabrique d'objets (Factory)
- Parallélisme et asynchronisme
- Activation automatique de serveurs
- Conclusion du chapitre
- Conclusion générale

Des RPC aux RMI (Remote Method Invocation)

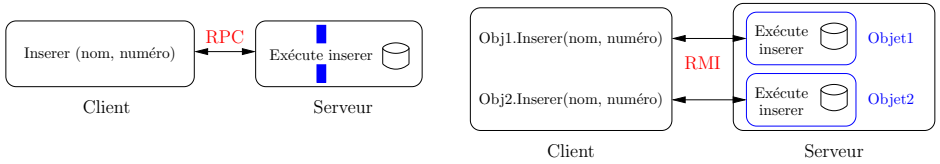
Retour sur les RPC

- Avantage principal : *Abstraction* (masquage des communications)
- Limitations :
  - Structure d'application statique, Schéma synchrone
  - Relativement difficile à mettre en œuvre

Programmation par objet des applications réparties

- 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

RPC *vs.* RMI



Java RMI (Remote Method Invocation)

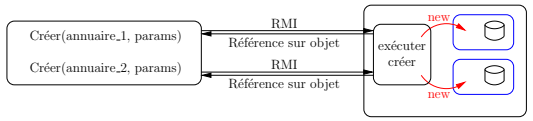
Motivation : applications réparties en Java

Principe : même schéma que RPC

- Le programmeur fournit
  - Description(s) d'interface (en Java)
  - Les objets réalisant les interfaces
  - Le programme du serveur, instanciant ces objets
  - Le programme du client, invoquant ces objets
- L'environnement Java fournit
  - Un générateur de talons nommé `rmic` (dans 1.4, inutile dans 1.5)
  - Un service de noms (Object Registry)
  - Un middleware pour l'invocation à distance (ensemble de classes)
  - La faculté d'exécuter du code généré ailleurs

Cf. `http://java.sun.com/docs/books/tutorial/rmi/`

Objets distants : problèmes à résoudre



Création d'objets à distance

- Comment faire un `new()` dans une autre machine virtuelle ?

Référence d'objet distant

- Concept clé pour objets répartis :
- Objet opaque + méthodes d'usage
  - Localisation : serveur (site et port) et interne (thread) + Protocole d'accès
- Exemples
  - Java RMI : talon client lui-même
  - CORBA : *IOR – Interoperable Object Reference*
  - DCOM : format propriétaire

Chargement de code

- Le bytecode java est portable, on peut le télécharger dynamiquement
- Simplification du déploiement d'applications

Troisième chapitre

Invocations de méthodes à distance (Java RMI)

Principes de programmation (1/2)

Grandes lignes

1. L'interface **avant** le programme
2. Réalisation de la classe qui implémente l'interface (**servant**)
3. Réalisation du programme qui lance le servant (**serveur**)
4. Réalisation du programme qui invoque le servant à distance (**client**)

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, retour de résultats

Locaux passage par copie profonde
⇒ `Serializable` (copie automatique, mais inefficace) ou `Externalizable`
Si non, erreur du compilateur

Distants passage par référence (le talon)

Principes de programmation (2/2)

2) Réalisation d'une classe distante (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 1.4, généré au vol en 1.5)
  - 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`)

3) Rôle et contenu du serveur : programme lançant le servant

- Créer et installer le gestionnaire de sécurité
- Créer au moins une instance de la classe servant
- Enregistrer au moins une instance dans le serveur de noms

Le serveur de noms (`registery`)

- Gère les associations entre noms symboliques et références d'objets
- Implémente `java.rmi.registry.Registry`
- Méthodes d'instance : `bind, rebind, unbind, lookup, list`
- Accès aux objets avec une syntaxe URL : `rmi://machine:port/NomObjet`

Exemple : Hello World (1/2)

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

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

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

```
import java.rmi.*;
import java.rmi.server.*;
public  class HelloImpl extends UnicastRemoteObject implements HelloInterface {
    private String message; /* data */

    /* le constructeur */
    public HelloImpl (String s) {
         message = s;
    };

    /* implémentation de la méthode */
    public String sayHello () throws RemoteException {
        return message;
    };
}
```

Exemple : Hello World (2/2)

Programme du serveur

```
import java.rmi.*;
public  class HelloServer {
  public static void main (String[ ] argv) {
    System.setSecurityManager (new RMISecurityManager ()); /* lancer SecurityManager */
    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) ;
} } }
```

Programme du client

```
import java.rmi.*;
public  class HelloClient {
  public static void main (String[ ] argv){
    System.setSecurityManager (new RMISecurityManager ()); /* lance le SecurityManager */
    try { /* cherche référence objet distant */
      HelloInterface hello = (HelloInterface) Naming.lookup("rmi://blaise.loria.fr/Hello1") ;
      /* appel de méthode à distance */
      System.out.println (hello.sayHello()) ;
    } catch (Exception e) {
      System.out.println ("Erreur client : " + e) ;
} } }
```


Le jeu des 8 erreurs

```
public interface AddInterface extends Remote {
  public Integer add2(int, int) ;
}
```

```
public class AddServant implements AddInterface {
    public Add () {  } ;
    public int add2 (int a, int b) throws RemoteException {
        return a+b ;
}   }
```

```
public class AddServer {
  public static void main (String[ ] argv) {
    System.setSecurityManager (new RMISecurityManager ()) ;
    Naming.rebind ("additionneur", new AddServant (1,2)) ;
    System.out.println ("Serveur prêt.") ;
} }
```

```
public class AddClient {
  public static void main (String[ ] argv){
    System.setSecurityManager (new RMISecurityManager ()) ;
    AddServant add = Naming.lookup("rmi://blaise.loria.fr:4242/Add1") ;

    System.out.println (add.add2(1,3)) ;
} }
```


Compilation et déploiement

Compilation

1. Il faut bien sûr compiler toutes les classes

   ```
   javac HelloInterface.java HelloImpl.java HelloServer.java HelloClient.java
   ```

2. Avec Java 1.4 : Créer les talons client et serveur

   ```
   rmic HelloImpl
   ```

   Génère `Hello_Stub.class` (talon client) et `Hello_Skel.class` (talon serveur)
   Inutile depuis 1.5 : les talons sont générés automatiquement (au vol)

Déploiement des classes (si pas de téléchargement dynamique)

Sur le client

- HelloInterface
- HelloClient
- HelloImpl_Stub (Java 1.4 only)

Sur le serveur

- HelloInterface
- HelloImpl
- HelloServer
- HelloImpl_Stub (Java 1.4 only)
- HelloImpl_Skel (Java 1.4 only)

Faire des `jar` peut aider au déploiement

Exécution (sans chargement dynamique)

Coté serveur

1. Lancer le serveur de noms en associant un port (1099 par défaut)

   ```
   rmiregistry 4242 &
   ```

   - On peut ajouter `-J-Dsun.rmi.loader.logLevel=BRIEF` (ou VERBOSE) : rend le registry bavard pour trouver les problèmes

2. Lancer le serveur

   ```
   java -Djava.security.policy=java.policy HelloServer &
   ```

   `-D...` précise la politique de sécurité, on y revient

Coté client

1. Lancer le client

   ```
   java HelloClient &
   ```

Sécurité

Motivation

- Accepter des connexions réseau de machines distantes peut être dangereux
- Exécuter du code téléchargé peut être dangereux

Politique de sécurité : spécification des actions autorisées

Exemples de `java.policy` (dans le répertoire courant)

Seules utilisations autorisées

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

 permission java.net.SocketPermission
  "*:80", "connect";
};
```

Dangereux ! (dans la vraie vie)

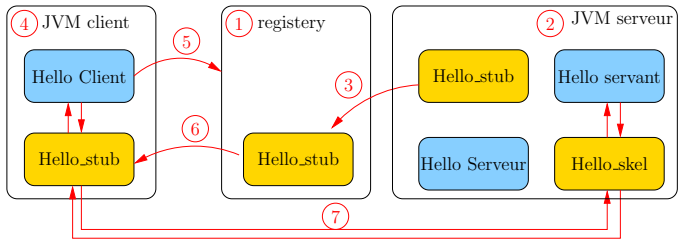
```
grant {
  permission java.security.AllPermission;
};
```

```
-Djava.security.policy=<nom du fichier>
```

Sans fichier policy, les connexions externes sont refusées

Déroulement d'une RMI de base en Java

(sans téléchargement dynamique de code)



1. Lancement du `registery`
2. Lancement du serveur et instanciation du servant
3. Enregistrement du talon client auprès du `registry`
4. Lancement du client
5. Recherche du serveur par le client
6. Téléchargement du talon client
7. Appel, puis retour

Troisième chapitre

Invocations de méthodes à distance (Java RMI)

Téléchargement de code distant (1/2)

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/"
```

- CLASSPATH est prioritaire sur codebase !

Téléchargement de code distant (2/2)

Cas des applets

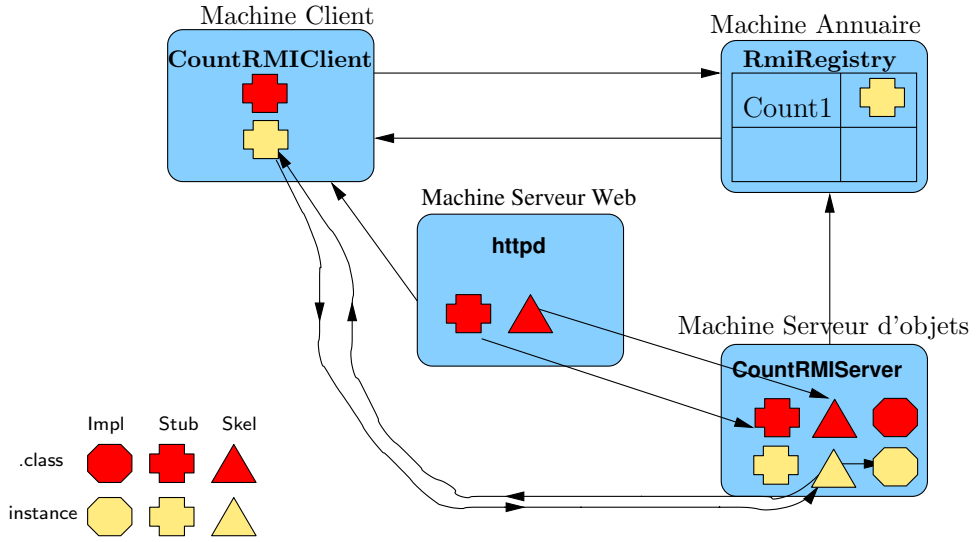
Seule source possible : serveur web, même pas local. Sauf applet signée

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

Exemple de RMI avec téléchargement de code



Polymorphisme et RMI

Polymorphisme

- Comme d'habitude : _____
- Téléchargement du code du sous-type (automatique, transparent)

⇒ Généricité et évolution du code

Par passage de paramètres

- Serveur offre une méthode `toto(param1 p);`
- Client invoque `serveur.toto(param2 p);` (`param2` dérive `param1`)

Exemple : un serveur de calcul générique (tâche en paramètre de `solve()`)

Par retour de résultat

- Client exécute `Resultat1 res = serveur.toto(...);`
- Serveur résultat de type `Resultat2` (sous-classe de `Resultat1`)

Troisième chapitre

Invocations de méthodes à distance (Java RMI)

Fabrique d'objets Factory

Motivation

Comment créer des objets `C` à distance ? `new` que mémoire locale

⇒ Appel d'un objet distant `FabriqueC` réalisant `new(C)` sur le serveur

Exemple

Un mécanisme d'annuaire (cf. exemple introductif aux objets répartis, p50)

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 num_tel;
}
public class ExisteDeja extends Exception{};
public class PasTrouve extends Exception{};
```

public **interface FabAnnuaireInterface** extends Remote {

```
  public AnnuaireInterface newAnnuaire(String titre) throws RemoteException;          }
```


Implémentation de la fabrique

public **class Annuaire** implements AnnuaireInterface extends UnicastRemoteObject {

```
  private String letitre;
  public Annuaire(String titre) {  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** implements FabAnnuaireInterface extends UnicastRemoteObject {

```
  public FabAnnuaire{};
  public AnnuaireInterface newAnnuaire(String titre) throws RemoteException {
    return new Annuaire(titre)
  };
}
```


Mise en œuvre de la fabrique

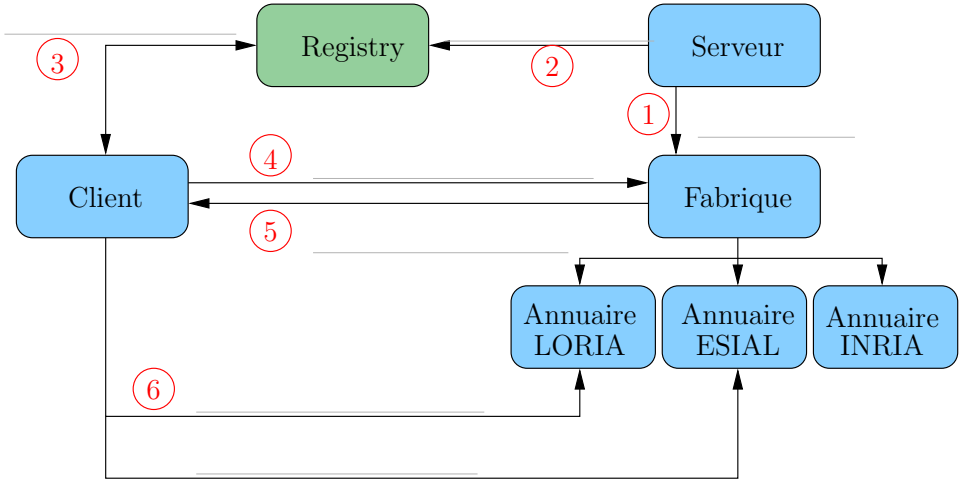
Le serveur

```
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);
} } }
```

Le client

```
import java.rmi.*;
public class Client {
  public static void main (String [ ] argv) {
    System.setSecurityManager (...);
    try {
      /* trouver une référence vers la fabrique */
      FabAnnuaireInterface fabrique = Naming.lookup("rmi://blaise.loria.fr/Fabrique");
      /* créer et utiliser des annuaires */
      annuaireLORIA = fabrique.newAnnuaire("LORIA");   annuaireLORIA.inserer(..., ...);
      annuaireINRIA = fabrique.newAnnuaire("INRIA");   annuaireINRIA.inserer(..., ...);
    } catch (Exception e) {
      System.out.println("Erreur client : " + e);
} } }
```


Vue d'ensemble d'une fabrique



Troisième chapitre

Invocations de méthodes à distance (Java RMI)

Parallélisme

Problème

- Comportement si plusieurs clients pas spécifié par RMI
  Possibilités : exécution parallèle ou séquentielle (FIFO, ...)

Solution

- Implémentation actuelle : threads
  ⇒ appels en parallèle
  ⇒ accès concurrents aux données
  ⇒ gestion explicite de la cohérence des données globales

Appels en retour

Problème : Appels RMI sont synchrones (client bloqué en attente)
Besoin de mieux si :

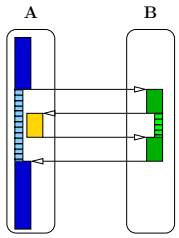
- Informations complémentaires serveur→client lors de l'exécution
- Évite le scrutation explicite (*pooling*)
- Exécution du service nécessite le client (traitement d'exception)

Une solution : les appels en retour *(callback)*
Permettre au serveur d'invoquer une méthode du client

1. appel client→serveur avec retour immédiat (demande service)
2. rappel serveur→client en cours d'exécution du service

Comment

- client implémente lui-même `Remote`
- objet de la VM cliente (références mutuelles avec client)



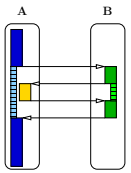
Accès concurents aux données ; deadlocks
(callbacks devraient être `synchronize`)

Appels en retour : les interfaces

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

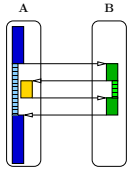
```
public interface ICallback extends Remote {
  public void doCallback(String message) throws RemoteException;
} /* s'exécute sur le client (sur demande du serveur) et affiche une chaîne */
```

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



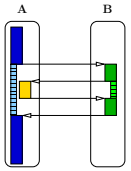
Appels en retour : le serveur

```
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",this);
    System.out.println("Serveur pret");
  }
  public void callMeBack(int time, String param, ICallback callback) throws RemoteException {
    Servant servant = new Servant(time, param, callback);
    servant.start();
} }
```



Appels en retour : 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("dmarrage 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");
} }
```

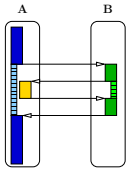


Appels en retour : le servant

```
public class Servant extends Thread {
  private int time;
  private String param;
  private ICallback callback;

  public Servant(int time, String param, ICallback callback) {
    this.time = time;
    this.param = param;
    this.callback = callback;                                    }

  public void run() { /* thread séparé */
    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();
} }
```



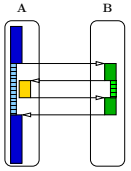
Il s'exécute sur le serveur et appelle le client en retour

Exercice : Pourquoi les deux lignes marquées "nettoyage" ?

Appels en retour : 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);
} }
```

Exercice : Où a lieu le `println` (dans quelle JVM) ?



Troisième chapitre

Invocations de méthodes à distance (Java RMI)

Activation automatique de serveurs

Objectif et motivation

- Libérer les ressources quand le service n'est pas utilisé
- Pérénité 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 (1/2)

```
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());

    // Crée 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);

    // Crée 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 reçu");

    // Informe registry
    Naming.rebind("ActivServer", mri);
    System.out.println("Servant exporté");
} }
```


Objets activables : mise en œuvre (2/2)

```
import java.rmi.*;
import java.rmi.activation.*;
public class ActivServer extends Activatable implements MyRemoteInterface {

  // Constructeur pour réactivation
  public ActivServer(ActivationID id, MarshalledObject data) throws RemoteException {
    // Enregistre l'objet au système d'activation
    super(id, 0);
  }

  public Object callMeRemotely() throws RemoteException {
    return "Success";
} }
```

Marche à suivre

1. Compilation des classes
2. rmic sur interface
3. `rmiregistry & rmid &`
4. Exécuter le programme setup
5. Exécuter le client

Conclusion sur Java RMI : extension RPC aux objets

Points forts

-
-
-
-

Points faibles

-
-
  -
  -
  -
-

Conclusion générale (1/4)

Objectifs pour la réalisation d'une application répartie

- Facilité de développement (donc coût réduit)
- Capacité d'évolution et de croissance
- Ouverture (intégration, supports hétérogènes, etc)

Conclusion générale (2/4)

Limites de la programmation traditionnelle

- Tout est à la charge du programmeur
  Construction des différents modules, interconnexion
- Manque d'abstraction pour limiter la complexité
  Structure de l'application et constituants masqués
- Évolution / modification des fonctionnalités difficiles

⇒ Besoins importants d'ingénierie logicielle et compétence technique

⇒ Développement complexe ⇒ temps ↗ ; fiabilité ↘

Amélioration apportées par la POO

- Simplifie le découpage de l'application en modules
- Simplifie la réutilisation du code au sein de l'application

Conclusion générale (3/4)

Limites de la programmation par objets

- Absence de vision globale d'architecture logicielle
  - Concepts dispersés dans chaque objets, sans description globale
  - Rarement configurable depuis l'extérieur
- Pas de découpage code fonctionnel vs. non-fonctionnel
  - Programmation explicite de code non-fonctionnel dans l'applicatif (persistance, sécurité, tolérance, nommage, cycle de vie, *etc.*)
- Absence d'outils standards pour composition et déploiement

Réponses des composants

- Description interactions entre composants (types ou instances)
- Dichotomie composant (code métier) / conteneur (code technique)
- Format d'archives avec descripteur XML ; API de déploiement

Conclusion générale (4/4)

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)

Les composants visent à fournir ces compléments