Chapitres du cours, par concepts

Chapitre	Séance
1. Le langage Java 1. Concepts objets, de base, Eclipse	Séance 1
2. POO, concepts avancés	Séance 2, TP1
2. Rappels 1. Les applications réparties 2. Architectures réparties 3. Client-Serveur et Java	Séance 3, TP2
3. Objets répartis : CORBA	Séance 4, Projet
4. Annuaires pour applications réparties **1. JNDI** **2. LDAP**	**Séance 5, TP3**
5. JMS	Séance 6, TP4
6. Services Web	Séance 7, **Projet**

Annuaires pour applications réparties

Chap #4

Introduction

L'objectif de ce chapitre de cours est de montrer comment une architecture répartie va se déployer autour d'un annuaire centralisant les références d'objets réseau.

En effet, la gestion et l'organisation de ces références est vitale pour le déploiement et la supervision des applications réparties.

Nous avons rapidement abordés les annuaires de type JNDI dans le cadre de notre étude de la technologie CORBA.

➡ Dans ce chapitre nous aborderons des notions d'ordre général sur les annuaires pour architectures réparties, puis nous étudierons plus spécifiquement les technologies **JNDI** et **LDAP** d'annuaires d'entreprise.

Notion de références

Gestion en réseau

Pour assurer l'appel d'objets distants via le réseau, il est nécessaire de découvrir où se trouvent ces objets, comment y accéder et comment gérer leur références distantes.

Ex des pages blanches : pour trouver le numéro de téléphone de quelqu'un dans l'annuaire, il faut déjà connaître son nom !

En POO, les objets ne sont pas désignés par leur nom, mais par **une référence.** Comme nous l'avons vu dans le chapitre de rattrapage sur Java, c'est l'équivalent d'un pointeur vers une zone mémoire.

- Pb : cet emplacement mémoire est propre à une JVM sur une machine donnée.
- De ce fait, ce type de référence mémoire ne permet pas d'accéder à un objet distant au sein d'une application répartie.

Notion de références
Gestion en réseau

Les services d'annuaire permettent de résoudre ce problème de référencement local en associant des noms valides et uniques dans l'annuaire à des **références distantes** vers ces objets répartis.

On parle alors d'un fonctionnement en mode 'pages blanches' des annuaires par rapport à un fonctionnement en mode 'pages jaunes' où l'on va rechercher un objet non plus en fonction de son nom, mais selon différents critères de sélection (catégorie, ...) :

- **Mode pages blanches** ("Naming service") : recherche par nom. On parlera par anglicisme de **Service de Nommage** au lieu de Service d'Annuaire.

- **Mode pages jaunes** ("Directory service") : recherche par critères de sélection.

Notion de références
Références distantes

Les références distantes sont ordinairement gérées par le bus logiciel : il doit s'assurer que la même référence pointe toujours sur le même objet physique.

Cette gestion se heurte à différentes problématiques au niveau du cycle de vie et du fonctionnement du Garbage-Collector (GC) :

- En effet, pour un serveur réel, le cycle de vie d'un objet peut être très long, comment s'assurer qu'un GC n'est pas passé par là ou que le serveur distant n'est pas éteint ?
- En RMI il est possible de gérer les références distantes avec le Distributed Garbage Collector (DGC).
- Avec CORBA cela est impossible car on ne peut faire aucune hypothèse sur les différents interlocuteurs : ils peuvent être de langages différents et gérer automatiquement ou non la mémoire.

Notion de références

Points d’entrée dans le système

Problématique : comment trouver le point d’entrée d’un système réparti qui va permettre d'accéder de proche-en-proche à l’ensemble de l’application ?

- **Avec des sockets :**

  Le point d’entrée est composé d’une référence vers la machine (nom DNS, adresse IP) ainsi que du port de communication. Mais ce point d’entrée ne permet d'accéder qu’à un seul serveur.
  Ex. `192.168.1.1:8080`

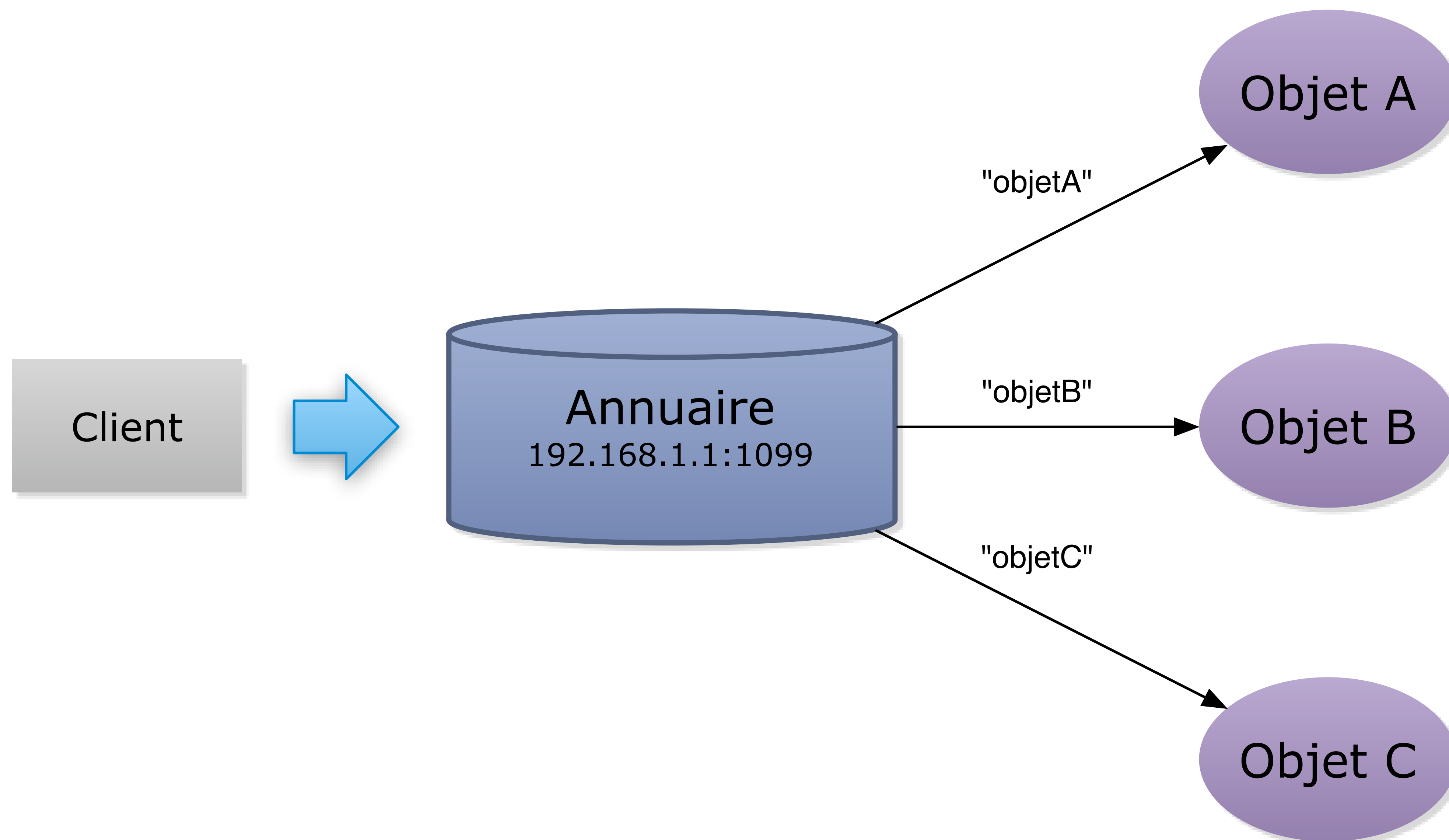
- **Avec un annuaire :**

  Les services de nommage vont permettre d’abriter de nombreux objets derrière le port de communication exposé de l’annuaire.
  On communique donc toujours en premier avec le service de nommage au travers d’une socket, avant de communiquer avec les objets en utilisant un protocole de plus haut niveau.

Notion de références

Points d'entrée dans le système

Un service de nommage contient les références de plusieurs objets distants :



Enregistrement d'un objet
Nommage

Afin d'être accessible au travers d'un service de nommage, il faut enregistrer les objets dans celui-ci.

On appellera **serveur** ou **servant** le programme qui effectue cette opération et met l'objet à disposition.

Un service de nommage doit s'assurer, au moment de l'enregistrement, **que le nom choisi par le serveur est unique** et qu'il ne pointe que vers une seule référence.

- Pour des applications réparties simples, il est facile de répertorier tous les objets à disposition et de choisir leurs noms.
- Pour des applications réparties complexes de grande taille, il faut envisager un **plan de nommage**.

Enregistrement d’un objet
Nommage

En CORBA, il est possible de hiérarchiser les noms dans des contextes, tout comme on peut hiérarchiser des fichiers dans des répertoires.
Ex. `/MonApplication1/MonObjet1`

Cette notation hiérarchique est aussi utilisée pour les URL sur Internet :
Ex. http://java.sun.com qui reflète une hiérarchie implicite du sous-domaine `java` dans le domaine `sun` de la zone `com`.

Ces URL sont utilisées naturellement pour designer des Services Web (voir chapitre du cours correspondant), mais CORBA a récemment introduit la notion d’URI (Universal Resource Identifier).

Contrairement à l’annuaire CORBA, l’annuaire `rmiregistry` de RMI n’est pas hiérarchique, ce qui peut mener à des collisions de noms.

Enregistrement d’un objet

Mise à disposition d’un objet

Que ce soit en RMI ou CORBA, le serveur doit enregistrer l’objet sous un nom unique dans un service de nommage :

- En RMI l’application servante va alors être bloquée en attente de requêtes. Il s’agit de la méthode classique qui permet de conserver l’objet dans un état actif. Il existe depuis le JDK 1.4 une méthode utilisant un démon pour activer l’objet à la demande, sans laisser le programme bloqué en attente.

- En CORBA, le processus d’activation est plus élaboré et s’appuie sur le POA (“Portable Object Adapter”, cf. chapitre du cours sur CORBA).

Enregistrement d’un objet
Performances

Le surcoût d’un appel de méthode sur un objet distant par rapport à l’appel en local va comporter deux facteurs :

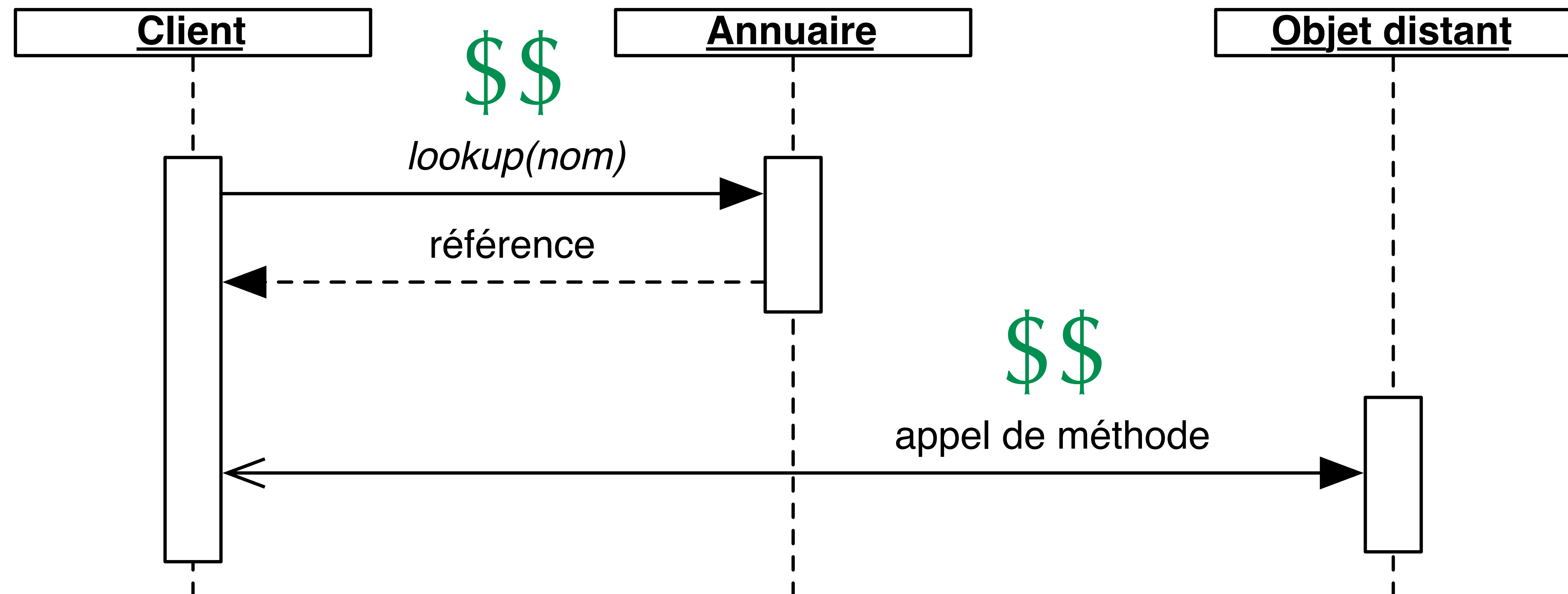
1. L’accès au service d’annuaire, de manière à obtenir une référence sur un objet distant.

2. La communication avec l’objet distant à proprement parler. Le client n’a alors plus besoin du service de nommage pour communiquer. C’est en ce sens que l’on parle de courtier (“broker”) en CORBA : le bus logiciel intervient dans la phase de négociation entre les objets, mais pas dans la transaction qui s’ensuit (cf. figure transparent suivant).

Ordinairement, la phase la plus longue est l’accès au service de nommage. Il est donc envisageable d’utiliser des techniques de mise en cache de la référence coté client de manière à optimiser cette étape. Les appels ultérieurs à cet objet éviterons ainsi la recherche coûteuse de la référence distante.

Enregistrement d'un objet
Performances

La performance va ensuite dépendre du transfert des arguments de la méthode appelée sur l'objet distant ainsi que sa valeur de retour.



Enregistrement d’un objet

Modification de référence

Par rapport à ses client, le rôle du service de nommage est de toujours présenter sous les mêmes noms des objets aux fonctionnalités similaires.

En effet, il est important de comprendre qu’un objet donné peut être remplacé par un autre dans le service de nommage :

- Si sa localisation change (application est migrée sur un autre serveur)
- Si une nouvelle implémentation de l’objet est injectée dans l’application. répartie
- ...

Le serveur doit donc dans ce cas enregistrer dans l’annuaire un nouvel objet derrière un nom et une interface communément admise par les clients.

La mise à disposition d’un objet par un serveur est un contrat qu’il passe avec les clients : dans les coulisses, le serveur peut faire ce qu’il veut, en particulier jouer sur l’activation de l’objet.

Enregistrement d’un objet
(De)activation des objets

Dans les EJB, ce mécanisme d’activation permet de décharger de la mémoire les objets non-utilisés vers le disque, puis de recharger leurs informations à la demande.

C’est le bus logiciel qui est responsable de réactiver l’objet à la demande et de s’assurer que la référence est proprement restaurée : tout cela doit être transparent pour le client.

Nb : Si plusieurs objets sont interchangeables et que leur état n’est pas modifié par les appels clients, il est possible de les grouper afin de répartir la charge. Il est possible de faire ceci simplement avec des pointeurs intelligents (“smart pointers”), qui peuvent être implémentés avec des proxies dynamiques.

Conception d’un service d’annuaire

Quels objets y stocker ?

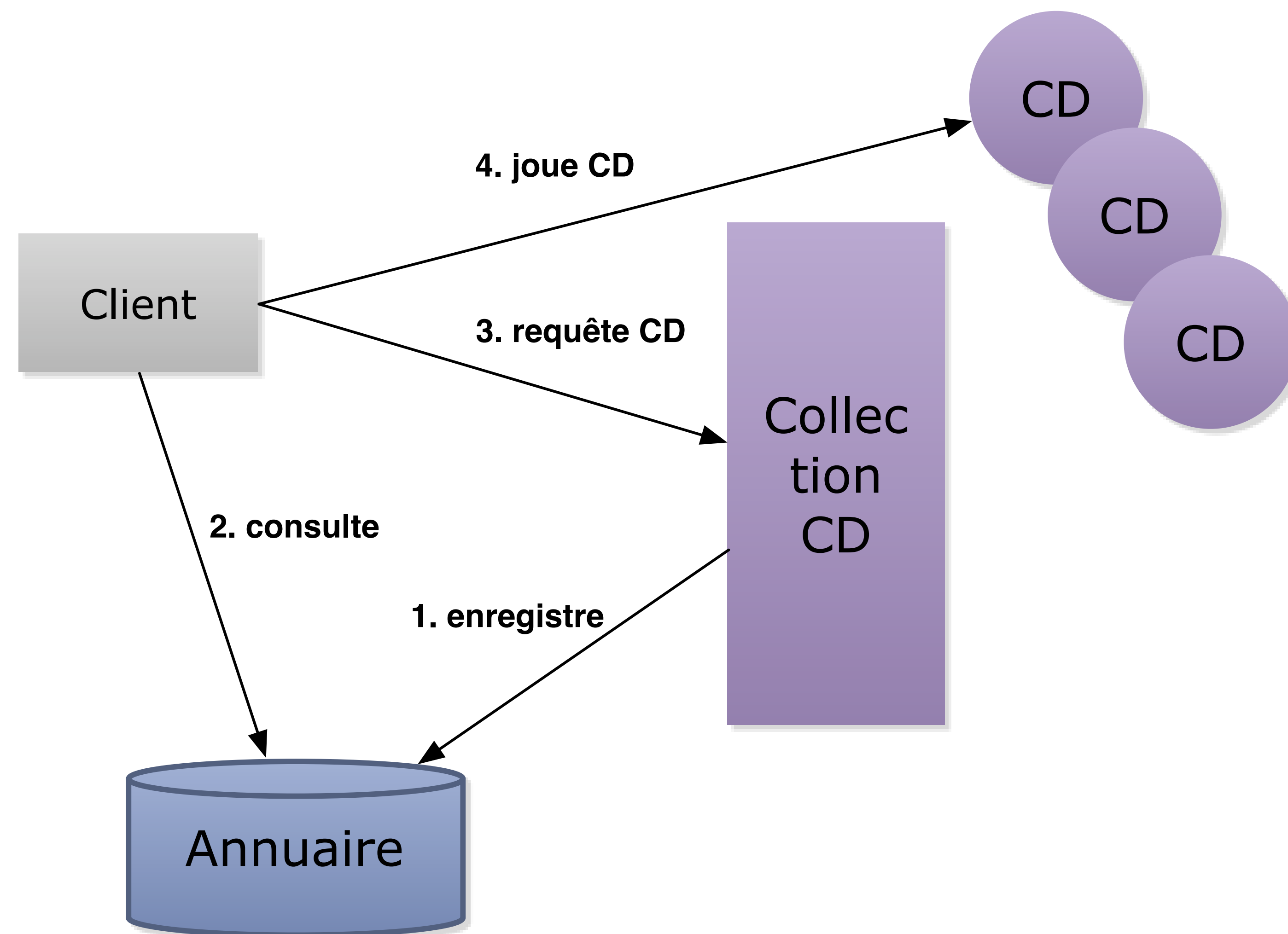
Ordinairement, le service d’annuaire assure une bonne montée en charge, mais il n’est pas toujours conseillé d’y mettre tous les objets d’une application répartie. Par exemple, dans les cas suivants :

- Il y a une large collection d’objets de même nature et il est fastidieux de les nommer un par un.
- Certains objets sont fortement dynamiques, avec un cycle de vie court, et il est difficile de tracer exactement leur apparition et disparition afin d’éviter les références mortes dans l’annuaire.
- Les objets doivent être passés **par valeur** (et non par référence distance) et il est plus performant de les faire directement transiter par le réseau plutôt que d’implémenter *n* méthodes d’accès à tous leurs champs.

Conception d’un service d’annuaire

Quels objets y stocker ?

Prenons l’exemple d’une bibliothèque de CD, au lieu d’enregistrer chaque CD dans le service de nommage, on préférera donner accès à la collection de tous les CD et implémenter une méthode sur celle-ci permettant d’obtenir des CD en fonction de leur titre, par valeur ou référence.



Conception d’un service d’annuaire

Quels objets y stocker ?

Lorsque l’on veut pouvoir créer et détruire des objets à distance, il est nécessaire de passer par un intermédiaire, la **fabrique à objets**.

En effet, **les classes Java ne sont pas traitées comme des objets**, et il n’est donc pas possible d’enregistrer une classe “distante” dans un service d’annuaire et d’appeler la méthode new sur cette dernière.

Pour que les classes soient traitées comme des objets, Java devrait supporter la notion de métaclasse (des classes de classes)...

Pour ce faire, on utilisera donc un autre objet chargé de créer les objets locaux en appelant la classe en local. On utilise donc le pattern de “Factory”, et la fabrique ainsi créée permettra d’instancier cette classe.

Nb: ce pattern est à rapprocher des interfaces EJBHome, qui permettent la création d’EJBs à distance.

Conception d’un service d’annuaire
Parallélisation d’annuaires

Le registre RMI (rmiregistry) est lancé par défaut sur le port 1099. Mais il possible de modifier ce port par exemple pour faire tourner deux registres en parallèle.

Pour quelle utilité ? Par exemple pour le partitionnement d’applications à des fins de sécurité : certains objets ne seront ainsi accessibles qu’à certaines applications.

Nb : Il est possible de sécuriser l’accès à ce registre à l’aide de sockets SSL (Secure Socket Layer). Bien que nous ne détaillerons pas ici le processus de création des clés de chiffrement, voici un exemple de code à utiliser dans ce cas :

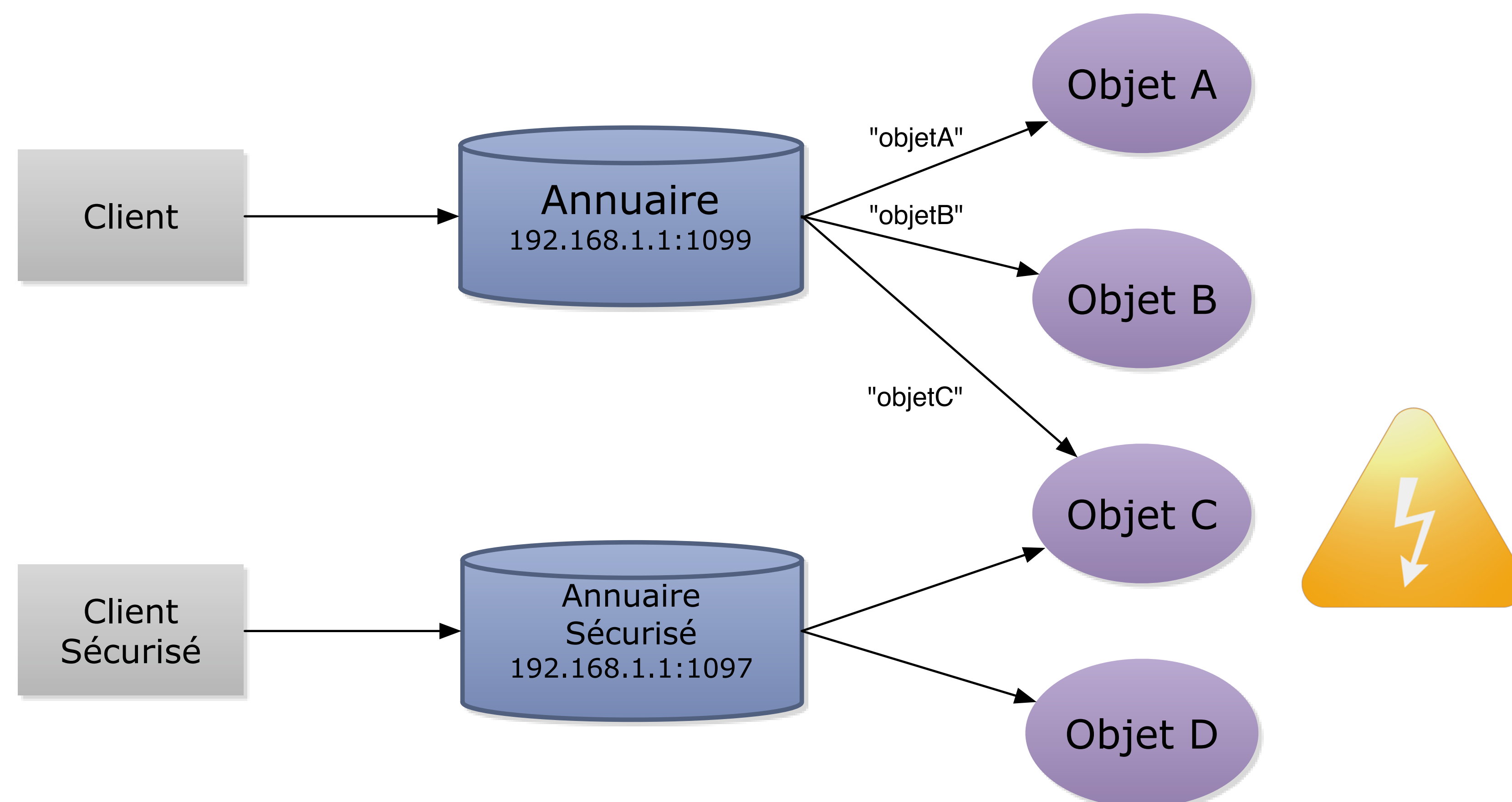
```
Registry registry = LocateRegistry.createRegistry(PORT,
   new SslRMIClientSocketFactory(),
   new SslRMIServerSocketFactory());
```

Conception d’un service d’annuaire

Parallélisation d’annuaires

Nous voyons sur la figure suivante que **plusieurs annuaires peuvent avoir des droits d’accès différents aux mêmes objets**. Il est donc nécessaire de faire une bonne partition entre les objets à sécuriser et les objets normaux.

Bien que ceci semble trivial sur la figure, il serait particulièrement ardu de démontrer, par une analyse de code, qu’un objet sécurisé ne sera jamais passé en valeur de retour d’une méthode d’un objet non-sécurisé.

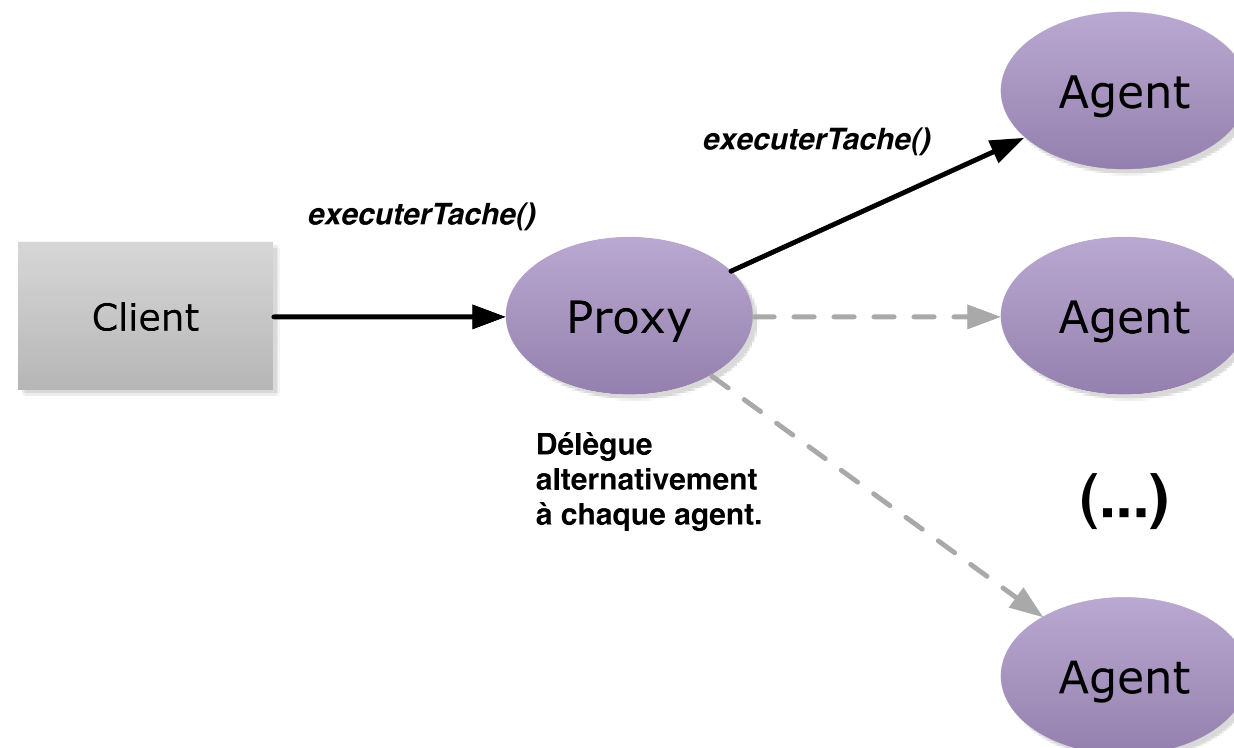


Conception d’un service d’annuaire

Pointeurs intelligents

Il est possible d’utiliser des pointeurs “intelligents” pour référencer un ensemble d’objets :

- Il est très facile de représenter un ensemble d’objets de même nature avec un objet **en façade**.
- Par exemple, si l’on souhaite représenter un centre d’appel où tout les agents ont le même rôle. On peut alors utiliser un pattern de “Proxy” :



Conception d’un service d’annuaire
Pointeurs intelligents

Le code pour le centre d’appels (le Proxy) ressemblerait à ceci :

```
public String executerTache(String action) {
   Agent candidat = selectionCandidat(action);
   return (String) candidat.executerTache(action);
}
```

Ceci est simple tant que nous n’avons qu’une seule méthode executerTache(), mais peut devenir fastidieux s’il faut écrire le même code pour toutes les méthodes. Le design pattern de **“proxy dynamique”** peut alors s’avérer être une meilleure solution.

➡ Voir transparents suivants.

Conception d’un service d’annuaire

Proxy dynamique

L’inconvénient des proxies est qu’ils sont intimement liés à l’objet qu’ils représentent et doivent notamment implémenter **la totalité** des méthodes de l’objet.

Depuis Java 1.3, Sun à introduit les proxies dynamiques qui peuvent implémenter plusieurs interfaces, à condition qu’elles n’aient pas de conflit de noms de méthodes. Il existe d’autre restrictions :

- Les objets qu’ils représentent doivent être l’implémentation d’interfaces.
- Les interfaces doivent être dans le même package et visibles du même “Class Loader” (un chargeur de classes Java dans le runtime) que celle des objets.

Dans l’exemple suivant, on souhaite journaliser toutes les actions d’une voiture. On va donc utiliser un proxy dynamique pour ajouter des écritures à chaque action de la voiture.

Conception d’un service d’annuaire

Proxy dynamique

La voiture implémente l’interface simple suivante :

```
public interface InterfaceVoiture {
   public void demarre();
   public void arrete();
}
```

Et voici la classe Voiture associée à cette interface :

```
public class Voiture implements InterfaceVoiture {
   public void demarre() { ... };
   public void arrete() { ... };
}
```

Il faut alors écrire une classe qui implémente java.lang.reflect.InvocationHandler et qui représente le corps du proxy avec délégation des appels vers le sujet.

Conception d’un service d’annuaire

Proxy dynamique

Cette classe possède un pointeur vers la voiture, le sujet du pattern de proxy. Elle se contente de rajouter une trace de l’appel de toutes les méthodes de la voiture :

```
public class JournalInvocationHandler implements InvocationHandler {
   Voiture sujet;

   //methode proxy pour toutes les methodes de ‘sujet’
   public Object invoke(Object objet, Method methode, Object[] arguments)
   throws Throwable {
      System.out.printlnt(“Invocation de :” + “methode.getName());
      return methode.invoke(sujet, arguments);
   }

   //constructeur
   public JournalInvocationHandler(Voiture v) {
      sujet = v;
   }
```

Conception d’un service d’annuaire

Proxy dynamique

Nous pouvons à présent tester le programme.

On crée une voiture et un InvocationHandler. Il est également nécessaire d’associer le proxy à un ClassLoader, qu’on prendra ici identique à celui de la voiture. On doit également spécifier la liste des interfaces implémentées par le proxy dynamique, ici seulement l’interface de voiture.

Lorsque ce programme sera exécuté, on observera la trace des 2 opérations effectuées sur la voiture :

```
Voiture voiture = new Voiture();

ClassLoader cl = voiture.getClass().getClassLoader();
Class[] listeClasses = new Class[]{InterfaceVoiture.class};
JournalInvocationHandler journal = new JournalInvocationHandler(voiture);

InterfaceVoiture proxy = (InterfaceVoiture) Proxy.newProxyInstance(cl,
listeClasses, journal);

proxy.demarre();
proxy.arrete();
```

Conception d’un service d’annuaire
Pointeurs intelligents

Retour sur le proxy dynamique d’un centre d’appel.

L’implémentation du InvocationHandler associé au proxy peut utiliser le code suivant. L’implémentation de la méthode invoke permet de centraliser la redirection plutôt que d’avoir à redéfinir chaque méthode de l’objet cible.

```
public class GroupInvocationHandler implements InvocationHandler {
   private int numero = 0;
   private AgentInterface[] agents;

   //constructeur
   public GroupInvocationHandler(AgentInterface[] aGroup) {
      super();
      agents = aGroup;
   }

   public Object invoke(Object proxy, Method methode, Object[] args) throws Throwable {
      AgentInterface agent = agent[numero++];
      return method.invoke(agent, args);
   }
}
```

JNDI

Chap #4.1

JNDI
Service de nommage et d’annuaire

Nous avons vu qu’un **service de nommage** permet de faire correspondre un nom à une référence sur un objet.

- Ce service de noms va le plus souvent être hiérarchique, afin de permettre une recherche plus rapide et une isolation des différents contextes.

Les annuaires sont typiquement optimisés pour la lecture des données, contrairement aux bases de données qui optimisent à la fois la lecture et l’écriture.

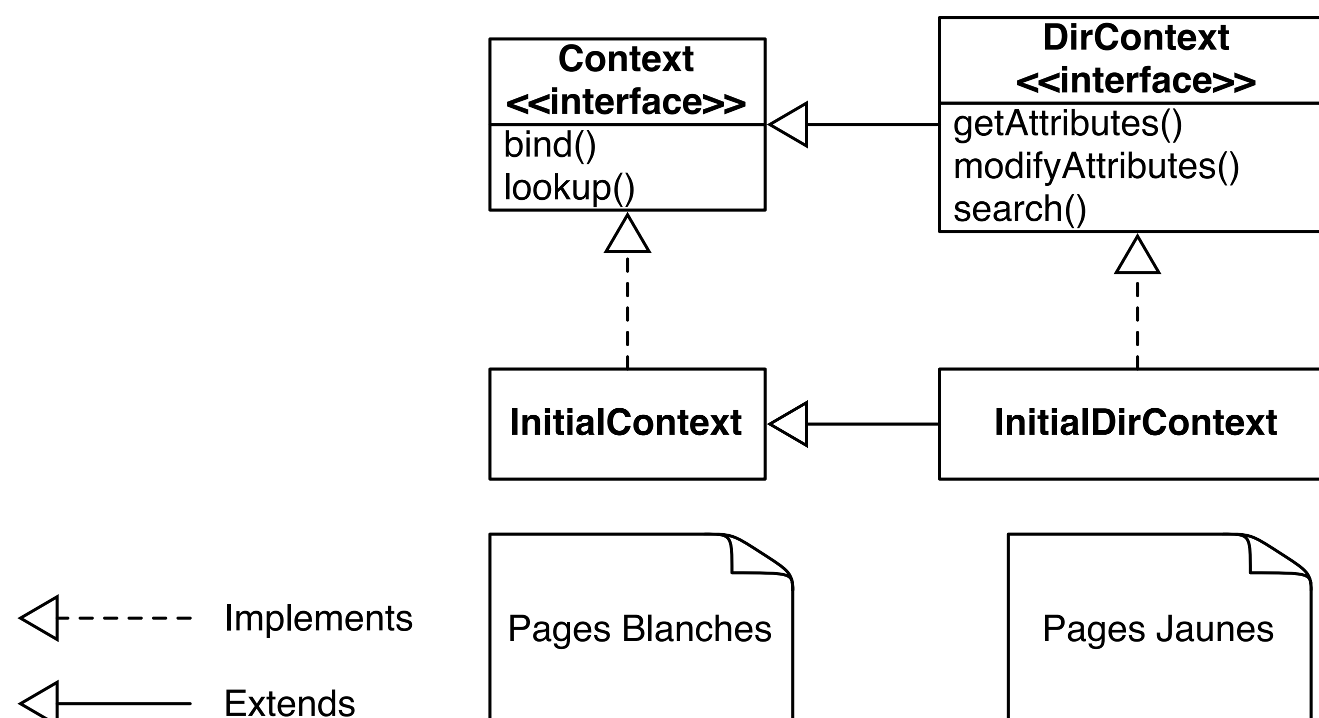
Un **service d’annuaire** (Directory Service) va enrichir un service de nommage en rajoutant des attributs sur les références, afin de pouvoir effectuer des recherches non plus en mode page blanches, mais **en mode pages jaunes**.

JNDI
Service de nommage et d’annuaire

L’interface JNDI (Java Naming and Directory Interface) va refléter cette double fonctionnalité en proposant au développeur essentiellement deux packages centraux :

- javax.naming.* pour le service de nommage.
- javax.naming.directory.* pour le service d’annuaire.

Les 2 interfaces pivots seront Context et DirContext, l’une pour le service de noms, l’autre pour le service d’annuaire comme le montre la figure suivante.



JNDI
Implémentations

JNDI fournit une API uniforme pour accéder à des ressources diverses dans un système (cf. figure suivante) :

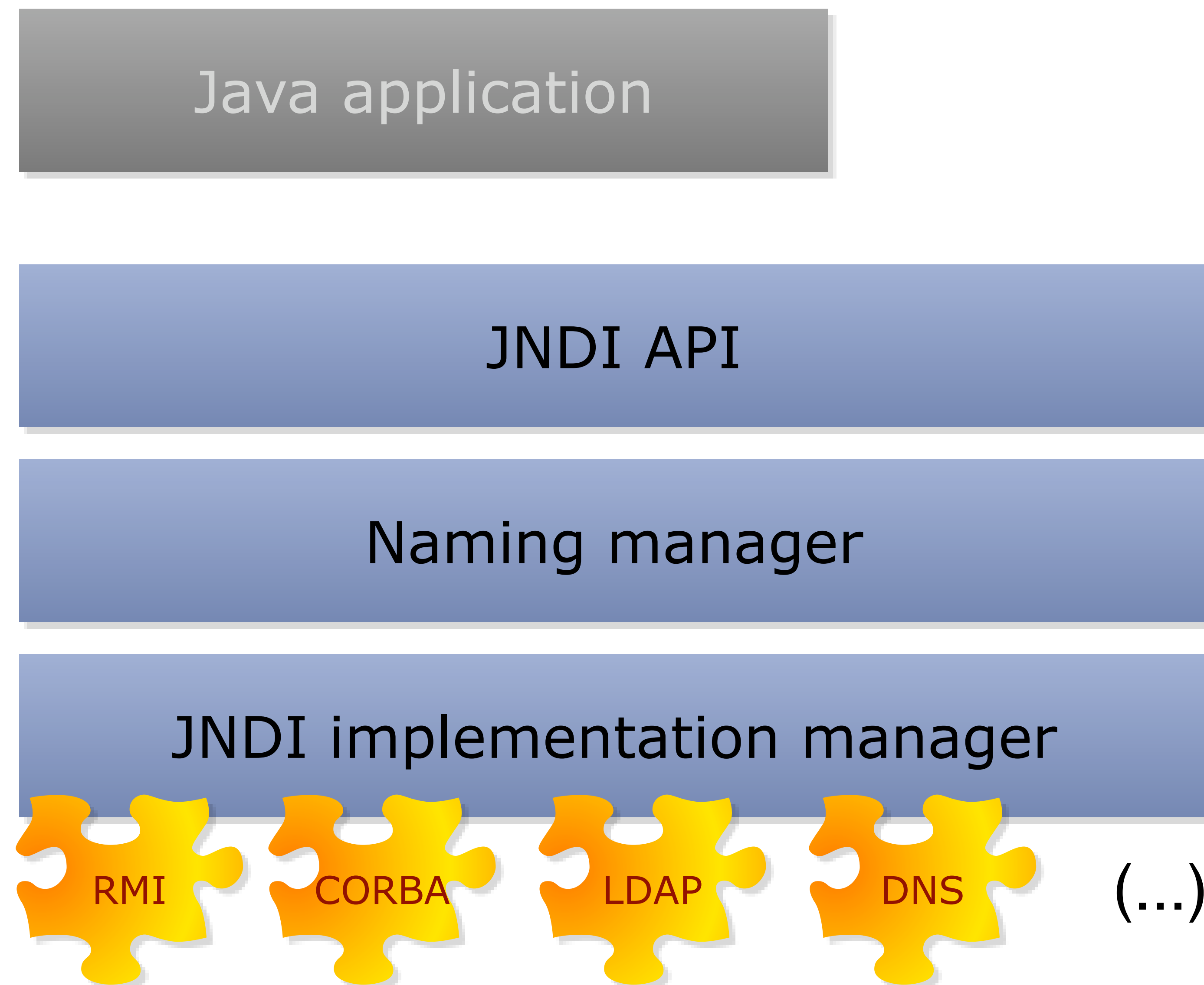
- Service de nommage *CORBA*, le registre *RMI rmiregistry*.
- Mais aussi bien d'autres annuaires tels que les annuaires *LDAP* (Light Directory Access Protocol), *Active Directory* de Microsoft, le serveur *DNS* (Domain Name Server) des noms de machines en réseau, etc ...

JNDI existe en standard dans la distribution Java (Java SE), mais des implémentations font parfois appel à des tierces parties.

- Il existe par exemple des implémentations pour s'interfacer en DXML (représentation d'annuaire sous forme XML).

JNDI
Implémentations

Une seule API pour de multiples implémentations :

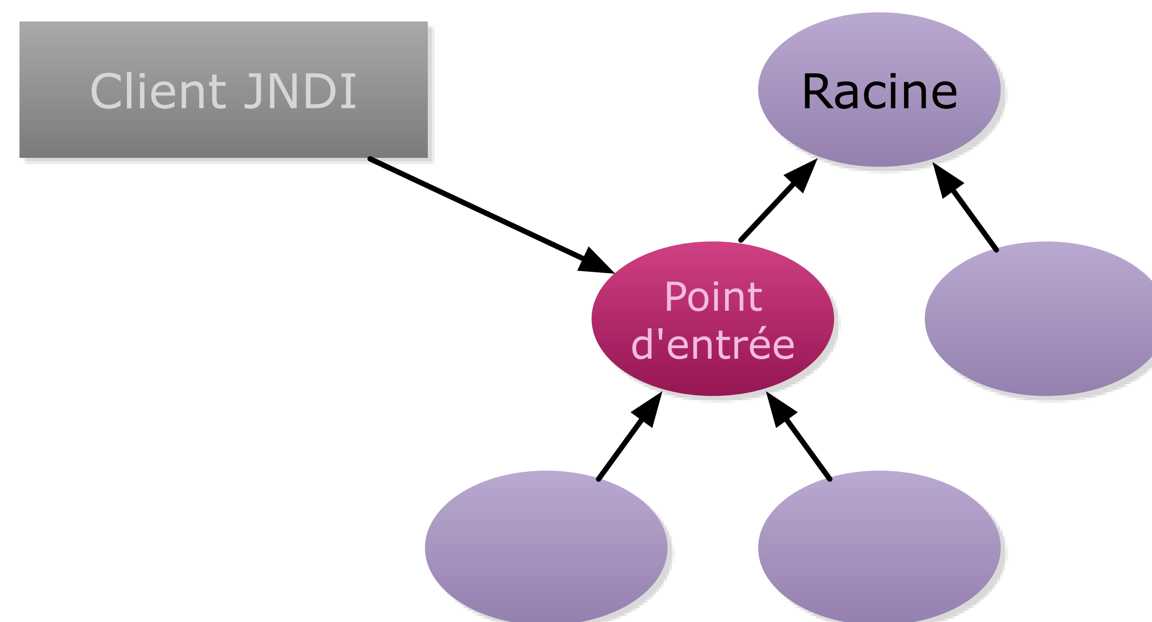


JNDI
Paramètres d’implémentation

Une implémentation de JNDI peut être identifiée de manière unique par deux paramètres :

- Une URL pour le point d’entrée dans la hiérarchie de l’annuaire (**Context.PROVIDER_URL**).
- Le nom de la fabrique de contextes (**Context.INITIAL_CONTEXT_FACTORY**), qui indique l’implémentation de JNDI utilisée.

Le point d’entrée dans l’annuaire peut être à un endroit quelconque de la hiérarchie, comme le montre la figure suivante :



JNDI
Paramètres d'implémentation

Pour ce qui concerne les différents types de fabriques, le tableau suivant liste un certain nombre d'implémentations usuelles de JNDI avec le nom des fabriques de contexte associées :

DNS	com.sun.jndi.dns.DnsContextFactory
RMI	com.sun.jndi.rmi.registry.RegistryContextFactory
LDAP	com.sun.jndi.ldap.LdapCtxFactory
CORBA	com.sun.jndi.cosnaming.CNCtcFactory
File System	com.sun.jndi.fscontext.RefFSContextFactory

Très souvent, les serveurs d'application implémentent également leur propre annuaire sous forme de fichier ou de mini base de données. Pour travailler avec de tels annuaires, il faut connaître le contexte initial ainsi que la classe d'implémentation.

Utilisation de JNDI

Nommage : lier et consulter les objets

Les opérations de nommage utilisent des classes Context qui disposent des opérations :

- rebind, qui va écraser la référence si elle existe déjà
- bind qui lèvera au contraire une exception dans ce cas.
- unbind permet de défaire une association.
- lookup, list, pour consulter des objets et rechercher la référence associée.

Il est également possible de créer des sous-contextes ainsi :

```
Context sousContexte = ctx.createSubContext(“departement”);
```

Utilisation de JNDI

Nommage : Ex. d’implémentation RMI

On peut utiliser JNDI pour accéder à un annuaire RMI. Le contexte va être créé avec les paramètres décrits précédemment.

On suppose que rmiregistry est lancé :

```
LocateRegistry.createRegistry(1099);
```

On peut alors y accéder au travers de l’interface JNDI en utilisation la fabrique RegitryContextFactory :

```
Hashtable<String, String> env = new Hashtable();
env.put(Context.INITIAL_CONTEXT_FACTORY,
“com.sun.jndi.rmi.registry.RegistryContextFactory”);
env.put(Context.PROVIDER_URL, “rmi://localhost:1099”);

Context ctx = new InitialContext(env);
Modele modele = new Modele();
ctx.bind(“Modele”, modele);
```

Utilisation de JNDI
Nommage : Ex. d’implémentation RMI

On peut également utiliser des **pseudos-URL.**

Ces pseudos-URL vont résument les deux paramètres essentiels :

- Le protocole indique directement le type de fabrique à utiliser.
- Le reste de l’URL le contexte initial.

Un client RMI pourra ainsi rechercher l’objet modèle par le code suivant :

```
Context contexte = new InitialContext();
ModeleInterface modele = (ModeleInterface) contexte.lookup(“rmi://localhost:1099/
Modele”);
```

Utilisation de JNDI

Nommage : Ex. d'implémentation CORBA

Nous avons vu dans un chapitre précédent du cours comment stocker un objet RMI IIOP dans un service de nommage CORBA. On lance le démon orbd sur le port 1050 par exemple :

```
orbd -ORBInitialPort 1050
```

Puis on écrit le code du serveur :

```
Hashtable<String, String> env = new Hashtable();
env.put(Context.INITIAL_CONTEXT_FACTORY, “com.sun.jndi.cosnaming.CNCtxFactory”);
env.put(Context.PROVIDER_URL, “iiop://localhost:1050”);

Context ctx = new InitialContext(env);
Modele modele = new Modele();
ctx.bind(“Modele”, modele);
```

Utilisation de JNDI

Nommage : Ex. d'implémentation CORBA

Ainsi, pour le client il est possible également d'utiliser une pseudo-URL qui utilise le protocole iiop au lieu de rmi.

Ceci serait aussi possible pour le serveur.

```
Context ctx = new InitialContext();
Object objref = ctx.lookup("iiop://localhost:1050/modele");
ModeleInterface modele = (ModeleInterface) PortableRemoteObjet.narrow(objref,
ModeleInterface.class);
```

Utilisation de JNDI

Nommage : lister les éléments

On peut lister les éléments de rmiregistry tout comme ce serait possible pour le service de nommage CORBA :

```
NamingEnumeration liste = ctx.list("rmi://localhost:1099);

while(liste.hasMore()) {
   System.out.println(liste.next());
}
```

Utilisation de JNDI

Annuaire : l'interface DirContext

Les opérations d'annuaire utilisent l'interface DirContext qui hérite de l'interface Context.

Les opérations d'annuaire permettent de travailler sur les attributs. Ce sont des objets de la classe Attribute ayant un nom et une ou plusieurs valeurs.

Un contexte d'annuaire va donc exposer la liste de ses attributs et va permettre d'effectuer des recherches sur une valeur d'attribut grace aux méthodes suivantes :

- `getAttributes()`
- search(...)

Utilisation de JNDI

Annuaire : exemple d’accès au DNS

Le DNS (Domain Name Server) est présent dans la grande majorité des réseaux d’entreprise ou chez votre fournisseur d’accès internet.

L’exemple suivant va permettre de donner la liste des machines présentes sur le réseau, ainsi que le nom de leur serveur de mail, un attribut de nom “MX”.

```
DirContext contexte = new InitialDirContext();
Attributes attrs = contexte.getAttributes(“dns://80.10.246.1/wanadoo.fr”, new
String[] { “MX” });
```

On remarquera l’utilisation d’une pseudo-URL, ainsi que le filtrage de la liste des attributs renvoyés.

Le resultat est une liste d’attributs de type Attributes qu’il faut parcourir avec une NamingEnumeration.

```
for(NamingEnumeration enum1 = attrs.getAll(); enum1.hasMore();) {
   System.out.println(enum1.next());
}
```

LDAP

Chap #4.2

LDAP
Annuaire d'entreprise

LDAP (Light Directory Access Protocol) est une implémentation d'annuaire d'entreprise très répandue. Elle s'appuie sur des schémas de données qui définissent la liste des attributs attendus pour un enregistrement de l'annuaire.

LDAP est un protocole initialement développé par Novell pour permettre la centralisation d'annuaires d'entreprise, de manière à simplifier l'utilisation des annuaires X500.

Il existe notamment des schémas permettant de représenter les informations sur des utilisateurs tels que leur nom, login, mot de passe, mail et éventuellement leur photo, voire leur certificat.

Il est possible pour un même serveur de faire cohabiter plusieurs schémas pour décrire plusieurs types de ressources.

LDAP

Annuaire d’entreprise

Le schéma d’annuaire le plus connu s’appel cosine.schema et définit la classe inetorgperson (pour une personne d’une organisation sur Internet) qui permet de décrire un utilisateur avec son login et mot de passe, son numéro de téléphone, son mail ....

Il est possible d’ajouter des schémas de données tels que java.schema ou corba.schema pour stocker des objets Java ou des références à des objets Java sous forme sérialisée dans l’annuaire. Il est aussi possible de rajouter un schéma DNS pour coopérer avec un serveur d’adresses IP.

On peut trouver comment un serveur LDAP tel que OpenLDAP et l’équiper de Java dans les tutoriels Sun. Il existe également une implémentation Open Source de LDAP v3 en Java appelée Apache Directory Server (Apache DS).

LDAP

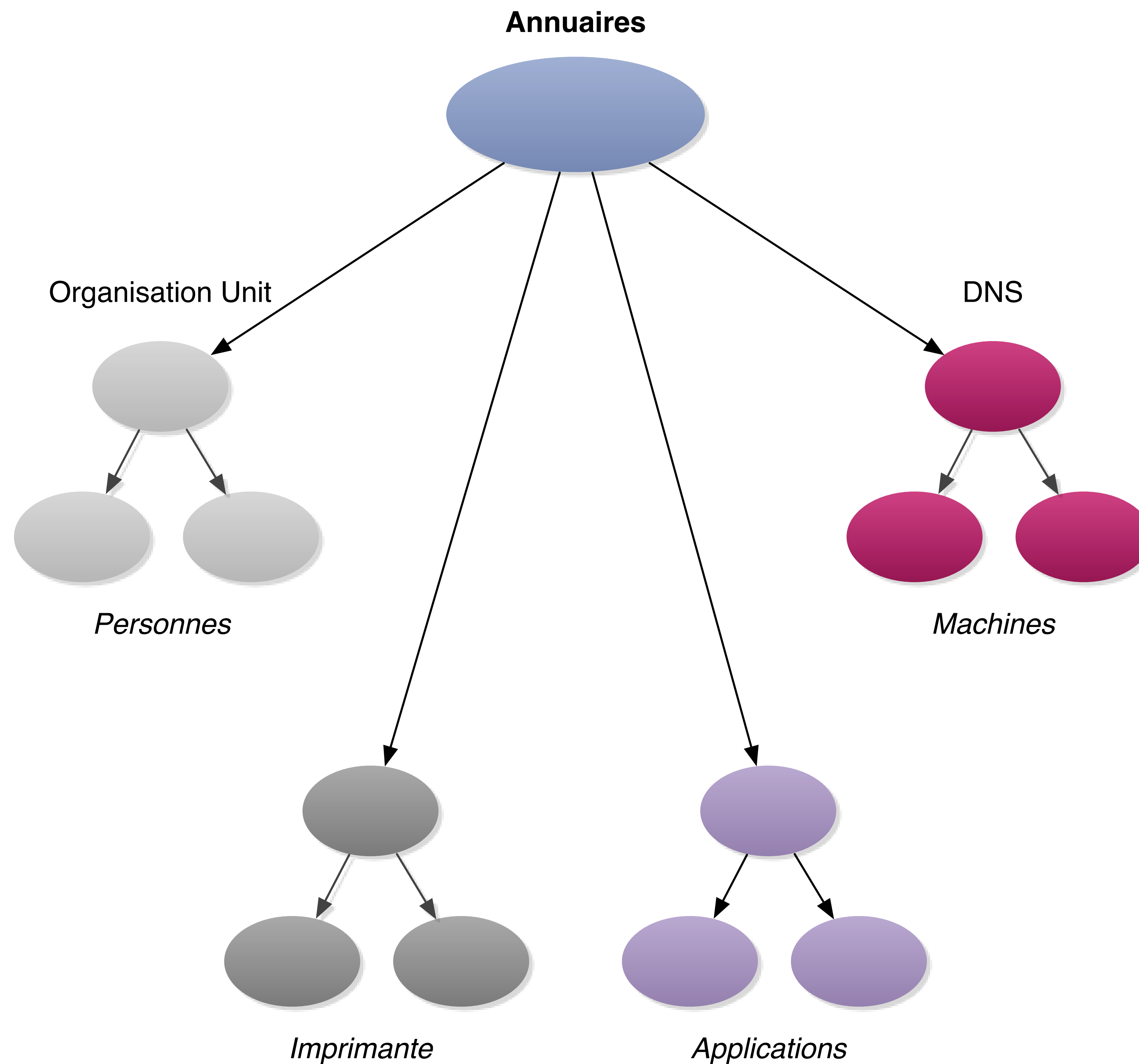
Annuaire d'entreprise

Un annuaire peut fédérer également des références de natures très multiples permettant ainsi la gestion de nombreux aspects de l'entreprise (cf. figure suivante) :

- Des annuaires de personnes, avec les mails et les numéros de téléphone, mais aussi les certificats d'identification, les mots de passe et les répertoires utilisateurs.
- Des annuaires de matériels, comportant les noms des machines comme le comporte typiquement un DNS, ou bien des imprimantes.
- Des annuaires d'applications, avec le stockage des références vers des serveurs distants (RMI, CORBA, serveurs web, etc...) ou encore des fabriques JMS (ConnectionFactory) ou des DataSource pour les accès aux bases de données.

LDAP
Annuaire d'entreprise

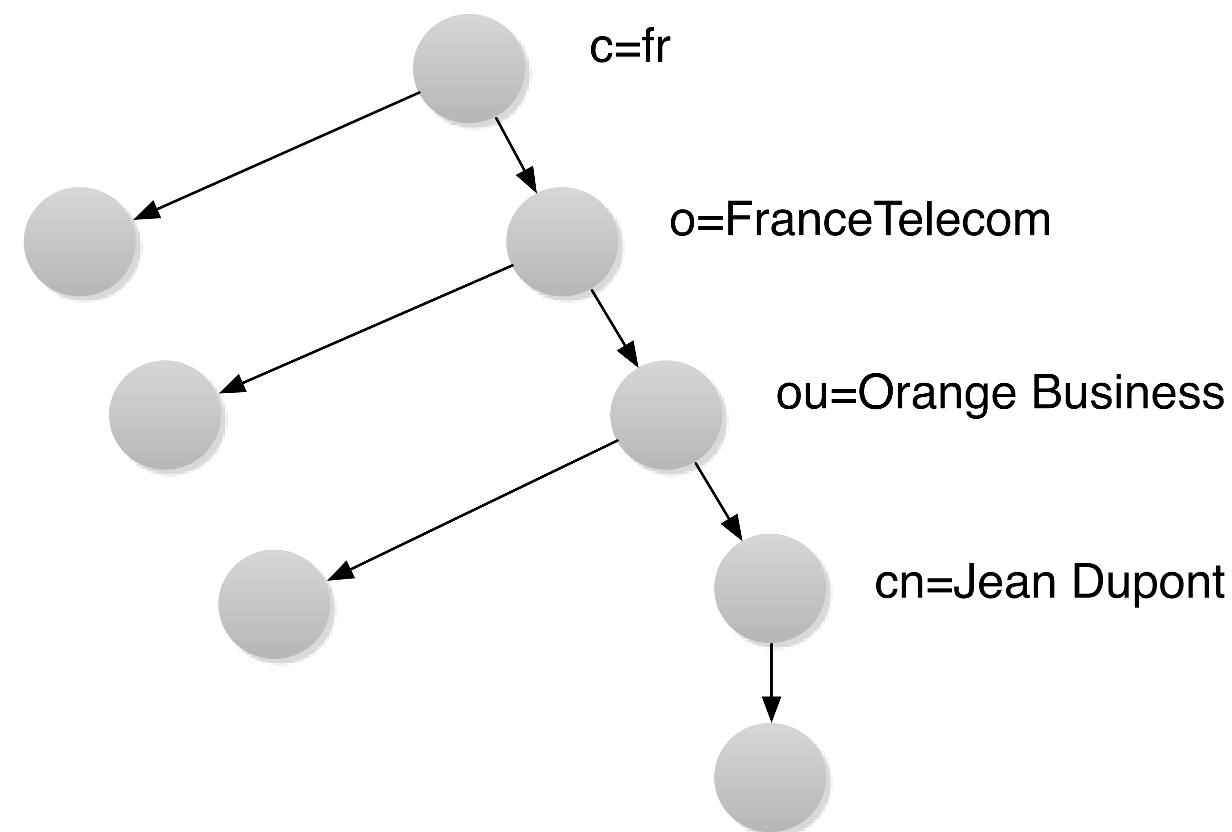
Vue globale de l'entreprise dans un annuaire :



LDAP

La hiérarchie des noms

LDAP utilise une convention hiérarchique de noms propres à l'organisation d'entreprise :



c (country) désigne le pays; **o** (organisation) désigne l'organisme; **ou** (organization unit) désigne un service; **cn** (common name) désigne une personne ou une feuille de l'arbre.

LDAP

La hiérarchie des noms

On appel **DN ("Distinguished Name")** la suite de noms désignant un contexte de manière unique.

- Ainsi le dn de France Telecom sera :
  o=FranceTelecom, c=fr
- Celui de Jean Dupont :
  cn=Jean Dupont, ou=Orange Business, o=FranceTelecom, c=fr

On désigne par **RDN ("Relative Distinguished Name")** le nom unique désignant une feuille dans un contexte particulier.

- Par exemple, dans le contexte Orange Business (ou=Orange Businesss, o=FranceTelecom, c=fr) le RDV de Jean Dupont sera
  cn=Jean Dupont
- Ainsi, pour insérer des objets dans l'annuaire, il faut utiliser le RDN par rapport au contexte d'origine.

LDAP
... comme fédérateur d'annuaires

LDAP va pouvoir servir de proxy pour plusieurs annuaires plus spécialisés, ou répartis sur le réseau.

LDAP peut ainsi servir de point d'entrée à des services de noms RMI ou CORBA répartis sur le réseau.

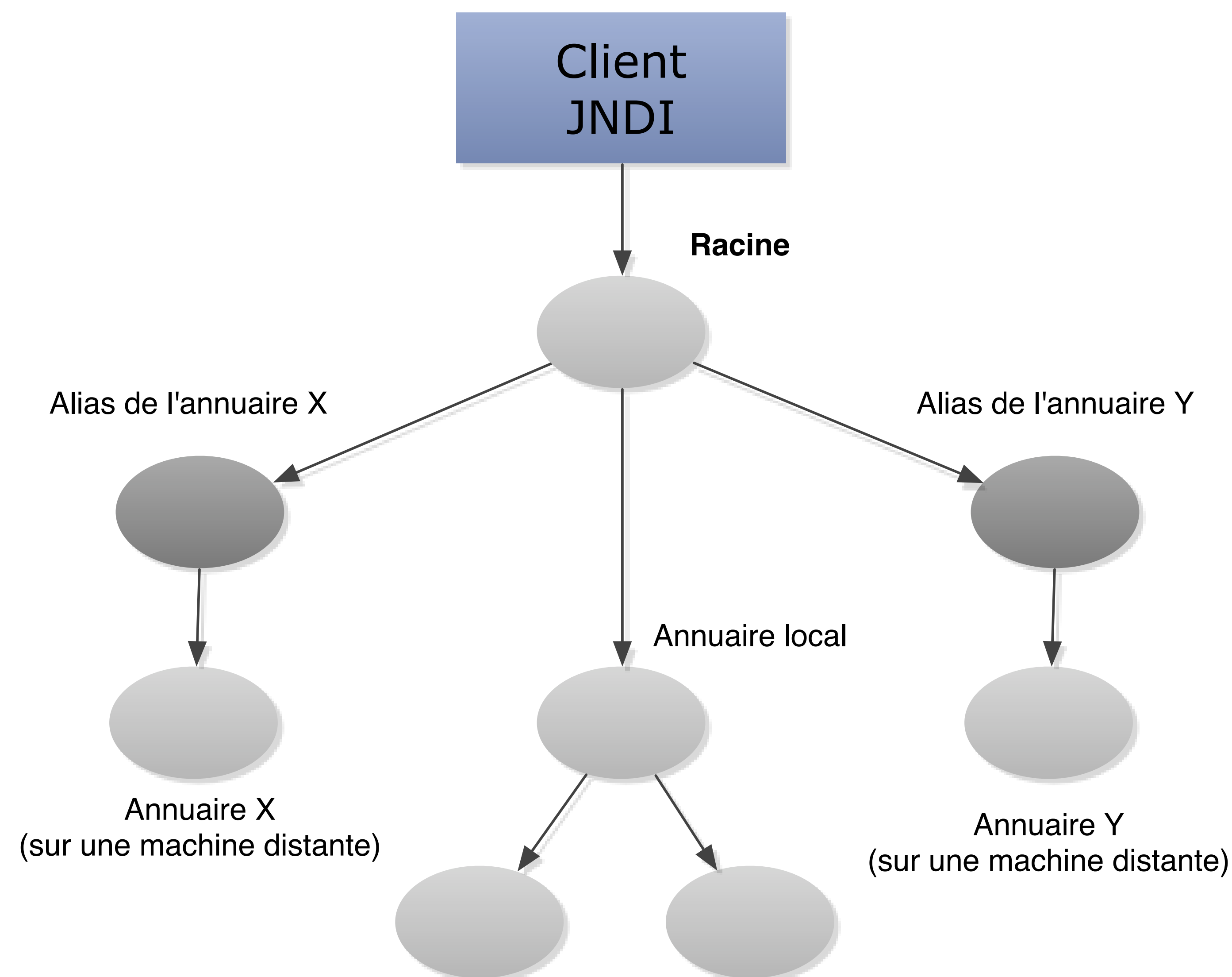
- En effet, rmiregistry doit être local aux objets serveur qu'il référence, et il est donc impossible de partager le même rmiregistry entre plusieurs objets serveurs sur des machines différentes.
- LDAP permet un point d'accès unique aux informations de configuration, ce qui va faciliter le déploiement d'une applications répartie.

Il est également possible de crypter l'accès à LDAP en utilisant SSL (Secure Socket Layer), notamment pour rechercher des informations d'authentification des utilisateurs.

LDAP

... comme fédérateur d'annuaires

LDAP permet enfin de relier plusieurs instances LDAP entre elles par des liens logiques (cf. ci-dessous), ce qui permet de fédérer des annuaires distants ("Referral") à l'aide de liens logiques ("Alias") comme sur les fichiers Linux, ce qui ne serait pas possible avec un modèle relationnel de base de données.



Utilisation de JNDI avec LDAP

Pour stocker un objet Java dans un annuaire LDAP, il suffit que celui-ci implémente :

- soit l’interface Referenceable,
- soit l’interface Serializable.

On va donc pouvoir stocker des souches d’objets distants CORBA ou RMI dans l’annuaire. **Toutefois, pour RMI, cela ne dispensera pas de mettre l’objet à disposition dans rmiregistry. Seule une référence sera stockée dans l’annuaire.**

Utilisation de JNDI avec LDAP

Stockage d’un objet sérialisé

Il faut en général un mot de passe pour accéder à l’annuaire LDAP. Une fois que l’on a obtenu le contexte avec son login, il suffit d’enregistrer l’objet sérialisé dans l’annuaire :

- On suppose que l’on a une classe Fleur sérialisable :
```
public class Fleur implements Serializable { ... }
```

- On doit connaître les informations d’accès à l’annuaire, et pour LDAP il est nécessaire de fournir un login et un mot de passe. Ici le mot de passe est “secret” et le RDN de l’utilisateur “cn=TutorialAdmin, o=JNDITutorial”.

```
Hashtable env = new Hashtable(11);
env.put(Context.INITIAL_CONTEXT_FACTORY, “com.sun.jndi.ldap.LdapCtxFactory”);
env.put(Context.PROVIDER_URL, “ldap://localhost:389M/o=JNDITutorial”);

//Identification de l’utilisateur avec mot de passe
env.put(Context.SECURITY_AUTHENTICATION, “simple”);
env.put(Context.SECURITY_PRINCIPAL, “cn=TutorialAdmin,o=JNDITutorial”);
env.put(Context.SECURITY_CREDENTIALS, “secret”);
DirContext ctx = new InitialDirContext(env);
```

Utilisation de JNDI avec LDAP

Stockage d’un objet sérialisé

On peut alors utiliser l’annuaire, tout comme on l’a vu avec RMI et CORBA et appeler les méthodes bind et lookup sur un objet sérialisé.

```
Fleur hibiscus = new Fleur();
hibiscus.setCouleur(“rouge”);
ctx.bind(“cn=maFleur”, hibiscus);
```

Plus tard, on récupère l’objet avec la même création de contexte :

```
Fleur fleurRetrouvee = ctx.lookup(“cn=maFleur”);
```

Ou en utilisant une pseudo-URL, car en lecture il n’est pas nécessaire de préciser le login :

```
Context ctx = new InitialContext();
Fleur fleurRetrouvee = ctx.lookup(“ldap://localhost:389/cn=maFleur,
o=JNDITutorial”);
```

Utilisation de JNDI avec LDAP
Stockage d’un objet CORBA

Il est possible d’utiliser un annuaire LDAP pour le service de noms CORBA. Pour cela on crée un objet CORBA :

```
ORB orb = ORB.ini(args, null);

//Création de l’objet Bourse
BourseImpl bourse = new BourseImpl();
POA rootpoa = POAHelper.narrow(org.resolve_initial_references(“RootPOA”));
rootpoa.the_POAManager().activate();
org.omg.CORBA.Objet ref = rootpoa.servant_to_reference(bourse);
Bourse href = BourseHelper.narrow(ref);

Hashtable env = new Hashtable();
env.put(Context.INITIAL_CONTEXT_FACTORY, “com.sun.jndi.ldap.LdapCtxFactory”);
env.put(Context.PROVIDER_URL, “ldap://localhost:389/o=JNDITutorial”);
env.put(“java.naming.corba.orb”, orb);

//Création du contexte initial
DirContext ctx = new InitialDirContext(env);

//Stockage de l’objet dans l’annuaire
ctx.bind(“cn=CorbaHello”, ref);
```

Utilisation de JNDI avec LDAP

Stockage d’une référence RMI

Il est possible de stocker une référence à un objet RMI dans LDAP. LDAP ne sert alors que de “proxy” à un rmiregistry local, mais comme rmiregistry ne peut enregistrer des objets sur une autre machine, c’est un moyen commode de fédérer plusieurs annuaires.

Comme les EJB (Entreprise Java Beans) sont la plupart du temps des objets RMI, il est aussi possible de les stocker dans LDAP.

On obtient le contexte d’écriture LDAP comme précédemment :

```
Context ctx = ...
```

On crée ensuite une référence sur le nom de l’objet lorsqu’il sera enregistré dans le rmiregistry, et on la dépose dans l’annuaire :

```
String rmiurl = “rmi://localhost/modele”;
Reference ref = new Reference(“Modele”, new StringRefAddr(“URL”, rmiurl));

ctx.bind(“cn=RefModele”, ref);
```

Utilisation de JNDI avec LDAP
Stockage d’une référence RMI

On crée l’objet distant et on l’enregistre dans rmiregistry avec le nom défini dans l’annuaire :

```
Modele modele = new Modele();
Naming.rebind(rmiurl, modele);
```

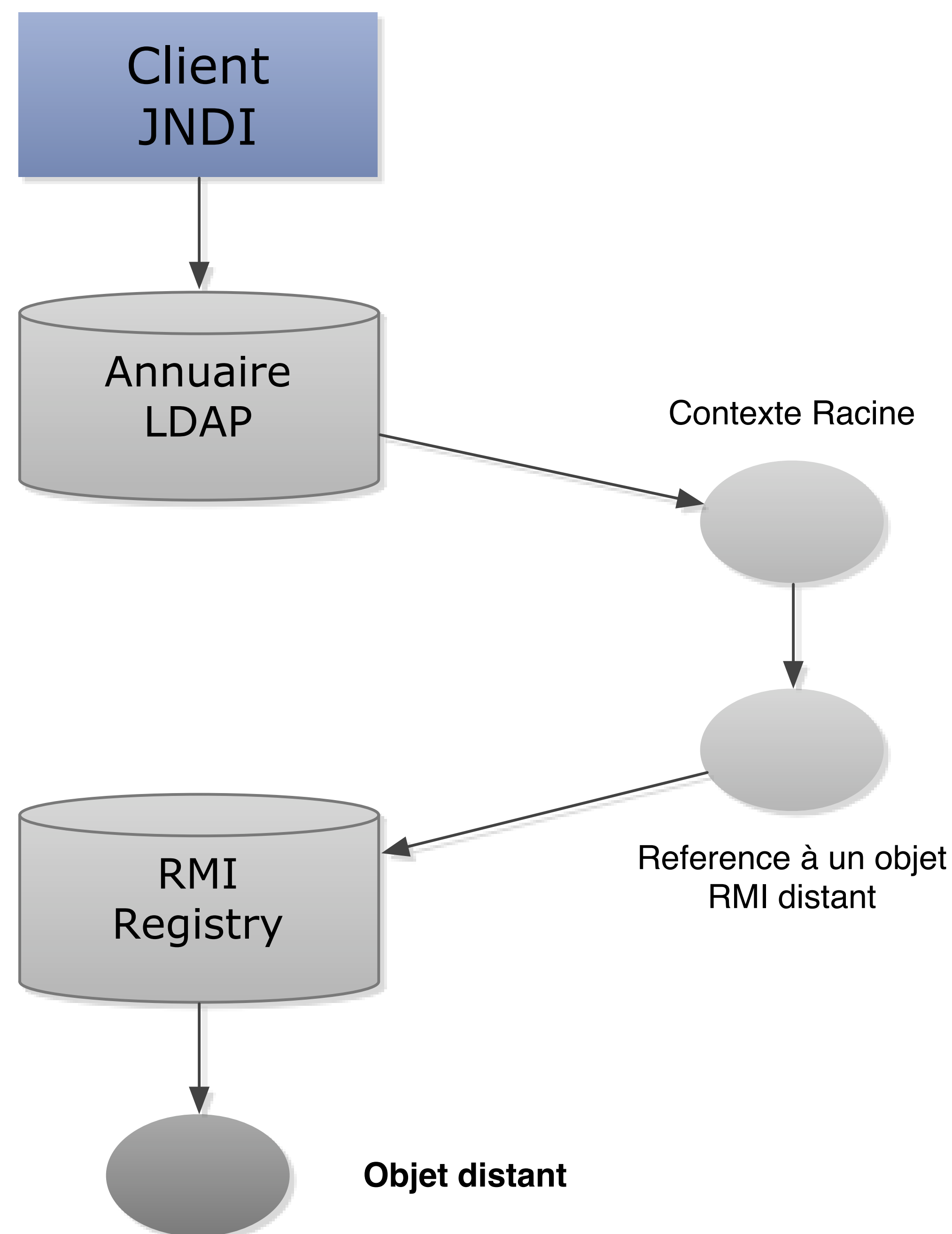
On peut alors tester la redirection effectuée automatiquement du LDAP vers le rmiregistry :

```
ModeleInterface modele2 = (ModeleInterface) ctx.lookup(“cn=RefModele”);
System.out.println(modele2.appelDeMethode());
```

... sans oublier de fermer la connexion à la fin avec `ctx.close();`

Utilisation de JNDI avec LDAP
Stockage d’une référence RMI

Redirection d’un annuaire RMI dans LDAP :



Utilisation de JNDI avec LDAP

Stockage d’une ConnectionFactory JMS

Nous verrons dans un chapitre suivant du cours qu’une ConnectionFactory va regrouper tous les paramètres de connexion à un serveur JMS.

Il est possible de stocker une ConnectionFactory dans LDAP afin de fournir un point d’accès unique et préconfiguré lorsque plusieurs clients partagent le même serveur.

Ces clients n’auront alors plus besoin de fichier de configuration, ce qui simplifie considérablement le déploiement d’une application répartie bâtie autours de JMS.

```
Context ctx = ...
ConnectionFactory factory = ...
ctx.bind(“cn=jmsFactory”, factory);
```

Ce mécanisme permet également de changer l’implémentation JMS ou de déplacer le serveur sans impact sur le client qui utilise la même API.

Recherche d’éléments dans LDAP

LDAP permet d’utiliser des attributs (mode pages jaunes) pour effectuer des recherches à l’aide de la méthode search de DirContext.

La recherche peut s’effectuer récursivement sur toute la hiérarchie ou à un seul niveau : c’est ce qu’on appelle les contrôles.

Ce cours ne développera pas ces aspects présents dans JNDI. Voici simplement un exemple de recherche d’enregistrements liés à un numéro de téléphone qui utilise une pseudo-URL :

```
DirContext ctx = new InitialDirContext();
Attributers attrs = new BasicAttributes();
attrs.put(new BasicAttribute(“telephoneNumber”, “+1 408 555 5252”));
NamingEnumeration enum = ctx.search(“ldap://localhost:389/
ou=People,o=JNDITutorial”, attrs);

while(enum.hasMore()) {
   System.out.println((SearchResult) enum.next());
}
```

Conclusion

Les services de noms et d’annuaires sont un des pivots du déploiement et de la supervision des applications réparties.

JNDI est une API Java qui permet d’uniformiser l’accès à de nombreuses ressources, tant matérielles qui logicielles, notamment avec l’utilisation de pseudo-URL.

- **Grâce à cette abstraction, l’accès à un annuaire RMI ou CORBA se fait de la même façon.**

LDAP permet une vision d’entreprise des annuaires et peut fédérer plusieurs implémentations ou plusieurs sites.

Fin du cours #5