Leuville Objects

EXPERTS EN TECHNOLOGIES JAVA, SOA ET UML,
DU SERVEUR AU MOBILE

Architectures Distribuées

Java / Remote Method Invocation
Web Services SOAP / JAX-WS

Leuville Objects
3 rue de la Porte de Buc
F-78000 Versailles
FRANCE

tel : + 33 (0)1 39 50 2000
fax: + 33 (0)1 39 50 2015

www.leuville.com
contact@leuville.com

29 rue Georges Clémenceau
F-91310 Leuville sur Orge
FRANCE

http://www.leuville.com

Table des matières

Architectures distribuées (Polytech)
Client-serveur à base d'objets

Version 3.0

- Objets répartis
  - CORBA de l'Object Management Group
  - Le modèle DCOM de Microsoft
  - Java-RMI de Sun Microsystems

Architectures à base d'objets répartis

Besoins

- Constituer une application par assemblage d'objets répartis sur un réseau
- Dissimuler les mécanismes réseau

Forces en présence

- CORBA de l'Object Management Group

- Microsoft : Distributed COM
- Sun / Java : Remote Method Invocation et Enterprise JavaBeans

Concurrents non-objet

- Middlewares orientés messages (MOM)

Architectures à base d’objets répartis

Besoins

Les différents constituants d’un système logiciel sont de plus en plus fréquemment répartis, pour tenir compte de différentes contraintes:

- répartition de différents sites de production,
- utilisation optimale d’un réseau de machines,
- proximité avec certaines ressources (base de données, matériels spécifiques, ...).

Ces applications communicantes sont constituées d’objets qui ont besoin d’inter-opérer de la façon la plus simple possible. Il est notamment important que les mécanismes de communication soient encapsulés de façon à simplifier la programmation.

Forces en présence

L’OMG, association regroupant plus de 800 sociétés informatiques, propose le standard CORBA: Common Object Request Broker Architecture.

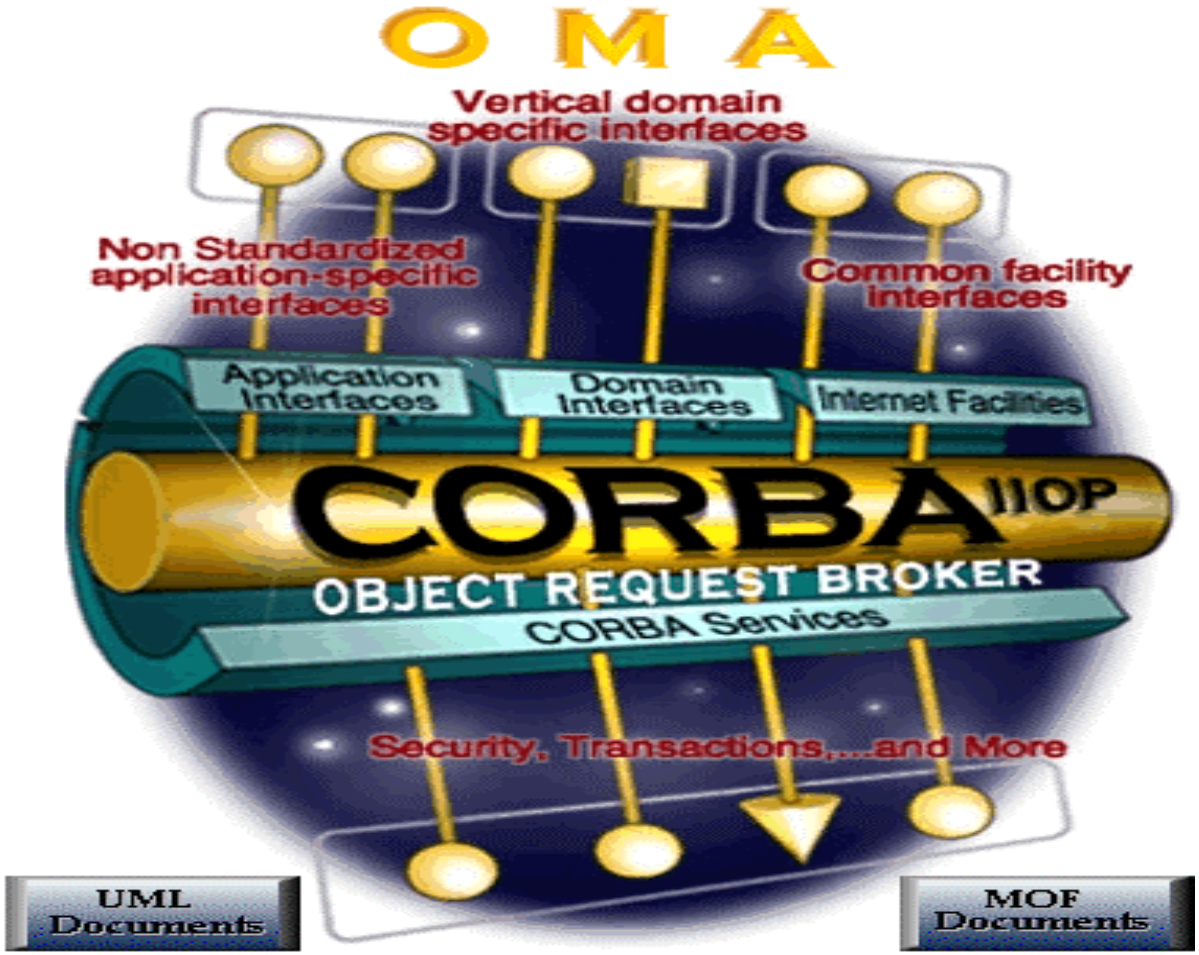
De son côté, Microsoft propose une solution aux principes proches, mais non compatible: DNA.

Concurrents non-objet

Les middlewares orientés messages permettent d’échanger des données, souvent de façon asynchrone, et à travers différents langages de programmation, Objet ou non.

Common Object Request Broker Architecture

CORBA: un 'bus logiciel Objet'



- Assure le transport des requêtes sur le réseau
- Propose des services de haut niveau: sécurité, persistance, transactions, cycle de vie, ...
- Indépendant des langages de programmation

Common Object Request Broker Architecture

CORBA: un ’bus logiciel Objet’

CORBA est une spécification de l’OMG qui définit les composants et services que l’on est en droit d’attendre d’une architecture à base de composants répartis:

- protocole de transport de requêtes et d’objets IIOP (Internet Inter-ORB Protocol)
- modèle Objet (classes, attributs, opérations, ...),
- définitions d’interfaces en IDL,
- mécanismes d’invocations statiques et dynamiques,
- les référentiels d’interfaces et d’implémentations,
- les services.

Les services proposés par CORBA comportent notamment:

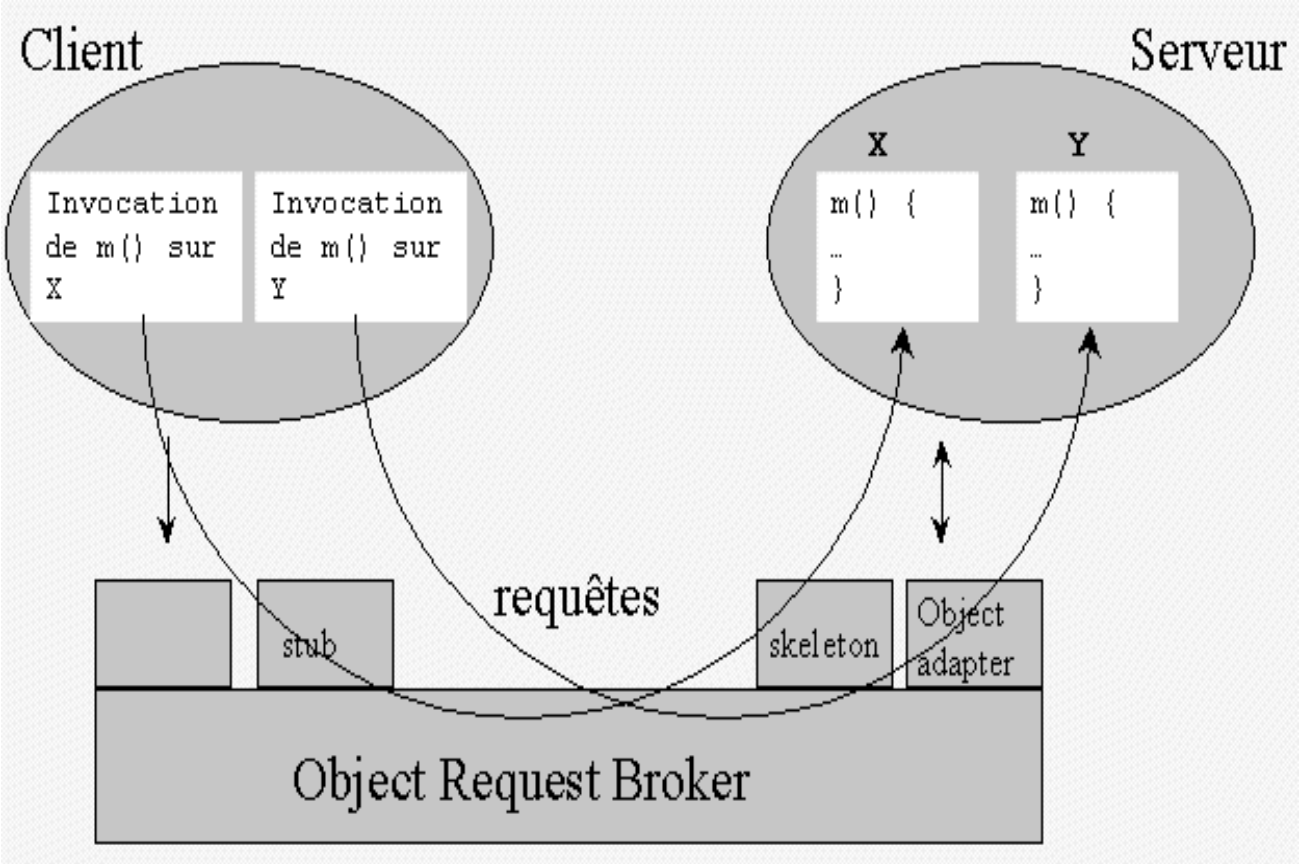
- le nommage, indispensable pour accéder à un objet serveur depuis une application cliente,
- les transactions (Object Transaction Service), qui permettent de travailler avec des objets comme avec une base de données,
- la persistance qui permet de sauvegarder les objets,
- les événements,
- la sécurité, ...

Principes d'un courtier de requêtes Objet (ORB)

- l'ORB localise l'objet serveur à l'aide du service de nommage
- il transporte les requêtes depuis le client vers le serveur
- l'opération est exécutée sur le serveur et les résultats retournent au client

Deux modes d'invocation

- statique
- dynamique



Principes d’un courtier de requêtes Objet (ORB)

L’ORB offre le moyen d’invoquer depuis le code client les fonctionnalités des objets serveur, presque comme si ceux-ci étaient situés au sein de l’espace mémoire du programme client.

Le service de nommage est un programme qui tourne sur le serveur et contient les noms de tous les objets que l’on veut rendre utilisables dans le code client. La première opération à effectuer au sein du code client est d’obtenir un représentant local de l’objet serveur. C’est sur ce représentant local que l’on effectuera les invocations de méthodes.

Il assure le transport des paramètres et des valeurs de retour quand les types utilisés sont primitifs (entiers, flottants, caractères, ...). Lorsque le type manipulé est une classe, seule une référence réseau est transportée.

Deux modes d’invocation

CORBA définit deux façons d’invoquer les services d’un objet distant:

- l’invocation statique permet d’exécuter une méthode connue au moment de la compilation des parties serveur et client,
- l’invocation dynamique permet d’exécuter une méthode non connue au moment de la compilation et rendue accessible grâce à une sorte de bases de données des interfaces: le référentiel d’interfaces.

Ces deux modes d’invocation peuvent être utilisés conjointement et permettent de faire face à toutes les situations.

Phases de développement d'une application CORBA

Sélectionner une implémentation CORBA

- Le niveau de services offerts est très variable
- Certaines solutions comportent des passerelles vers DCOM, DCE, CICS, Tuxedo, ...

Définir les interfaces des services distants en IDL

- Interface Definition Language
  - langage déclaratif de définition d'interfaces et de structures
  - une interface contient des définitions de types, d'attributs et de méthodes

Implémenter les services à l'aide d'un 'mapping' officiel

- C
- C++
- Smalltalk
- ADA 95
- JAVA

Phases de développement d’une application CORBA

Sélectionner une implémentation CORBA

De nombreuses solutions CORBA sont disponibles sur le marché. Elles comportent généralement:

- un sous-ensemble des services spécifiés par le standard CORBA,
- différentes passerelles permettant de communiquer avec l’existant (base hiérarchiques, applications COBOL, ...) ou avec d’autres techniques de programmation client-serveur réseau (DCE, DCOM, EJB, ...).

Définir les interfaces des services distants en IDL

IDL permet de définir l’ensemble des services accessibles à distance en restant indépendant du langage de programmation. Cependant, sa syntaxe reste fortement inspirée de la partie déclarative de C++.

Implémenter les services à l’aide d’un ’mapping’ officiel

CORBA définit plusieurs correspondances officielles entre IDL et différents langages de programmation. Chaque structure IDL possède une équivalence unique dans chacun des langages de programmation retenu. Ces équivalences ne dépendent pas d’un fournisseur spécifique.

Exemple CORBA

Interface IDL

```
interface Grid {
        readonly attribute short height;
        readonly attribute short width;

        void set (in short n, in short m, in long value);
        long get (in short n, in short m);
};
```

Interface C++ correspondante

```
class Grid : public virtual CORBA::Object {
public:
        short   height();
        short   width();
        void    set (short, short, long);
        long    get (short, short);
};
```

Utilisation

```
#include «grid.hh» // généré par le compilateur IDL
#include <stream.h>

main() {
        GridRef p = Grid::_bind(); // une Grid quelconque

        cout << p->height() << « « << p->width() << endl;
        p->set (2, 4, 123);
        cout << «grid[2, 4]=« << p->get(2, 4) << endl;
}
```

Exemple CORBA

Interface IDL

L'interface définit l'ensemble des caractéristiques publiques de tout objet de la classe Grid. Cette classe devra être ensuite implémentée côté serveur. Elle devra comporter une implémentation pour chacun des services défini dans l'interface IDL. Elle pourra également comporter d'autres méthodes, non définies dans l'interface IDL et donc réservées à un usage local au serveur.

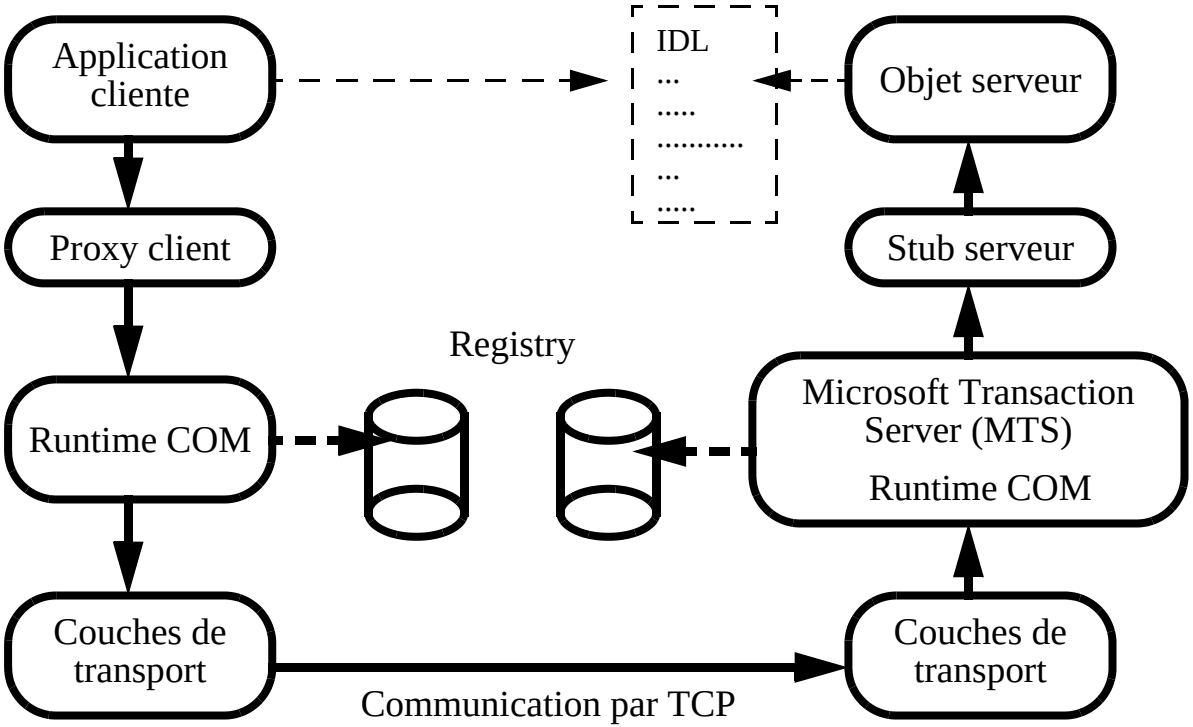
Interface C++ correspondante

Cette interface est générée par le compilateur IDL livré par le produit CORBA. La traduction IDL vers C++ est normalisée par le standard et est donc indépendante du fournisseur de l'outil.

Utilisation

Le service de nommage permet d'obtenir une référence locale sur un objet distant. Le travail sur cette référence locale s'effectue alors comme s'il s'agissait d'un objet commun. Ce sont les classes générées par le compilateur IDL et les librairies CORBA qui se chargent d'intercepter les envois de messages et de les expédier à leurs véritables destinataires à travers le réseau.

Modèle Microsoft COM-DCOM



- DCOM est basé sur DCE (Distributed Computing Environment)
- L'interface des services est définie avec un IDL Microsoft
- La programmation s'effectue par Visual Basic, Visual C++, ou différents L4G
- Les objets serveurs sont enregistrés dans la base de registres

- Les invocations de services peuvent être statiques ou dynamiques
- DCOM est très lié avec les système et nécessite les API Win32
- MTS fournit les services transactionnels

Modèle Microsoft COM-DCOM

DCOM (Distributed COM) est une extension réseau de COM (Component Object Model) basée sur DCE.

Microsoft Transaction Server (MTS) offre des services permettant de définir des composants COM en ignorant les aspects de la programmation liés au système d'exploitation.

MTS prend en charge:

- la gestion des transactions,
- la concurrence d'accès,
- la gestion des ressources ...

MTS permet aux concepteurs de composants de spécifier tous ces aspects par de simples paramètrages. Cependant, MTS comporte quelques inconvénients qui en limitent l'usage:

- une forte liaison avec le système d'exploitation et les APIs WIN32 qui le rendent presque inutilisable en dehors de cet environnement,
- l'absence de support direct pour les composants persistants.

Java Remote Method Invocation de Sun Microsystems

Principes

- Permet l'invocation distante de méthodes sur un objet Java
- Comparable à un CORBA très simplifié
  - Les moins: aucun service en dehors du service de nommage,
  - Les plus: simplicité, transport d'objets par valeur sur le réseau
- Solution propriétaire Java
- Inter-opérabilité avec CORBA via le protocole IIOP

Java Remote Method Invocation de Sun Microsystems

Principes

Java RMI est une technique de programmation distribuée Java qui est à mi-chemin entre les Remote Procedure Call du langage C et CORBA.

La mise en place d'une application RMI suit les mêmes étapes que pour une application CORBA:

- définition des interfaces des services accessibles à distance, mais en langage Java au lieu de IDL,
- implémentation des services côté serveur,
- accès aux objets serveur depuis les applications clientes et invocations de méthodes.

En plus du transport de références d'objets sur le réseau que l'on trouve dans CORBA 2, RMI définit un mode de transport d'objet par valeur. Ce mode a depuis été également adopté par CORBA dans la spécification 3.0.

RMI utilise également le protocole de transport IIOP, ce qui permet aux deux modes de cohabiter et coopérer.

Exemple RMI

- Définition de l'interface

```
public interface Compte extends Remote
{
        public float lireSolde();
        public void depot (float montant);
        public void retrait (float montant);
}
```

- Nommage des objets côté serveur

```
Compte cpt = new CompteImpl ();

Naming.bind («rmi://mybank.com/cpt512», cpt);
```

- Accès côté client

```
Compte cpt = (Compte) Naming.lookup («rmi://mybank.com/cpt512»);

cpt.depot (10000);
```

RMI nécessite l'utilisation d'un serveur de noms qui enregistre des associations entre un nom (une chaîne de caractères) et un objet serveur.

L'application cliente obtient une référence d'un objet serveur en interrogeant ce serveur de noms. Elle peut ensuite manipuler l'objet serveur à travers la référence obtenue.

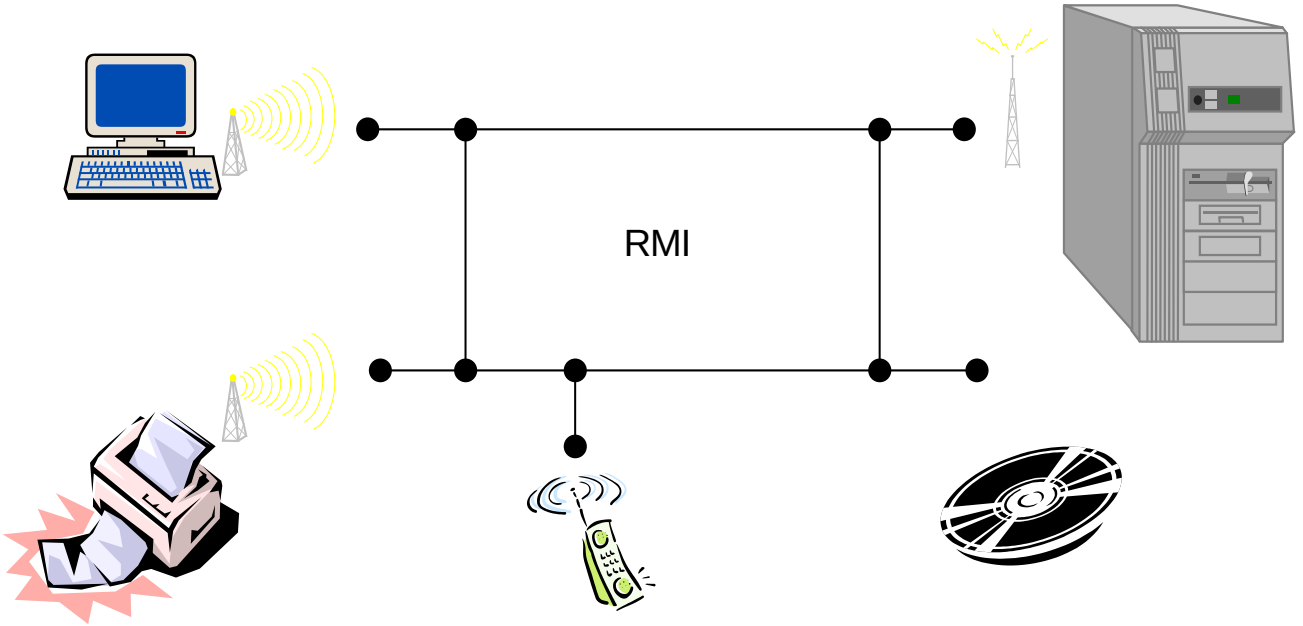
Exemple RMI

Notes :

Exemple d'application RMI: JINI

Le réseau universel

- Tous les appareils connectés publient leur services sous forme d'interfaces RMI
- Les services peuvent être utilisés sans drivers spécifiques



Exemple d’application RMI: JINI

Le réseau universel

JINI est une tentative pour simplifier l’usage de différents appareils électroniques connectables en réseau:

- ordinateurs,
- unités de stockage (disques durs),
- imprimantes,
- scanners,
- appareils photo numériques,
- caméscopes,
- ...

Chaque appareil publie ses services sous la forme d’interfaces RMI. Il est alors possible pour un autre appareil connecté au réseau d’utiliser ces services.

Synthèse

De nombreux besoins restent à couvrir

- L'architecture demeure complexe à établir
- Il faut fournir beaucoup de code technique
  - gestion du cycle de vie des objets
  - gestion des threads
  - gestion des transactions
  - gestion des connexions vers le SGBD

Serveurs d'applications et composants métier

- Les services techniques sont pris en charge par le serveur d'applications
- Les composants comportent seulement du code métier
- Les spécificités techniques sont contrôlées par des mécanismes de paramètrage
- Solutions en présence : Enterprise JavaBeans, Microsoft COM +, CORBA semble en retrait

Synthèse

De nombreux besoins restent à couvrir

Les applications client-serveur multi-niveaux nécessitent de nombreuses optimisations au niveau du code purement technique. En effet, il faut fréquemment fournir des mécanismes permettant d'optimiser:

- la gestion des connexions réseau afin d'éviter les créations dynamiques,
- le lancement de threads de traitements client,
- la gestion mémoire et l'instanciation des objets.

Serveurs d'applications et composants métier

La solution consiste à séparer nettement les services techniques du code métier, en laissant seulement au développeur la possibilité de travailler sur ce dernier. Le serveur d'applications est une structure d'accueil pour composants métier qui offre toutes les infrastructures nécessaires à la réalisation d'applications client-serveur multi-niveaux. Il comporte notamment tous les services techniques que l'on doit toujours réaliser pour ce type d'applications:

- cycle de vie des objets avec pools d'objets pré-construits,
- accès réseau,
- gestion de pools de connexions réseau (client et SGBD),
- gestion transactionnelle, ...

Les applications sont alors réalisées par assemblage de composants comportant presque exclusivement du code métier.

Architectures distribuées (Polytech)
Remote Method Invocation, les bases

Version 3.0

- Appel de méthodes distantes avec `RMI`
- Définition d'une interface de services distants
- Invocation de méthodes distantes et règles de passage de paramètres
- Applications clientes et serveurs
- Architecture-type d'une solution `RMI`

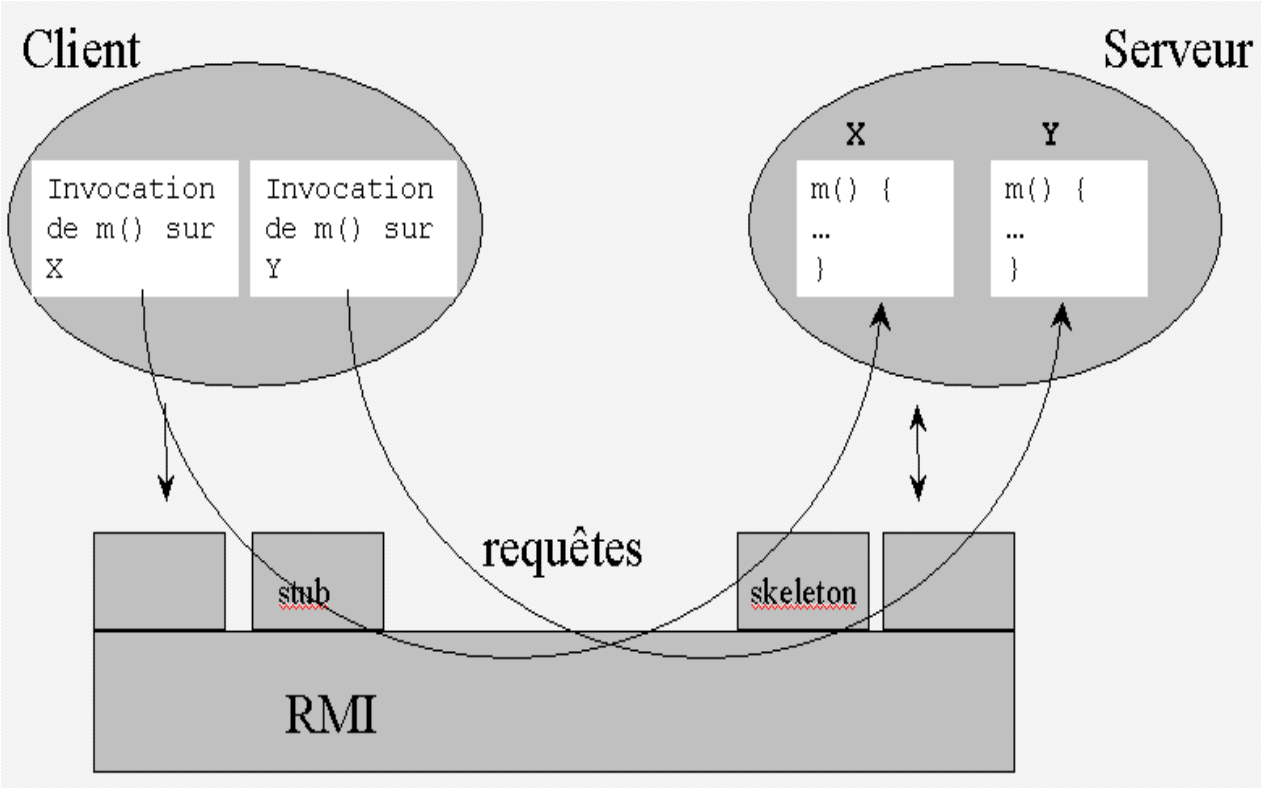
Présentation

Invocation de méthodes distantes

- A mi-chemin entre RPC du langage C et CORBA de l'OMG
- Fonctionne sur tout type de réseau IP

Caractéristiques

- Définition des services utilisables à distance à l'aide d'interfaces JAVA
- Transport d'objets par valeur et par référence
- Identification des objets serveur par nommage à syntaxe de type URL



Présentation

Invocation de méthodes distantes

Java permet de réaliser des applications client-serveur au dessus d'un réseau IP à l'aide de différentes techniques: Sockets UDP ou TCP, Programmation CORBA, Programmation RMI.

Cette dernière technique est propre à Java et permet d'invoquer une méthode sur un objet Java depuis une autre application Java, située sur la même machine ou sur tout autre machine connectée à la première par un réseau IP.

Caractéristiques

RMI est un mécanisme utilisable uniquement depuis le langage de programmation Java. Une implémentation RMI-IIOP permet de faire inter-opérer des objets RMI et des objets CORBA.

Les méthodes invocables à distance doivent être 'publiées' à l'aide d'interfaces Java et implémentées sur le serveur.

Elles seront alors accessibles depuis toute application cliente, à partir de représentants locaux des objets serveur. Ces représentants sont appelées des stubs.

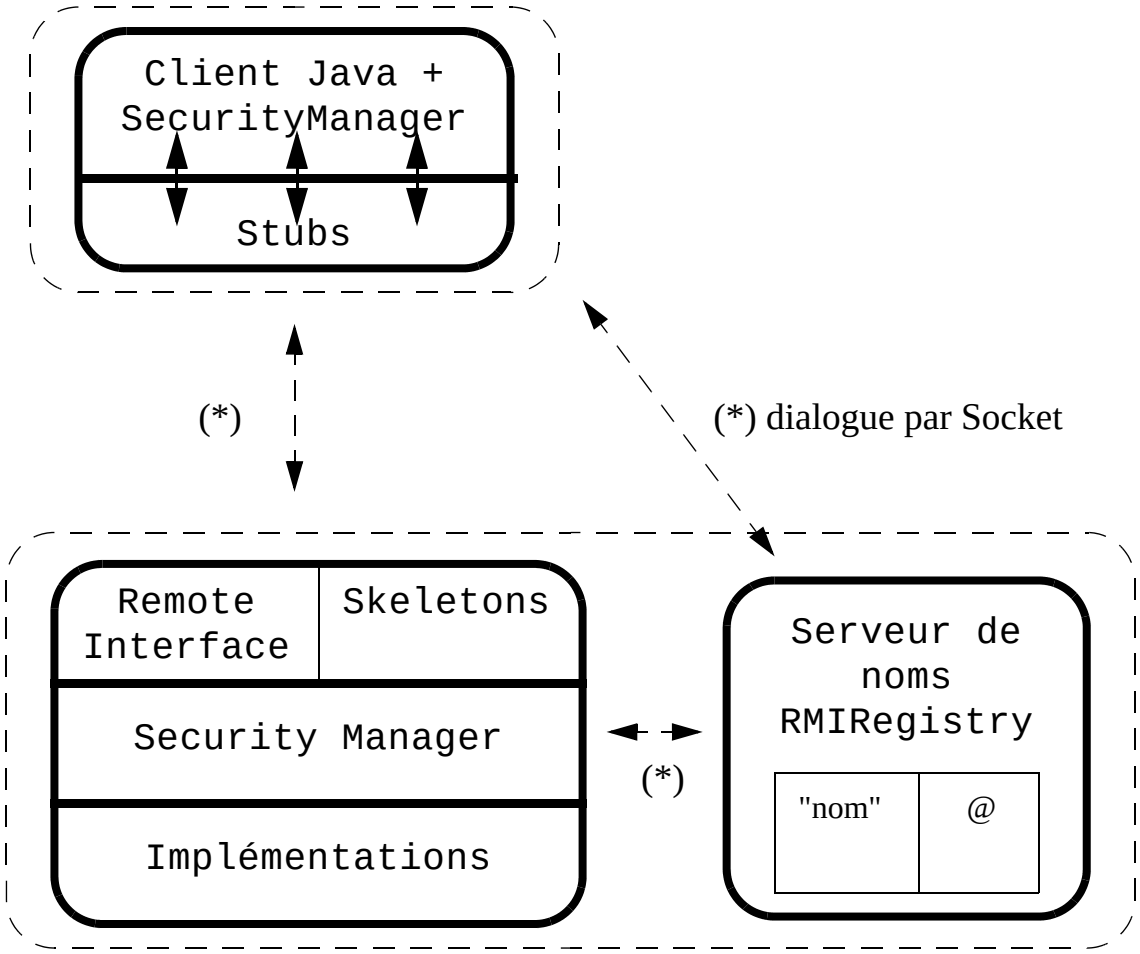
L'obtention d'un représentant local se fait par interrogation d'un serveur de noms qui répertorie l'ensemble des racines d'objets serveur.

Le transfert des informations sur le réseau est effectué suivant le protocole JRMP, propre à RMI. Ce protocole autorise le transport d'objet par référence et par valeur.

Architecture RMI

Schéma général

- Serveur: application autonome
- Client: application autonome ou applet
- Le serveur de noms fournit les objets initiaux aux clients
- Les communications s'effectuent par sockets



Architecture RMI

Schéma général

- Un serveur RMI est une application autonome Java dont les tâches sont les suivantes:
  - installation éventuelle d'une politique de sécurité spécifique pour protéger le serveur si celui-ci télécharge des classes, par exemple en provenance d'un client qui jouerait également le rôle d'un serveur
  - instanciation des objets serveurs
  - enregistrement des objets dans le serveur de noms
  - l'écoute des requêtes clients est assurée automatiquement par RMI

- Un client RMI peut être:
  - une application autonome
  - une applet

- rmiregistry est un serveur de noms minimal qui offre uniquement un service permettant de retrouver un objet serveur d'après son nom. Si l'on désire un service plus évolué (authentification par exemple), il est tout-à-fait possible de lui substituer un autre mécanisme (LDAP, serveur de noms CORBA, ...)

- Les communications entre client-serveur, client-rmiregistry et serveur-rmiregistry s'effectuent par Sockets. Les numéros de ports utilisés sont paramétrables. Il est également possible d'utiliser des sockets de types particuliers (SSL par exemple).

Etapes de réalisation d'une application RMI

Conception de l'architecture

- Services et relations entre objets serveur
- Nature des objets paramètres et valeurs de retour des services
- Politique de sécurité pour le client et le serveur

Code Java

- Définition des interfaces des classes d'objets serveur
- Implémentation des classes d'objets serveur conformes aux interfaces
- Réalisation du processus serveur
- Réalisation de l'application cliente

Mise en place

- Compilation des différents fichiers à l'aide du compilateur habituel
- Génération des stubs et skeletons avec le compilateur rmic
- Mise en oeuvre d'un démon httpd en fonction du mode de fonctionnement choisi

Etapes de réalisation d’une application RMI

Conception de l’architecture

Il convient tout d’abord de définir le modèle Objet des différentes classes d’objets serveur. Ce modèle devra intégrer l’ensemble des interfaces nécessaires et mettre en évidence les relations entre objets serveur.

Le choix de la nature de certains objets peut s’effectuer en considérant différents facteurs comme la fréquence d’utilisation, le volume des données, le nombre de services invoqués sur l’objet, ...

Code Java

Quelques règles d’écriture doivent être respectées quant à l’écriture des interfaces RMI:

- héritage de l’interface java.rmi.Remote
- ajout d’une clause throws RemoteException sur chaque méthode invocable à distance

Mise en place

Différents modes d’installation sont possibles:

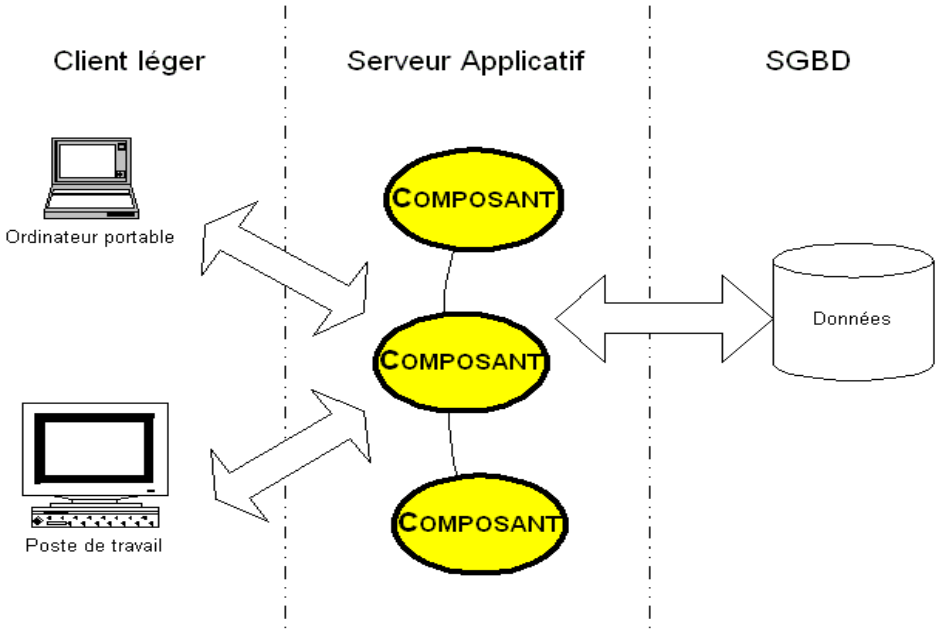
- client de type applet avec téléchargement de toutes les classes, y compris les classes RMI (stubs)
- client de type application autonome à installation fixe
- client de type application autonome possédant la capacité de se télécharger à partir d’une classe de chargement (bootstrap)

Les modes offrant un téléchargement de classes requièrent la mise en oeuvre d’un serveur WEB.

Déroulement sur un exemple

Mise en place d'un serveur bancaire

- Description de l'architecture
  - Un serveur central stocke les comptes des clients dans un SGBD
  - Accès aux opérations classiques de gestion de comptes depuis les postes des agences



- Services disponibles sur le serveur
  - ouverture d'un compte bancaire
  - accès à un compte à partir de son numéro de compte
  - obtention du solde d'un compte
  - dépot et retrait d'une somme sur un compte
- Contraintes particulières
  - pouvoir accéder aux services à partir d'un simple navigateur
  - limiter les opérations de configuration du poste client

Déroulement sur un exemple

Mise en place d'un serveur bancaire

- Description de l'architecture

Une application en grandeur réelle comporterait trois niveaux:

- une base de données comportant les informations sur les comptes et les clients,
- un serveur applicatif proposant l'ensemble des services à des clients distants,
- un ensemble de clients distants légers.

Ici, nous allons nous focaliser sur le serveur applicatif intermédiaire et le client léger, qui communiquent à l'aide de RMI.

- Services disponibles sur le serveur

Le serveur offre deux niveaux de services distincts:

- la création de comptes bancaires,
- la réalisation d'opérations sur un compte particulier.

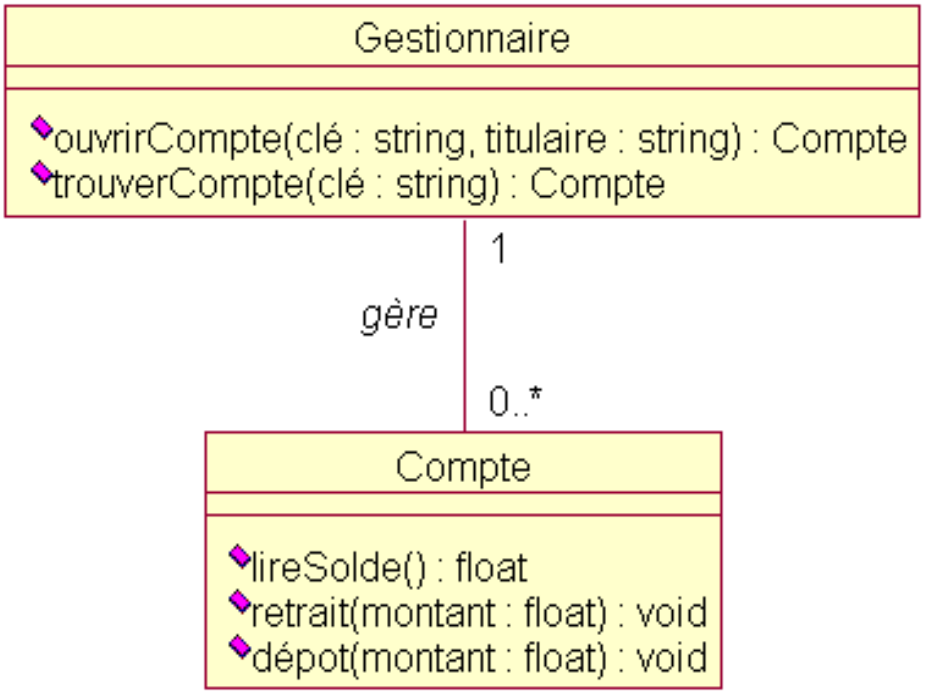
- Contraintes particulières

Un client de type applet permettra de bénéficier des fonctionnalités Java de téléchargement de classes, y compris les classes spécifiques à RMI (stubs).

Modélisation du problème

Deux niveaux de services distincts

- Gestion de la liste des comptes bancaires
  - Création
  - Modification
  - Destruction
- Opérations sur chaque compte bancaire
  - Obtention du solde
  - Retrait
  - Dépot



La solution RMI doit respecter la modélisation Objet.

Modélisation du problème

Deux niveaux de services distincts

La modélisation Objet de notre problème montre qu'il existe deux niveaux distincts de services:

- opérations sur la liste des comptes bancaires: ajout, retrait, accès ...
- opérations sur un compte: dépot, solde, retrait.

Ces deux types de services vont être assurés par deux classes différentes.

La solution RMI doit respecter cette modélisation Objet afin d'en conserver tous les avantages:

- identification simple du rôle de chaque classe
- plus grande simplicité du code
- meilleure encapsulation.

Nous allons donc définir deux interfaces RMI correspondant aux classes d'objets que nous souhaitons manipuler à partir de l'application cliente.

Définition des interfaces des services

- Dériver l'interface *java.rmi.Remote*
- Prototyper chaque méthode, en utilisant des types de données Remote ou Serializable
- Ajouter une clause *throws RemoteException* sur chaque signature de méthode

Interface Compte avec gestion d'exceptions "métier"

```
import java.rmi.*;
public interface Compte extends Remote {

   public float lireSolde() throws RemoteException;
   public void dépot(float montant) throws RemoteException, TransactionException;
   public void retrait(float montant) throws RemoteException, TransactionException;
}
// avec

public class TransactionException
        extends RemoteException {}
```

Définition des interfaces des services

Interface Compte avec gestion d’exceptions "métier"

Chaque méthode est prototypée avec son type de retour, son nom et les types de ses paramètres éventuels. Nous y ajoutons une clause throws RemoteException imposée par RMI. Cette clause impose au code client de prendre en compte les éventuels problèmes qui peuvent survenir pendant l’invocation des méthodes:

- serveur ne répondant plus,
- rupture du réseau,
- ...

Les prototypes des méthodes retrait() et solde() sont enrichis pour signaler au code client qu’une exception TransactionException peut être levée.

Cette classe d’exception appartient au domaine des classes applicatives. Lorsqu’une exception de cette classe sera levée par le code serveur, le client pourra la traiter à l’aide d’un code try {} catch() {} classique.

Typage des valeurs de retour et paramètres

Types autorisés

- **Types primitifs**
  - boolean, int, float, ...
- Classes implémentant l'interface **java.io.Serializable**
  - de nombreuses classes du JDK: String, Vector,
  - classes utilisateur implémentant explicitement Serializable
- Interfaces dérivées de l'interface **java.rmi.Remote**
  - interfaces de description des méthodes invocables à distance par RMI

Deux modes de passage de paramètres

- **Par valeur**: types primitifs et java.io.Serializable
  - la totalité de la donnée est transférée sur le réseau
- **Par référence**: les sous-interfaces de java.rmi.Remote
  - seule la référence de la donnée est transférée sur le réseau
  - l'objet reste dans la machine virtuelle où il a été instancié

Typage des valeurs de retour et paramètres

Types autorisés

Les seuls types qui ne peuvent pas être utilisés en RMI pour spécifier les valeurs de retour ou paramètres sont constitués par les classes qui ne sont:

- ni Serializable,
- ni Remote.

Dans ce cas, l'erreur est détectée à l'exécution uniquement.

Deux modes de passage de paramètres

- Par valeur
  Lorsque l'on utilise des objets dont la classe est Serializable, la totalité des objets est passée sur le réseau lors d'un passage de paramètre ou lors d'un retour d'invocation. Si l'objet est volumineux, cela peut augmenter le temps d'invocation des méthodes.
  Il existe différents moyens d'agir sur le mécanisme de Serialization. Ils sont décrits par la suite.

- Par référence
  Seule une référence permettant d'agir sur l'objet est transférée sur le réseau. Toutes les méthodes décrites dans l'interface de type Remote sont utilisables.

Contrôle de la serialization

Exemple

- Une classe est Serializable si tous ses attributs le sont
- Un attribut static n'est jamais serialisé
- transient permet d'éliminer un attribut
- Les références circulaires sont gérées par le mécanisme
- Les classes des objets ne sont pas sérialisées

```
import java.io.*;

public class Param implements Serializable
{
   // cet attribut sera Serializable si AutreClasse
   // implémente java.io.Serializable
   private AutreClasse unObjet;

   // type primitif toujours Serializable
   private int x;

   // les variables statiques ne sont jamais sérialisées
   static int y;

   // le mot-clé transient permet de ne pas sérialiser
   // un attribut
   transient private float taux;
}


class AutreClasse implements Serializable
{
   // référence circulaire possible
   Param monParam;

   String nom;
}
```

Contrôle de la serialization

Notes

Lorsqu'un objet est sérialisé, un graphe à plat contenant non seulement l'objet initial mais également tous ceux qu'ils référence est fabriqué. Ce graphe ne contient pas les classes des objets, mais uniquement leurs noms complets ainsi que des identifiants de version.

Ce graphe est transporté sur le réseau sous la forme d'un paquet d'octets. A son arrivée, le mécanisme crée une arborescence d'objets identiques aux objets originaux, à condition que toutes les classes des objets sérialisés soient présentes localement.

La serialization peut également être utilisée pour sauvegarder des objets dans un fichier. Il s'agit alors d'un mécanisme de persistance.

Un attribut marqué comme transient n'est pas sérialisé. Lors de la reconstruction de l'objet, la valeur par défaut du type de l'attribut est affectée à l'attribut:

- 0 pour les entiers,
- false pour boolean,
- 0.0 pour les flottants,
- null pour les références d'objets.

Interface Gestionnaire

Solution 1

- Transformer Compte en objet Serializable et définir Gestionnaire en objet Remote
  - l'accès à un Compte entraîne un transfert par valeur sur le réseau
  - le Compte est ensuite utilisable localement sur le poste client

---

Attention aux problèmes de synchronisation clients - serveur.

Solution 2

- Conserver Compte sous la forme d'un objet Remote et définir gestionnaire en objet Remote
  - l'accès à un Compte entraîne seulement le transfert de sa référence sur le réseau
  - l'utilisation de l'objet se fait obligatoirement par invocation de méthodes à travers le réseau

---

Solution retenue.

Interface gestionnaire

```
public interface Gestionnaire extends Remote {
public Compte ouvrirCompte(String clé, String titulaire) throws RemoteException;
public Compte trouverCompte(String clé) throws RemoteException;
}
```

Interface Gestionnaire

Gestionnaire permet de construire des objets de type Compte, manipulables à distance. En cela, il remplace le constructeur que Java ne permet pas de spécifier dans une interface.

Cette interface permet également de retrouver un Compte à partir de sa clé, afin de pouvoir effectuer les opérations bancaires de dépot, retrait et obtention de solde.

Solution 1

Cette solution est intéressante si l'on est amené à utiliser la quasi-totalité des possibilités de l'objet Compte sur le poste client.

Dans ce cas, le coût initial du transfert est compensé par la possibilité d'accéder en local à l'objet Compte.

Mais il ne faut pas perdre de vue que le Compte présent sur le poste local n'est qu'une copie de l'objet présent sur le serveur. En cas de modification sur plusieurs clients, des problèmes de synchronisation sont à prévoir.

Solution 2

Cette solution sera plus intéressante si l'on accède à un sous-ensemble des possibilités de l'objet Compte. En effet, chaque invocation de méthode doit transiter par le réseau.

Cette solution est retenue pour la suite de la présentation.

Implémentation des interfaces Remote

Services à fournir

- Une classe pour chaque interface de type Remote
- Une implémentation de toutes les méthodes spécifiées par les différentes interfaces Remote
- Convention d'usage
  Pour toute interface **X**, nommer la classe d'implémentation **XImpl**

Compléments

- Constructeur(s)
- Sections critiques éventuelles (accès concurrents possibles)

Implémentation des interfaces Remote

Services à fournir

Une classe d'implémentation d'une interface de type Remote doit fournir une implémentation pour chaque méthode de l'interface.

Elle doit également être en mesure de traiter les différentes invocations de méthodes qui seront effectuées depuis les postes clients. Pour cela, il faut se mettre à l'écoute d'un port socket particulier, décoder les requêtes entrantes, les exécuter et renvoyer les résultats dans la socket.

Les requêtes RMI provenant d'un client sont automatiquement exécutées au sein d'un thread créé par RMI. Le concepteur devra donc éventuellement définir des sections critiques à l'aide de **synchronized**.

Implémentation de l'interface Compte

CompteImpl

- La classe implémente l'interface Compte
- Toutes les méthodes de l'interface doivent être implémentées
- Les implémentations n'ont pas à signaler la levée de RemoteException
- Il est possible d'ajouter d'autres méthodes:
  - publiques
  - privées ou protégées

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

public class CompteImpl implements Compte {
   private float solde = 0.0f;
   private String titulaire;

   // constructeur
   public CompteImpl(String titulaire){
       this.titulaire = titulaire;
   }
   // implémentation de l'interface Compte
   public float lireSolde() {
       return solde;
   }
   public void dépot(float montant)
       throws TransactionException {
       if (montant <= 0)
           throw new TransactionException();
       solde += montant;
   }
   public void retrait(float montant)
       throws TransactionException {
       if ((montant <= 0) || (solde < montant))
           throw new TransactionException();
       solde -= montant;
   }
   // autres méthodes possibles ...
}
```

Implémentation de l’interface Compte

CompteImpl

La classe CompteImpl peut comporter:

- des définitions d’attributs statiques ou non, suivant tous les types et possédant toutes les visibilités,
- toute autre méthode supplémentaire.

Le code applicatif est très peu perturbé par RMI.

Implémentation de l'interface Gestionnaire

GestionnaireImpl

- La méthode ouvrirCompte instancie un autre objet d'implémentation
- Cet objet est retourné suivant son type d'interface, seul connu côté client

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

public class GestionnaireImpl implements Gestionnaire {
   private Hashtable comptes;

   // constructeur
   public GestionnaireImpl() {
       comptes = new Hashtable();
   }

   // implémentation de l'interface Gestionnaire
   public Compte ouvrirCompte(String clé, String nom) {
       Compte c = new CompteImpl(nom);
       comptes.put (clé, c);
       return c;
   }
   public Compte trouverCompte(String clé) {
       Compte c = comptes.get(clé);
       return c;
   }
   // autres méthodes
   public String toString() {
       return comptes.toString();
   }
}
```

Implémentation de l’interface Gestionnaire

GestionnaireImpl

Cette implémentation met en évidence les différences de fonctionnement entre le serveur et le client:

- les objets d’implémentation peuvent instancier d’autres objets d’implémentation, directement à partir des classes d’implémentation,
- ces objets sont manipulés côté client uniquement à travers leurs interfaces Remote.

Réalisation du serveur de Banque

Tâches à effectuer

- Installer un SecurityManager
- Instancier un ou plusieurs objets d'implémentation
- Créer des souches qui seront utilisées par les clients
- Obtenir la référence du serveur de noms
- Exporter leurs souches dans le serveur de noms

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

public class Serveur {
 public static void main(String[] args) {
   try {
     if (System.getSecurityManager() == null) {
       System.setSecurityManager(new SecurityManager());
     }
     Compte c = new CompteImpl("MoiMême");
     Compte stub = (Compte)
                  UnicastRemoteObject.exportObject(c, 0);
     Registry registry = LocateRegistry.getRegistry();
     registry.rebind("UnCompte", stub);
   } catch (Exception e) {
     e.printStackTrace();
   }
 }
}
```

UnicastRemoteObject.exportObject (objet Remote, numéro de port)

- Crée la classe et l'instance de Stub qui sera utilisée par le client
- Permet de définir le port socket utilisé pour la communication entre le client et le serveur, par l'emploi d'un second paramètre après la référence du stub.
  - 0 correspond à un numéro de port quelconque

Réalisation du serveur de Banque

Tâches à effectuer

- Le SecurityManager permet de protéger le serveur contre les opérations indésirables, et bien qu'il ne soit pas utile dans tous les cas de figure, son utilisation est fortement recommandée.
- L'application serveur doit instancier un ou plusieurs objets serveur qui permettront, à leur tour d'accéder à d'autres objets au travers des méthodes de leurs interfaces Remote. Ces objets sont des racines d'arborescences d'objets comparables à des racines de persistance par exemple.
- Le serveur de noms gère une liste de couples (nom, objet). Il permet à l'application cliente d'accéder aux objets serveur à partir de leur nom. La composition du nom est libre, mais un nom doit être lié à un unique objet.
  La classe Registry comporte des méthodes d'accès aux fonctionnalités du serveur:
  - bind() permet de nommer un objet,
  - rebind() permet de nommer ou renommer un objet (elle est souvent utilisée de préférence à bind() car elle peut être utilisée plusieurs fois sans arrêter le serveur de noms),
  - lookup() permet de rechercher un objet à partir de son nom depuis l'application cliente.

UnicastRemoteObject.exportObject (objet Remote, numéro de port)

Attention, il existe une méthode exportObject qui accepte en paramètre unique l'objet Remote et qui génére un Stub à condition que sa classe ait été préalablement créée par appel au compilateur rmic. Cette opération était indispensable avant l'arrivée de JAVA 5.

Le serveur de noms

Lancement manuel

- Lancer **rmiregistry en ligne de commande**

```
rmiregistry 1099
```

- Obtenir sa référence dans le code de l'application serveur

```
Registry registry = LocateRegistry.getRegistry(N° port);
```

Lancement effectué par l'application serveur

- Ne pas lancer le processus rmiregistry en ligne de commande
- Placer dans le code de l'application serveur:

```
Registry registry = LocateRegistry.createRegistry(N° port)
```

Ce mode de lancement interdit le téléchargement des classes Remote par le client.

Le serveur de noms

Notes

rmiregistry est un binaire présent parmi les binaires du JDK.

Réalisation de l'application cliente

Tâches à effectuer

- Installer un SecurityManager si le client est une application autonome
- Obtenir une référence sur le serveur de noms
- Obtenir une référence sur un objet Remote
- Invoquer des méthodes en tenant compte des exceptions de type RemoteException

```
import java.rmi.*;

public class Client {
 public static void main(String[] args) {
   try {
     if (System.getSecurityManager() == null) {
       System.setSecurityManager(new SecurityManager());
     }
     Registry registry =
       LocateRegistry.getRegistry("localhost", 1099);
     Compte c = (Compte) registry.lookup ("UnCompte");

     c.dépot(1000.0f);
     c.retrait(500);
     System.out.println (c.lireSolde());
   } catch (Exception e) {
       e.printStackTrace();
   }
 }
}
```

Réalisation de l’application cliente

Tâches à effectuer

La syntaxe à utiliser pour obtenir une référence distante est de la forme:

```
rmi://host:port/nom
```

- serveur est le nom ou l’adresse IP de la machine sur laquelle tourne le serveur de noms rmiregistry. Par raison de sécurité, cette machine est forcément la même que celle du processus serveur.
- port est le numéro de port utilisé par le serveur de noms rmiregistry (valeur par défaut: 1099).
- nom est la chaîne de caractères utilisée côté serveur pour enregistrer l’objet (premier paramètre de bind() ou rebind()).

Client de type applet

Exemple

- Le SecurityManager est déjà installé
- Si la sécurité n'est pas spécifiquement paramètrée, le serveur RMI doit se trouver sur le serveur WEB

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

public class Client extends java.applet.Applet {
 private Gestionnaire gref;

 public void init() {
   try {
     // obtention d'une référence Remote
     String host = getCodeBase().getHost();
     Registry registry =
              LocateRegistry.getRegistry(host, 1099);
     gref = (Gestionnaire) registry.lookup("MaBanque");
   } catch (Exception e) {
       // traitement d'erreur
   }
 }
 public void start() {
   try {
     // invocations de méthodes distantes
     Compte cref = gref.trouverCompte("12345");
     cref.dépot(1000);
     cref.retrait(2000);
     repaint();
   } catch (Exception e) {
       // traitement d'erreur
   }
 }
}
```

Client de type applet

Exemple

La référence du serveur RMI peut être obtenue par l'emploi de getCodeBase().getHost():

- getCodeBase() retourne l'URL de la classe d'applet sur le serveur,
- getHost() retourne le nom d'hôte inclus dans cette URL.

Compilation des différents fichiers

Fichiers Java

- Interfaces de type Remote
- Classes d'implémentation et classes associées
- Applications serveur et client

```
javac -d . package/Interface.java
javac -d . package/InterfaceImpl.java
javac -d . Serveur.java
javac -d . Client.java
```

Stubs

- Il n'est plus nécessaire d'employer le compilateur rmic depuis JAVA 5.
- Les classes des Stubs sont générées dynamiquement en stockées en mémoire.

Compilation des différents fichiers

Fichiers Java

La compilation des différents fichiers sources s’effectuent de façon habituelle, sans aucune option spécifique à RMI.

Types d'installations RMI et procédures de lancement

Client de type Applet

- RMI est totalement transparent pour le client
- Une procédure de lancement particulière doit être respectée pour rmiregistry et le serveur
- Le dialogue client-serveur peut franchir un firewall

Client application autonome non téléchargée

- Les applications sont installées manuellement

Client application autonome téléchargée

- Un module générique présent sur le poste client permet de télécharger l'application cliente
- Ce mode requiert un serveur HTTP sur le serveur
- La procédure de lancement des applicatifs serveur est identique à celle du mode applet

Types d’installations RMI et procédures de lancement

Client de type Applet

L’application cliente doit pouvoir télécharger les classes RMI (stubs notamment) depuis le serveur. Pour cela, ce dernier doit être paramétré spécifiquement lors de son lancement.

RMI est capable d’encapsuler ses requêtes au sein de requêtes HTTP pour franchir un firewall. Dans ce cas, un script CGI doit être utilisé côté serveur afin de transmettre les requêtes du serveur HTTP au serveur RMI. Ce script est fourni dans le répertoire bin de l’installation standard.

Client application autonome non téléchargée

Dans ce mode, les différentes classes sont installées sur chaque poste:

- toutes les classes à l’exception de celles du client sur le poste serveur,
- toutes les classes relatives à l’applicatif client sur le poste client.

Client application autonome téléchargée

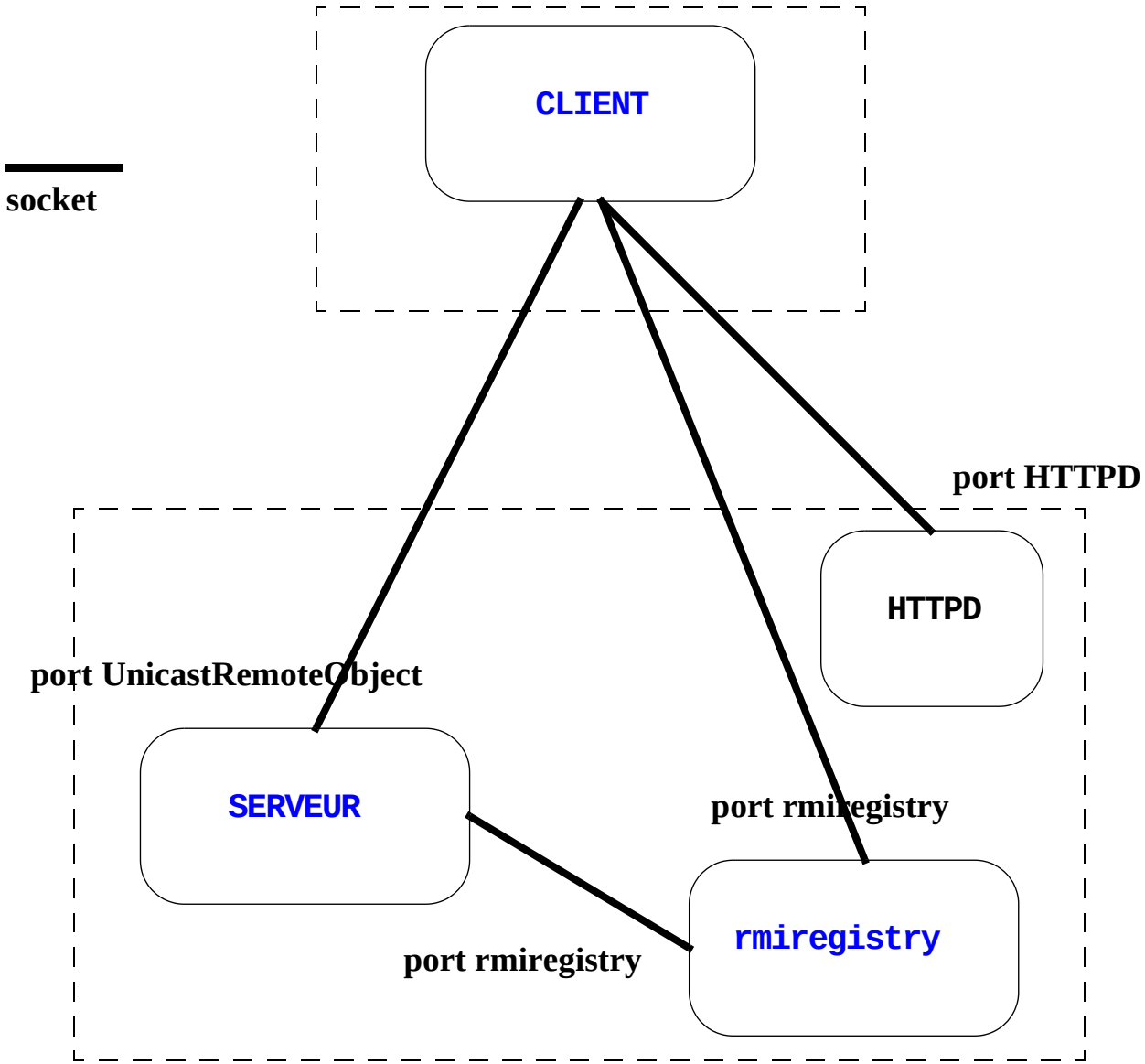
Ce mode permet de bénéficier du mécanisme de téléchargement des classes habituellement dévolu aux applets.Un applicatif de téléchargement est installé sur le poste client. Son rôle est de télécharger le véritable applicatif client depuis le serveur et de le lancer. Pour cela, il s’appuie sur un démon HTTP tournant sur le poste serveur.

Ce mode n’est pas étudié dans ce chapitre. Il est présenté de façon détaillé dans les spécifications RMI disponibles sur le site Javasoft (www.javasoft.com).

Echanges entre processus et rmiregistry

Schéma général

- port UnicastRemoteObject peut être déterminé par le programmeur pour chaque objet d'implémentation
- port rmiregistry fixé par défaut à 1099



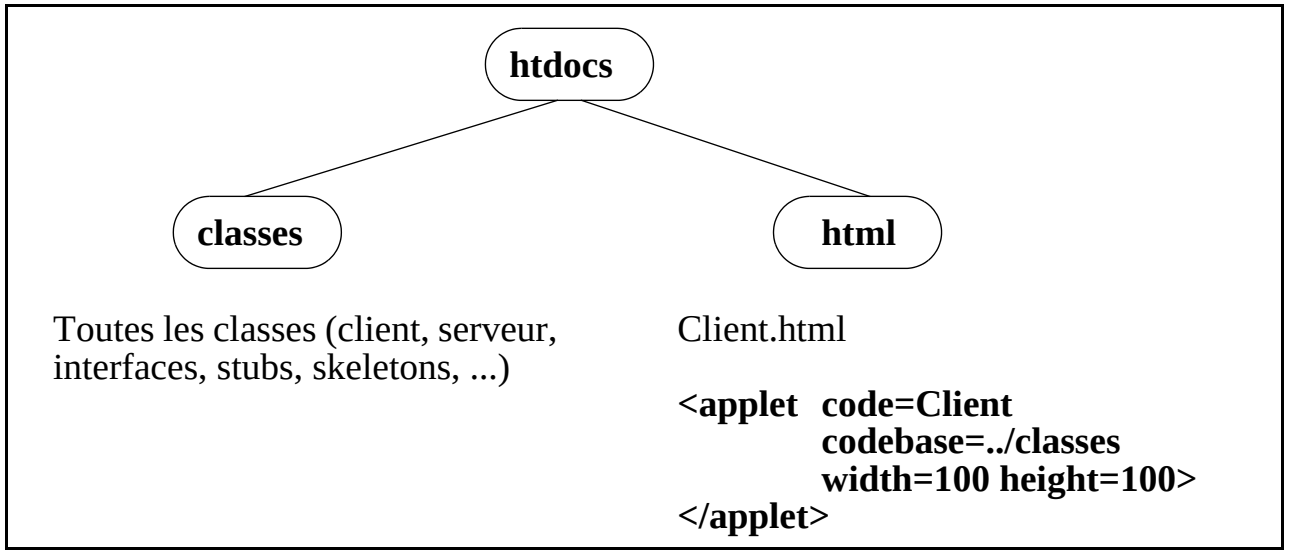
Echanges entre processus et rmiregistry

Schéma général

- Le serveur de noms rmiregistry écoute les requêtes sur un port qui est fixé par défaut à **1099** et qui peut être modifié au moment du lancement.
  Ce serveur de noms est destiné à offrir au minimum une référence d'objet de type **Remote** au client distant. Les autres objets de type Remote peuvent être obtenus par le moyen d'envoi de messages à cet objet initial. rmiregistry a donc essentiellement une fonction permettant d'initier les échanges client-serveur (**bootstrap**).
- Chaque objet d'implémentation RMI peut utiliser un port socket spécifique dans son dialogue avec le client. Ce port peut être déterminé lors de l'appel de la méthode exportObject de UnicastRemoteObject.
- HTTPD n'est pas utile dans le cas où l'application RMI est de type autonome et ne fait appel à aucun téléchargement de classe. Ce point est étudié plus en détail par la suite.

Client RMI de type applet

Installation du serveur



Lancement du serveur

- Lancer **rmiregistry**

```
rmiregistry 1111
```

CLASSPATH ne doit pas permettre d'atteindre les stubs

- Lancer le **serveur** (hypothèse: httpd utilise le port 8080)

```
java -Djava.rmi.server.codebase=http://serveur:8080/classes/ Serveur
```

Client RMI de type applet

Installation du serveur

Le répertoire destiné à contenir les classes de l'application cliente (l'applet ici) doit notamment contenir:

- la classe de l'applet et toutes les classes qu'elle référence,
- les interfaces de type Remote.

En ce qui concerne le serveur RMI, nous aurons besoin:

- de la classe du serveur et de toutes les classes qu'il référence,
- des interfaces de type Remote et toutes les classes d'implémentation associées.

Lancement du serveur

- rmiregistry

Il faut veiller à ce que rmiregistry ne dispose pas d'un CLASSPATH permettant d'atteindre les stubs localement. Si c'était le cas, le client ne serait plus en mesure de télécharger ces classes.

- Le serveur

Le serveur doit connaître l'URL du répertoire contenant les stubs utilisés par le client. Cette URL est passée à rmiregistry, qui s'en servira pour permettre au client de télécharger les stubs.

Client RMI de type application autonome non téléchargée

Installation

- Serveur
  - L'application serveur: Serveur.class
  - Les interfaces Remote: Compte.class et Gestionnaire.class
  - Les classes d'implémentation: CompteImpl.class et GestionnaireImpl.class
- Client
  - L'application cliente: Client.class
  - Les interfaces Remote: Compte.class et Gestionnaire.class

Lancement

- Pour chaque application, CLASSPATH doit être définie de façon à permettre un chargement local de toutes les classes.
- Serveur

```
rmiregistry 1099
java Serveur
```

- Client

```
java Client
```

Client RMI de type application autonome non téléchargée

Installation

Les applications doivent être installées de façon identique aux applications autonomes Java classiques.

Lancement

Il faut veiller à ce que la variable d'environnement CLASSPATH permette d'atteindre les classes dans tous les cas:

- rmiregistry doit pouvoir atteindre tous les stubs,
- le serveur doit pouvoir charger toutes ses classes, les interfaces Remote,
- le client doit pouvoir référencer sa classes et toutes celles qu'il utilise ainsi que toutes les interfaces Remote et stubs.

Architectures distribuées (Polytech)
Remote Method Invocation: aspects avancés

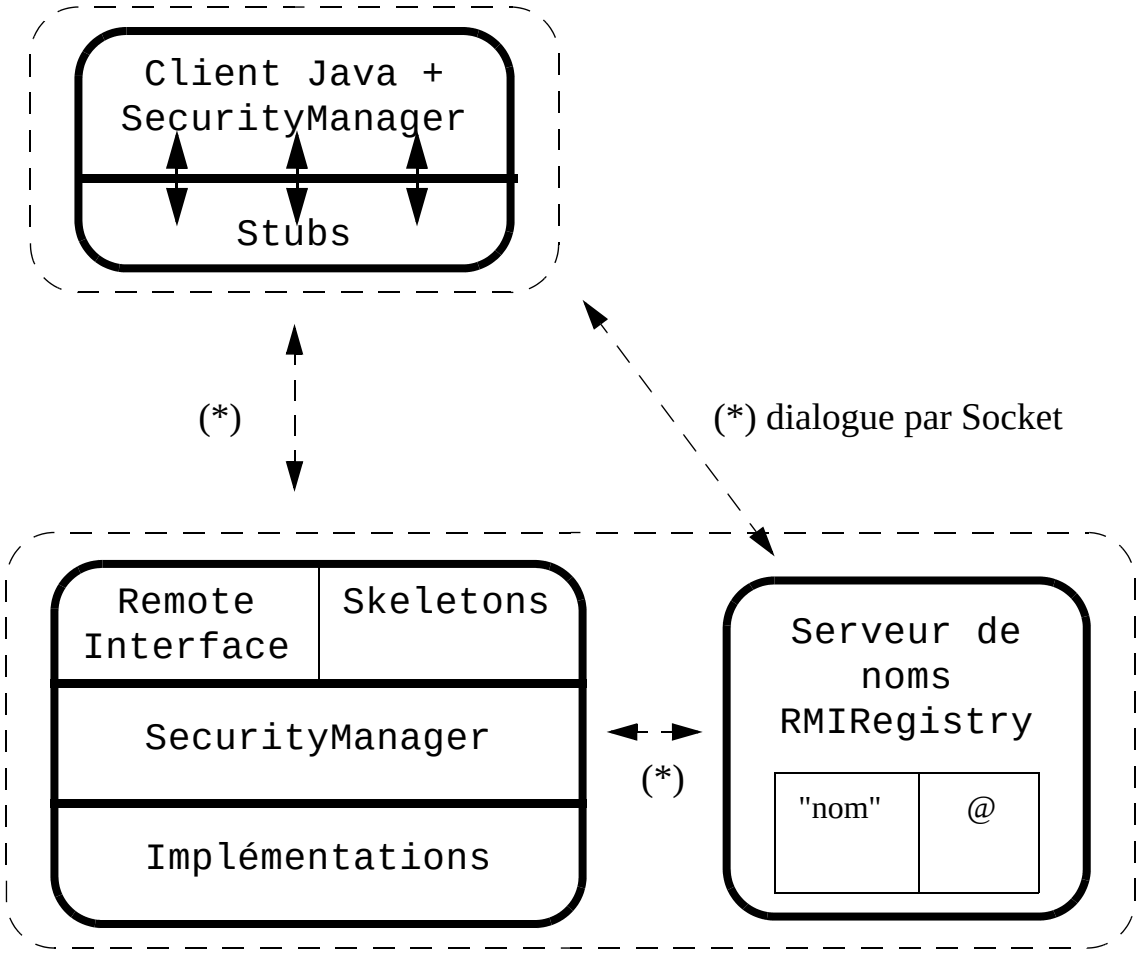
Version 3.0

- Réalisation d'un client RMI autonome téléchargeable
- Utilisation de sockets spécifiques avec les SocketFactory
- Mise en place de callbacks entre le serveur et les clients
- Activation des objets serveurs

Architecture RMI

Schéma général

- Serveur: application autonome
- Client: application autonome ou applet
- Le serveur de noms fournit les objets initiaux aux clients
- Les communications s'effectuent par sockets



Architecture RMI

Schéma général

- Un serveur RMI est une application autonome Java dont les tâches sont les suivantes:
  - installation éventuelle d'une politique de sécurité spécifique pour protéger le serveur si celui-ci télécharge des classes, par exemple en provenance d'un client qui jouerait également le rôle d'un serveur
  - instanciation des objets serveurs
  - enregistrement des objets dans le serveur de noms
  - l'écoute des requêtes clients est assurée automatiquement par RMI

- Un client RMI peut être:
  - une application autonome
  - une applet

- rmiregistry est un serveur de noms minimal qui offre uniquement un service permettant de retrouver un objet serveur d'après son nom. Si l'on désire un service plus évolué (authentification par exemple), il est tout-à-fait possible de lui substituer un autre mécanisme (LDAP, serveur de noms CORBA, ...)

- Les communications entre client-serveur, client-rmiregistry et serveur-rmiregistry s'effectuent par Sockets. Les numéros de ports utilisés sont paramétrables. Il est également possible d'utiliser des sockets de types particuliers (SSL par exemple).

Types d’installations RMI et procédures de lancement

Client de type Applet

- RMI est totalement transparent pour le client
- Une procédure de lancement particulière doit être respectée pour rmiregistry et le serveur
- Le dialogue client-serveur peut franchir un firewall

Client application autonome non téléchargée

- Les applications sont installées manuellement

Client application autonome téléchargée

- Un module générique présent sur le poste client permet de télécharger l’application cliente
- Ce mode requiert un serveur HTTP sur le serveur
- La procédure de lancement des applicatifs serveur est identique à celle du mode applet

Types d’installations RMI et procédures de lancement

Client de type Applet

L’application cliente doit pouvoir télécharger les classes RMI (stubs notamment) depuis le serveur. Pour cela, ce dernier doit être paramétré spécifiquement lors de son lancement.

RMI est capable d’encapsuler ses requêtes au sein de requêtes HTTP pour franchir un firewall. Dans ce cas, un script CGI doit être utilisé côté serveur afin de transmettre les requêtes du serveur HTTP au serveur RMI. Ce script est fourni dans le répertoire bin de l’installation standard.

Client application autonome non téléchargée

Dans ce mode, les différentes classes sont installées sur chaque poste:

- toutes les classes à l’exception de celles du client sur le poste serveur,
- toutes les classes relatives à l’applicatif client sur le poste client.

Client application autonome téléchargée

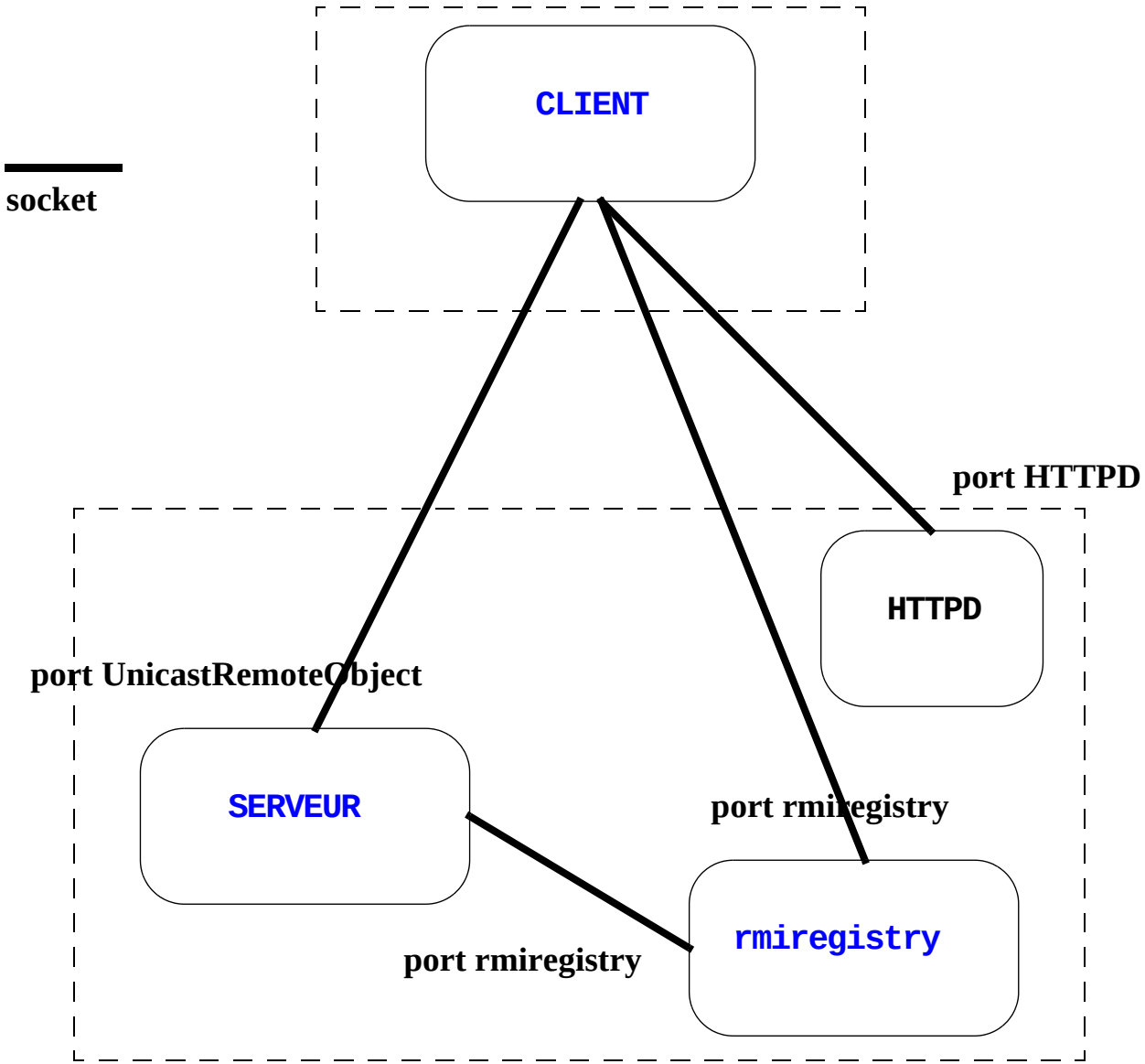
Ce mode permet de bénéficier du mécanisme de téléchargement des classes habituellement dévolu aux applets.Un applicatif de téléchargement est installé sur le poste client. Son rôle est de télécharger le véritable applicatif client depuis le serveur et de le lancer. Pour cela, il s’appuie sur un démon HTTP tournant sur le poste serveur.

Ce mode n’est pas étudié dans ce chapitre. Il est présenté de façon détaillé dans les spécifications RMI disponibles sur le site Javasoft (www.javasoft.com).

Echanges entre processus et rmiregistry

Schéma général

- port UnicastRemoteObject peut être déterminé par le programmeur pour chaque objet d'implémentation
- port rmiregistry fixé par défaut à 1099



Echanges entre processus et rmiregistry

Schéma général

- Le serveur de noms rmiregistry écoute les requêtes sur un port qui est fixé par défaut à **1099** et qui peut être modifié au moment du lancement.
  Ce serveur de noms est destiné à offrir au minimum une référence d'objet de type **Remote** au client distant. Les autres objets de type Remote peuvent être obtenus par le moyen d'envoi de messages à cet objet initial. rmiregistry a donc essentiellement une fonction permettant d'initier les échanges client-serveur (**bootstrap**).
- Chaque objet d'implémentation RMI peut utiliser un port socket spécifique dans son dialogue avec le client. Ce port peut être déterminé lors de l'appel de la méthode exportObject de UnicastRemoteObject.
- HTTPD n'est pas utile dans le cas où l'application RMI est de type autonome et ne fait appel à aucun téléchargement de classe. Ce point est étudié plus en détail par la suite.

Client RMI de type application autonome téléchargeable

Principes de fonctionnement

- Une application cliente générique télécharge la véritable application cliente depuis le serveur
- Ce processus requiert un démon **httpd** sur le serveur
- Code du client de chargement

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

public class Bootstrap {
 public static void main(String[] args)
 {
   try {
     // SecurityManager
     System.setSecurityManager(
                  new SecurityManager());
     // téléchargement de la classe du client
     Class c = RMIClassLoader.loadClass("Client");
     // instanciation du client
     Runnable r = (Runnable)c.newInstance();
     // lancement du 'véritable' client
     r.run();
   } catch (Exception e) {
     e.printStackTrace();
   }
 }
}
```

- Lancement du client

```
java -Djava.rmi.server.codebase=http://serveur:8080/classes/ Boot
```

Client RMI de type application autonome téléchargeable

Principes de fonctionnement

- RMIClassLoader offre la possibilité de télécharger une classe à l'aide HTTP
- Cette classe est manipulable à travers le type java.lang.Class, qui offre, outre les services d'auto-description du package java.lang.reflect, la capacité d'instanciation
- L'appel à newInstance() permet d'instancier la classe téléchargée, à condition que celle-ci contienne un constructeur public sans paramètre
- L'instance obtenue est manipulée à travers le type java.lang.Runnable, qui n'a pas à être téléchargé car toujours présent dans la machine virtuelle locale. Cela impose que la classe du 'véritable client' implémente l'interface Runnable. Ce choix a été fait afin de faciliter l'écriture de la classe Bootstrap
- Lorsque l'application Bootstrap est lancée, il faut définir la propriété **java.rmi.server.codebase** afin de spécifier l'URL permettant d'accéder au code du véritable client

Client RMI de type application autonome téléchargeable

Lancement du serveur

- Processus identique au lancement du serveur RMI avec des clients de type applet
- Lancer **httpd** (par exemple sur le port 8080)
- Lancer **rmiregistry**

```
rmiregistry 1111
```

CLASSPATH ne doit pas permettre d'atteindre les stubs

- Lancer le **serveur** avec la propriété **java.rmi.server.codebase**

```
java -Djava.rmi.server.codebase=http://serveur:8080/classes/ Serveur
```

Ne pas oublier le / final dans la définition de java.rmi.server.codebase

Client RMI de type application autonome téléchargeable

Lancement du serveur

- httpd

Le démon httpd est utilisé par le client de type Bootstrap pour télécharger les classes. Cette opération est effectuée par RMIClassLoader.

- rmiregistry

Le serveur de noms rmiregistry est une application Java qui permet au client distant d'obtenir au moins la référence d'un objet Remote initial.

Il faut veiller à ce que rmiregistry ne dispose pas d'un CLASSPATH permettant d'atteindre les stubs localement. Si c'était le cas, le client ne serait plus en mesure de télécharger ces classes.

- Le serveur

Le serveur doit connaître l'URL du répertoire contenant les stubs utilisés par le client. Cette URL est passée à rmiregistry, qui s'en servira pour permettre au client de télécharger les stubs.

Sockets spécifiques

Les interfaces SocketFactory

- Une paire de sockets factory serveur/client par objet d'implémentation
- Il faut paramétrer la sécurité pour pouvoir créer des sockets spécifiques

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

public class MiseAjourImpl implements MiseAjour { // ... }
public class ServSF implements RMIServerSocketFactory {
  public ServerSocket createServerSocket(int port) { // ...}
}
public class ClientSF implements RMIClientSocketFactory {
  public Socket createSocket(String host, int port) { // ...}
}
public class Serveur {
 public static void main(String[] args) {
   try {
     if (System.getSecurityManager() == null) {
       System.setSecurityManager(new SecurityManager());
     }
     MiseAJour m = new MiseAJourImpl();
     MiseAJour stub = (MiseAJour)
       UnicastRemoteObject.exportObject(
           c, 0, new ClientSF(),new ServerSF()
         );
   } catch (Exception e) {
     e.printStackTrace();
   }
 }
}
```

```
grant {
   permission java.lang.RuntimePermission "setFactory";
};
```

Sockets spécifiques

Les interfaces SocketFactory

Ces interfaces permettent de définir des objets responsables de la création des Sockets utilisées par RMI. Chaque objet d'implémentation RMI peut être utilisé avec une paire de SocketFactory différente.

La création de sockets spécifiques est une opération assez complexe, qui n'est pas abordée par ce cours.

Ce mécanisme permet d'agir aisément sur les couches de communication de RMI. Il est par exemple possible de s'appuyer sur des sockets de type SSL plutôt que sur des sockets standards. Plusieurs éditeurs proposent des solutions prêtes à l'emploi:

- www.phaos.com
- www.baltimore.ie
- www.jcp.co.uk

Réalisation de callbacks RMI

Définition d'une callback

- Méthode appelée par le serveur sur un client
- Avec RMI, l'objet client doit également être de type **java.rmi.Remote**

Exemple: application bancaire

- Un service permet à des clients distants d'obtenir en temps-réel les informations concernant les opérations passées sur le compte
- Interface du service Serveur MiseAJour

```
import java.rmi.*;
public interface MiseAJour extends Remote {
public void connecter(Client c) throws RemoteException;
public void deconnecter(Client c)
        throws RemoteException;
}
```

- Interface du Client avec méthode informer() de type callback

```
import java.rmi.*;
public interface Client extends Remote {
public void informer(String operation)
        throws RemoteException;
}
```

Réalisation de callbacks RMI

Définition d'une callback

Une callback est une méthode que le serveur invoque sur un client, généralement à la suite d'une action de ce dernier sur le serveur.

Avec RMI

Le serveur doit connaître l'interface des méthodes accessibles à distance du client. Il faut donc que celles-ci soient décrites par une interface de type java.rmi.Remote.

Exemple

Une extension du programme Bancaire permet à des utilisateurs distants d'obtenir en temps-réel des informations sur les opérations concernant leurs comptes en banque. Le service fonctionne à partir du moment ou l'utilisateur s'est préalablement connecté. Il reçoit alors toutes les mises-à-jour, et ce jusqu'à sa déconnexion.

La méthode informer() est invoquée par le serveur dès qu'une opération est effectuée.

Sur le poste client doit se trouver la classe d'implémentation des services de l'interface Client. Cette classe **ClientImpl** héritera de UnicastRemoteObject ou redéfinira les méthodes d'écoute des requêtes RMI.

Réalisation de callbacks RMI

Enregistrements et notifications

- Ne pas oublier de prendre en charge:
  - UnicastRemoteObject.exportObject
  - UnicastRemoteObject.unexportObject

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

public class MiseAjourImpl implements MiseAjour {
   private ArrayList connectes;

   public MiseAJourImpl(String nom)
            throws RemoteException {
       connectes = new ArrayList()
   }
   public synchronized void connecter(Client c) {
       connectes.add(c);
       notifier(c, "Bienvenue");
   }
   public synchronized void deconnecter(Client c) {
       connectes.remove(connectes.lastIndexOf(c));
       notifier(c, "Au revoir");
   }
   // invocation de la callback
   public void notifier (Client client, String info)
       try {
           client.informer(info);
       } catch (RemoteException e) {
           try {
               deconnecter(client);
           } catch (RemoteException e2) {
           }
       }
   }
}
```

Réalisation de callbacks RMI

Invocation des callbacks

La callback est invoquée en suivant les mêmes règles que pour l'invocation de méthodes distantes classiques:

- passage de paramètres de type **java.io.Serializable** ou **java.rmi.Remote** uniquement,
- gestion des exceptions de type **java.rmi.RemoteException**

Le concepteur doit veiller à l'intégrité des données manipulées en utilisant éventuellement des sections critiques définies à l'aide de **synchronized**.

Politique d'activation des objets serveurs

- **java.rmi.server.UnicastRemoteObject** permet de définir des objets serveurs tout le temps actifs
- **java.rmi.activation.Activatable** permet de définir des objets serveurs activables sur demande

Caractéristiques

- les objets Activable sont chargés en mémoire uniquement lorsqu'ils reçoivent une requête
- une démon **rmid** est chargé de lancer des machines virtuelles Java si nécessaire
- il faut préparer un ensemble d'informations qui seront utilisées pour paramétrer l'objet au moment de son activation
- la politique d'activation dépend uniquement du serveur
  - aucune modification du code client
  - pas de modification des interfaces Remote

Rendre un objet serveur activable

- Hériter de java.rmi.activation.Activatable
- Définir un constructeur pour enregistrer l'objet dans le système d'activation
- Créer une classe de paramétrage de l'activation (setup)

Politique d’activation des objets serveurs

Caractéristiques

JAVA 2 introduit la possibilité d’instancier les objets serveurs RMI uniquement à la demande, c’est-à-dire au moment où une requête RMI leur est envoyée.

C’est un programme particulier, **rmid**, qui est chargé d’activer les objets d’implémentation en cas de besoin. Ce démon tourne au sein d’une machine virtuelle Java et est capable de lancer d’autres machines virtuelles.

Rendre un objet serveur activable

La classe de paramétrage de l’activation devra notamment effectuer les tâches suivantes:

- installer un SecurityManager
- définir les paramètres d’activation
  - groupes d’objets activables attachés à différentes machines virtuelles d’exécution,
  - informations permettant d’instancier les objets serveurs
- enregistrer les informations dans rmid
- nommer l’objet serveur

Activation: adaptation de la classe GestionnaireImpl

Exemple

- Fournir un constructeur à deux arguments pour enregistrer l'objet serveur dans le système d'activation
- Le second paramètre contient les données d'initialisation sous la forme d'un objet sérialisé

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

public class ActivatableGestionnaire extends Activatable
                          implements Gestionnaire {
 private Hashtable comptes;

 public ActivatableGestionnaire(
        ActivationID id, MarshalledObject data)
        throws RemoteException {
   super (id, 0);
   comptes = new Hashtable();
 }

 public Compte ouvrirCompte(String clé, String nom)
                     throws RemoteException {
   Compte c = new CompteImpl(nom);
   comptes.put (clé, c);
   return c;
 }
 public Compte trouverCompte(String clé)
                     throws RemoteException {
   Compte c = comptes.get(clé);
   return c;
 }
 // autres méthodes
 public String toString() {
   return comptes.toString();
 }
}
```

Activation: adaptation de la classe GestionnaireImpl

```
public ActivatableGestionnaire(
                        ActivationID id, MarshalledObject data)
        throws RemoteException {
   super (id, 0);
   ...
}
```

- id est un identificateur unique de l'objet,
- data contient les données d'initialisation de l'objet, définies au moyen de la classe de type Setup étudiée par la suite,
- l'appel à super (id, 0) permet d'enregistrer l'objet d'implémentation dans le système d'activation. Le paramètre 0 indique qu'un port anonyme sera utilisé. Il est également possible d'utiliser un numéro de port explicite. Cela peut-être utile dans un cadre Internet, où l'on souhaiterait paramètrer le filtrage d'un firewall.

De façon identique à ce qui existe dans UnicastRemoteObject, la classe Activatable offre des constructeurs permettant de spécifier des fabriques de sockets spécifiques.

Classe de paramètrage de l'activation (setup)

Structure générale

- Installation d'un SecurityManager

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

public class Setup {
 public static void main (String[] args) {
  try {
   System.setSecurityManager(new SecurityManager());
   //
   // code de paramétrage de l'activation
   // ...
   //
  } catch (Exception e) {
   e.printStackTrace();
  }
 }
}
```

Classe de paramètrage de l’activation (setup)

Structure générale

La classe de paramètrage de l’activation remplace la classe de lancement du serveur que l’on utilise avec une solution de type UnicastRemoteObject.

Ses tâches sont les suivantes:

- installation d’un SecurityManager,
- création d’un ActivationGroup,
- création d’un ActivationDesc,
- enregistrement auprès de rmid,
- nommage du stub créé par la précédente opération.

Le code présenté par la suite est à insérer à la place des commentaires du bloc **try { } catch () {}.**

Classe de paramètrage de l'activation (setup)

ActivationGroup

- Regroupement d'objets serveurs activables
- Une machine virtuelle d'exécution est attachée à un groupe

```
Properties props = new Properties();
props.put ("java.security.policy",
           "/home/moi/mypolicy");
ActivationGroupDesc.CommandEnvironment ace = null;
ActivationGroupDesc grDesc =
      new ActivationGroupDesc(props, ace);
ActivationGroupID id =
      ActivationGroup.getSystem().registerGroup(grDesc);
ActivationGroup.createGroup(id, grDesc, 0);
```

- Contrôle l'activation/désactivation des objets serveurs

ActivationDesc

- Indique l'endroit ou réside la classe de l'objet activable
- Permet de définir les paramètres qui seront passés au constructeur de l'objet serveur

```
// URL de la classe de l'objet à activer
String location = "file:/home/moi/mesclasses/";
// paramètres
MarshalledObject par = new MarshalledObject("RMI Bank");
//
ActivationDesc desc = new ActivationDesc (
   "ActivatableGestionnaire", location, par);
```

Classe de paramètrage de l’activation (setup)

ActivationGroup

Pour définir un ActivationGroup, il faut disposer du privilège java.lang.RuntimePermission intitulé "setFactory".

ActivationDesc

Ce descripteur permet au groupe d’avoir toutes les informations nécessaires à l’activation de l’objet:

- nom de la classe d’objet serveur (attention le nom complet de la classe est requis si celle-ci est membre d’un package),
- localisation de la classe compilée sous la forme d’une chaîne à la syntaxe de type URL,
- paramètres à passer éventuellement au constructeur de l’objet serveur,
- groupe de rattachement.

Classe de paramètrage de l'activation (setup)

Enregistrement dans le système d'activation

- rmid doit être actif avant l'exécution de ce code
- le nommage est effectué sur le bouchon et non pas sur l'objet d'implémentation, qui n'existe pas encore

```
//
// enregistrement dans rmid
//
Gestionnaire gest =
            (Gestionnaire)Activatable.register(desc);
//
// nommage du stub
//
Naming.rebind ("MaBanque", gest);
//
// fin de setup
//
System.exit(0);
```

Classe de paramètrage de l’activation (setup)

Enregistrement dans le système d’activation

- Enregistrement dans rmid

La méthode d’enregistrement du descripteur d’activation retourne une instance dont le type est celui de l’interface Remote. Il faut que rmid soit actif au moment de l’exécution de ce code.

- Nommage du stub

L’objet d’implémentation n’est plus nommé directement car il n’existe pas encore. C’est le bouchon destiné au poste client qui est nommé.

- Fin de setup

Cette classe sert uniquement au paramètrage de l’activation et n’est pas responsable de l’écoute des requêtes des clients. Il est donc possible de quitter l’application.

Lancement côté serveur

Dans cet ordre

- rmiregistry
- rmid

```
rmid -J-Djava.security.policy=mysecurity.policy
```

- la classe Setup

Lancement côté serveur

Notes

Il faut penser à lancer rmiregistry et rmid avant la classe de paramètrage de l’activation.

Suivant le mode d’installation et de lancement désiré, on se référera à la procédure concernant les serveurs de type UnicastRemoteObject.

Architectures distribuées (Polytech)
Problématiques liées aux architectures à base d'objets répartis

Version 3.0

- Montée en charge
- Administration
- Accès aux ressources
- Séparation entre code fonctionnel et code technique
- Concurrence d'accès et gestion des transations

Problèmes courants

Conception des applications

- L'architecture demeure complexe à établir
- Il faut fournir beaucoup de code technique
  - gestion du cycle de vie des objets
  - gestion des threads
  - gestion des transactions
  - gestion des connexions vers le SGBD
- Il faut séparer le code fonctionnel du code technique

Administration

- Fragilité liée au nombre de serveurs
- Complexité

Problèmes courants

De nombreux besoins restent à couvrir

Les applications client-serveur multi-niveaux nécessitent de nombreuses optimisations au niveau du code purement technique. En effet, il faut fréquemment fournir des mécanismes permettant d'optimiser:

- la gestion des connexions réseau afin d'éviter les créations dynamiques,
- le lancement de threads de traitements client,
- la gestion mémoire et l'instanciation des objets.

Serveurs d'applications et composants métier

La solution consiste à séparer nettement les services techniques du code métier, en laissant seulement au développeur la possibilité de travailler sur ce dernier. Le serveur d'applications est une structure d'accueil pour composants métier qui offre toutes les infrastructures nécessaires à la réalisation d'applications client-serveur multi-niveaux. Il comporte notamment tous les services techniques que l'on doit toujours réaliser pour ce type d'applications:

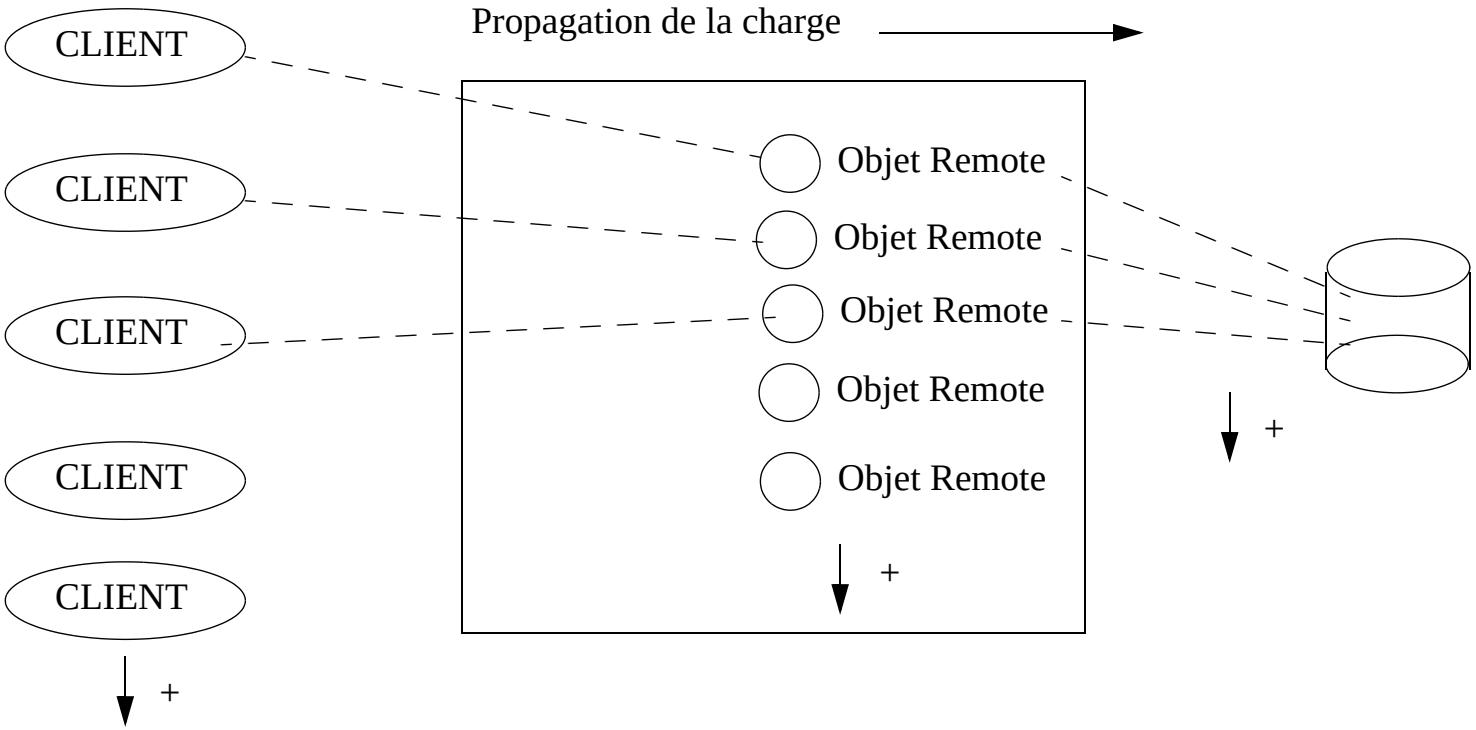
- cycle de vie des objets avec pools d'objets pré-construits,
- accès réseau,
- gestion de pools de connexions réseau (client et SGBD),
- gestion transactionnelle, ...

Les applications sont alors réalisées par assemblage de composants comportant presque exclusivement du code métier.

Montée en charge

Que se passe-t-il quand le nombre de clients augmente ?

- Avec une gestion dynamique des allocations:



- Augmentation du nombre d'objets présents sur le serveur liée à l'augmentation du nombre de clients
- Augmentation des ressources nécessaires au fonctionnement: connexions vers SGBD

Montée en charge

Notes

Si on alloue dynamiquement les objets Remote sur le serveur en fonction de la demande des clients, on s'expose à différents risques:

- la non-maîtrise de l'espace mémoire utilisé par l'application si le nombre d'objets Remote devient trop important,
- la propagation des problèmes de charge aux ressources exploitée par le serveur telles que les connexions vers un SGBD par exemple.

Montée en charge

Mécanismes d'accompagnement

- Mise en oeuvre du design-pattern Pool
  - une réserve d'objets est allouée au lancement
  - une demande d'allocation provenant du client se traduit par la mise à disposition d'un objet pré-alloué
  - une fin de connexion provenant du client se traduit la remise à disposition de l'objet sur le serveur et non pas par sa désallocation
  - un dimensionnement préalable de la réserve est indispensable au bon fonctionnement du mécanisme
- Avantages
  - l'usage mémoire au niveau du serveur est connu à l'avance
  - le travail de l'allocateur mémoire est simplifié
  - l'obtention d'un objet est rapide
- Inconvénients
  - limitation des ressources
  - discipline des utilisateurs indispensable

Montée en charge

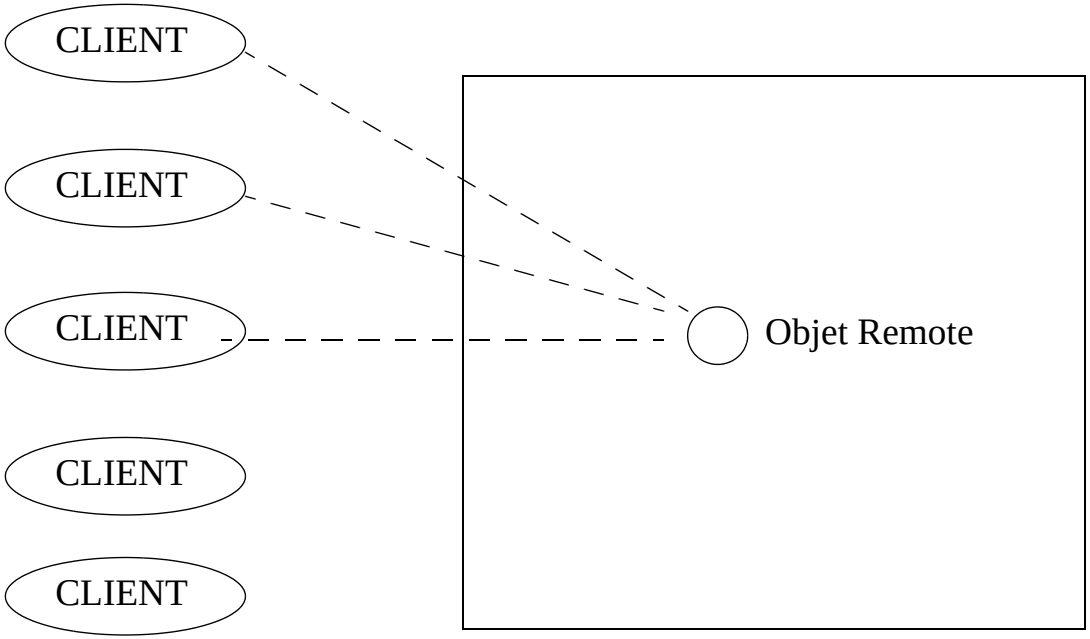
Notes

On parle souvent de "scalabilité" des applications côté serveur.

Concurrences d'accès

Exemple: un objet Remote partagé entre plusieurs clients

- Certains clients peuvent modifier cet objet ...
- ... alors que d'autres cherchent à obtenir son état



Il faut gérer les sections critiques

- soit manuellement, à l'aide de blocs "synchronized"
- soit en mettant en oeuvre des "skeletons" spécifiques qui offrent un mécanisme dédié (EJB par exemple)

Concurrences d’accès

Notes

Gestion des transactions

Complexité liée à la "dissémination" potentielle des objets

- Une transaction pourrait impliquer plusieurs objets répartis sur différents serveurs
  - protocole de type "Commit à deux phases" (2PC)
- Il est nécessaire de mettre en oeuvre un mécanisme permettant la gestion de transactions distribuées
  - Object Transaction Service de CORBA
  - Java Transaction Service de JavaEE
  - Microsoft Transaction Service
- La réalisation d'un tel mécanisme par ses propres moyens est difficilement envisageable.

Gestion des transactions

Notes

Administration

Plus grande complexité des tâches d'administration

- Importance du réseau
- Nécessité de penser spécifiquement la sécurité
- Fragilité inhérente à la répartition
  - impose des mécanismes supplémentaires de reprise sur panne, avec équilibrage de charge et réplication d'état

Administration

Notes

Accès aux ressources

Nature des ressources

- Connexions SGBD
- Connexions applications tierces
- Descripteurs de fichiers
- ...

Problèmes posés par l'allocation dynamique des ressources

- Demandes liées au nombre de clients pouvant conduire à l'épuisement des ressources
- Performances médiocres
  - la connexion à la ressource entraine souvent l'exécution de mécanismes lourds, comme les contrôles de sécurité par exemple
  - la ressource est fréquemment distante, ce qui implique l'emploi du réseau

Accès aux ressources

Notes

Accès aux ressources

Optimisations

- Mise en oeuvre du design-pattern Pool
  - Réserves de connexions ouvertes au lancement du serveur
  - Etude de dimensionnement nécessaire
  - Impose une discipline des utilisateurs
- Plusieurs implémentations sont disponibles
  - mécanismes intégrés aux conteneurs des serveurs d'applications
  - librairies tierces

Accès aux ressources

Notes

Proximité du code fonctionnel avec le code technique

Le risque

- Produire des composants mélangeant code fonctionnel et code technique

Solutions

- Organisation rigoureuse
- Sélection de mécanismes proposant des implémentations répondant aux problèmes soulevés et permettant une meilleur organisation:
  - conteneurs des serveurs d'applications

Proximité du code fonctionnel avec le code technique

Notes

Architectures distribuées (Polytech)
Introduction aux Services Web

Version 3.0

- Services Web : principes de base
- Les besoins qui expliquent l'avènement des Web Services
- Services WEB et protocoles associés : SOAP, WSDL, UDDI

Les services WEB

Les besoins

- Permettre les échanges inter-entreprises et inter-systèmes
- Avoir une réelle inter-opérabilité indépendante des contraintes techniques : système, langage, ...
- Ne pas perturber l'existant au niveau des infrastructures de gestion de la sécurité

La solution

- Utiliser l'existant plutot qu'imposer des changements
  - Protocoles Internet déjà répandus, comme HTTP et HTTPS
  - Format de données XML
- Constituants essentiels
  - Un protocole de requétage basé sur un format de données universel : XML et pouvant utiliser HTTP : SOAP
  - Un langage de description des services : WSDL
  - Un système d'annuaire : UDDI

Les services WEB

Les besoins

L’inter-opérabilité des systèmes et des applications est un souhait partagé depuis très longtemps par de nombreuses organisations et entreprises. Jusqu’à présent, les technologies permettant l’invocation distante de services comportaient de très nombreuses contraintes qui rendaient difficile leur utilisation:

- incompatibilité des protocoles de communication : CORBA avec DCOM par exemple,
- incapacité à traverser les protections pare-feu des entreprises sans modification des règles de sécurité,
- lourdeurs de développement et de mise en oeuvre.

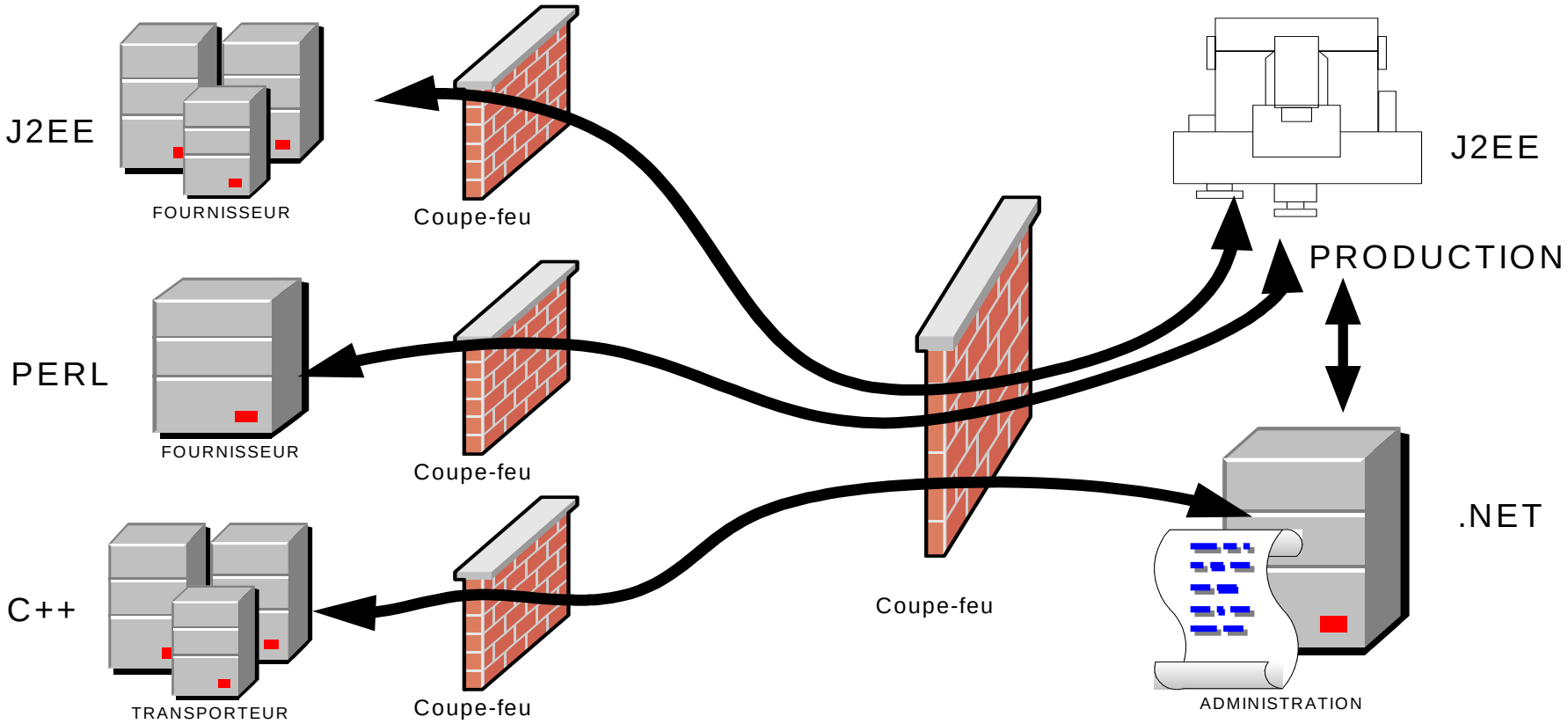
La solution

L’idée forte des services WEB consiste à utiliser ce qui existe déjà au niveau des entreprises plutôt qu’à proposer de nouvelles solutions trop contraignantes en termes d’équipements et de procédures:

- XML qui est un format de données largement accepté et répandu,
- HTTP qui est un procole de communication universel pour lequel les procédures de sécurité réseau des entreprises sont déjà adaptées.

Bénéfices des services WEB

- Intégrer des plate-formes techniques incompatibles jusqu'à maintenant
- Ouvrir l'entreprise à ses fournisseurs, ses partenaires et ses clients en exposant ses services à travers des protocoles simples et normalisés
- S'adapter aux changements plus facilement et plus rapidement



Bénéfices des services WEB

Avant l'avénement des services WEB, il était assez complexe de développer des mécanismes d'échanges inter-applications en milieu hétérogène. Ainsi, pour par exemple communiquer entre un serveur Corba sous Unix et un serveur DCOM sous Windows, il fallait utiliser des passerelles propriétaires et difficiles à mettre en oeuvre.

S'affranchir des technologies sous-jacentes rend virtuellement possible une inter-opérabilité extrémement grande entre systèmes d'informations du monde entier.

Cette liberté (relative !) donne aussi la possibilité de suivre plus facilement les évolutions technologiques sans rompre le contrat applicatif qui peut exister envers les utilisateurs d'un système d'informations.

Les constituants d'une architecture à base de services WEB

SOAP : Simple Object Access Protocol

- Protocole requete-réponse d'invocation des services WEB
- Synchrone ou asynchrone
- Basé sur XML

WSDL : Web Service Description Language

- Langage de description technique d'un service WEB : signatures des opérations, adresses, ...
- S'apparente au langage de description de services de CORBA : IDL

UDDI : Universal Description Discovery and Integration

- Système d'annuaire normalisé de services WEB
  - système d'enregistrement de services
  - système d'interrogation

Les constituants d’une architecture à base de services WEB

SOAP

SOAP est le protocole d’invocation d’un service WEB. Il a été défini de façon à pouvoir être mis en oeuvre avec n’importe quelle technologie de développement. Il suffit de pouvoir:

- ouvrir une socket de communication vers le serveur hébergeant le service WEB,
- lui transmettre une requête formattée en XML, donc essentiellement textuelle,
- analyser une réponse également formattée en XML.

WSDL

Cette description comporte notamment:

- les coordonnées du service : URL, numéro de port, protocole, ...
- la signature du service : nom, types des paramètres, ...

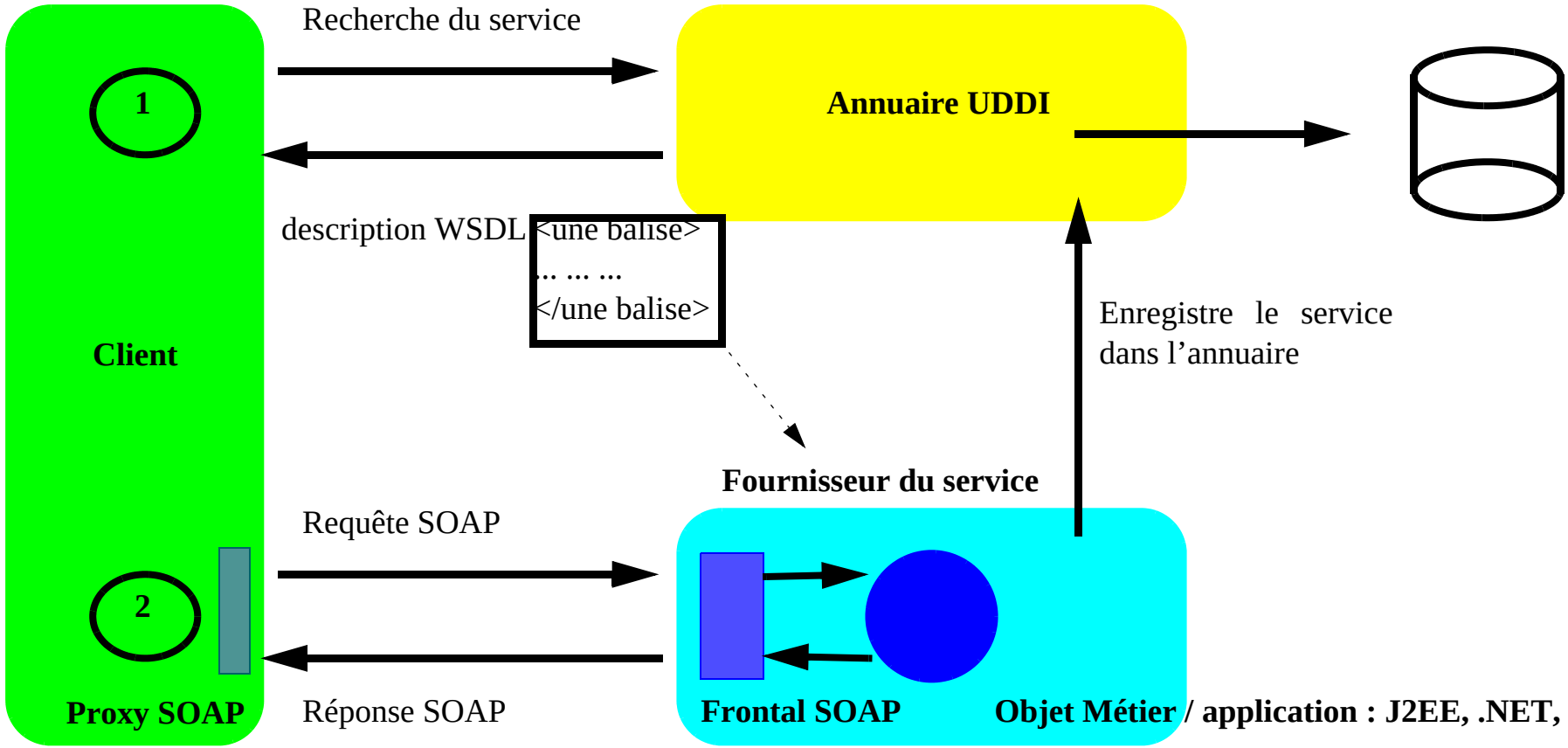
UDDI

UDDI est une spécification élaborée par IBM, Microsoft et Ariba. Elle normalise le fonctionnement d’un annuaire permettant d’enregistrer des descriptions WSDL et de les restituer à différents utilisateurs.

Les annuaires UDDI peuvent être de deux types:

- accessibles à tout utilisateur à partir d’Internet,
- locaux à une entreprise ou à une organisation.

Principes de fonctionnement



- L'usage de l'annuaire est optionnel

Principes de fonctionnement

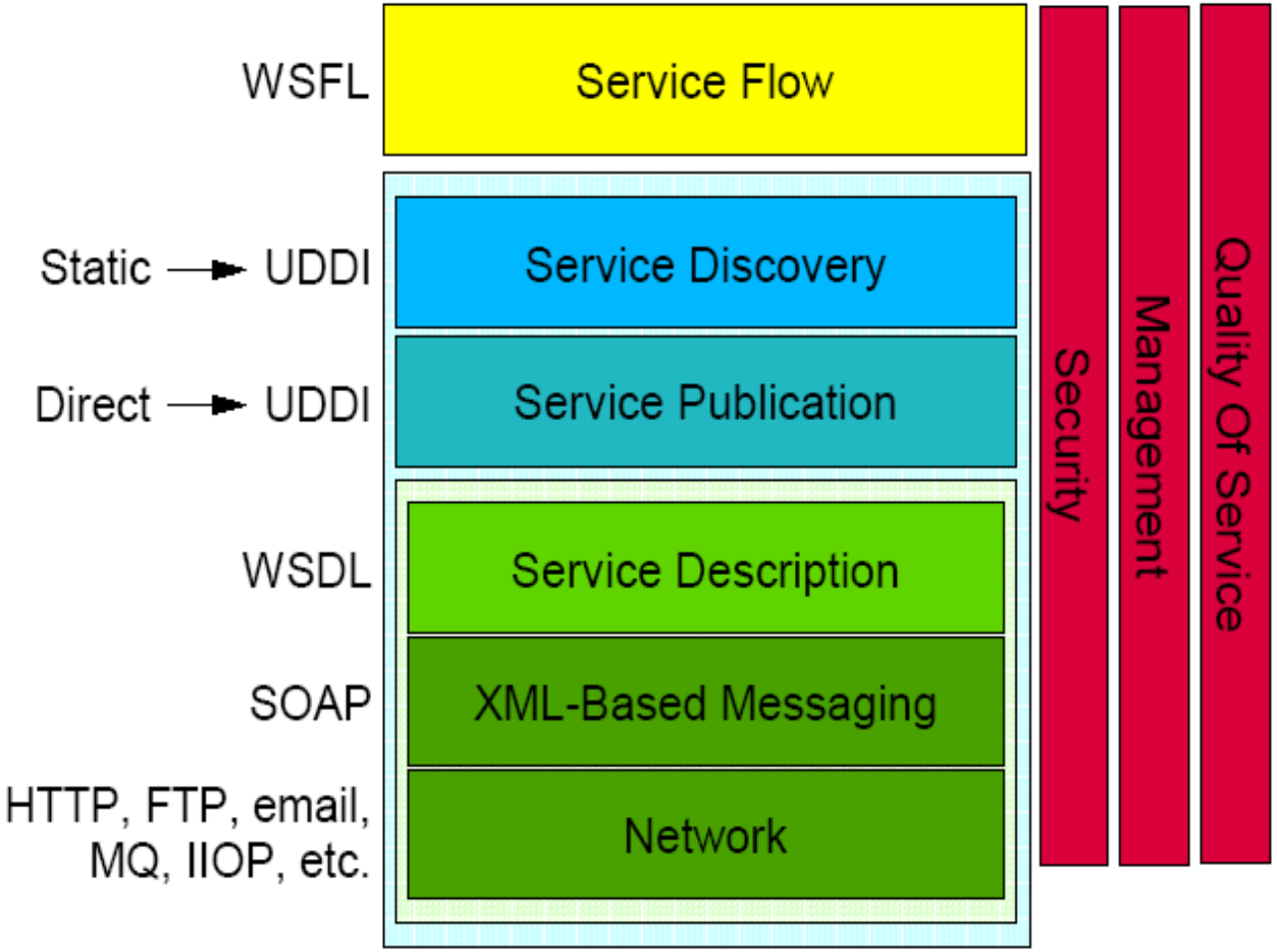
Le fournisseur du service WEB peut l'enregistrer au sein d'un annuaire UDDI, soit à l'intérieur de son entreprise, soit à l'extérieur si ce service est destiné à être utilisé par des acteurs externes.

Un client est alors en mesure d'interroger l'annuaire à partir de critères qui lui sont propres pour obtenir la description WSDL du service WEB.

Cette description permet d'automatiser et donc de simplifier la phase de requêtage via SOAP. En effet, de nombreux outils permettent de produire des proxy d'interrogation des services WEB à placer côté client afin de masquer les complexités de SOAP.

Un service WEB peut être réalisée avec n'importe quelle technologie de développement. Les serveurs d'applications de type J2EE ou .NET proposent généralement des mécanismes permettant de simplifier énormément la réalisation d'un service WEB. Il s'agit la plupart du temps de frontaux SOAP générés automatiquement qui permettent au concepteur de se concentrer sur le développement de ses objets métier.

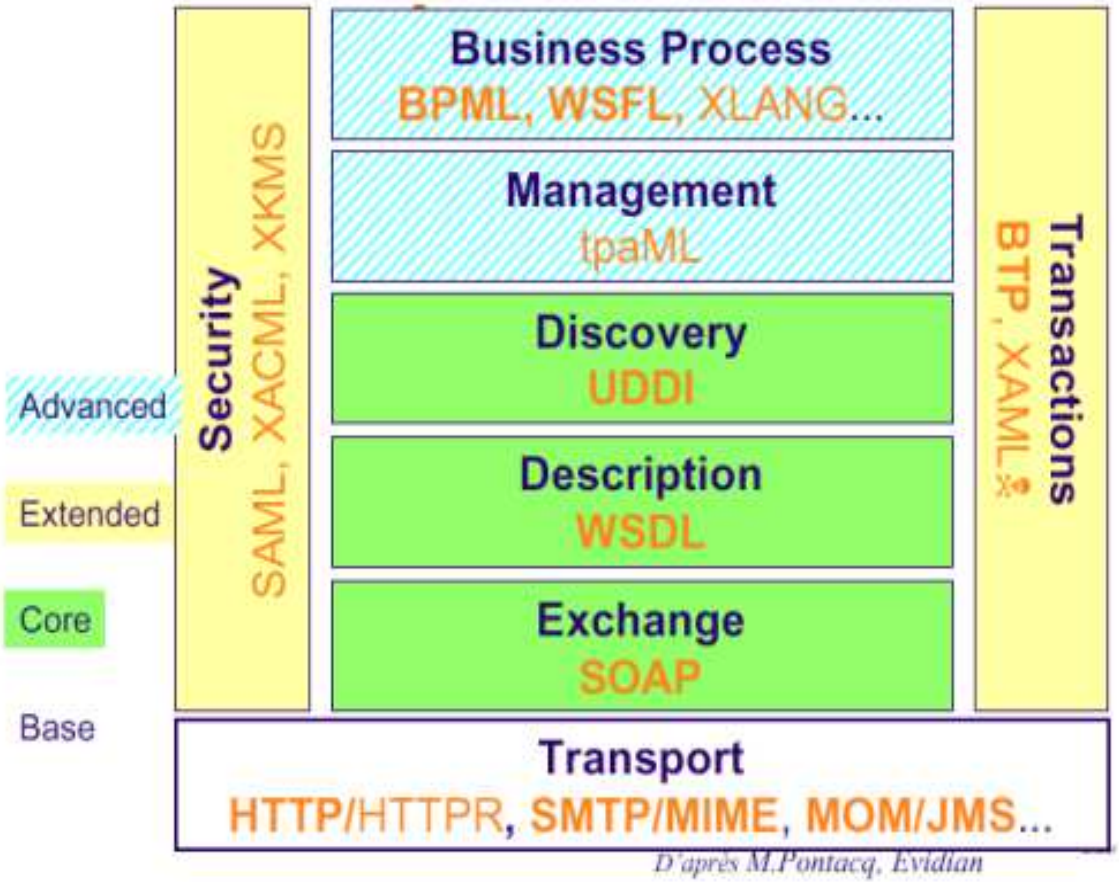
Pile Web Services



Pile Web Services

Notes

Pile Web Services

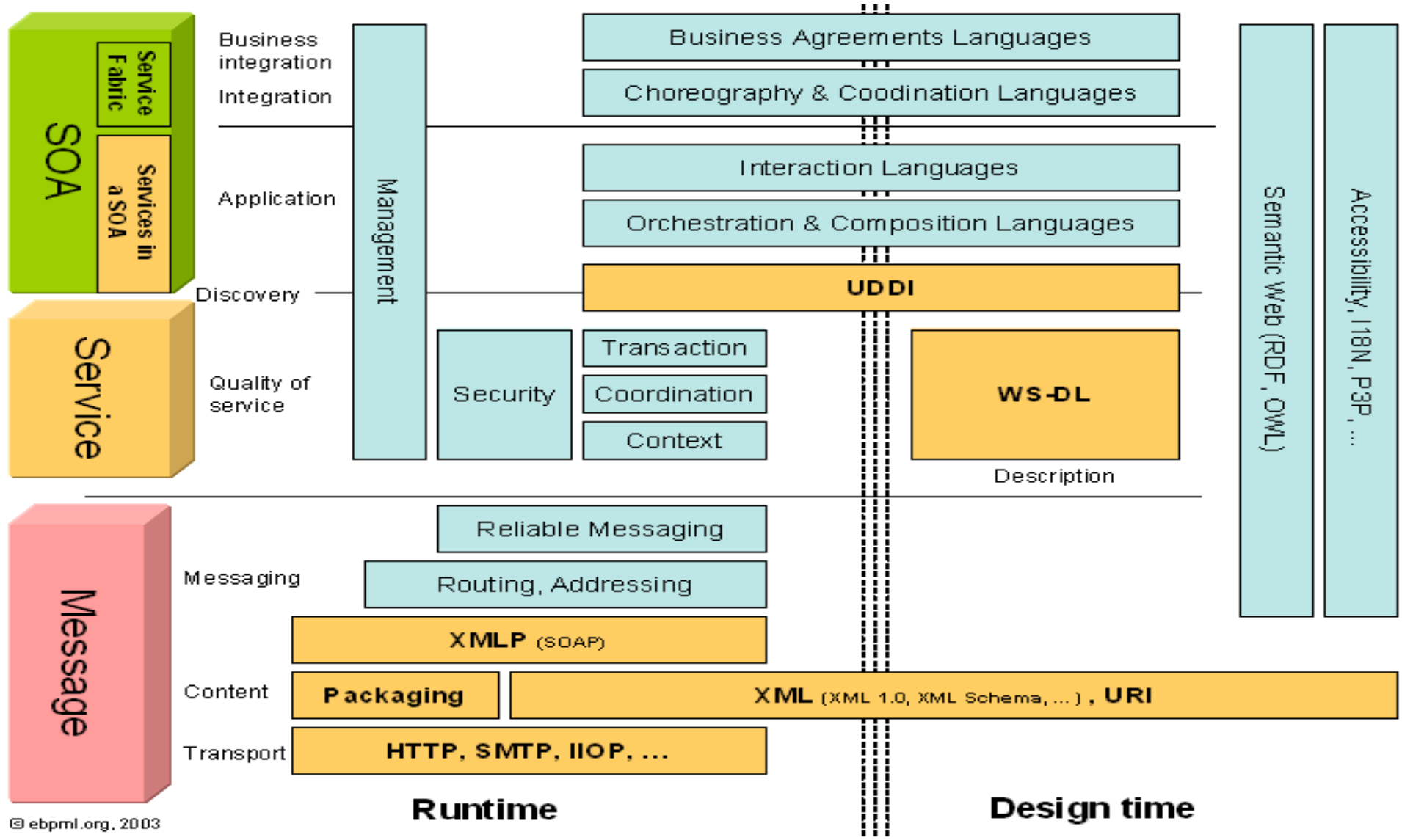


D'après M.Pontacq, Evidian

Pile Web Services

Notes

Pile Web Services

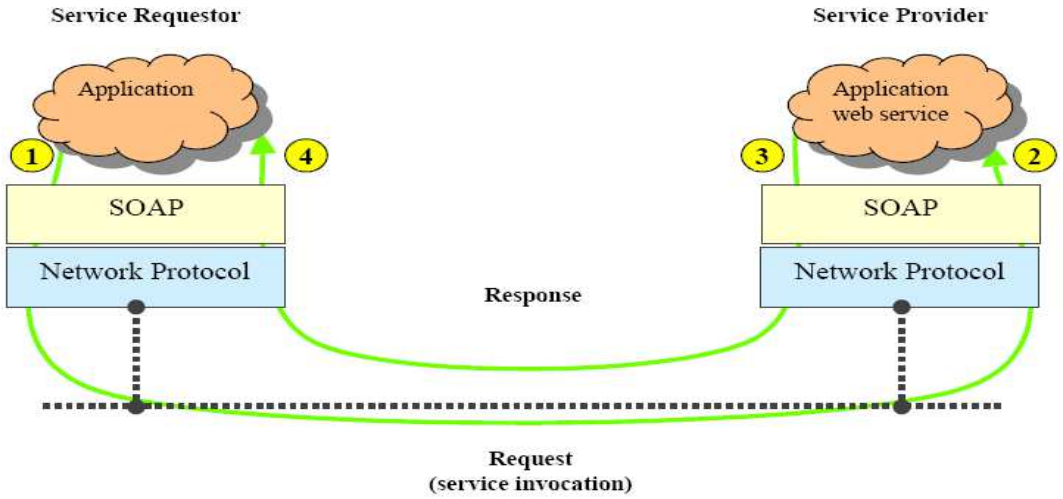


Pile Web Services

Notes

SOAP

Protocole léger d'échange d'informations



- Messages dont la structure est basée sur XML
  - Attachements possibles pour les données binaires
  - Produits et analysés automatiquement
- Normalisation des requêtes distantes et des réponses
  - Synchronisme ou asynchronisme
- Règles d'encodage et de décodage des types de données
  - Correspondances normalisées avec les types primitifs de tous les langages de programmation
  - Mécanismes de sérialisation - désérialisation pour les types complexes

Ne normalise pas certaines modalités d'échanges évoluées

- Sessions et Transactions
- Sécurité intrinsèque
- Fiabilité de la transmission

SOAP

Protocole léger d’échange d’informations

SOAP est un protocole simplifié d’invocation de services à distance. Il normalise donc uniquement ce qui est strictement indispensable.

Ne normalise pas certaines modalités d’échanges évoluées

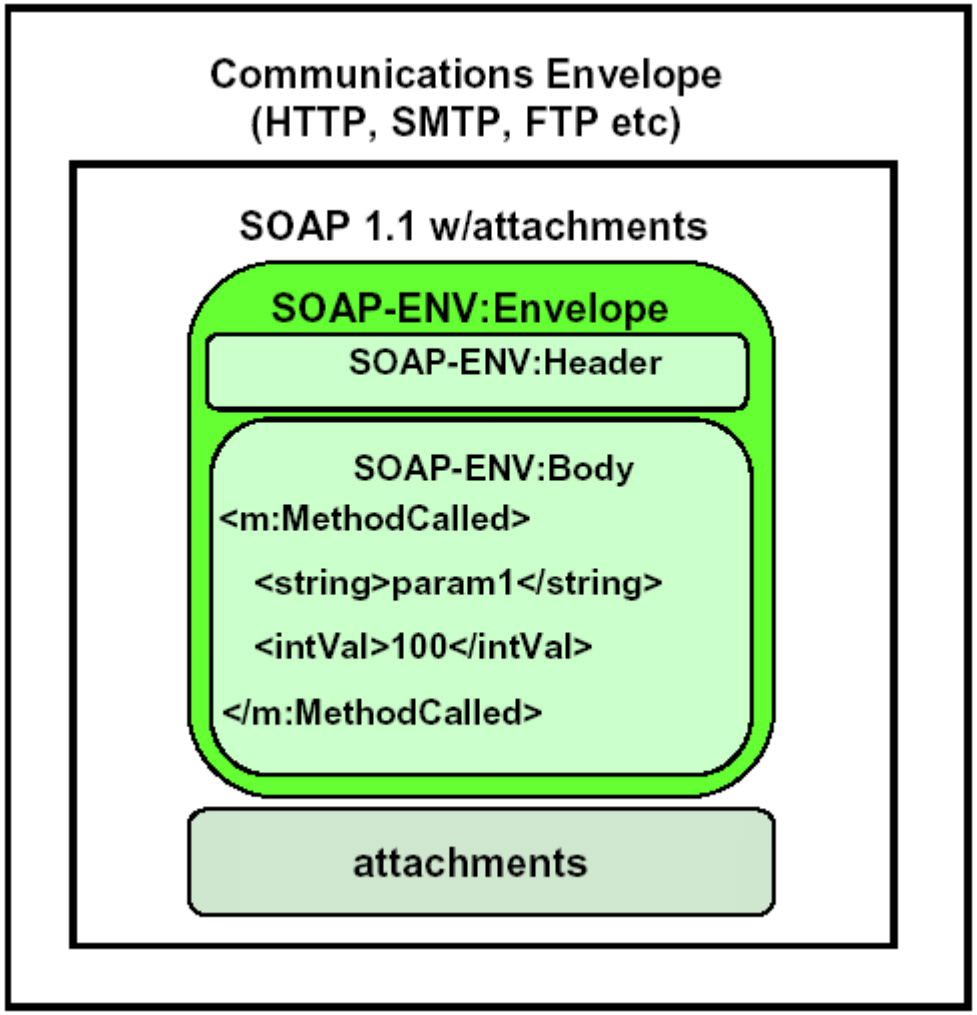
SOAP comporte plusieurs limitations:

- l’absence de mécanisme intrinsèque permettant de gérer les différents aspects de la sécurité:
  - authentification et droits d’accès
  - confidentialité
- l’absence de mécanisme intrinsèque de gestion des sessions applicatives
- l’absence de prise en charge des transactions globales ou transactions distribuées.

Cependant, plusieurs groupes de travail se sont constitués afin de faire des propositions dans ces différents domaines. Nous les exposerons dans la suite de ce document.

Structure d'un message SOAP

- Enveloppe de communication souvent basée sur HTTP
- Un header pour tous les aspects non normalisés : gestion de session par exemple
- Un body comportant les informations sur le service appelé, les paramètres effectifs, ...
- Des attachements optionnels pour les données binaires



Structure d’un message SOAP

SOAP normalise la structure des messages de définition des requêtes et des réponses.

Cette structure comporte notamment:

- Une enveloppe de communication correspondant au protocole utilisé pour acheminer les données. Ce protocole n’est pas normalisé par SOAP et peut théoriquement être de toute nature. En pratique, on rencontre couramment HTTP ou HTTPS.
- Une zone d’entête utilisée pour transmettre des informations complémentaires non prévues par le standard comme par exemple un identifiant de session applicatice.
- Un corps permettant de transmettre le message lui-même, qu’il s’agisse d’une requête ou d’une réponse.
- Un ou plusieurs attachements optionnels (à partir de SOAP 1.1) qui permettent d’associer aux messages des pièces jointes binaires, de façon similaire à un courrier électronique.

Un exemple de message SOAP

Requête

```
POST /StockQuote HTTP/1.1
Host: www.stockquoteserver.com
Content-Type: text/xml; charset="utf-8"
Content-Length: nnnn
SOAPAction: "Some-URI"

<SOAP-ENV:Envelope
xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
<SOAP-ENV:Body>
    <m:GetLastTradePrice xmlns:m="Some-URI">
        <symbol>DIS</symbol>
    </m:GetLastTradePrice>
</SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Réponse

```
HTTP/1.1 200 OK
Content-Type: text/xml; charset="utf-8"
Content-Length: nnnn

<SOAP-ENV:Envelope
xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"/>
<SOAP-ENV:Body>
    <m:GetLastTradePriceResponse xmlns:m="Some-URI">
        <Price>34.5</Price>
    </m:GetLastTradePriceResponse>
</SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Un exemple de message SOAP

Comme le montre cet exemple, les deux messages SOAP correspondant respectivement à une requête et à sa réponse sont assez verbeux. C’est un inconvénient mineur de SOAP.

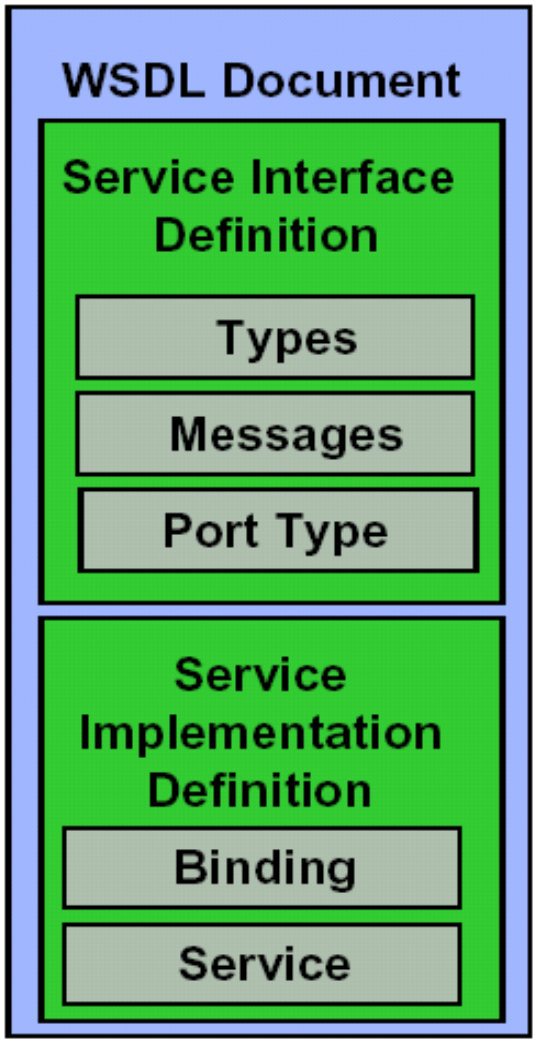
Par contre, ils ont l’avantage d’être en format texte ce qui facilite la mise-au-point d’une solution à base de services WEB.

En cas de nécessité d’une confidentialité accrue, on pourra utiliser HTTPS à la place de HTTP.

WSDL

Format XML de description de services réseau

- Défini à l'initiative d'IBM et de Microsoft
- Particulièrement adapté à la description des services WEB
- Description comportant:
  - informations concernant la localisation du service : adresse, numéro de port, procole, ...
  - description de la structure des requêtes et réponses associées
  - description des opérations disponibles : noms, paramètres, types, ...
  - informations sur les types de données nécessaires pour les arguments et les valeurs de retour



WSDL

Format XML de description de services réseau

WSDL a été défini par IBM et Microsoft dans le but de fournir un moyen de décrire des services applicatifs accessibles par réseau. Ces services sont invocables par l'intermédiaire de messages dont le format est défini à l'aide de WSDL.

Un document WSDL peut contenir:

- **Types–** des définitions de types basées sur un formalisme de définition tel que XSD.
- **Message–** une définition abstraite des données à communiquer.
- **Operation–** une définition abstraite des actions supportées par le service.
- **Port Type–** un ensemble d'opérations abstraites supportées par les points d'entrée du service.
- **Binding–** une définition concrète d'un protocole et d'un format de données pour un Port Type.
- **Port–** un point d'entrée défini comme une combinaison d'un Binding et d'une adresse réseau.
- **Service–** un ensemble de points d'entrée.

WSDL

Message Exchange Patterns

- Spécifient les modalités du dialogue
- 4 dans WSDL 1.1
  - requête client / réponse serveur synchrone
  - invocation par le client sans retour (One Way)
  - requête serveur / réponse client synchrone
  - invocation par le serveur sans retour
- Enrichis dans WSDL 2.0

WS-Policy

- Politique de sécurité applicable

WSDL

Notes

Exemple WSDL

Entête et définition des types

```
<?xml version="1.0"?>
<definitions     name="StockQuote"
                 targetNamespace="http://example.com/stockquote.wsdl"
                 xmlns:tns="http://example.com/stockquote.wsdl"
                 xmlns:xsd1="http://example.com/stockquote.xsd"
                 xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
                 xmlns="http://schemas.xmlsoap.org/wsdl/">
<types>
    <schema targetNamespace="http://example.com/stockquote.xsd" xmlns="http://www.w3.org/2000/10/XMLSchema">
        <element name="TradePriceRequest">
            <complexType> <all> <element name="tickerSymbol" type="string"/> </all> </complexType>
        </element>
        <element name="TradePrice">
            <complexType><all> <element name="price" type="float"/> </all> </complexType>
        </element>
    </schema>
```

Exemple WSDL

Entête et définition des types

L'entête d'un document WSDL comporte des références à différents espaces de nommage, identifiés par une URL et un préfixe:

- wsdl : http://schemas.xmlsoap.org/wsdl/WSDL namespace pour le framework WSDL.
- soap : http://schemas.xmlsoap.org/wsdl/soap/ WSDL namespace pour le binding WSDL SOAP.
- http : http://schemas.xmlsoap.org/wsdl/http/ WSDL namespace pour le binding WSDL HTTP GET & POST.
- mime : http://schemas.xmlsoap.org/wsdl/mime/ WSDL namespace pour le binding WSDL MIME.
- soapenc : http://schemas.xmlsoap.org/soap/encoding/ namespace d'encodage SOAP 1.1.
- soapenv : http://schemas.xmlsoap.org/soap/envelope/ namespace de définition de l'enveloppe SOAP 1.1.
- xsi : http://www.w3.org/2000/10/XMLSchema-instance namespace d'instance XSD.
- xsd : http://www.w3.org/2000/10/XMLSchema namespace de schéma XSD .
- tns : "this namespace" (tns) référence le document courant par convention.

La partie réservée à la définition des types peut être écrite en utilisant XSD.

Exemple WSDL

Message et portType

```
<message name="GetLastTradePriceInput">
      <part name="body" element="xsd1:TradePriceRequest"/>
</message>
<message name="GetLastTradePriceOutput">
      <part name="body" element="xsd1:TradePrice"/>
</message>

<portType name="StockQuotePortType">
      <operation name="GetLastTradePrice">
            <input message="tns:GetLastTradePriceInput"/>
            <output message="tns:GetLastTradePriceOutput"/>
      </operation>
</portType>
```

Exemple WSDL

Message et portType

Une opération est définie par association de deux messages, l'un en entrée et l'autre en sortie.

Elle permet de constituer un élément appelé portType.

Exemple WSDL

Binding et service

```
<binding name="StockQuoteSoapBinding" type="tns:StockQuotePortType">
    <soap:binding style="document" transport="http://schemas.xmlsoap.org/soap/http"/>
    <operation name="GetLastTradePrice">
        <soap:operation soapAction="http://example.com/GetLastTradePrice"/>
        <input> <soap:body use="literal"/> </input>
        <output> <soap:body use="literal"/> </output>
    </operation>
</binding>
<service name="StockQuoteService">
    <documentation>My first service</documentation>
    <port name="StockQuotePort" binding="tns:StockQuoteBinding">
        <soap:address location="http://example.com/stockquote"/>
    </port>
</service>
```

Exemple WSDL

Binding et service

Le binding associe une opération à un protocole de transport, HTTP sur cet exemple.

Le service associe un binding et une adresse, sous la forme d’une URL.

UDDI

- Fonctionnement comparable aux DNS réseau

Publication de services WEB

- Permet la création, l'édition et la suppression d'enregistrements via des messages SOAP avec accès sécurisé
- Publisher API

Recherche de services WEB

- Accessible par l'intermédiaire de SOAP
- Quatre types d'informations définies avec XML schéma
  - <businessEntity> : identification d'une entreprise ou d'une organisation permettant une recherche de type "Pages blanches"
  - <businessService> : informations métier sur un service particulier ("Pages jaunes")
  - <bindingTemplate> : éléments techniques indispensables à l'invocation du service ("Pages vertes")
  - <tModels> : références aux spécifications utilisées
- Inquiry API

UDDI

La spécification UDDI a été établie suite aux travaux initiaux de Ariba, IBM et Microsoft. Elle définit un accès programmatique normalisé à des services métier invocables indépendamment des technologies.

Les registres UDDI fonctionnent avec un système de réplication d'informations, à la façon des serveurs DNS, afin d'offrir une fiabilité maximale.

Publication de services WEB

Le volet publication de la spécification UDDI prévoit plusieurs modalités de publication d'un service WEB:

- l'utilisation d'une API dédiée qui permet d'automatiser cette t,
- l'utilisation d'un service en ligne avec une interface de type WEB, de façon analogue à ce que certains moteur de recherche proposent sur Internet.

Recherche de services WEB

Dans le cadre d'échanges de type B2B, la présence d'une API normalisée de recherche est indispensable.

Cette API permet d'accèder aux pages blanches et jaunes d'un registre, mais également à sas pages vertes, ce qui permet d'automatiser la recherche et l'invocation d'un service WEB.

Historique des services WEB

- **1992 : naissance de HTML**
- **1994 : création du W3C (World Wide Web Consortium)**
- 1996 : document de travail XML
- 1997 : adoption de XML par Microsoft
- **1998 : XML 1.0 par le W3C**
- **1999 : SOAP 0.9 par DevelopMentor, Microsoft, Userland**
- **1999 : SOAP 1.0 soumis à l'IETF**
- 2000 : SOAP 1.1 contributions IBM
- 2001 : 28 groupes de travail au W3C !
- **2001 : soumission de WSDL au W3C à l'initiative de IBM et Microsoft**
- 2001 : soumission de Business Transaction Protocol à l'OASIS par BEA Systems
- 2002 : définition de WSCI 1.0 (Web Service Choregraphy Interface) par Sun Microsystems, BEA Systems, Intalio, SAP
- 2002 : UDDI 3.0
- Avril 2002 : définition de WS-Security par Microsoft, IBM, Verisign
- Août 2002 : Business Process Execution Language for Web Services (BPEL4WS)
- Octobre 2002 : WS-Coordination et WS-Transactions par IBM et Microsoft
- 2003 : SOAP 1.2

Historique des services WEB

Notes

Les enjeux

Standardisation

- Des acteurs ambitieux : IBM, Microsoft, Sun, BEA, ...
- De multiples organismes de normalisation : W3C, OASIS, WS-I
- La promesse initiale de l'interopérabilité sera-t-elle tenue ?

Evolutions importantes

- WSCI : description du comportement des services Web (dépendances logiques, règles, ...)
- WS-Transactions : prise en charge des transactions distribuées sur plusieurs services Web
- WS-Security : sécurité intrinsèque
- Description des processus B2B : ebXml, BPEL4WS
- Architectures Orientées Services ou SOA - Services Oriented Architectures

Les enjeux

Standardisation

Devant la multiplication des alliances de circonstances et des propositions de standards, on peut légitimement s'inquiéter de l'universalité des services Web.

Evolutions importantes

Au départ, les services Web comportaient le minimum de ce qu'il fallait pour faire des invocations inter-applications sur le Web. Ces mécanismes initiaux ne proposaient pas ce qui est indispensable pour supporter un fonctionnement de type B2B:

- les transactions distribuées
- des mécanismes de sécurité
- une sémantique riche.

Devant l'importance des enjeux, de nombreux fournisseurs tentent de prendre une longueur d'avance en proposant différentes solutions, généralement au travers d'alliances de circonstance.

Architectures distribuées (Polytech)
Introduction à l'API JAX-WS

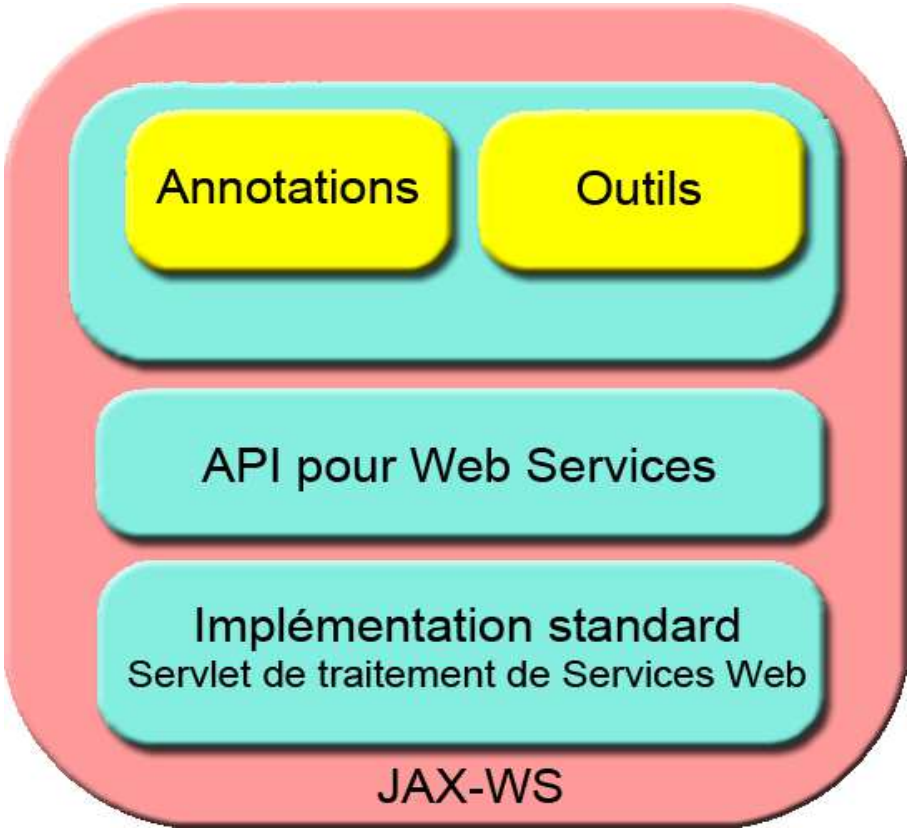
Version 3.0

- Présentation
- Développement d'un service
- Développement d'un client

Ce qu'est JAX-WS

Java API for XML Web Services

- Spécification JSR 224.
- Evolution de JAX-RPC facilitant le développement et la consommation de Services Web orientés RPC et Document.
- Standardisation des frontaux SOAP Java
- Standardisation de l'utilisation de JAXB pour le marshalling / unmarshalling



Contenu de JAX-WS

- Une API basée sur Java et XML.
- Une implémentation standard (SI) de Servlet Web Service.
- Un ensemble d'annotations utilisées par le Annotation ProcessingTool (APT) pour générer des composants coté serveur ou coté client.
- Des outils de génération.

Ce qu’est JAX-WS

Notes

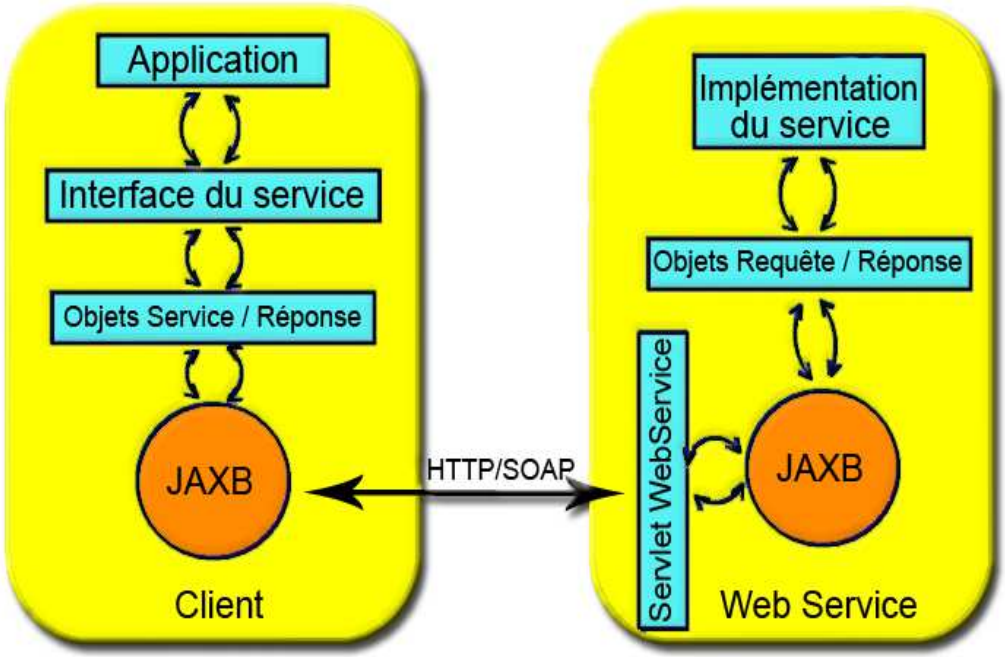
Principe de fonctionnement

Coté client

- La seule classe développée est celle consommant le Web Service.
- Les autres classes sont générées automatiquement par l'outil 'wsimport'.

Coté Service

- La seule classe à développer est celle implémentant le service.
- Les autres sont générées automatiquement par apt à partir des annotations.



Principe de fonctionnement

Notes

Développement d'un WebService avec JAX-WS

```
package com.leuville.formation.inscription;

import javax.jws.*;
import javax.xml.ws.WebFault;

@WebService(name="inscription", targetNamespace="http://www.leuville.com/formation")
@WebFault(name="erreur-inscription", targetNamespace="http://www.leuville.com/formation/erreur")
public class InscriptionService {

  @WebMethod(operationName="inscription")
  @WebResult(name="inscrit")
  public boolean inscrire(
       @WebParam(name="formulaire-inscription", mode=WebParam.Mode.IN,
                   targetNamespace="http://www.leuville.com/formation")
       FormulaireInscription form)
     throws InscriptionException {

     System.out.println(form.getPrenomClient() + " " + form.getNomClient());
     System.out.println(form.getCodeCours());
     System.out.println(form.getDateSession());

     return true;
  }
```

Développement d’un WebService avec JAX-WS

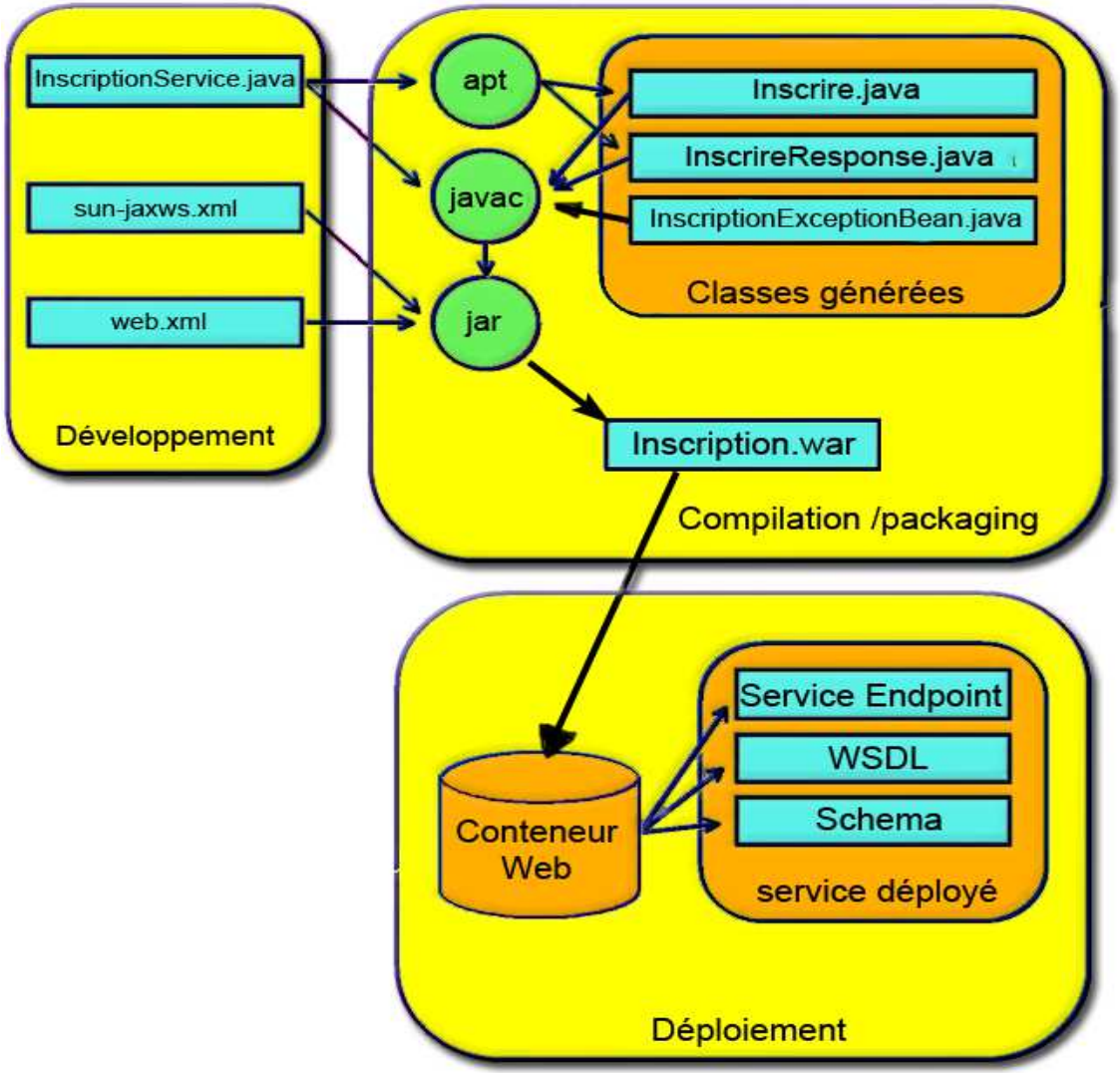
Notes

La classe d’implémentation du service et développée et annotée.

L’outil ’wsgen’ se base sur cette classe pour générer les classes qui seront chargées de faire le lien entre la servlet WebService et l’implémentation (classes de marshalling / unmarshalling)

Déploiement d'un WebService JAX-WS

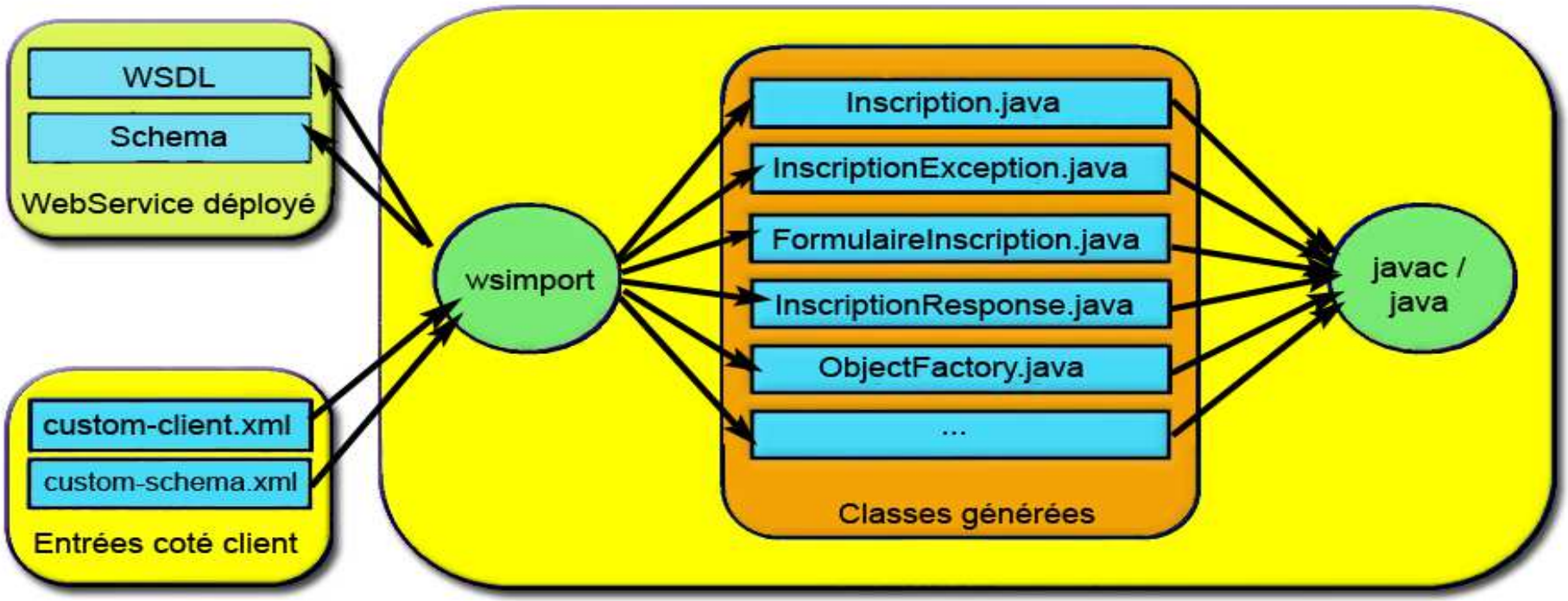
- Les classes générées par apt sont compilées.
- Une archive web est créée à partir des classes compilées et des fichiers de déploiement.
- Le service est déployé comme toute autre application web JavaEE.



Déploiement d’un WebService JAX-WS

Notes

Développement d'un client JAX-WS



- L'outil 'wsimport' permet de générer des classes coté client.
- Des fichiers XML peuvent être donnés en entrée afin de personnaliser les bindings.

Développement d’un client JAX-WS

Notes

Autres fonctionnalités de JAX-WS

- Développement et configuration de Handlers SOAP
- Support des pièces attachées avec MTOM + XOP coté service et coté client
- Accès aux méta données du message (données non incluses dans le message SOAP)
- Maintien de session entre le client et le service (WS-Context)
- Support des paramètres OUT et INOUT
- Invocation synchrone ou asynchrone
- Selon les implémentations, d'autres fonctionnalités de type WS-* sont disponibles

Autres fonctionnalités de JAX-WS

Notes

Architectures distribuées (Polytech)
JAX-WS : Endpoint API

Version 3.0

- Présentation
- Mise en oeuvre
- Configuration

Présentation

API Endpoint

- L'API Endpoint permet de publier des services web de façon programmative.
- Un serveur HTTP léger est créé dynamiquement pour traiter les requêtes.
- Cette API permet de publier des services web sans Serveur d'Applications.

Présentation

Notes

Mise en oeuvre de l'API Endpoint

Etapes

- Ecrire une classe Java de type POJO.
- Ajouter des annotations.
- Générer les classes (portables artifacts).
- Développer une classe chargée de publier le service.

Remarques

- Les trois premières étapes sont les mêmes que celles du développement de services JAX-WS standards.
- La quatrième étape consiste à l'utiliser l'API Endpoint pour publier le service.

Mise en oeuvre de l’API Endpoint

Notes

La classe javax.xml.ws.EndPoint

Création avec l'une des méthodes statiques de la classe EndPoint

- Endpoint create(Object implementor)
- Endpoint create(String bindingId, Object implementor)
- Endpoint publish(String address, Object implementor)

- La troisième méthode créé et publie le Endpoint.
- Les deux premières nécessitent d'invoquer la méthode publish() pour que le service soit publié.
- La méthode stop() permet d'arréter le service.

La classe javax.ws.Endpoint

Notes

Exemple

```
public static void main(String[] args) {
     Endpoint endpoint = Endpoint.publish("http://localhost:8080/hello", new HelloWorldService());


     System.out.println("Saisir car+return pour stopper le endpoint");
     Scanner sc = new Scanner(System.in);
     sc.next();


     endpoint.stop();
}
```

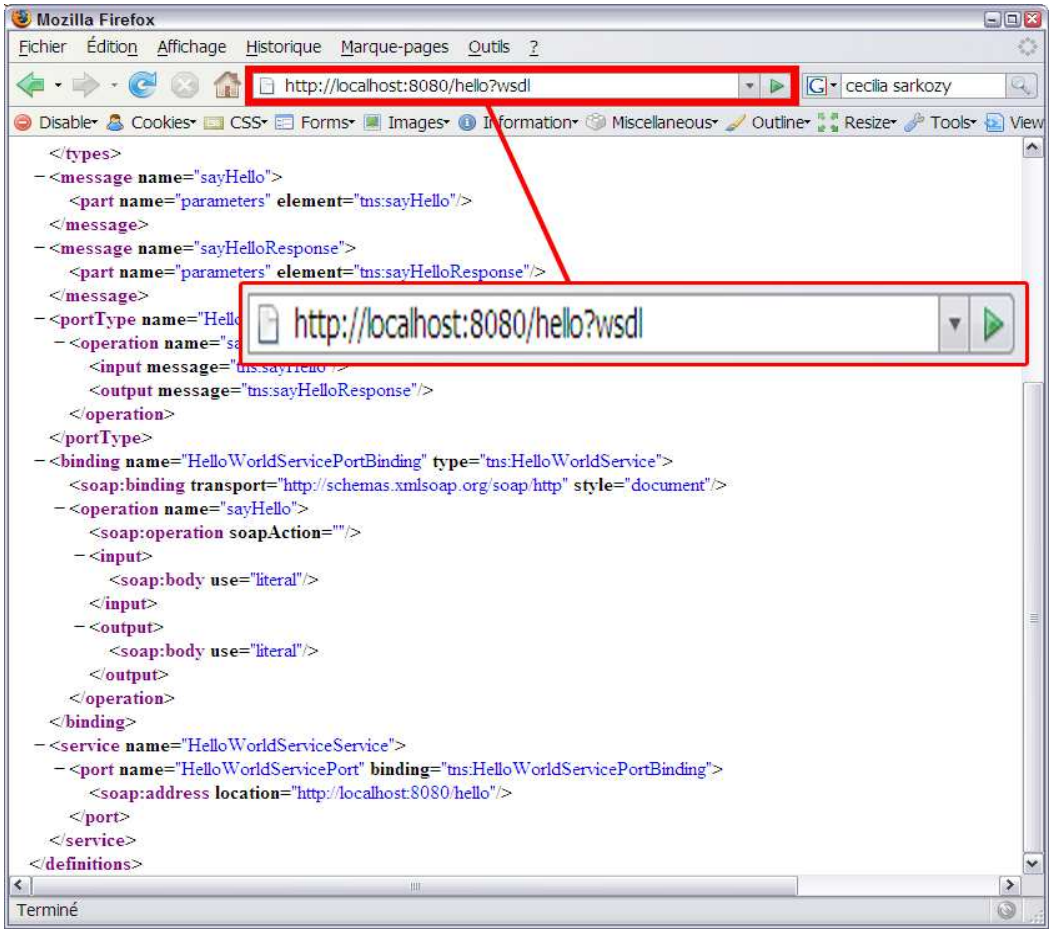
- HelloWorldService est une classe annotée @WebService.
- Le Endpoint est publié sur une URL choisi arbitrairement
  - Attention, tenir compte des contraintes de la machine (ports disponibles)
- Après saisie clavier, le Endpoint est arrêté.

Exemple

Notes

Configuration d'un Endpoint

- Le WSDL du service publié peut être obtenu en invoquant le Endpoint avec le paramètre HTTP WSDL.
- Par défaut :
  - Le nom du service est composé du nom de la classe d'implémentation suffixé par "Service"
  - Le nom du port est composé du nom de la classe d'implémentation suffixé par "Port".
- Ces deux éléments sont configurables en associant des propriétés au Endpoint.
- Pour cela, on utilise la méthode setProperties(Map).



Configuration d’un EndPoint

Notes

Configuration d'un Endpoint

```
public static void main(String[] args) {
     Endpoint endpoint = Endpoint.create(new HelloWorldService());

     Map<String, Object> map = new HashMap<String, Object>();
     map.put(Endpoint.WSDL_SERVICE, new QName("http://www.leuville.com/hello", "HelloService"));
     map.put(Endpoint.WSDL_PORT, new QName("http://www.leuville.com/hello", "HelloPort"));

     endpoint.setProperties(map);
     endpoint.publish("http://localhost:8080/hello");

     System.out.println("Saisir car+return pour stopper le endpoint");
     Scanner sc = new Scanner(System.in);
     sc.next();

     endpoint.stop();
}
```

Configuration d’un Endpoint

Notes

Configuration d'un Endpoint

Bindings

- Un EndPoint peut être configuré pour utiliser plusieurs bindings. La configuration des bindings peut se faire :
  - avec l'annotation @BindingType
  - en utilisant la méthode Endpoint create(String bindingId, Object implementor)
- Si rien n'est spécifié, le binding SOAP1.1/HTTP est utilisé
- Les objets Binding de l'API JAX-WS permettent également de configurer le support de MTOM et la déclaration des chaînes de Handlers.

Configuration d’un Endpoint

Notes

Configuration d'un Endpoint

WSDL

- Lorsque le service est créé à partir d'une classe, le WSDL est généré à la volée par le Endpoint
- Lorsque l'implémentation du service a été créée à partir d'une description WSDL, on souhaite que le WSDL fourni par le Endpoint soit celui qui a servi de base à la création du service.
- Pour référencer les documents contenant la description du service, l'API Endpoint utilise des Metadata :

```
// metadata processing for WSDL, schema files
   List <File> metadataFile = ...
   List<Source> metadata = new ArrayList<Source>();
   for(File file : metadataFile) {
       Source source = new StreamSource(new FileInputStream(file));
       source.setSystemId(file.toURL().toExternalForm());
       metadata.add(source);
   }
   endpoint.setMetadata(metadata);
```

Configuration d’un Endpoint

Notes

Architectures distribuées (Polytech)
Annexes

Version 3.0

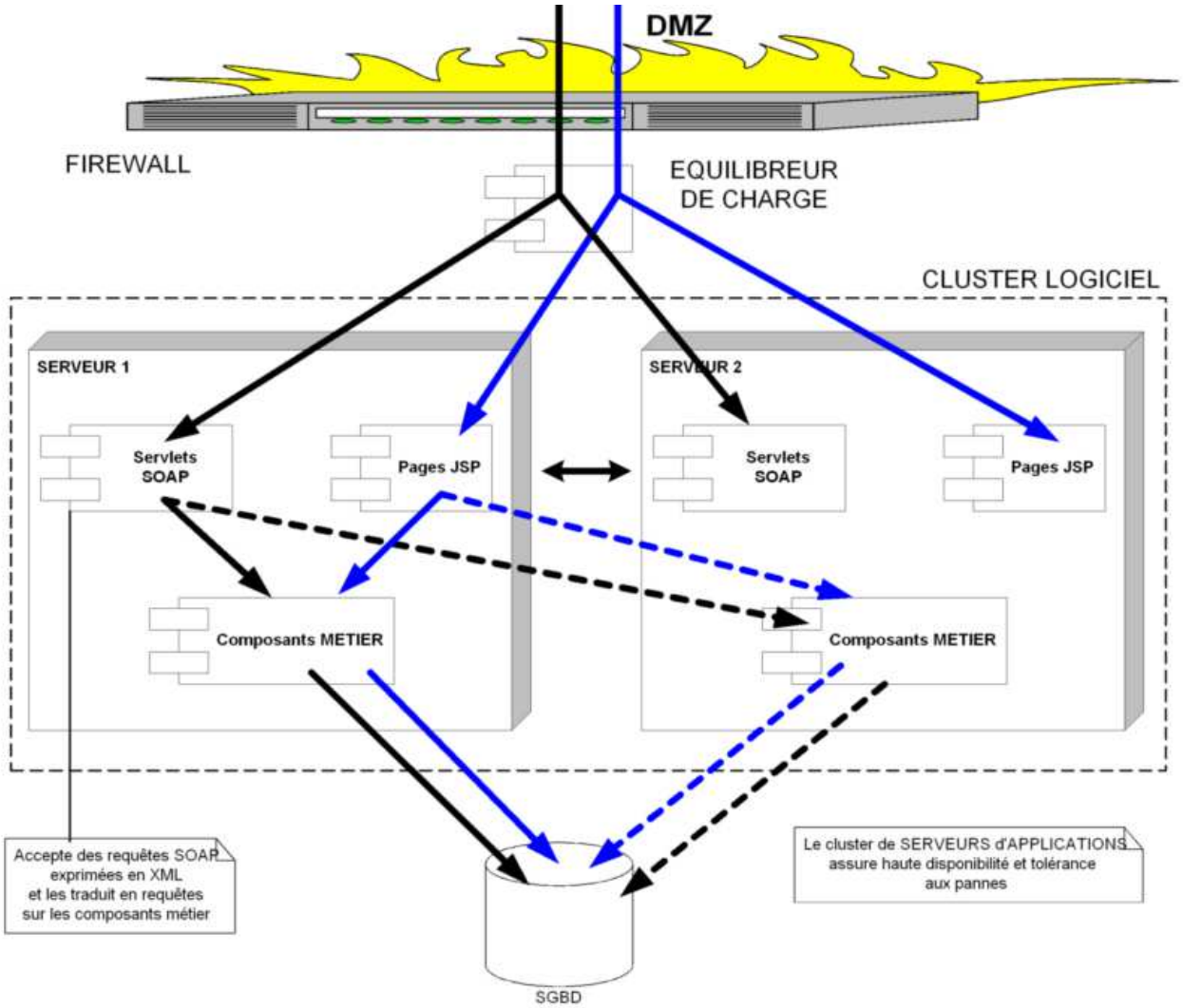
- Compléments

Architectures distribuées (Polytech)
Caractéristiques de base de JavaEE

Version 3.0

- Notion de confinement
- Architecture et API Java Enterprise Edition (Java EE)

Architecture multi-niveaux et tenue à la charge



Niveau intermédiaire

- composants "métier"
- serveur WEB et modules de génération de pages HTML
- services techniques : transactions, persistance, accès aux données, invocations à distance, ...
- proxy d'équilibrage de charge
- mécanismes de tolérance aux pannes

Architecture multi-niveaux et tenue à la charge

Couche intermédiaire

Cette couche est la plus complexe à concevoir car elle rassemble de nombreux services:

- les composants métier qui comportent la totalité de la logique fonctionnelle à destination des utilisateurs,
- les infrastructures techniques telles que :
  - la gestion transactionnelle distribuée (commit à deux phases),
  - l'accès aux données de la couche données,
  - la persistance des données,
  - les optimisations de gestion des threads et des 'pools' de connexions vers les serveurs de données,
  - les possibilités d'invocation des services à distance,
- les solutions permettant de faire de l'équilibrage de charge statique et/ou dynamique:
  - proxy logiciels,
  - clusters de serveurs,
  - ...

Un serveur applicatif devra prendre en charge au maximum ces infrastructures techniques afin que le concepteur puisse se concentrer sur la définition des services "métier".

Le serveur d’applications

Composant

- Définit la totalité de la logique métier

Conteneur

- "Intercepte" les requêtes des clients et fournit un ensemble de services :
  - cycle de vie (création / desctruction d’objets)
  - sécurité applicative (autorisations d’exécuter des méthodes)
  - gestion des transactions
  - persistance
- Gère les ressources du serveur

Serveur

- Héberge un ou plusieurs conteneurs

Le serveur d’applications

Composant

Dans la plupart des cas, le composant EJB contient uniquement du code fonctionnel. Mais il est parfois indispensable qu’il contienne également du code de gestion de sa persistance. Ce mélange entre code métier et code technique peut être limité par une bonne organisation du code.

Conteneur

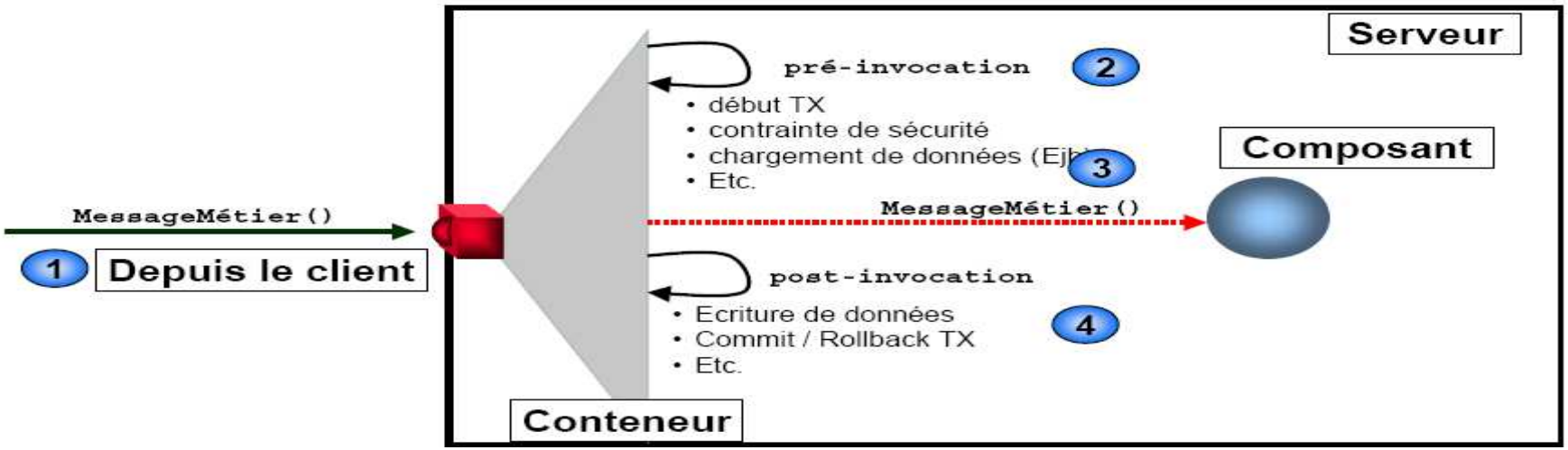
La portabilité des EJB d’un serveur à un autre repose sur la définition précise des interactions entre le serveur d’application et les conteneurs, ainsi que celles entre les composants EJB et les conteneurs.

Serveur

Le serveur doit posséder la capacité d’héberger plusieurs conteneurs de types différents. Il comporte également certaines infrastructures, notamment les capacités d’accès distants.

Rôle du conteneur

Protection des ressources serveur



- objectif principal: créer des réserves de ressources construites dès le démarrage du serveur
- gestion de caches d'objets
- gestion de pools
  - permettent une obtention immédiate des ressources par les clients
  - sont dimensionnés de façon statique afin de protéger le serveur en cas d'augmentation des requêtes provenant des clients
  - garantissent la qualité de service des clients en cours d'exécution

Rôle du conteneur

Protection des ressources serveur

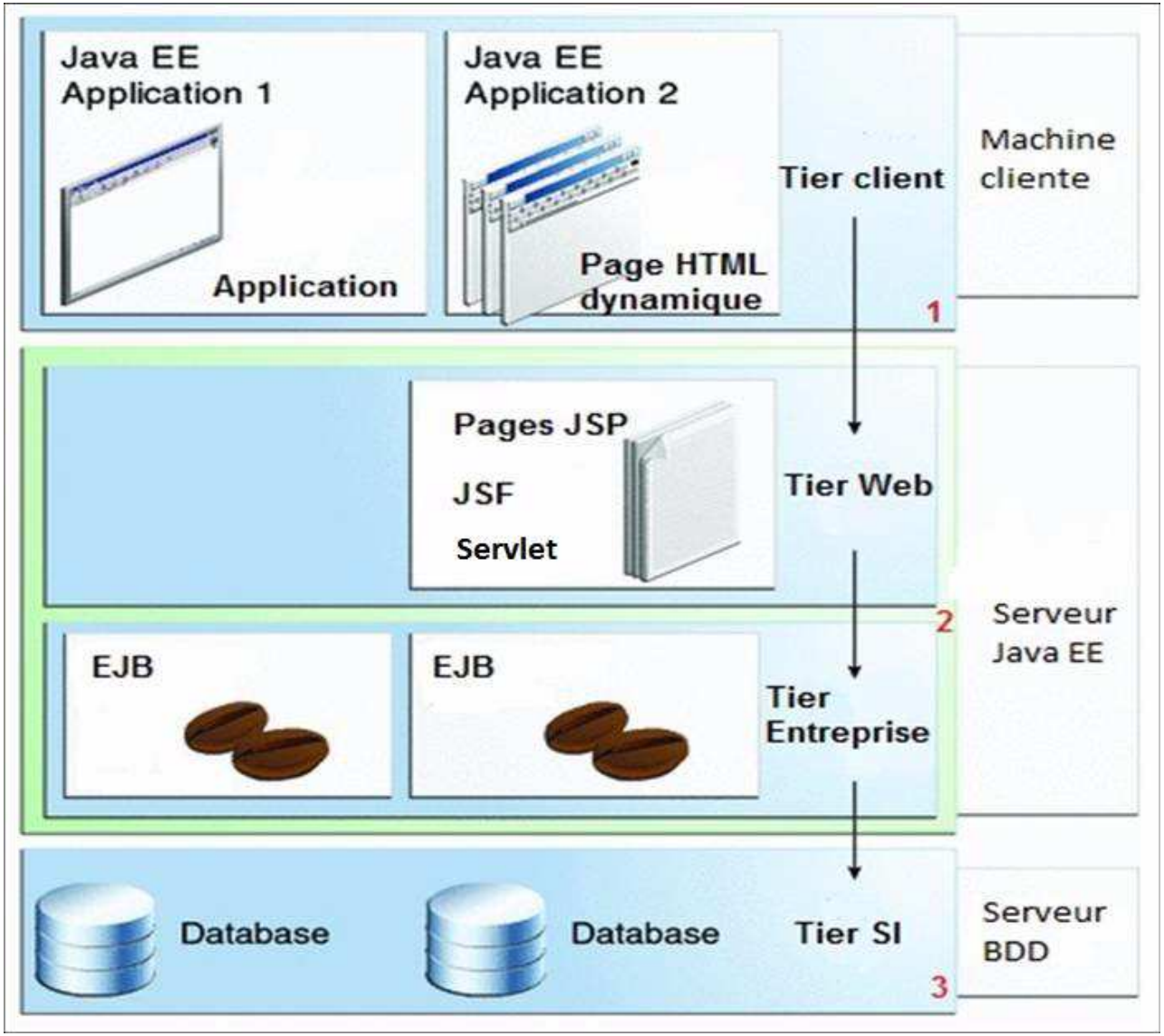
Le conteneur a également pour rôle de construire des réserves de ressources afin d'une part en améliorer les performances d'obtention et d'autre part protéger le serveur contre un trop grand nombre de demandes simultanées.

Par l'intermédiaire de ce mécanisme sont notamment gérées:

- les connexions vers les serveurs de base de données,
- les connexions vers les middlewares orientés messages (MOM),
- les connexions vers les moniteurs transactionnels,
- les réserves de composants EJB,
- les processus légers d'exécution ou threads,
- ...

Architecture Java EE simplifiée

1 : Appel coté client
2 : Serveurs Web et serveurs d'applications
3 : Système d'information



Architecture Java EE simplifiée

Ce schéma présente une architecture multi-niveaux Java EE typique.

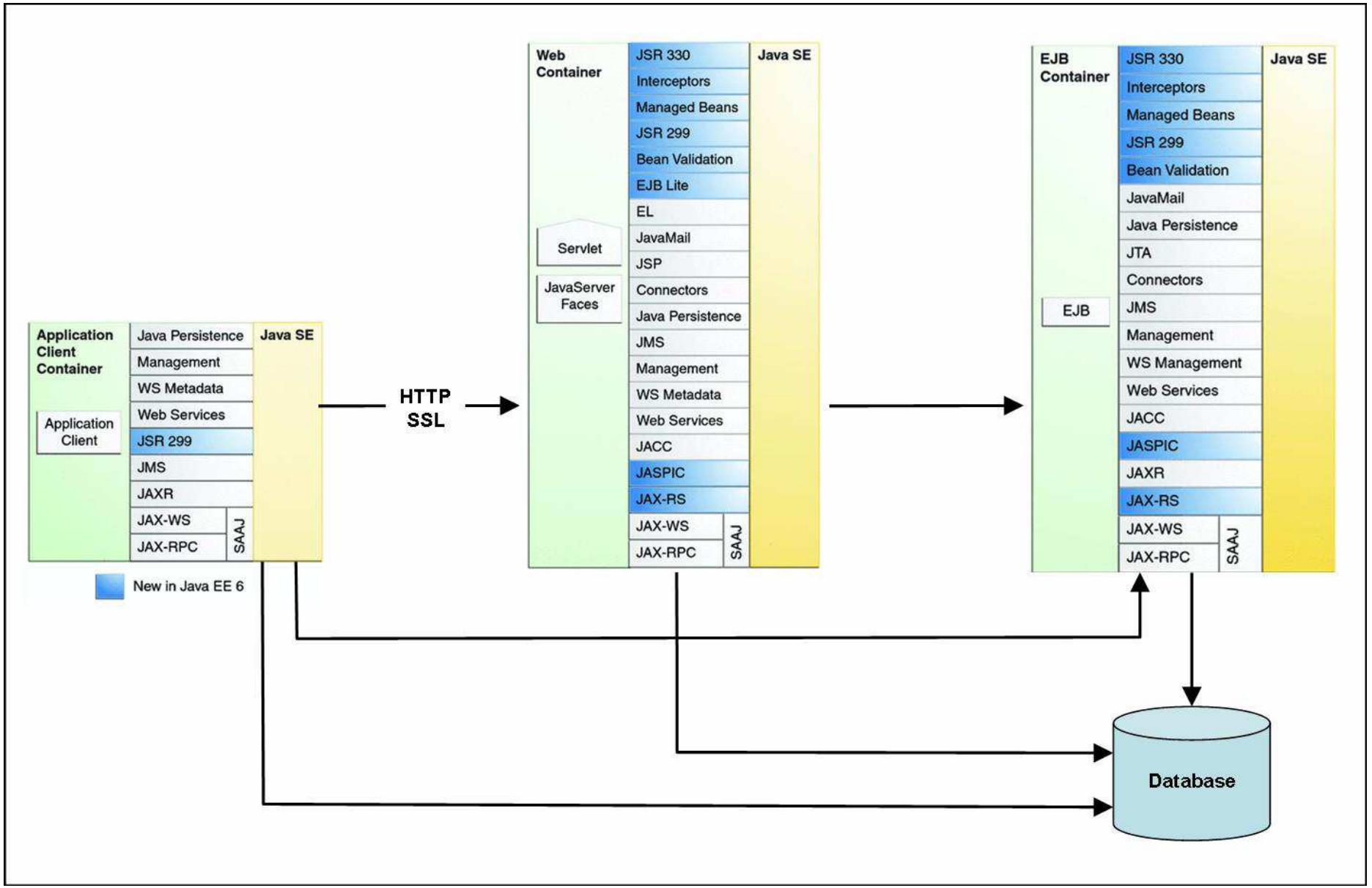
Elle comporte:

- un client léger affichant du HTML
- un serveur intermédiaire comportant deux constituants distincts:
  - un serveur WEB avec des composants servlets et JSP/JSF agissant comme couche de présentation
  - un serveur de composants métiers EJB
- un existant, parfois appelé ’legacy’ comportant SGBDR, applications mainframe, ...

Découpage en couche (tier) permettant de séparer le code métier du code Web

Assure une meilleure sécurité

Architecture Java EE complète



Architecture Java EE complète

Notes :

- D’après http://docs.oracle.com/javaee/6/tutorial/doc/index.html
- Notion de conteneur (*container*): structures d’exécution qui fournissent des services techniques aux composants qu’ils hébergent:
  - Conteneur Web: exécution des JSP/Servlet
  - Conteneur EJB: exécution des Entreprise Java Bean
  - Conteneur client : exécution des applications standalone sur les postes qui utilisent des composants J2EE
- Communication "*Web Container et EJB Container*:" protocole RMI/IIOP: extension du protocole RMI pour intégration avec CORBA

API Java EE

Regroupe un ensemble d'APIs réunis dans 4 groupes distincts

- ***Technologies des Services Web***
  - Java API for RESTful Web Services (JAX-RS) 1.1 (JSR 311)
  - Implementing Enterprise Web Services 1.3 (JSR 109)
  - Java API for XML-Based Web Services (JAX-WS) 2.2 (JSR 224)
  - Java Architecture for XML Binding (JAXB) 2.2 (JSR 222)
  - Web Services Metadata for the Java Platform (JSR 181)
  - Java API for XML-Based RPC (JAX-RPC) 1.1 (JSR 101)
  - Java APIs for XML Messaging 1.3 (JSR 67)
  - Java API for XML Registries (JAXR) 1.0 (JSR 93)
- ***Technologies des applications Web***
  - Java Servlet 3.0 (JSR 315)
  - JavaServer Pages 2.2/Expression Language 2.2 (JSR 245)
  - Standard Tag Library for JavaServer Pages (JSTL) 1.2 (JSR 52)
  - JavaServer Faces 2.0 (JSR 314)

API Java EE

Notes :

API JavaEE

- ***Technologies des applications d'entreprise***
  - Common Annotations for the Java Platform 1.1 (JSR 250)
  - EnterpriseJava Beans 3.1 (avec Interceptors 1.1) (EJB) (JSR 318)
  - Java EE Connector Architecture 1.6 (JCA) (JSR 322): API d'accès à des Entreprise Information System)
  - JavaMail 1.4 (JSR 919): API d'envoi d'emails
  - Java Message Service API 1.1 (JMS) (JSR 914): API de communication asynchrone à base de messages
  - Java Persistence API 2.0 (JPA) (JSR 317): mapping objet/relationnel
  - Java Transaction API (JTA) 1.1 (JSR 907): API de démarcation de transactions
  - Contexts and Dependency Injection for Java (Web Beans 1.0) (JSR 299)
  - Dependency Injection for Java 1.0 (JSR 330)
  - Bean Validation 1.0 (JSR 303)
- ***Technologies de gestion et de sécuritée***
  - J2EE Application Deployment (JSR 88)
  - J2EE Management (JSR 77) et Java Authentication Service Provider Interface for Containers (JSR 196)
  - Java Authorization Contract for Containers (JSR 115) et

API JavaEE

Notes :

Architectures distribuées (Polytech)
JAX-WS : Mise en oeuvre

Version 3.0

- Choix d'une implémentation
- Développement d'un service à partir d'une classe Java
- Développement d'un service à partir d'une description WSDL
- Développement d'un client
- Annotations

Choix d'une implémentation

Open source

- Glassfish Metro : implémentation de référence
- JBossWS
- Apache CXF (anciennement XFire, aujourd'hui dans Incubator)

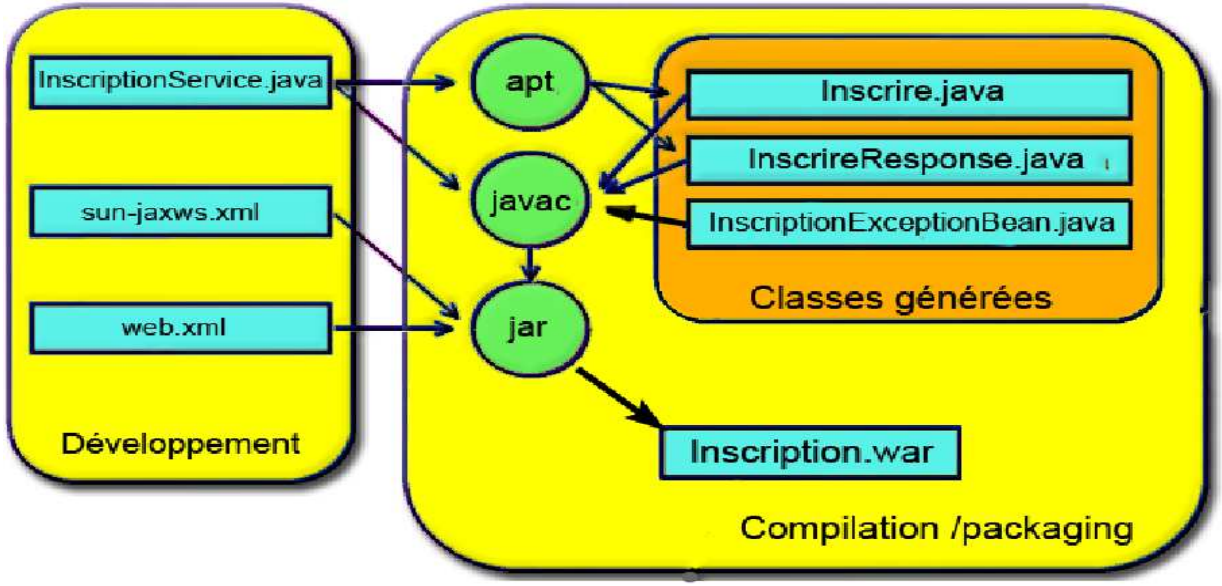
Commerciales

- Oracle Weblogic Server
- IBM Websphere
- ...

Choix d’une implémentation

Notes

Développement d'un service à partir d'une classe Java



- Ecrire une classe Java de type POJO
- Ajouter des annotations
- Générer les classes (portables artifacts)
- Ecrire les descripteurs de déploiement
- Packager sous la forme d'une application Web

Développement d’un service à partir d’une classe Java

Notes

Développement d'un service à partir d'une classe Java

Ecrire une classe de type POJO

- Cette classe constitue l'implémentation du Endpoint du Service.
- Les méthodes publiques de cette classe seront les opérations du Service.
- Les paramètres et types de retour de ces méthodes doivents être compatibles avec JAXB 2.x.
- Les méthodes peuvent lancer une java.rmi.RemoteException en plus de toute autre exception spécifique.
- Les paramètres et types de retour ne doivent pas implémenter l'interface java.rmi.Remote.

```
public class MessageManager {

     public String getMessage(String key, String langIsoCode) throws UnknownKeyException  {

          Locale locale = new Locale(langIsoCode);
          ResourceBundle bundle = ResourceBundle.getBundle("messages", locale);
          try  {
               String message = bundle.getString(key);
               return message;
          } catch  (MissingResourceException mre) {
               throw new UnknownKeyException(key);
          }
     }
}
```

Développement d’un service à partir d’une classe Java

Notes

Développement d'un service à partir d'une classe Java

Ajouter les annotations

- La classe d'implémentation doit porter l'annotation @WebService
- Les méthodes exposées peuvent porter l'annotation @WebMethod

```
@WebService(name="message", targetNamespace="http://www.leuville.com/message")
@WebFault(name="erreur-message", targetNamespace="http://www.leuville.com/message/erreur")
@SOAPBinding(style=SOAPBinding.Style.DOCUMENT, use=SOAPBinding.Use.LITERAL)
public class MessageManager {
	@WebMethod(operationName="recupererMessage")
	@WebResult(name="libelle")
	public String getMessage(
			@WebParam(name="message-key", mode=WebParam.Mode.IN, targetNamespace="http://www.leuville.com/message")
			String key,
			@WebParam(name="lang-iso-code", mode=WebParam.Mode.IN, targetNamespace="http://www.leuville.com/message")
			String langIsoCode)
		throws UnknownKeyException  {
		Locale locale = new Locale(langIsoCode);
		ResourceBundle bundle = ResourceBundle.getBundle("messages", locale);
		try  {
			return bundle.getString(key);
		} catch  (MissingResourceException mre) {
			throw new UnknownKeyException(key);
		}
	}
}
```

Développement d’un service à partir d’une classe Java

Notes

Développement d'un service à partir d'une classe Java

Générer les portable artifacts

- Deux solutions :
  - Utiliser la tâche ant apt :

```
<apt
     debug="false"
     verbose="false"
     destdir="./build"
     srcdir="./src"
     preprocessdir="./src">
     <classpath>
          <fileset dir="./WebContent/WEB-INF/lib" includes="*.jar" />
     </classpath>
</apt>
```

  - Utiliser l'outil wsgen :

```
%JAXWS_HOME%\bin\wsgen.bat -cp .;build\classes -s src -r WebContent\WEB-INF com.leuville.message.MessageManager
```

Développement d'un service à partir d'une classe Java

- La tâche apt fait partie des Core Tasks de Ant.
- L'outil wsgen est disponible avec les implémentations de JAX-WS

```
Usage: WSGEN [options] <SEI>

where [options] include:
  -classpath <path>          specify where to find input class files
  -cp <path>                 same as -classpath <path>
  -d <directory>             specify where to place generated output files
  -extension                 allow vendor extensions - functionality not specified by the specification.  Use of exten-
sions may result in applications that are not portable or may not interoperate with other implementations
  -help                      display help
  -keep                      keep generated files
  -r <directory>             resource destination directory, specify where to place resouce files such as WSDLs
  -s <directory>             specify where to place generated source files
  -verbose                   output messages about what the compiler is doing
  -version                   print version information
  -wsdl[:protocol]           generate a WSDL file.  The protocol is optional. Valid protocols are soap1.1 and Xsoap1.2,
the default is soap1.1.  Xsoap1.2 is not standard and can only be used in conjunction with the -extension option
  -servicename <name>        specify the Service name to use in the generated WSDL Used in conjunction with the -wsdl
option.
  -portname <name>          specify the Port name to use in the generated WSDL Used in conjunction with the -wsdl option.

Examples:
  wsgen -cp . example.Stock
  wsgen -cp . example.Stock -wsdl -servicename {http://mynamespace}MyService
```

Développement d'un service à partir d'une classe Java

Ecrire les descripteurs de déploiement

sun-jaxws.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<endpoints xmlns="http://java.sun.com/xml/ns/jax-ws/ri/runtime" version="2.0">
   <endpoint name="message" implementation="com.leuville.message.MessageManager" url-pattern="/message" />
</endpoints>
```

web.xml

```
<listener>
    <listener-class>com.sun.xml.ws.transport.http.servlet.WSServletContextListener</listener-class>
  </listener>
  <servlet>
    <servlet-name>message</servlet-name>
    <servlet-class>com.sun.xml.ws.transport.http.servlet.WSServlet</servlet-class>
    <load-on-startup>1</load-on-startup>
  </servlet>
  <servlet-mapping>
    <servlet-name>message</servlet-name>
    <url-pattern>/message</url-pattern>
  </servlet-mapping>
```

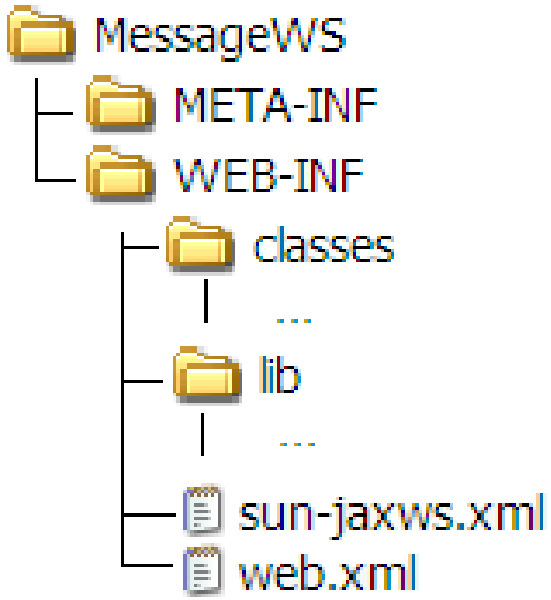
Développement d’un service à partir d’une classe Java

Notes

Développer un Service à partir d'une classe Java

Packager

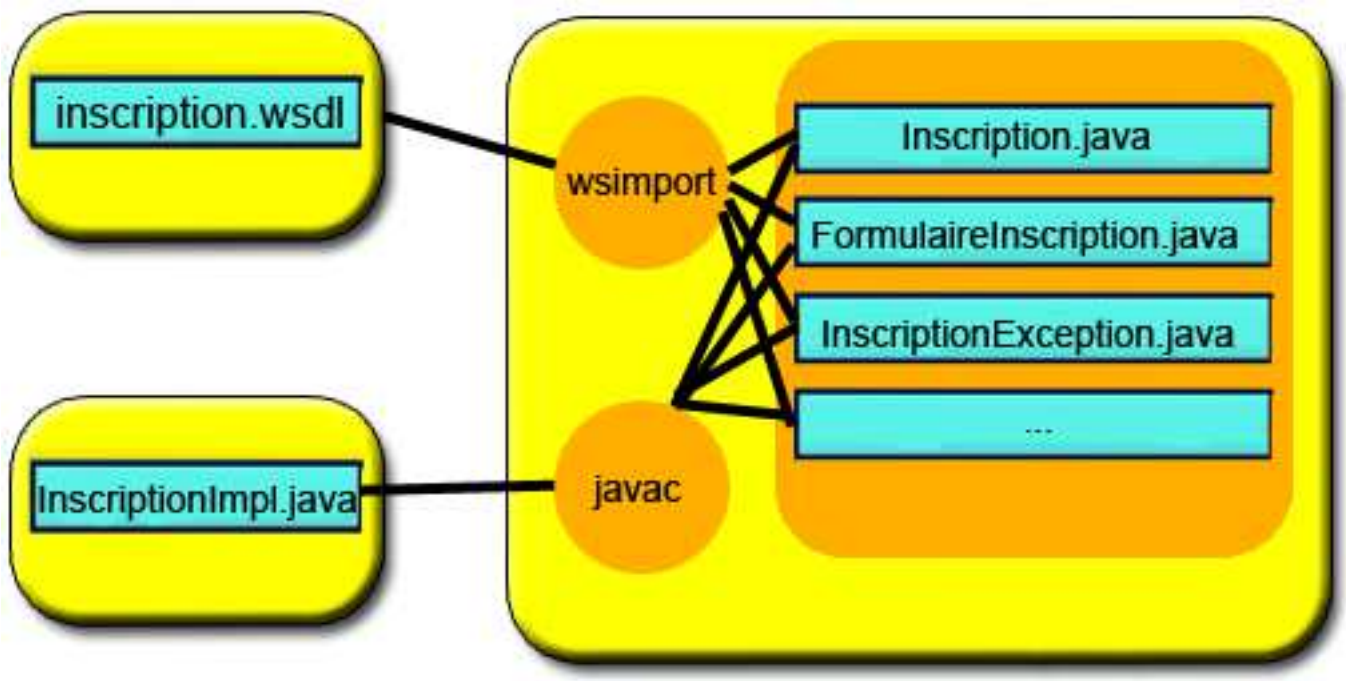
- Il s'agit de créer une archive au format WAR ayant la structure suivante :



Développer un Service à partir d'une classe Java

Notes

Développer un Service à partir d'une description WSDL



- Générer l'interface du service à partir de la description WSDL
- Implémenter l'interface du service
- Packager le service au format WAR

Développer un Service à partir d'une description WSDL

Notes

Développer un Service à partir d'une description WSDL

Générer l'interface du service

- Deux solutions :
  - Utiliser la tâche ant wsimport :

```
<target name="generateService">
    <wsimport
        verbose="false"
        keep="true"
        destdir="./build"
        sourcedestdir="./src"
        wsdl="inscription.wsdl"/>
</target>
```

  - Utiliser l'outil wsimport :

```
%JAXWS_HOME%\bin\wsimport.bat -s src inscription.wsdl
```

Développer un Service à partir d'une description WSDL

Notes

Développer un service à partir d'une description WSDL

Implémenter l'interface générée

Interface générée :

```
@WebService(name = "inscription", targetNamespace = "http://www.leuville.com/formation")
@XmlSeeAlso({
    ObjectFactory.class
})
public interface Inscription {

    @WebMethod
    @WebResult(name = "inscrit", targetNamespace = "")
    @RequestWrapper(localName = "inscription", targetNamespace = "http://www.leuville.com/formation",
            className = "com.leuville.formation.Inscription_Type")
    @ResponseWrapper(localName = "inscriptionResponse",
            targetNamespace = "http://www.leuville.com/formation",
            className = "com.leuville.formation.InscriptionResponse")
    public boolean inscription(
        @WebParam(name = "formulaire-inscription",
                          targetNamespace = "http://www.leuville.com/formation")
        FormulaireInscription formulaireInscription)
        throws InscriptionException_Exception
    ;
}
```

Développer un service à partir d'une description WSDL

Notes

Développer un service à partir d'une description WSDL

Implémenter l'interface générée

Implémentation :

```
@WebService(endpointInterface="com.leuville.formation.Inscription")
public class InscriptionImpl implements Inscription{

    public boolean inscription(FormulaireInscription formulaireInscription)
            throws InscriptionException_Exception {
        return false;
    }

}
```

- L'implémentation hérite de toutes les annotations de l'interface.

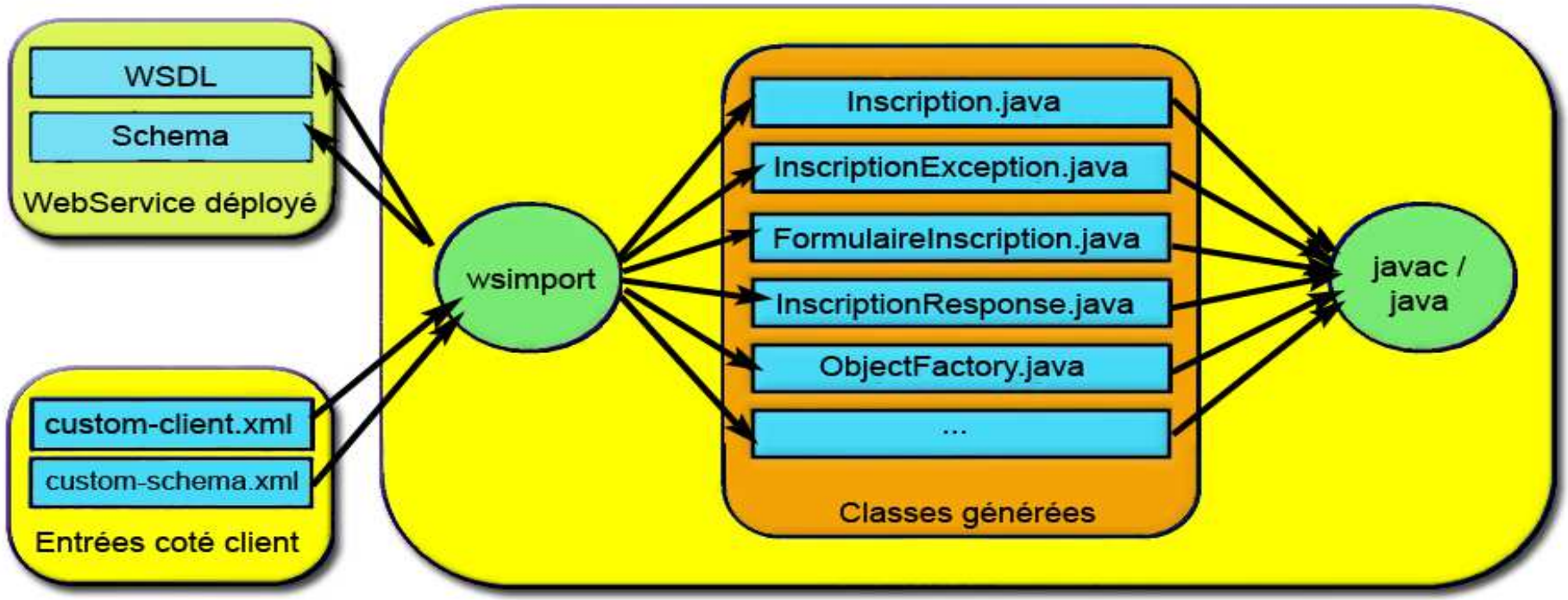
Packager

- Le packaging est le même que pour un service générée à partir d'une classe.

Développer un service à partir d'une description WSDL

Notes

Développer un client



- L'outil wsimport permet de générer des classes proxy à partir de la description WSDL du service.
- Il s'agit du même utilitaire que celui utilisé lors de la génération des classes du service depuis le WSDL.

Développer un client

Notes

Développer un client

Utilisation de la classe proxy

```
// Création du formulaire d'inscription
FormulaireInscription form = new FormulaireInscription();
form.setNomClient("Leuville");
form.setPrenomClient("Objects");
form.setCodeCours("JAX-WS");
form.setDateSession(
        DatatypeFactory.newInstance()
            .newXMLGregorianCalendar(
                    (GregorianCalendar) GregorianCalendar.getInstance()));

// Invocation du service
Inscription service = new InscriptionServiceService().getInscriptionPort();
boolean result = service.inscription(form);
```

Développer un client

Notes

Annotations

Annotations issues de la JSR 181 (Web Services Metadata)

Annotation	Description
javax.ws.WebService	Marque la classe comme étant l'implémentation du endpoint d'un service.
javax.ws.WebMethod	Marque une méthode comme étant une opération exposée par le service.
javax.ws.OneWay	Marque une méthode web comme étant asynchrone.
javax.ws.WebParam	Permet de modifier le mapping XML d'un paramètre d'une méthode web.
javax.ws.WebResult	Permet de modifier le mapping XML de la valeur de retour d'une méthodeweb.
javax.ws.HandlerChain	Permet de référencer un fichier XML de déclaration de chaine de Handlers a affecter au Service.
javax.ws.soap.SOAPBinding	Permet de configurer le style et l'encodage des opérations du service

Annotations

Notes

Annotations

Annotations issues de la JSR 224 (JAX-WS)

Annotation	Description
javax.xml.ws.BindingType	Indique le binding à utiliser pour invoquer le service
javax.xml.ws.RequestWrapper	Permet de personnaliser l'élément contenant les paramètres d'une méthode Document / Literal.
javax.xml.ws.ResponseWrapper	Permet de personnaliser l'élément contenant le retour d'une méthode Document / Literal.
javax.xml.ws.ServiceMode	Permet de spécifier le mode de récupération du message (MESSAGE ou PAYLOAD)
javax.xml.ws.WebEndpoint	Permet d'identifier un élément port au sein de la description WSDL du service
javax.xml.ws.WebFault	Permet de personnaliser l'élément fault pouvant être renvoyé par une opération
javax.xml.ws.WebServiceClient	Spécifie l'élément service associé à un client généré.
javax.xml.ws.WebServiceProvider	Permet de configurer une classe de type Provider.
javax.xml.ws.WebServiceRef	Permet de définir une référence à un WebService (ressource au sens Java EE).
javax.xml.ws.Action	Permet d'associer une action à une opération.
javax.xml.ws.FaultAction	Permet de préciser une Fault au sein d'une Action

Annotations

Notes

Annotations

Annotations issues de la JSR 222 (JAXB)

Annotation	Description
javax.xml.bind.annotation.XmlRootElement	Permet de configurer l'élément racine d'un document XML
javax.xml.bind.annotation.XmlAccessorType	Indique les données devant être sérialisées.
javax.xml.bind.annotation.XmlType	Permet d'associer une classe à un type XML.
javax.xml.bind.annotation.XmlElement	Permet d'associer un attribut à un élément XML.
javax.xml.bind.annotation.XmlSeeAlso	Indique à JAXB de binder d'autres classes

Annotations

Notes

Annotations

Annotations issues de la JSR 250 (Common Annotations)

Annotation	Description
javax.annotation.Resource	Permet de marquer une ressource de type WSContext.
javax.annotation.PostConstruct	Permet d'indiquer une méthode à exécuter après les injections de dépendances et avant que la classe ne commence à répondre aux requêtes.
javax.annotation.PreDestroy	Permet d'indiquer une méthode à exécuter avant la suppression de l'instance.

Annotations

Notes

Architectures distribuées (Polytech)
Le niveau métier

Version 3.0

- La représentation UML d'un objet métier
- Les design patterns Blueprints associés

La représentation d’un objet métier

Plusieurs points de vue possibles

- Point de vue logique ou conceptuel
  - apport des design patterns de type Blueprints
- Point de vue d’exploitation

Apports d’UML

- diagramme de classes ou d’objets
- diagramme de séquences, de collaborations ou d’activités
- diagramme de paquetages
- diagramme de composants et de déploiement
- UML 2 : diagramme de structure composite

La représentation d’un objet métier

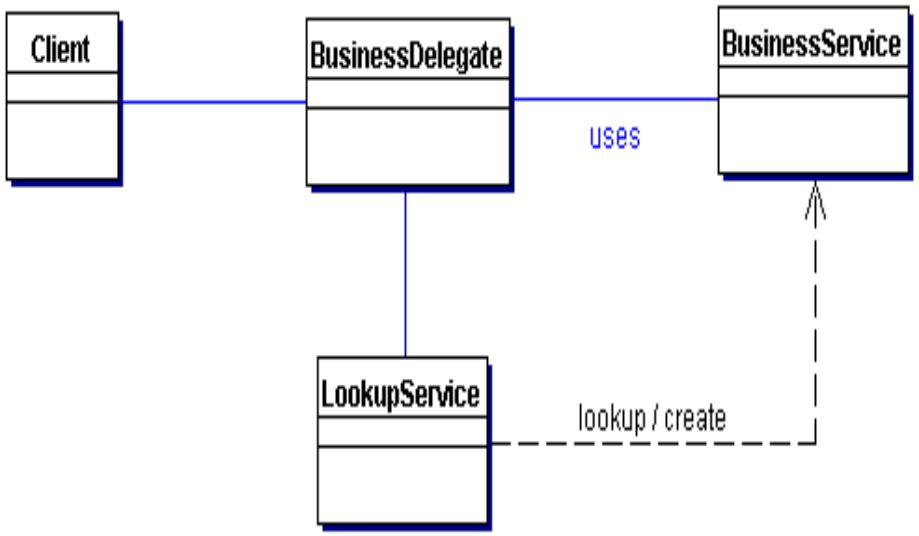
Notes

SessionFacade

Objectifs

- Définir une façade d'accès aux services métier avec un haut niveau d'abstraction
- Définir une séparation entre les services métier et leurs utilisateurs
  - les clients ne dépendent pas de la technologie utilisée dans la couche métier
  - la maintenance est facilitée
- Centraliser certains services techniques : authentification, permissions, transactions

Propositions



SessionFacade

Propositions

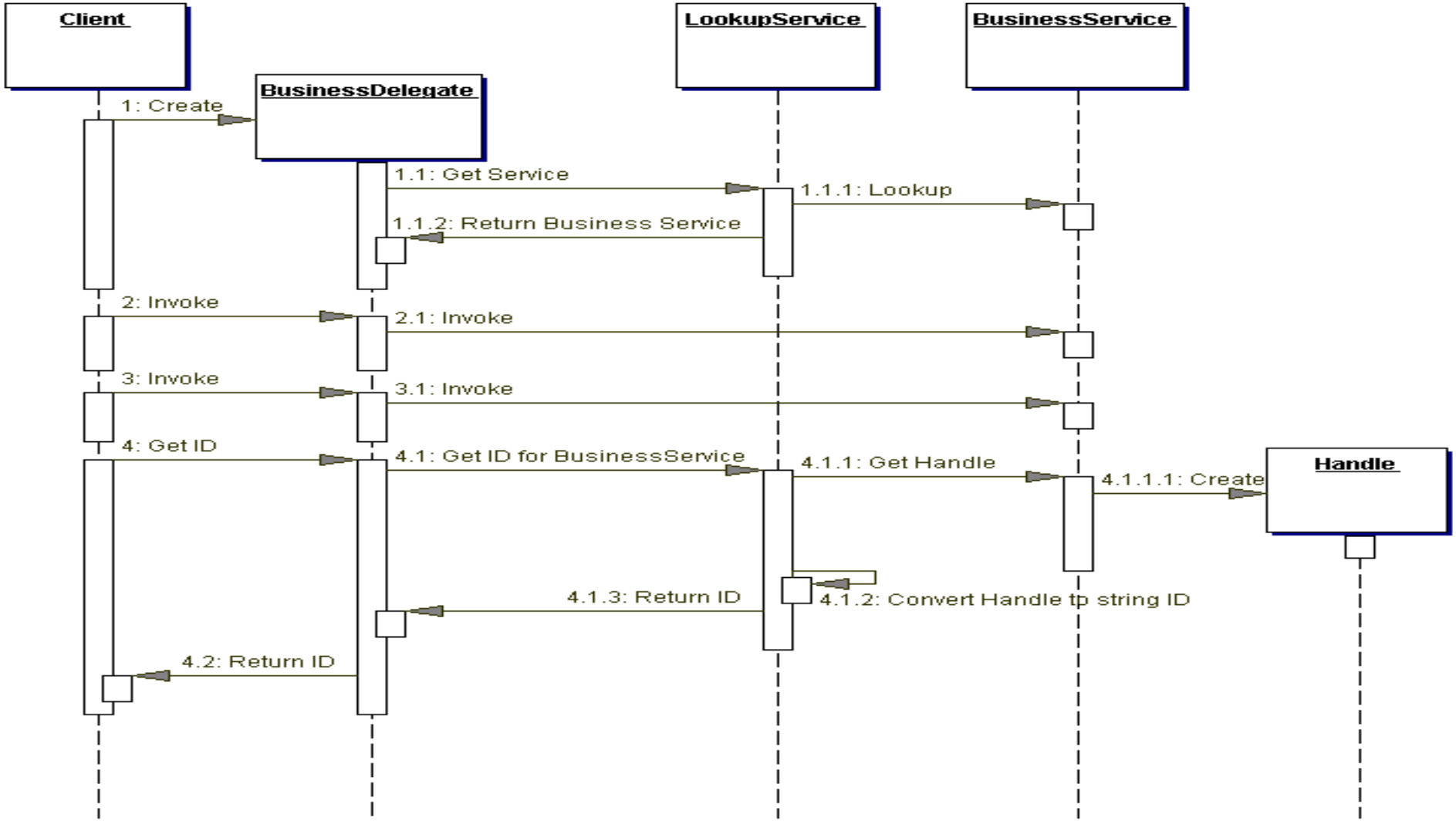
SessionFacade est accompagné d’un autre pattern appelé ServiceLocator (LookupService sur le diagramme).

Son objectif est d’encapsuler le code d’accès au service de nommage ou annuaire permettant de retrouver les différents objets nécessaires à la réalisation des traitements.

Il sera détaillé dans la suite de ce chapitre.

SessionFacade

Exemple de diagramme de séquences



SessionFacade

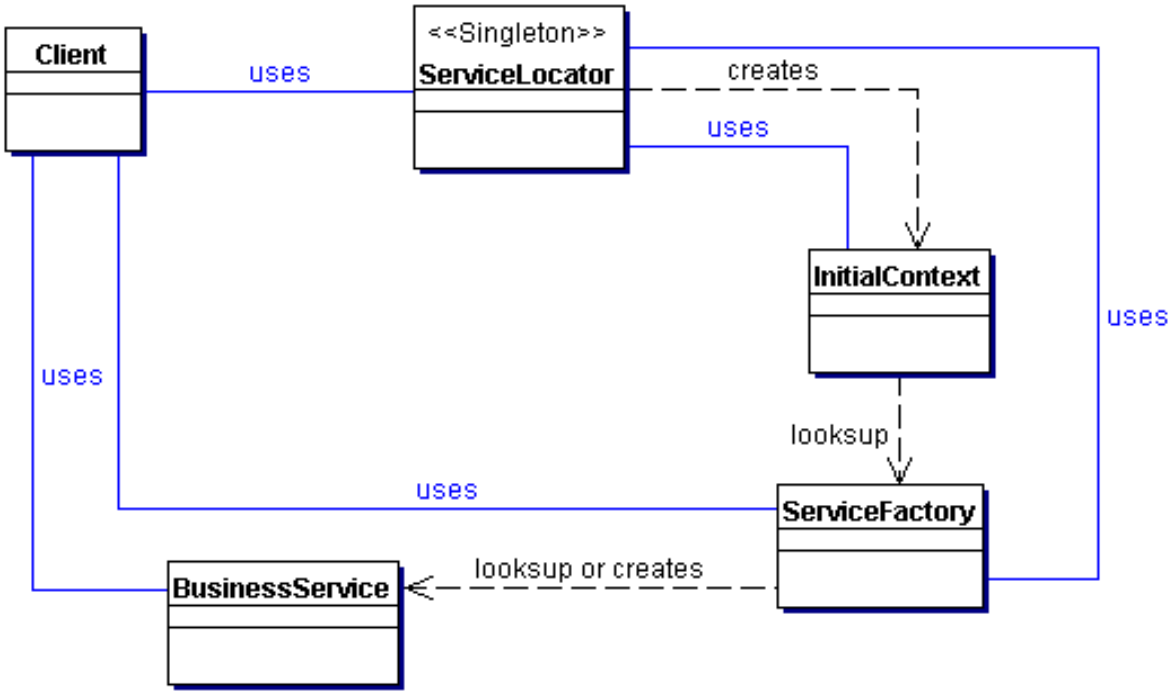
Notes

ServiceLocator

Objectifs

- Encapsuler le code technique d'accès au service de nommage d'un serveur d'applications J2EE
- Servir de cache pour les "fabriques" de composants EJB

Propositions



ServiceLocator

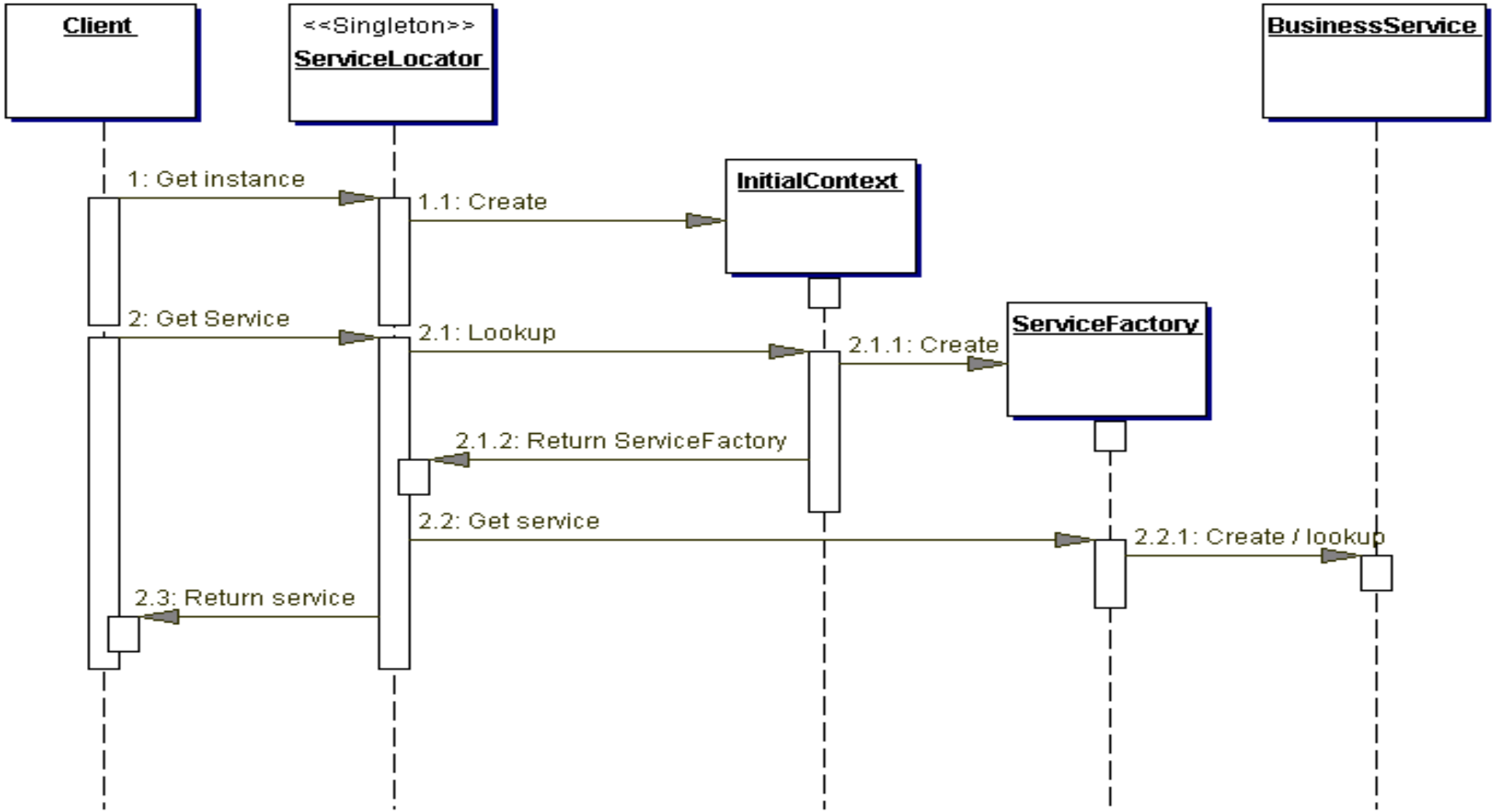
Notes

ServiceLocator gère des singletons de fabriques de composants EJB (EJBHome).

Pour cela, il est possible de se baser sur le pattern GOF Singleton.

ServiceLocator

Exemple de diagramme de séquences



ServiceLocator

Notes

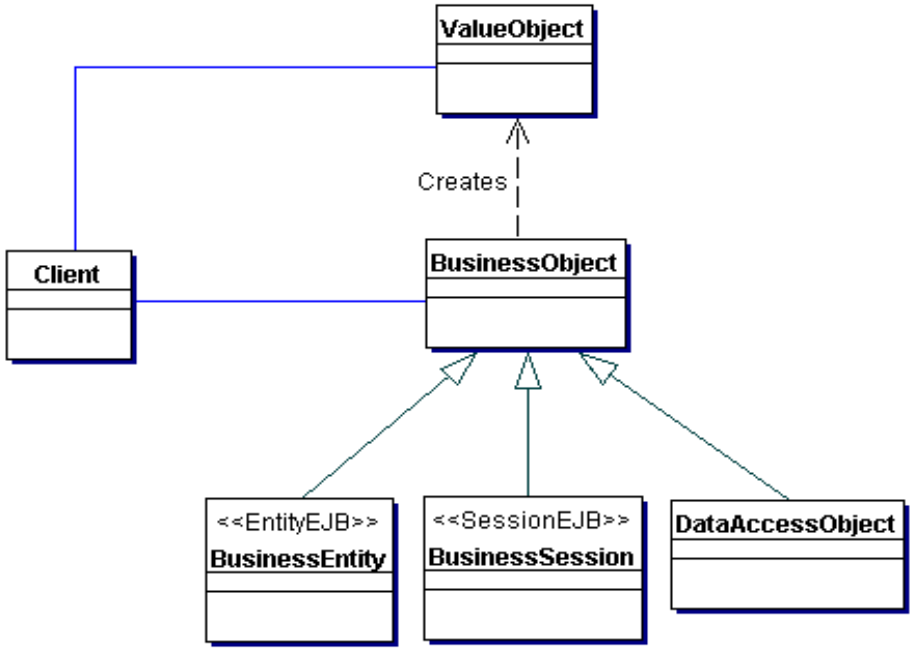
TransferObject ou ValueObject

Objectifs

- Eviter ou limiter les multiples accès réseau nécessaires à l'obtention de propriétés de composants métier

Propositions

- Agréger plusieurs propriétés d'un composant métier dans un seul objet sérialisable



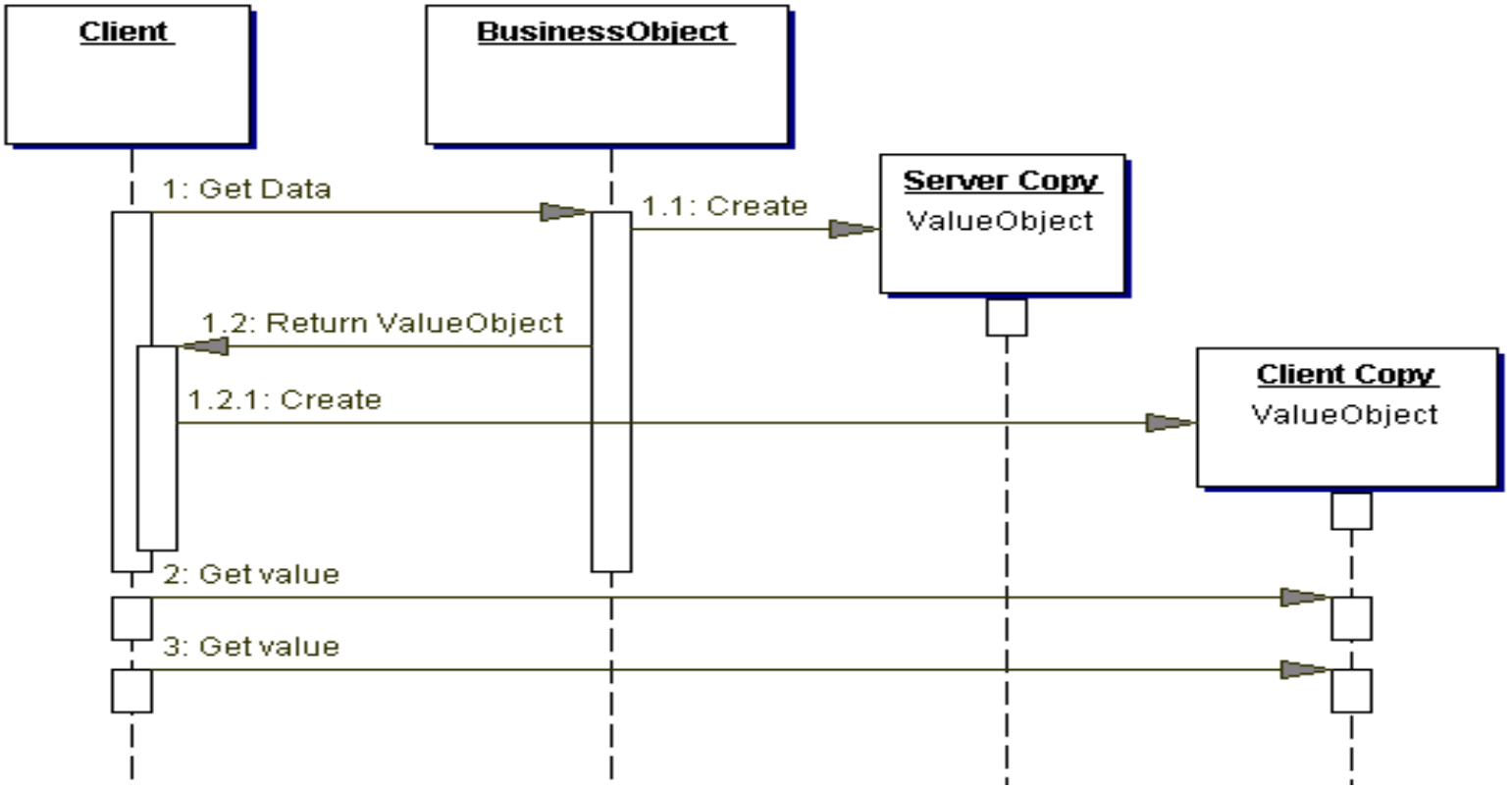
- Différentes stratégies possibles : données en lecture seulement, en lecture / écriture, sous-ensembles de données, ...

TransferObject ou ValueObject

Notes

TransferObject ou ValueObject

Exemple de diagramme de séquences



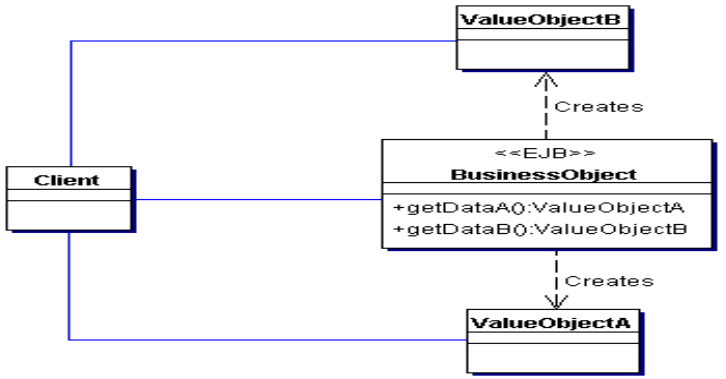
TransferObject ou ValueObject

Notes

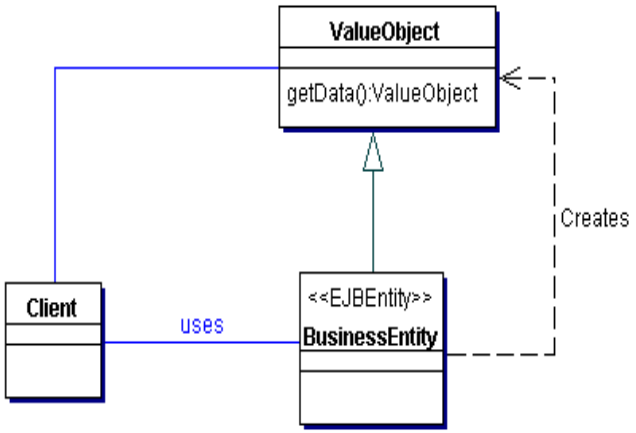
TransferObject ou ValueObject

Stratégies possibles

- En lecture seulement : stratégie la plus simple
- En lecture / écriture : difficultés en cas de modifications simultanées
- Avec retour de plusieurs types de TransferObject



- En utilisant l'héritage



TransferObject ou ValueObject

Stratégies possibles

Un objet métier peut retourner plusieurs types de TransferObjet/ValueObject suivant les besoins.

Il est possible de rendre le mécanisme générique grâce aux interface de Java:

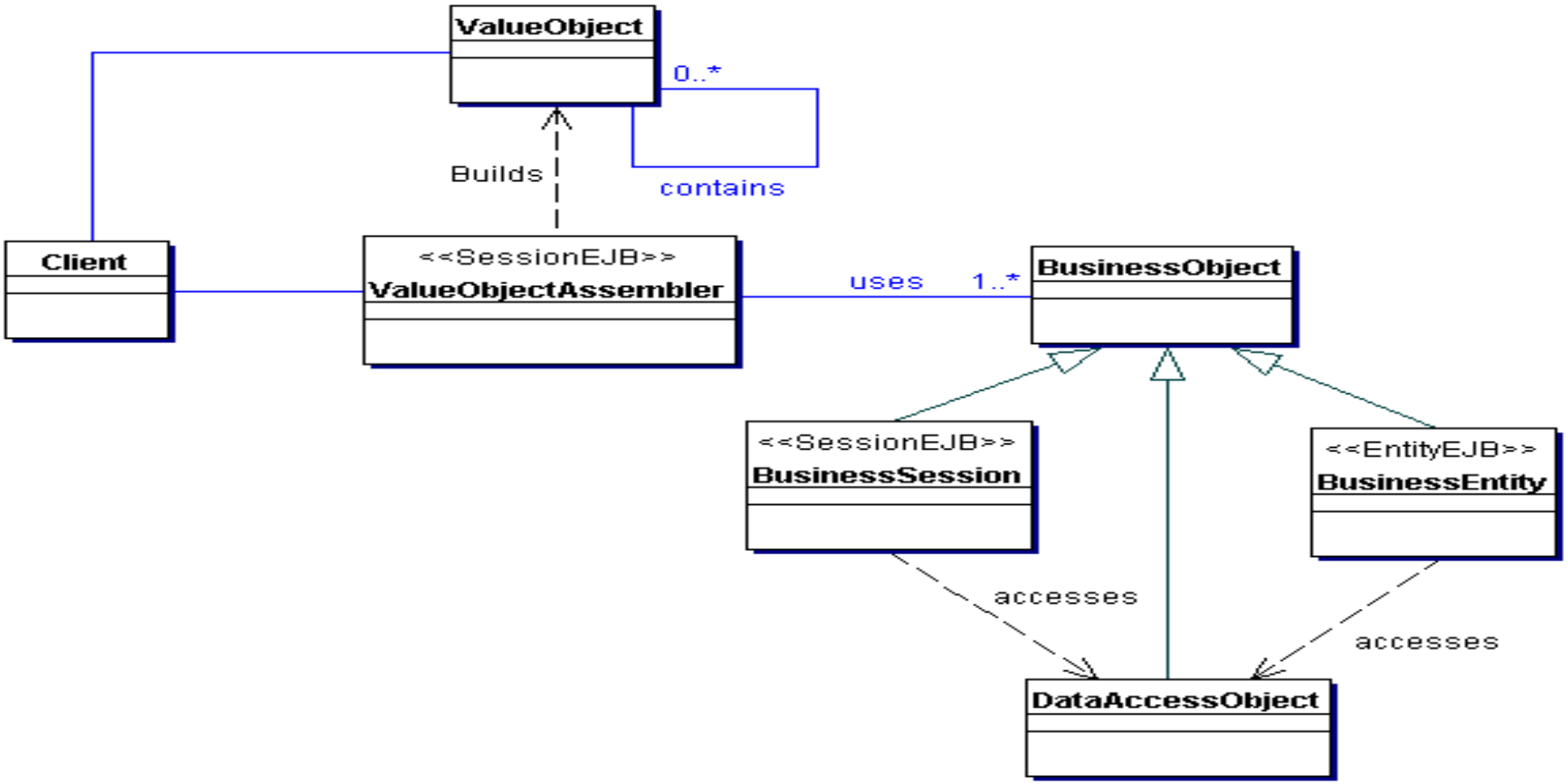
- chaque type de TransferObjet est décrit par une interface contenant les accesseurs intéressants
- le client passe en paramètre à l'objet métier l'interface désirée
- l'objet métier instancie un TransferObject correspondant à l'interface. Il peut ensuite le renseigner par introspection.

TransferObjectAssembler ou ValueObjectAssembler

Objectifs

- Définir un moyen d'élaborer les structures de données ou objets complexes nécessaire aux clients

Propositions

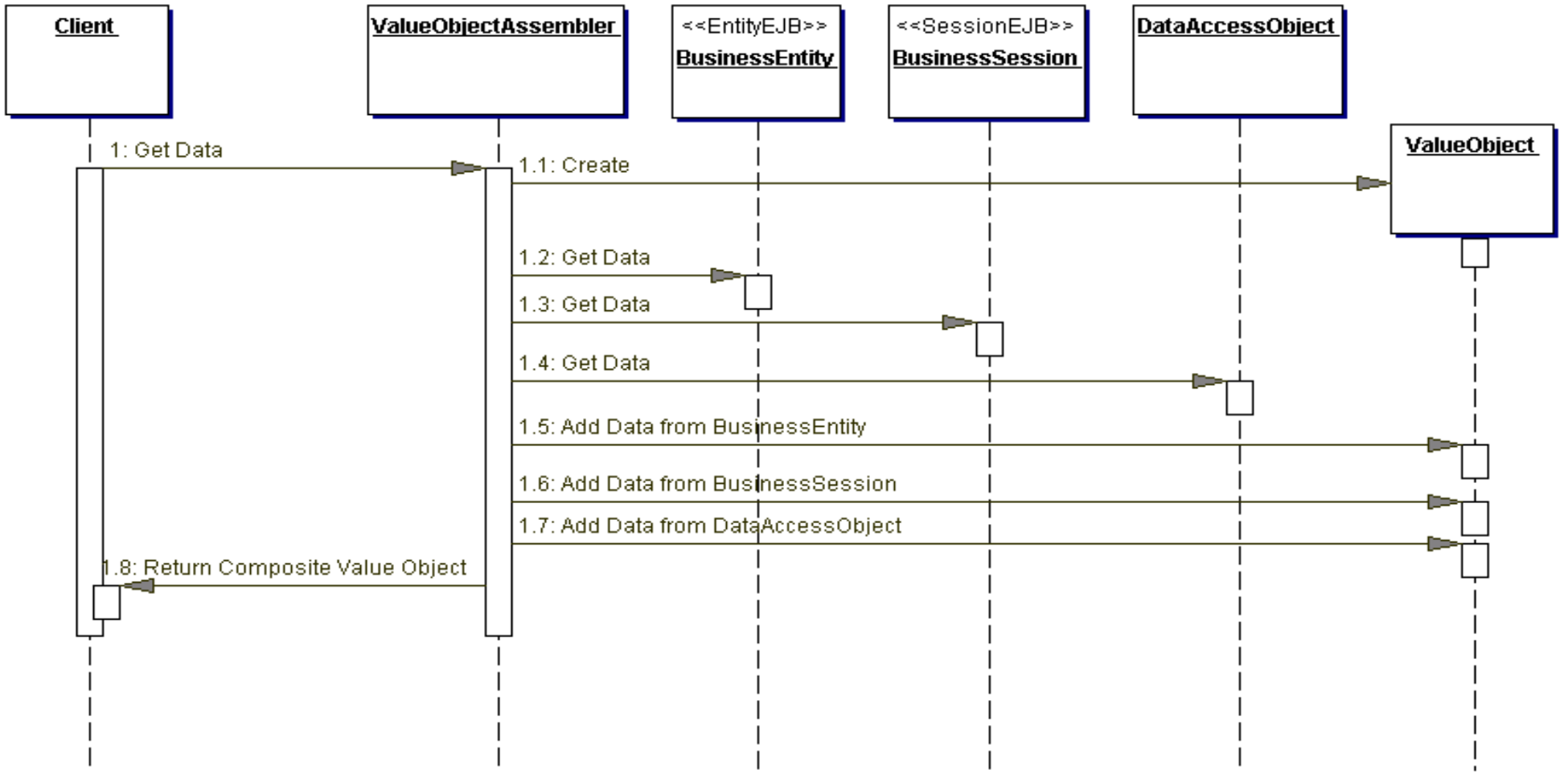


TransferObjectAssembler ou ValueObjectAssembler

Notes

TransferObjectAssembler ou ValueObjectAssembler

Exemple de diagramme de séquences



TransferObjectAssembler ou ValueObjectAssembler

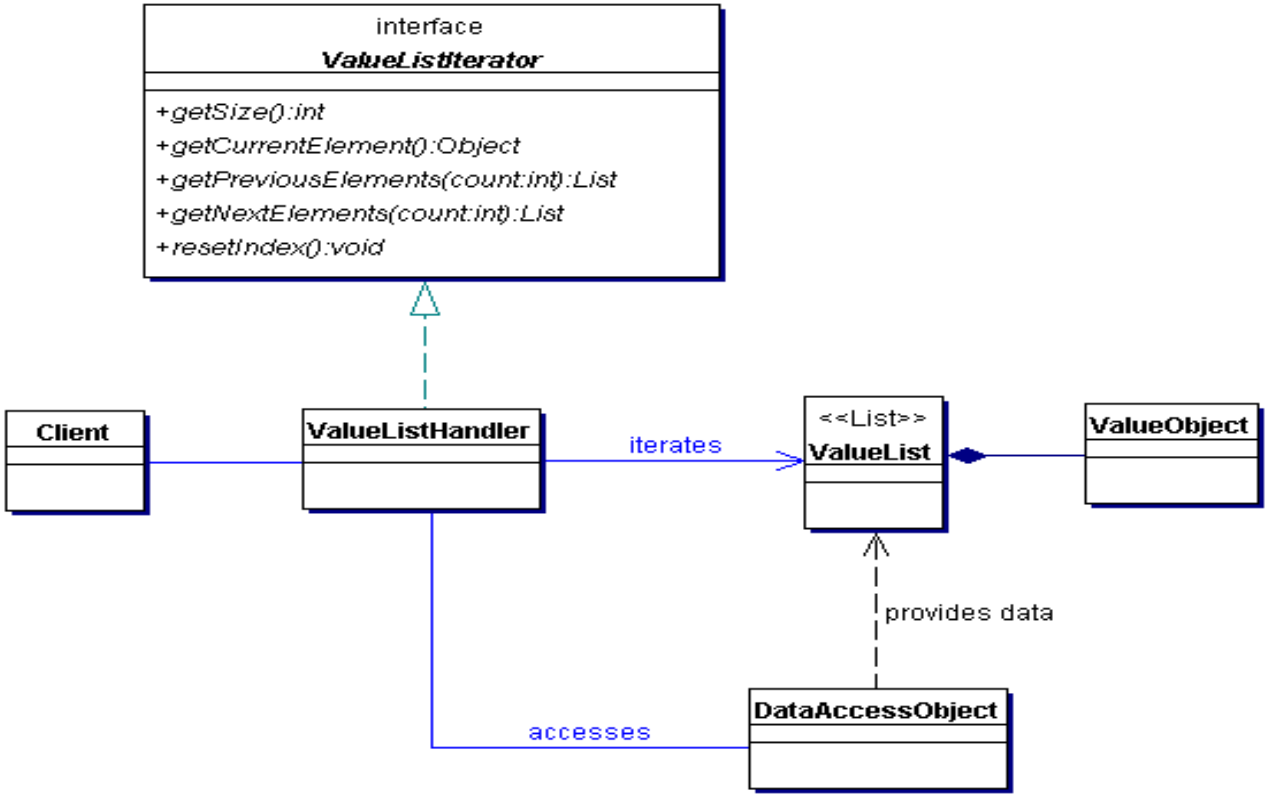
Notes

ValueListHandler

Objectifs

- Définir un moyen de naviguer dans une liste de résultats
- Proposer un mécanisme de cache

Propositions

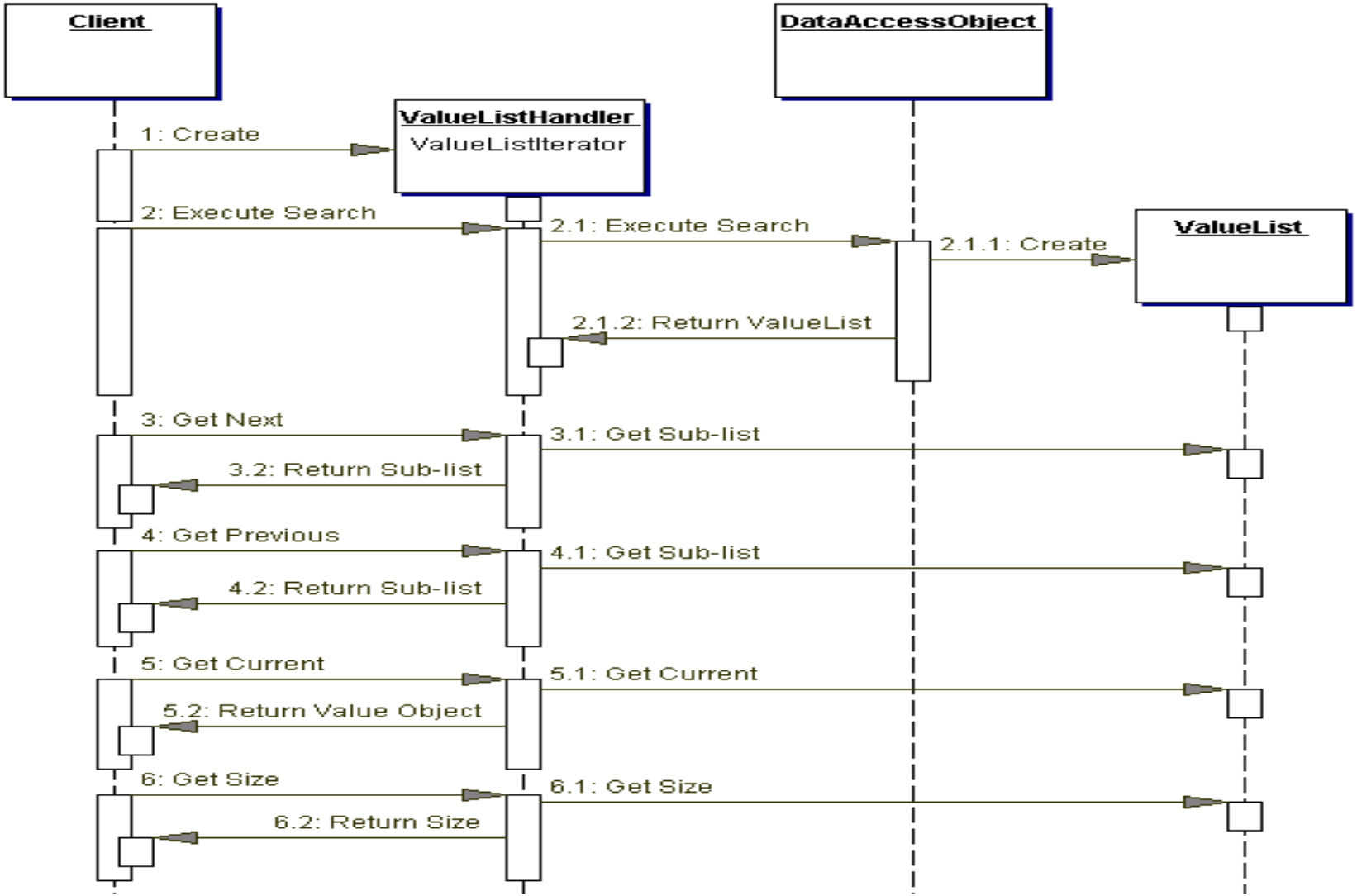


ValueListHandler

Notes

ValueListHandler

Diagramme de séquences



ValueListHandler

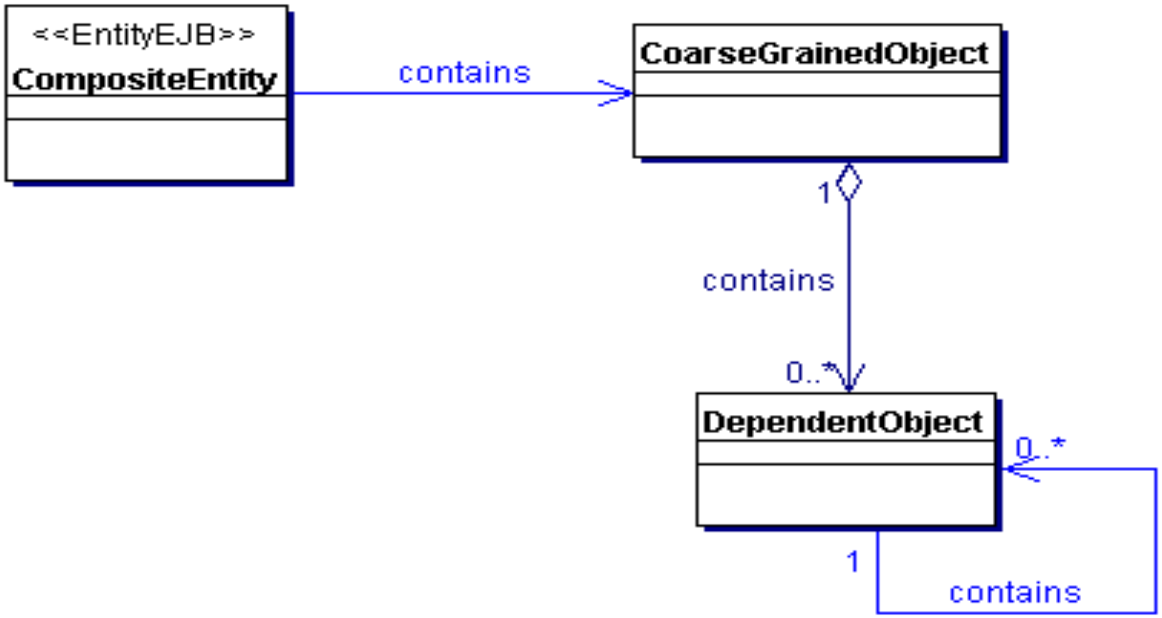
Notes

CompositeEntity

Objectifs

- Optimiser le fonctionnement d'applications utilisant des EJB Entité

Propositions



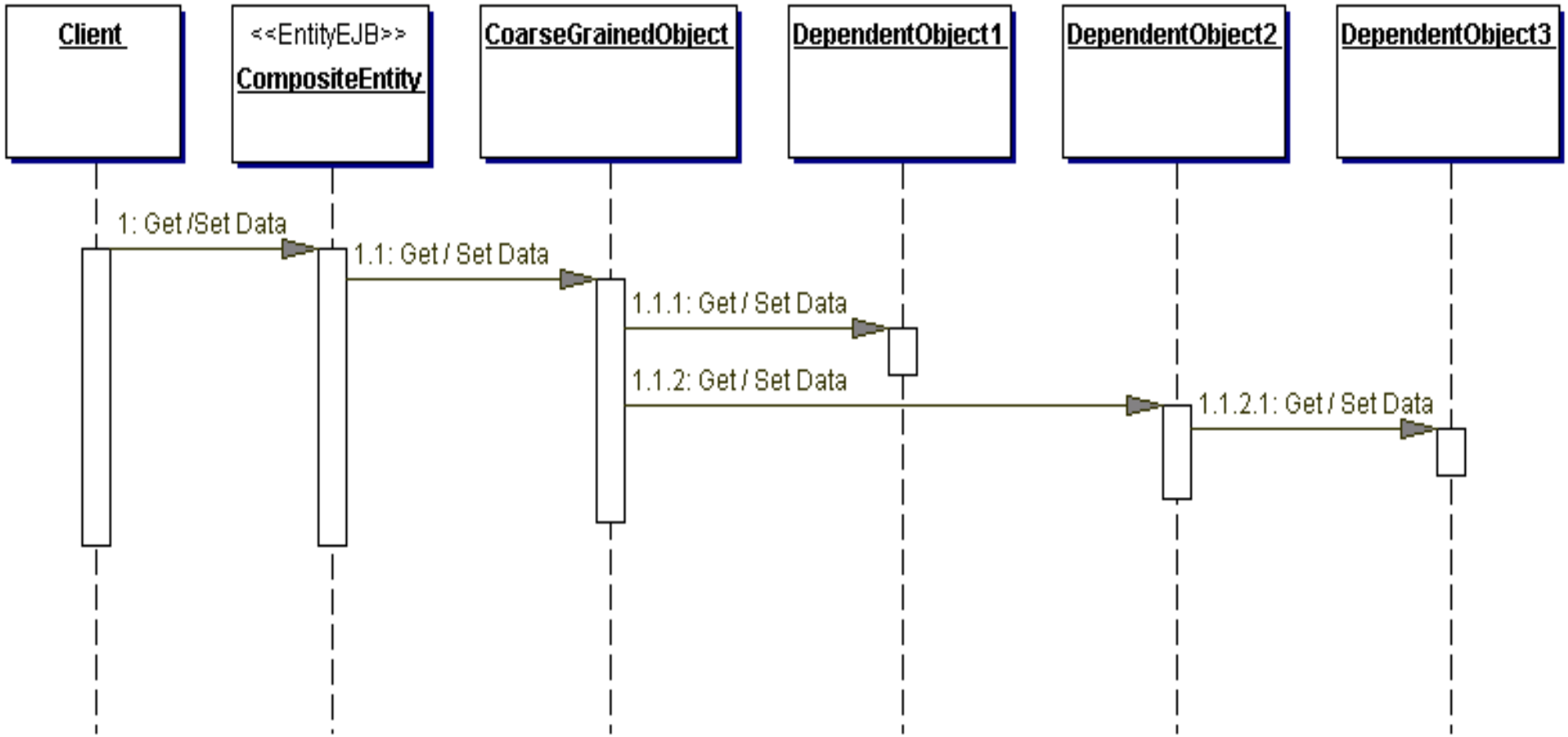
- Utiliser des EJB Entity pour représenter des objets à grosse granularité
- Utiliser des objets "classiques" pour représenter les objets dépendants

CompositeEntity

Notes

CompositeEntity

Exemple de diagramme de séquences



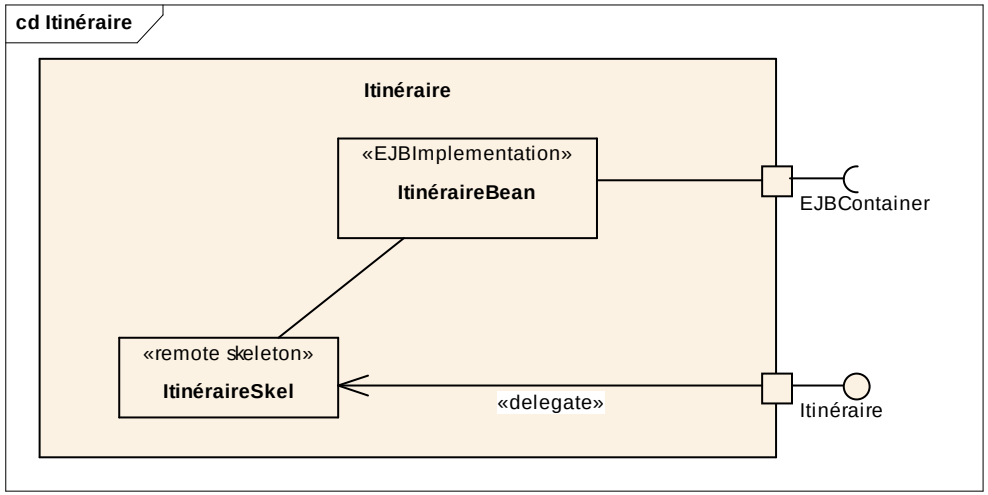
CompositeEntity

Notes

La représentation des EJB en exploitation

Diagramme de structure composite

- Présente la structure d'un objet complexe, tel qu'il se présente en phase d'exploitation



La représentation des EJB en exploitation

Notes