Cours Systèmes / Applications Réparti(e)s

Institut Supérieur d'Informatique et de Mathématique de Monastir

teaching.bhiri@gmail.com

2015/2016

Bibliographie et Webliographie

- Material: Web Services Concepts, Architectures and Applications
- Authors: Gustavo Alonso, Fabio Casati, Harumi Kuno, Vijay Machiraju
- Publisher: Springer Verlag 2004, ISBN 3-540-44008-9

- Material: Web services and Service Oriented Architectures
- Location: http://www.iks.inf.ethz.ch/education/ss08/ws_soa
- Author: Gustavo Alonso: ©Gustavo Alonso, D-INFK. ETH Zürich.

- Material: Enterprise Application Integration (Middleware)
- Location: http://www.iks.inf.ethz.ch/education/ws06/EAI
- Author: Gustavo Alonso and Cesare Pautasso: ©IKS, ETH Zürich.

- Material: Services Web
- Location: http://www-inf.int-evry.fr/cours/WebServices
- Author: Samir Tata

- Material: BPEL : SOA
- Location: http://mbaron.developpez.com/soa
- Author: Mickael Baron: © Mickael Baron

- Formation Java avancé
- Author : Mohamed Youssfi

DES SOCKETS AU SYSTÈMES RPC

Contenu du cours

- Limites des sockets
- Problèmes principaux à résoudre pour supporter les interactions entre deux modules distants
- Système RPC
  - idée et objectif
  - composants, fonctionnement et outils
  - étapes de développement
- RMI

Quelques réflexions sur les sockets

- Mécanisme de bas niveau
  - pour implémenter des programmes modulaires, voir orientés objet
    - communication par envoie de messages par comparaison à l'appel de procédure
    - Passage de paramètres
  - réconcilier différents formats de données dans un environnement hétérogène
    - codage des données primitives et représentation des données complexes : tableaux, arbres, objets
  - résoudre dynamiquement les références des modules à invoquer
    - au lieu de les coder en dur dans les programmes
  - détecter et traiter les problèmes de communication

➔ Nécessite un système /qui étend les/au dessus des/ sockets pour dépasser ces limites

INTERACTIONS DANS UN SYSTÈME D'INFORMATION DISTRIBUÉ

Interactions dans SID: comment interagir dans un système distribué (1)

- Pour permettre à deux modules logiciels distribués de communiquer, il faut résoudre 4 problèmes principaux:

1. Comment intégrer les primitives de communications dans les langages de programmation
   - Comment communiquer
   - Paradigme sous-jacent du langage
2. Comment réconcilier différentes représentations machines des donnés et faire l'appariement de différentes représentations des structures de données complexes
   - Différence au niveau machine
   - Représentation des types de données abstraits

7

Interactions dans SID: comment interagir dans un système distribué (2)

3. Comment trouver le module avec qui on veut communiquer
   - Mode d'adressage (qui connait l'adresse et quand la déterminer)
4. Comment traiter les erreurs qui peuvent surgir durant le processus de communication entre les modules
   - … et comment recouvrir de ces erreurs

8

Interactions dans SID: le langage de programmation (1)

- Communication implicite
  - Les primitives de communication sont cachées derrières des constructions de programmation normales
  - Quand on invoque ces constructions, le système sous-jacent prend en charge de transformer telle invocation à une interaction avec un module logiciel distant
  - Typiquement, cette forme de communication est bloquante (l'appelant doit attendre l'appelé)
  - Avantage: le programmeur n'a pas besoin de changer son mode de programmation pour développer une application distribuée
  - Inconvénient: quand des problèmes surgissent, c'est difficile au programmeur de les traiter
  - Exemples: mémoire partagée, Appel de procédure à distance (RPC), appelle de méthode à distance (RMI)

9

Interactions dans SID: le langage de programmation (2)

- Communication explicite
  - Cette communication s'effectue via des primitives explicites
  - Nécessite un modèle pour spécifier comment la communication se déroule
  - Typiquement, ces formes de communications sont non bloquantes
  - Avantage: les patrons de communications sont beaucoup plus flexibles
  - Inconvénient: le programmeur doit connaitre comment la communication se déroule
  - Exemples: intérgiciel orientés message, queues persistants, systèmes à base d'événements

10

Interactions dans SID: La représentation intermédiaire (1)

- Pour comprendre les messages échangés, les parties doivent se mettre d'accord comment représenter ces donné:
  - Ordre des octets et bits: Little-endian vs Big-endian
  - Représentation des types de données complexes (tableaux. Arbres, structures, objets)
  - Format des messages
- Marshalling: le processus de transformation d'une représentation de donnés particulière vers une autre représentation plus adéquate à la communication (et le stockage persistant)
- Serialisation: le processus de transformation de données en des chaines d'octets qui peuvent être envoyées comme des paquets dans le réseau

11

Interactions dans SID: La représentation intermédiaire (2)

- Il y a deux options pour résoudre ce problème de représentation de données hétérogènes:
  - NxN représentations intermédiaires
  - Adopter une représentation intermédiaire comprise par tout les systèmes
- Une représentation intermédiaire doit être standardisée pour être vraiment utile
- Une représentation intermédiaire non standard est typiquement dépendante du système. Par exemple:
  - SUN RPC: XDR (External Data Representation)

12

Interactions dans SID: La représentation intermédiaire (3)

- Les langages de définition d'interfaces (IDL) sont généralement utilisés: typiquement utilisés dans des approches de communications implicites.
  - Un langage de déclaration d'interfaces.
  - Représentation de données intermédiaire: Correspondance entre les représentation de données dans des langages particuliers et une représentation intermédiaire.
  - Une approche de (dé)sérialisation des données transmises.

➔ Utilisés par les compilateurs et les éditeurs de lien coté client (stubs) et sereveur (skeleton) pour savoir comment procéder pour le (dé)marshalling et la (dé)sérialisation lors d'un appel de procédure à distance.

13

Interactions dans SID : localiser un module (1)

- Pour qu'un module communique avec un autre, il doit le localiser (dans le réseau).
- Le processus de détermination de l'adresse d'un module à invoquer est appelé liaison (binding).

14

Interactions dans SID : localiser un module (2)

- La liaison (binding) peut être:
- Statique: l'appelant possède l'adresse de l'appelé et lui transmet les données directement
  - La liaison statique est très efficace (pas d'indirection)
  - La liaison statique conduit au couplage fort entre les différentes parties (assez mauvais en général)
- Dynamique: est implémenté via un module intermédiaire (registre de service, service de nommage, routeur)
  - Adresse: l'appelant connait certaines informations sur l'appelé qu'il veut invoquer et demande au module intermédiaire son adresse.
  - Interface et adresse: l'appelant possède quelques informations sur ce qu'il veut appeler et demande au module intermédiaire l'interface et l'adresse.
  - La liaison dynamique est utilisée pour le balancement de charges, améliorer la tolérance aux pannes, flexibilité de conception, et le découplage des parties impliquées
  - … comment trouver le service de nommage?

15

SYSTÈME DE « REMOTE PROCEDURE CALL » (RPC)

- Les sockets sont encore de bas niveau pour plusieurs applications ➔ systèmes de RPC ont émergé pour
  - Masquer les détails de communication dans un appel de procédure
  - Réconcilie des environnements hétérogènes

Communication inter processus (IPC)

- Permet la communication entre des processus
  - Transférer des données
  - Invoquer une action dans un autre processus (éventuellement s'exécutant dans une autre machine distante)

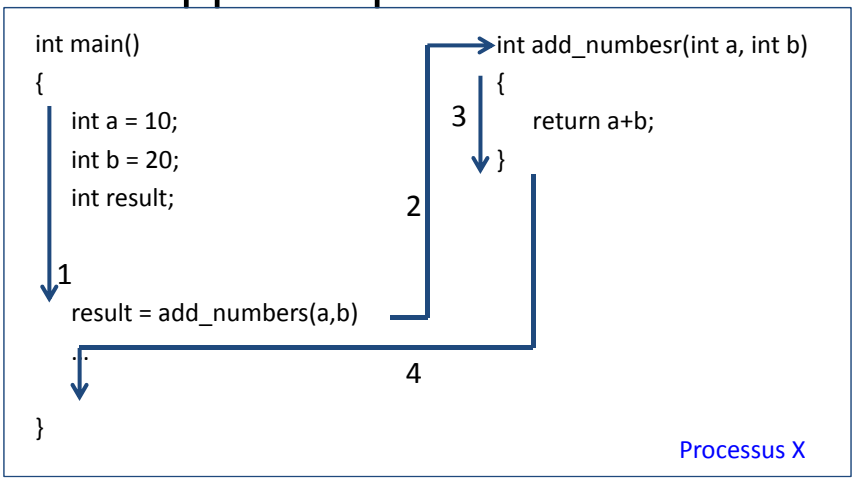
18

Appelle de procédure à distance - Remote Procedure Call (RPC)

- Une représentation de haut niveau de IPC à distance
- Utilise le même métaphore et style de programmation que l'appel de procédure classique
- Appel une procédure dans un autre processus
- Masque/fait abstraction des mécanismes de communication

19

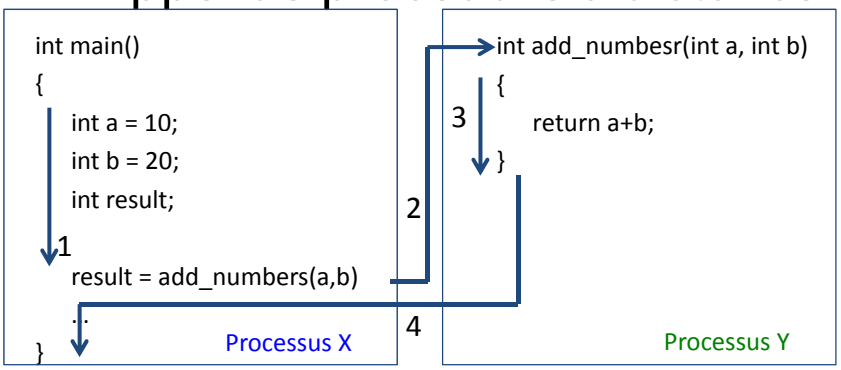
Appel de procédure locale



Taken from Chris Greenhalgh

20

Appel de procédure à distance



- Flot 1 et 3 sont les mêmes dans les deux cas (local et à distance)
  - Mais dans deux processus différents thread/processus
- Flot 2: l'appel passe de X à Y
- Flot 4: le retour passe de Y à X

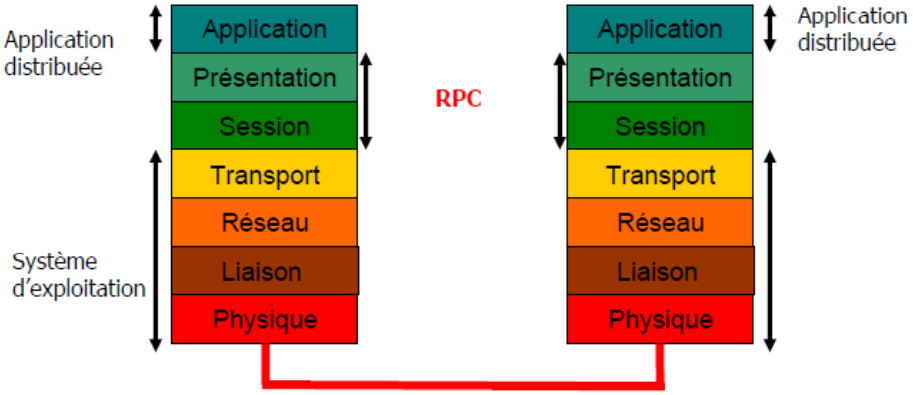
Taken from Chris Greenhalgh

21

RPC : Objectifs

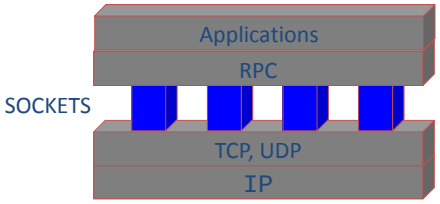
- Objectifs :
  - Model de programmation (classique) basé sur l'appel de procédures au lieu du mécanisme L'envoie/réception de messages
  - Cacher la complexité du réseau au développeurs d'applications distribuées (Sockets, numéro de port, Formatage des donnée, ordre des octets, …)

RPC dans le modèle OSI



RPC en pratique

- Le programmeur ne peut pas implémenter toutes les mécanismes nécessaires à chaque fois il développe une application distribuée. Plutôt , ils utilisent/reposent sur un système de RPC (premier exemple d'intergiciel de bas niveau)
- Un système RPC
  - Fournit un langage de définition d'interface (IDL) pour décrire les services
  - Génère tout le code additionnel nécessaire pour faire un appel de procédure à distance et traiter les différents aspects de communication
  - Fournit un "binder" en cas il utilise un service de nommage distribué



24

IDL (Interface Definition Language)

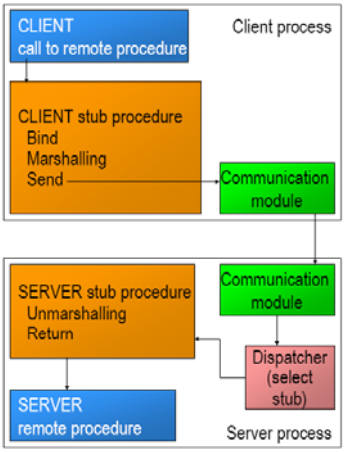
- Tous les systèmes RPC fournissent un langage, IDL, pour décrire les services d'une manière abstraite (indépendamment des langages de programmation)
- IDL définit aussi une représentation intermédiaire des données.
- Un compilateur d'interface est alors utilisé pour générer des ''stubs'' pour les clients et ''skeletons'' des serveurs. Il peut aussi générer des entêtes de procédures que les programmeurs peuvent utiliser et étendre pour compléter l'implémentation des serveurs et des clients.

- Étant donné une spécification IDL, le compilateur d'interface achève un ensemble de tâches pour générer les ''stubs'' et ''skeletons'' dans des langages de programmation particuliers (comme JAVA, C) :

1. Génère la procédure de stub du client pour chaque procédure dans l'interface. Le stub va alors être compilé et lié avec le code du client.
2. Génère le skeleton du serveur. Il peut aussi générer une ébauche du code de serveur. Ce code peut être alors complété par le développeur pour implémenter les procédures.
3. Il peut aussi générer des fichiers (d'entête *.h/des classes) des types à importer par les clients et serveurs.

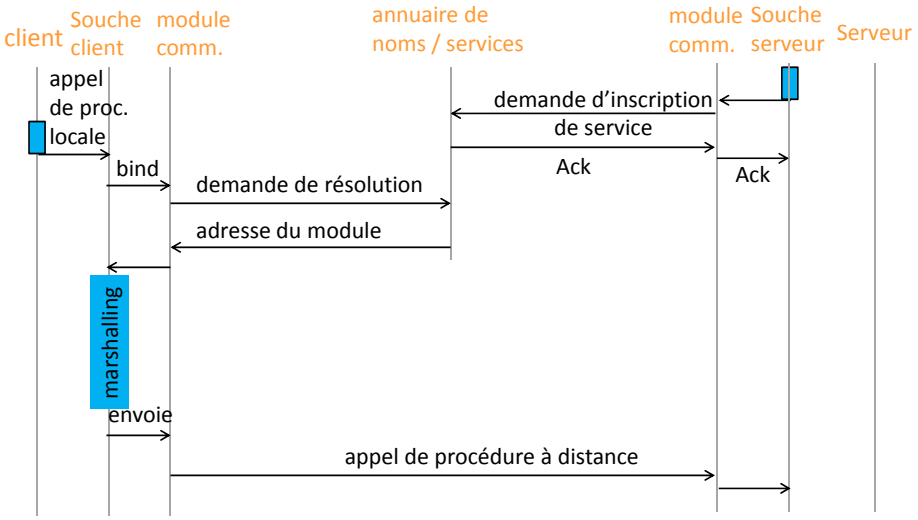
25

RPC en pratique

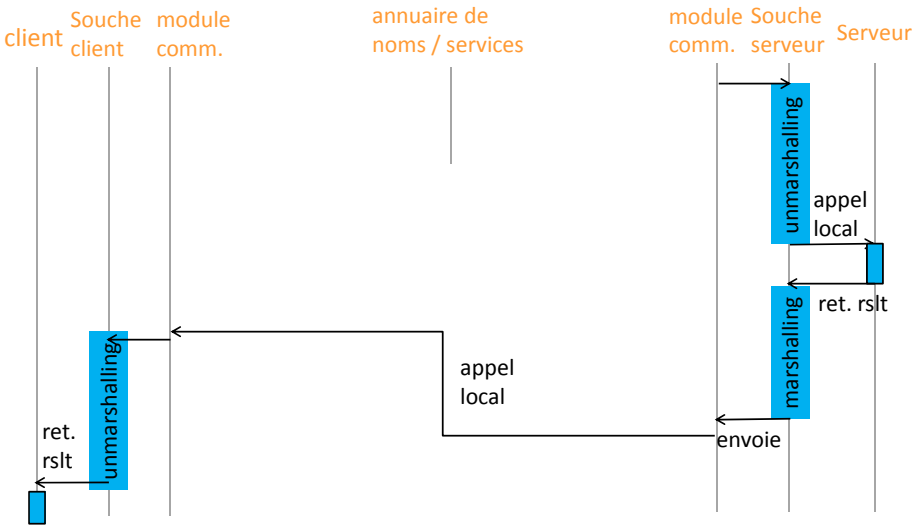


26

RPC: comment ça marche en pratique

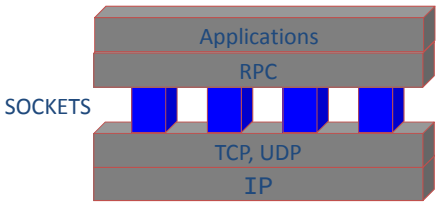


RPC: comment ça marche en pratique



RPC en pratique

- Le programmeur ne peut pas implémenter toutes les mécanismes nécessaires à chaque fois il développe une application distribuée. Plutôt , ils utilisent/reposent sur un système de RPC (premier exemple d'intergiciel de bas niveau)
- Un système RPC
  - Fournit un langage de définition d'interface (IDL) pour décrire les services
  - Génère tout le code additionnel nécessaire pour faire un appel de procédure à distance et traiter les différents aspects de communication
  - Fournit un "binder" en cas il utilise un service de nommage distribué



29

Démarche pour développement RMI

1. Créer les interfaces des objets distants
2. Créer les implémentation des objets distants
3. Générer les stubs et skeletons
4. Créer le serveur RMI
5. Créer le client RMI
6. Déploiement Lancement
   - Lancer l'annuaire RMIREGISTRY
   - Lancer le serveur
   - Lancer le client

1. Créer les interfaces des objets distants

- La première étape consiste à créer une interface distante qui décrit les méthodes que le client pourra invoquer à distance.
- Pour que ses méthodes soient accessibles par le client, cette interface doit hériter de l'interface **Remote.**
- Toutes les méthodes utilisables à distance doivent pouvoir lever les exceptions de type **RemoteException qui sont spécifiques à l'appel** distant.
- paramètres de type simple, objets Sérialisables (implements Serializable) ou Distants (implements Remote)
- Cette interface devra être placée sur les deux machines (serveur et client).

1. Créer les interfaces des objets distants

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

public interface Hello extends java.rmi.Remote {
   String sayHello(String input) throws
   java.rmi.RemoteException;
}
```

2. Créer les implémentation des objets distants

- Il faut maintenant implémenter cette interface distante dans une classe. Par convention, le nom de cette classe aura pour suffixe Impl.
- Notre classe doit hériter de la classe java.rmi.server.RemoteObject ou de l ’une de de ses sous-classes.
- Le plus facile à utiliser étant java.rmi.server.UncicastRemoteObject.
- C’est dans cette classe que nous allons définir le corps des méthodes distantes que pourront utiliser nos clients

2. Créer les implémentation des objets distants

```
import java.rmi.*;
import java.rmi.server.UnicastRemoteObject;
public class HelloImpl extends UnicastRemoteObject implements Hello {
        private String name;
        public HelloImpl() throws RemoteException {
                super(); }
// Implémentation de la méthode distante
        public String sayHello(String input) throws RemoteException {
                return “This is a message from the remote object.
   Hello from " + input; }
}
```

STUBS et SKELETONS

- Si l'implémentation est créée, les stubs et skeletons peuvent être générés par l'utilitaire rmic en écrivant la commande rmic

rmic HelloImpl

Création et enregistrement d'objets distants

- Un serveur crée un service distant en créant d'abord un objet local qui implémente ce service.
- Ensuite, il exporte cet objet vers RMI.
- Quand l'objet est exporté, RMI crée un service d'écoute qui attend qu'un client se connecte et envoie des requêtes au service.
- Après exportation, le serveur enregistre cet objet dans le registre de RMI sous un nom public qui devient accessible de l'extérieur.
- Le client peut alors consulter le registre distant pour obtenir des références à des objets distants.

Création et enregistrement d'objets distants

- Les clients trouvent les services distants en utilisant un service d'annuaire activé sur un hôte connu avec un numéro de port connu.
- RMI peut utiliser plusieurs services d'annuaire, y compris Java Naming and Directory Interface (JNDI).
- Il inclut lui-même un service simple appelé (rmiregistry).
- Le registre est exécuté sur chaque machine qui héberge des objets distants (les serveurs) et accepte les requêtes pour ces services, par défaut sur le port 1099.

Création et enregistrement d'objets distants

- Notre serveur doit enregistrer auprès du registre RMI l'objet local dont les méthodes seront disponibles à distance.
- Cela se fait grâce à la méthode statique **bind() ou rebind() de la classe Naming.**
- Cette méthode permet d'associer (enregistrer) l'objet local avec un synonyme dans le registre RMI.
- Le Naming enregistre le nom de l'objet serveur et une souche client (contenant l'@IP et le port de comm.) dans le rmiregistry
- L'objet devient alors accessible par le client.

```
ObjetDistantImpl od = new ObjetDistantImpl();
Naming.rebind("rmi://localhost:1099/NOM_Service",od);
```

Création et enregistrement d'objets distants

```
import java.rmi.Naming;
public class ServeurRMI {
    public static void main(String[] args) {
        try {
            // Créer l'objet distant
            HelloImpl hello=new HelloImpl();
            // Publier sa référence dans l'annuaire
            Naming.rebind("rmi://localhost:1099/bonjour",hello);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

Créer le client RMI

- Le client peut obtenir une référence à un objet distant par l'utilisation de la méthode statique **lookup() de la classe Naming.**
- La méthode lookup() sert au client pour interroger un registre et récupérer un objet distant.
- Elle retourne une référence à l'objet distant.
- Le Naming récupère la souche client de l'objet serveur, …
- La valeur retournée est du type Remote. Il est donc nécessaire de caster cet objet en l'interface distante implémentée par l'objet distant.

Créer le client RMI

```
import java.rmi.Naming;
public class ClientRMI {
    public static void main(String[] args) {
        try {
            Hello stub=
            (Hello)Naming.lookup("rmi://localhost:1099/bonjour");

            System.out.println(stub.sayHello("Galway"));
        } catch (Exception e) {
            e.printStackTrace();
        }
}}
```

Déploiement Lancement

- Il est maintenant possible de lancer l'application. Cela va nécessiter l'utilisation de trois consoles.
  - La première sera utilisée pour activer le registre. Pour cela, vous devez exécuter l'utilitaire **rmiregistry : start rmiregistry**
  - Dans une deuxième console, exécutez le serveur. Celui-ci va charger l'implémentation en mémoire, enregistrer cette référence dans le registre et attendre une connexion cliente.
  - Vous pouvez enfin exécuter le client dans une troisième console.
- Même si vous exécutez le client et le serveur sur la même machine, RMI utilisera la pile réseau et TCP/IP pour communiquer entre les JVM.

JNDI (Java Naming and Directory Interface)

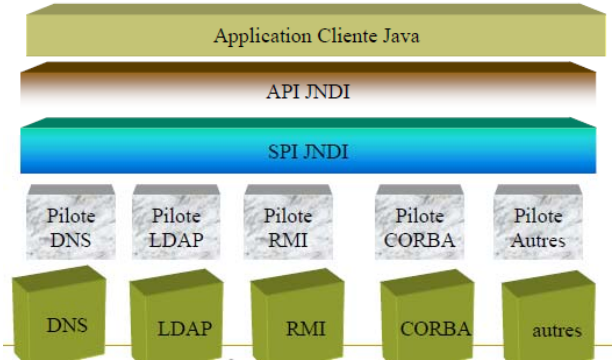
- JNDI est l'acronyme de Java Naming and Directory Interface.
- Cette API fournit une interface unique pour utiliser différents services de nommages ou d'annuaires et définit une API standard pour permettre l'accès à ces services.
- Il existe plusieurs types de service de nommage parmi lesquels :
  - DNS (Domain Name System) : service de nommage utilisé sur internet pour permettre la correspondance entre un nom de domaine et une adresse IP
  - LDAP(Lightweight Directory Access Protocol) :
  - annuaire NIS (Network Information System) : service de nommage réseau développé par Sun Microsystems
  - COS Naming (Common Object Services) : service de nommage utilisé par Corba pour stocker et obtenir des références sur des objets Corba
  - RMI Registry : service de nommage utilisé par RMI
  - etc, ...

JNDI (Java Naming and Directory Interface)

- Un service de nommage permet d'associer un nom unique à un objet et de faciliter ainsi l'obtention de cet objet.
- Un annuaire est un service de nommage qui possède en plus une représentation hiérarchique des objets qu'il contient et un mécanisme de recherche.
- JNDI propose donc une abstraction pour permettre l'accès à ces différents services de manière standard. Ceci est possible grâce à l'implémentation de pilotes qui mettent en oeuvre la partie SPI de l'API JNDI. Cette implémentation se charge d'assurer le dialogue entre l'API et le service utilisé.

Architecture JNDI

L'architecture de JNDI se compose d'une API et d'un Service Provider Interface (SPI). Les applications Java emploient l'API JNDI pour accéder à une variété de services de nommage et d'annuaire. Le SPI permet de relier, de manière transparente, une variété de services de nommage et d'annuaire ; permettant ainsi à l'application Java d'accéder à ces services en utilisant l'API JNDI



Architecture JNDI

- JNDI est inclus dans le JDK Standard (Java 2 SDK) depuis la version 1.3. Pour utiliser JNDI, vous devez possédez les classes JNDI et au moins un SPI (JNDI Provider).
- La version 1.4 du JDK inclut 3 SPI pour les services de nommage/annuaire suivant :
  - Lightweight Directory Access Protocol (LDAP)
  - Le service de nommage de CORBA (Common Object Request Broker Architecture) Common Object Services (COS)
  - Le registre de RMI (Remote Method Invocation) rmiregistry
  - DNS

Serveur RMI utilisant JNDI

```
package service; import java.rmi.Naming; import java.rmi.registry.LocateRegistry;
public class ServeurRMI {
    public static void main(String[] args) {
        try {
        // Démarrer l'annuaire RMI REgistry
        LocateRegistry.createRegistry(1099);
        // Créer l'Objet distant
          BanqueImpl od=new BanqueImpl();
    // Créer l'objet InitialContext JNDI en utilisant le fichier jndi.properties
         Context ctx=new InitialContext();
        // Publier la référence de l'objet distant avec le nom BK
         ctx.bind("BK", od);
    } catch (Exception e) { e.printStackTrace();}
}
```

Fichier jndi.properties :

```
java.naming.factory.initial=com.sun.jndi.rmi.registry.RegistryContextFactory
java.naming.provider.url=rmi://localhost:1099
```

Client RMI utilisant JNDI

```
import javax.naming.*; import rmi.IBanque;
public class ClientRMI {
    public static void main(String[] args) {
        try {
           Context ctx=new InitialContext();
           ctx.addToEnvironment(Context.INITIAL_CONTEXT_FACTORY,
           "com.sun.jndi.rmi.registry.RegistryContextFactory");
           ctx.addToEnvironment("java.naming.provider.url",
           "rmi://localhost:1099");
           IBanque stub=(IBanque) ctx.lookup("BK");
           System.out.println(stub.test("test"));
        } catch (Exception e) { e.printStackTrace();}
}}
```

- Dans ce cas, on pas besoin de créer le fichier jndi.properties