Université Lille 1
Master mention Informatique – M1

Construction d'applications réparties

IV. Client/serveur et objets

Lionel.Seinturier@univ-lille1.fr

Client/serveur orienté objet

But : coupler la notion d'objet avec les concepts client/serveur

Objectifs

- meilleure modularisation des applications
- entités logiciels plus réutilisables
- applications mieux *packagées*, portables et maintenables plus facilement

Résultats

- unité de distribution = objet
- un objet = un ensemble de traitement fournis par ses méthodes
- interaction client/serveur = invocation de méthode

⇨ Nombreux aspects techniques à intégrer pour y arriver

Client/serveur orienté objet

1.1 Nommage

Identifier les objets dans un environnement réparti

- deux objets ≠ sur le même site ou sur des sites ≠
  ne doivent pas avoir la même identité
- généralisation de la notion de pointeur à un environnement réparti

Deux niveaux

- référence d'objet distant
  adresse physique
  ex. : pointeur + @IP
- annuaire
  adresse logique
  facilier l'accès à l'objet
  annuaire = serveur "bien connu" de couples <@ logique, @ physique>

1.2 Ramasse-miettes

Détruire les objets qui ne sont référencés par aucun autre

Objectif : récupérer les ressources (mémoire, CPU, disque) inutilisées
Difficulté / aux systèmes centralisés : suivre les références entre sites distants

Deux techniques

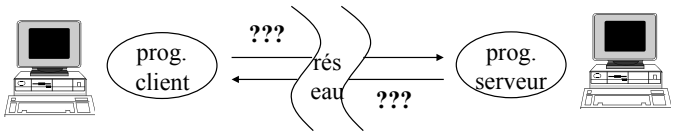
• **comptage** de références : on associe un compteur à chaque objet
+1 lorsque la référence est acquise
-1 lorsque elle est libérée
inconvénient : utilisation mémoire + pas de gestion des cycles de réf.
avantage : simple

• **traçage** (*mark and sweep*) : parcours graphe objets à partir d'une racine
objets non marqués détruits
inconvénient : temps CPU pour le traçage
avantage : pas de modification des objets

Client/serveur orienté objet

2. Langage de définition d'interface

Principe



- quels sont les services disponibles ?
- comment les invoquer ?

Langage de définition d'interface
en anglais : *Interface Definition Language* IDL

- serveur décrit les services
- client les utilise

=> contrat d'utilisation entre le client et le serveur

2. Langage de définition d'interface

Syntaxe

2 cas pour la syntaxe de l'IDL
- même syntaxe que le langage de programmation (ex. : Java RMI)
- syntaxe propre (ex. : Apache Thrift, SOAP, CORBA IDL)

Exemple

```
interface CompteItf {
   string getTitulaire();
   float solde();
   void deposer( in float montant );
   boolean retirer( in float montant );
};
```

2. Langage de définition d'interface

Syntaxe

IDL décrit
- des types de données
- des services/méthodes/procédures
- des exceptions
- des structures de données (tableau, struct, union, collection, etc.)

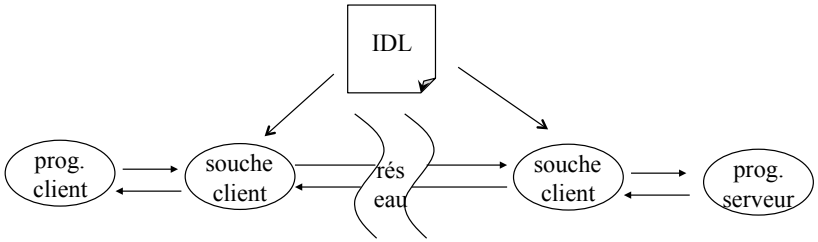
- éventuellement d'autres notions issues de langages de programmation
  - constantes, modules, héritage, etc.
- ou liées aux communications distantes
  - adressage, mode d'invocation, mode de passage de paramètres, etc.

⇒ les caractéristiques varient d'un IDL à l'autre

⇒ mais **PAS DE CODE**

2. Langage de définition d'interface

Génération des souches



- dans un langage de programmation donné (par ex. : Java, Python, C++, COBOL, etc.)
  ⇒ d'où l'intérêt d'une syntaxe IDL ≠ de celle des langages de programmation
- notion de mapping
  - traduction des notions de l'IDL vers le langage de programmation

rq. : si souche générique, pas de génération

2. Langage de définition d'interface

Ecriture de programmes client/serveur avec IDL

1. Définition des services => IDL
2. Ecriture du code des services => langage de programmation
3. Démarrage des services => langage de programmation
4. Utilisation des services => langage de programmation

Client/serveur orienté objet

3.1 Contrôle d'accès

Gérer le partage des objets dans un environnement réparti

Pour des raisons de sécurité, l'accès à certains objets peut être restreint

- en fonction de l'identité de l'objet appelant
  ex : seuls les accès en provenance de l'intranet sont autorisés
- à partir d'une liste de contrôle d'accès
  ex : mot de passe, mécanismes de clés de session, ...

La restriction peut

- interdire complètement l'accès à l'objet
- fournir une vue «dégradée»
  ex : autoriser les méthodes qui fournissent un service de consultation
  mais interdire celles qui modifient l'état de l'objet

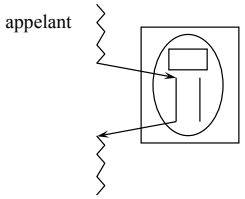
⇨ Nombreuses informations à ajouter aux objets

3.2 Concurrence

Deux types d'objets concurrents

Objet passif

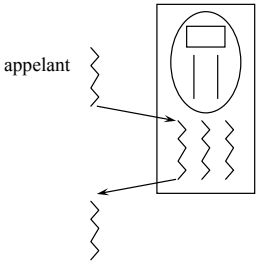
L'activité
- manipulée de façon explicite
- orthogonal à l'objet
- se «plaque» sur les méthodes



Ex :
thread Java

Objet actif

Une (+sieurs) activité dédiée
à l'objet exécutent les méthodes



3.3 Transaction

Assurer l'exécution cohérente d'une suite d'invocations de méthodes

Exemple : un transfert entre deux objets comptes bancaires

Assurer qu'un ensemble de 2 ou +sieurs invocations de méthodes

```
compte1.depot(50);
compte2.retrait(50);
```

s'effectuent complètement ou pas du tout (+ propriétés ACID /* cf. BD */)

Problématique de la théorie de la sérialisatibilité et des moniteurs transactionnels

⇨ intégration du moniteur dans le système réparti objet avec un
- protocole de **validation** (2PC ou 3PC)
- mécanisme de **verrouillage** des ressources
- mécanisme de **détection** et de **traitement** des conflits

3.4 Migration

Déplacer des objets

- d'un espace d'adressage à un autre
- d'une zone de stockage à une autre (mémoire, disque)
- d'un **site à un autre**

Objectifs

- réduire les coûts de communication entre objets «bavards»
- répartir la charge vers les sites inutilisés
- augmenter la disponibilité d'un service en le rapprochant de ses clients

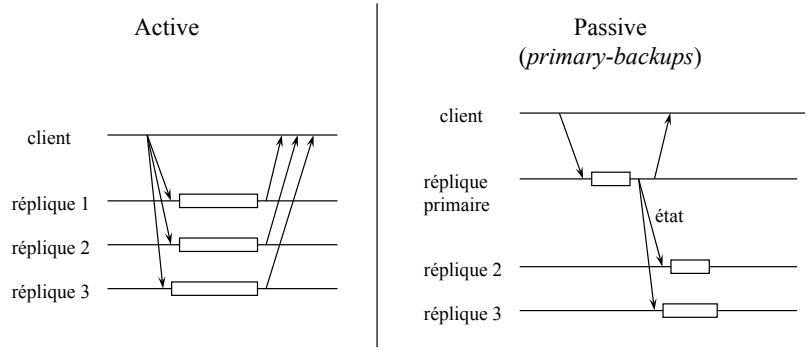
Nombreux problèmes à résoudre

- interrompre ? attendre la fin ? des traitements en cours avant de migrer
- impact de la migration sur la référence ?

3.5 Réplication

Dupliquer des objets sur +sieurs sites

Objectif principal : tolérance aux fautes



Université Lille 1
Master mention Informatique – M1

Construction d'applications réparties

V. Java RMI

Lionel.Seinturier@univ-lille1.fr

Plan

1. Caractéristiques

Java RMI

Un mécanisme d'invocation de méthodes distantes sur des objets Java

- inclus par défaut dans le JDK depuis 1.1 (évolutions dans JDK 1.2 puis 5)
- package `java.rmi`
- + outils : générateur de souches, serveur de noms, démon d'activation

- implantations alternatives
  - NinjaRMI, Jeremie, NRMI (copie-restauration)

2. Modèle de programmation

Principes

Chaque classe d'objets serveur doit être associée à une interface

⇨ seules les méthodes de l'interface pourront être invoquées à distance

1. Ecriture d'une interface
2. Ecriture d'une classe implantant l'interface
3. Ecriture du programme serveur
4. Ecriture du programme client

≡

1. déclaration des services accessibles à distance
2. définition du code des services
3. instanciation et enregistrement de l'objet serveur
4. interactions avec le serveur

2. Modèle de programmation

Ecriture d'une interface

- interface Java normale
- doit étendre `java.rmi.Remote`
- toutes les méthodes doivent lever `java.rmi.RemoteException`

```
import java.rmi.Remote;
import java.rmi.RemoteException;

interface CompteInterf extends Remote {
    public String getTitulaire() throws RemoteException;
    public float solde() throws RemoteException;
    public void deposer( float montant ) throws RemoteException;
    public void retirer( float montant ) throws RemoteException;
    public List historique() throws RemoteException;
}
```


2. Modèle de programmation

Ecriture d'une classe implantant l'interface

- classe Java normale implantant l'interface
- doit étendre `java.rmi.server.UnicastRemoteObject`
- constructeurs doivent lever `java.rmi.RemoteException`
- si pas de constructeur, en déclarer un vide qui lève `java.rmi.RemoteException`

```
import java.rmi.server.UnicastRemoteObject;
import java.rmi.RemoteException;

public class CompteImpl extends UnicastRemoteObject
                        implements CompteInterf {
    private String nom;
    private float solde;

    public CompteImpl(String nom) throws RemoteException {
        super();
        this.nom = nom;
    }
    public String getTitulaire() { return nom; }
    ... }
```


2. Modèle de programmation

Ecriture d'une classe implantant l'interface

1. Compilation de l'interface et de la classe avec `javac`
2. Génération de la souche cliente avec `rmic`

```
javac CompteInterf.java CompteImpl.java
rmic CompteImpl
```

Fichier généré

`CompteImpl_Stub.class` : souche cliente

Options ligne de commande rmic

-d path répertoire pour les fichiers générés

2. Modèle de programmation

Ecriture du programme serveur

1. Instanciation de la classe serveur
2. Enregistrement de l'instance dans le serveur de noms RMI

```
public class Serveur {

    public static void main(String[] args) throws Exception {

        CompteInterf compte = new CompteImpl("Bob");
        Naming.bind("Bob",compte);
    }
}
```

- `compte` prêt à recevoir des invocations de méthodes
- programme "tourne" en permanence
  tant que `compte` reste enregistré dans le runtime RMI

2. Modèle de programmation

Ecriture du programme client

1. Recherche de l'instance dans le serveur de noms
2. Invocation des méthodes

```
public class Client {
    public static void main(String[] args) throws Exception {
        CompteInterf compte = (CompteInterf) Naming.lookup("Bob");
        compte.deposer(10);
        System.out.println( compte.solde() );
    }
}
```


2. Modèle de programmation

Exécution des programmes

1. Lancer le serveurs de noms (`rmiregistry`)
   - une seule fois
   - doit avoir accès au *bytecode* de la souche cliente (→ CLASSPATH)

2. Lancer le programme serveur
3. Lancer le programme client

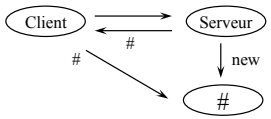
2. Modèle de programmation

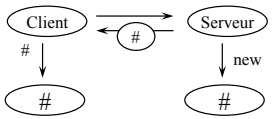
Compléments

Passage de paramètres avec RMI

- types de base (`int`, `float`, ...) par copie
- objets implantant `java.io.Serializable` passés par copie (sérialisés)
- objets implantant `java.rmi.Remote` passés par référence
  → la référence RMI de l'objet est transmise

- dans les autres cas une exception `java.rmi.MarshalException` est levée





2. Modèle de programmation

Compléments

Ecriture d'une classe d'objet serveurs

- héritage `UnicastRemoteObject`
- pas toujours possible (si classe ∈ à une hiérarchie d'héritage)

→ méthode
`static UnicastRemoteObject.exportObject(Remote)`
enregistrement côté serveur

→ méthode de désenregistrement
`static UnicastRemoteObject.unexportObject(Remote,boolean)`
false : attente fin les requêtes en cours / true : immédiat

2. Modèle de programmation

Compléments

Objets serveur prévus pour fonctionner avec ≠ types de liaisons

1. objet distant joignable en point à point
2. objets distants répliqués
3. objets distants joignables par diffusion

En pratique seul 1. mis en oeuvre (classe `UnicastRemoteObject`)

Références d'objets distants RMI avec `UnicastRemoteObject`

- adresse IP
- n° port TCP
- identifiant local d'objet (entier)

2. Modèle de programmation

Compléments

Invocations concurrentes de méthodes RMI

Un objet serveur RMI est susceptible d'être accédé
par plusieurs clients simultanément

⇨ toujours concevoir des méthodes RMI *thread-safe*
i.e. exécutable concurrement de façon cohérente

⇨ la création de *thread* est faite automatiquement par le *runtime* RMI

3. Services RMI

Services RMI

• service de nommage
• service d'activation d'objets
• ramasse-miettes réparti

3. Services RMI

Service de nommage

Permet d'enregistrer des liaisons
entre un objet serveur et un nom symbolique

• par défaut port 1099 (autre port : rmiregistry 12345)
• noms "plats" (pas de noms composés, pas de hiérarchies)

URL RMI `rmi://machine.com:1099/nomSymbolique`

`rmi://` et `:1099` facultatifs
`machine.com` par défaut `localhost`

Serveur de noms demarrable

• de façon autonome dans un *shell* avec l'outil `rmiregistry`
• dans un programme par appel de la méthode `static`
`java.rmi.registry.LocateRegistry.createRegistry(int port)`

3. Services RMI

Service de nommage

Classe `java.rmi.Naming` pour l'accès local

Toutes les méthodes sont `static`

`1. void bind(String,Remote)`	enregistrement d'un objet
`2. void rebind(String,Remote)`	ré enregistrement d'un objet
`3. void unbind(String)`	désenregistrement d'un objet
`4. String[] list(String)`	liste des noms d'objets enregistrés
`5. Remote lookup(String)`	recherche d'un objet

Les paramètres `String` correspondent à une URL d'objet RMI

1 2 3 accessibles localement uniquement

1 lève une exception si le nom est déjà enregistré

4 URL du `rmiregistry`

3. Services RMI

Service de nommage pour l'accès distant

Classe `java.rmi.registry.LocateRegistry`

```
static Registry getRegistry( String host, int port )
```

`java.rmi.registry.Registry` : même méthodes que `Naming`

3. Services RMI

Service d'activation d'objets

Objets serveurs activables à la demande (depuis JDK 1.2)
en fonction demande clients

- démon `rmid`
- package `java.rmi.activation`

Avantages

- évite d'avoir des objets serveurs actifs en permanence
  (ie "tournant" dans une JVM)
  → trop coûteux si beaucoup d'objets serveurs

- permet d'avoir des références d'objets persistantes
  → en cas de *crash* d'objet serveur
  le démon peut le relancer avec la même référence
  → les clients continuent à utiliser la même référence

3. Services RMI

Service d'activation d'objets

```
public class Serveur {

    public static void main(String[] args) throws Exception {

ActivationGroupDesc gDesc = new ActivationGroupDesc(null,null);
ActivationGroupID gID =
        ActivationGroup.getSystem().registerGroup(gDesc);
ActivationGroup.createGroup(gID,gDesc,0);

MarshalledObject mo = new MarshalledObject("Bob");
ActivationDesc desc = new ActivationDesc("CompteImpl","",mo);
CompteInterf compte = (CompteInterf) Activatable.register(desc);

Naming.bind("Bob",compte);

    }
}
```

Programme client inchangé
Lancer `rmiregistry` et **démon d'activation** (`rmid`)

3. Services RMI

Extension du principe : *pool* d'objets

Avoir un ensemble d'objets prêts à traiter les requêtes

Même problématique que le *pool* de *threads*

⇨ gestion de la taille du *pool* (fixe, variable)
⇨ à programmer

3. Services RMI

Ramasse-miettes réparti

Récupérer les ressources (mémoire, ...)
occupées par un objet que personne ne référence
→ i.e. que l'on ne pourra plus jamais accéder

Difficulté : environnement distribué donc référencement à distance

Dans une JVM : mécanisme *mark-and-sweep*

1. parcours du graphe de référencement des objets
2. destruction des objets non visités

Avec RMI : double mécanisme géré par le Remote Reference Manager

- comptage de référence : # de clients référençant l'objet
- bail (*lease*) : mémoire "louée" à l'objet pour un temps fini

3. Services RMI

Ramasse-miettes réparti

Comptage

- chaque transmission de référence +1
- chaque fin de référencement -1

Bail

- par défaut 10 min (réglable par la propriété `java.rmi.dgc.leaseValue`)
- but : se prémunir
  - partitions de réseaux
  - pertes de message de déréférencement

Si le compteur tombe à 0 ou si le bail expire,
l'objet devient candidat pour le ramassage local (*mark-and-sweep*)

3. Services RMI

Ramasse-miettes réparti

Attention : un objet serveur "normal"
instancié par un programme serveur qui "tourne" en permanence
est toujours référencé par ce programme

⇨ il n'est pas ramassé (même au delà des 10 min)

⇨ le ramassage concerne des objets créés dont on "perd" la référence

Client

```
BarRemote b = foo();
...
b = null;
```

Serveur

```
BarRemote foo() {
    return new BarRemote();
}
```

4. Fonctionnalités additionnelles

- chargement dynamique de classes (`RMIClassLoader`)
- personnalisation des communications
- génération dynamique des souches

4. Fonctionnalités additionnelles

Chargement dynamique de classes

- Java charge les `.class` à la demande à partir du disque (CLASSPATH)
- RMI introduit en + un mécanisme de chargement des classes à distance par HTTP ou FTP

Avantage : classes déployées sur 1 seul site (+ rapide + simple à gérer)
Inconvénient : *single point of failure*

Utilisation

- propriété `java.server.rmi.codebase` : URL du serveur de téléchargement
- classe `RMISecurityManager`

4. Fonctionnalités additionnelles

Personnalisation des communications

RMI utilise des *sockets* TCP

- côté client et serveur
- attribuées automatiquement par défaut

Possibilité de personnaliser ces sockets pour

- forcer l'utilisation de *sockets* précises
- tracer les communications
- crypter et/ou signer les données
- introduire des traitements "à l'insu" des objets clients/serveurs RMI

⇨ redéfinir le constructeur

```
protected UnicastRemoteObject( int port,
     RMIClientSocketFactory csf, RMIServerSocketFactory ssf )
```

- ou utiliser `static UnicastRemoteObject.export(Remote,int,..,..)`

4. Fonctionnalités additionnelles

Personnalisation des communications

```
interface RMIClientSocketFactory {
    java.net.Socket createSocket( String host, int port );
}
interface RMIServerSocketFactory {
    java.net.ServerSocket createServerSocket( int port );
}
```

⇨ déf. 2 classes qui implantent ces interfaces
⇨ déf. des sous-classes de `Socket` et `ServerSocket` pour personnaliser le fonctionnement des sockets
ex : `javax.net.ssl.SSLSocket`, `javax.net.ssl.SSLServerSocket`

4. Fonctionnalités additionnelles

Génération dynamique des souches

- rmic génère les souches de communication
- A partir JDK 5, possibilité de générer les souches dynamiquement
  - génération de *bytecode* à la volée
  - chargement dynamique de la classe générée

5. Protocole

Protocole JRMP

2 possibilités pour acheminer les invocations de méthodes distantes

- JRMP : protocole Sun (`UnicastRemoteObject`) utilisé par défaut
- IIOP : protocole OMG pour CORBA

Messages JRMP sortants

- *Call* : véhicule une invoc. de méth. (+ *callData*)
- *Ping* : teste le bon fonctionnement d'un serveur

Messages JRMP entrants

- *Return* : véhicule le retour de l'invoc. (+ *returnValue*)
- *PingAck* : acquittement d'un message *Ping*

5. Protocole

Structure des paquets échangés par JRMP

4 octets	2 octets	1 octet	n octets
En-tête	Version	Protocole	Message(s)

En-tête : *magic number* (JRMI)
Version : numéro de version du protocole

Protocole : 3 possibilités

- SingleOpProtocol : 1 seule invoc. par paquet
- StreamProtocol : +sieurs invoc. **vers un même obj.** les unes à la suite
- MultiplexProtocol : +sieurs invoc. **vers une même machine** multiplexées sur la même connexion

5. Protocole

Mécanisme de contrôle de flux JRMP

But : éviter qu'un buffer plein pour 1 connexion bloque toutes les autres
éviter qu'1 connexion qui ne se termine pas bloque toutes les autres
(par exemple en cas d'appels récursifs)

2 compteurs (en nombre d'octets) pour chaque cx RMI multiplexée
- *input request count* (irc)
- *output request count* (orc)

- irc et orc ne doivent jamais être négatifs
- irc ne doit pas dépasser une valeur (en nb d'octets) qui le bloquerait