Appel de procédures distantes

Introduction aux systèmes à base d'objets distribués,
Illustration sur RMI

Du modèle client/serveur …

- **Application Client/Serveur** :

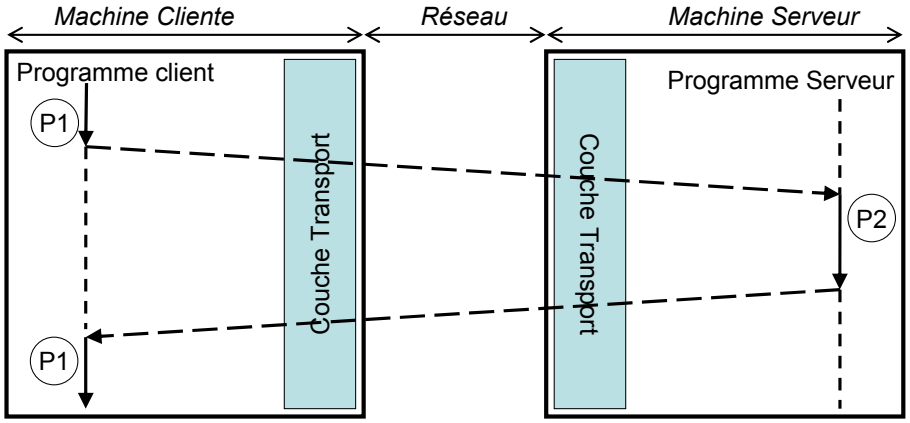
Déf. : Application qui fait appel à des services distants au travers d'un échange de messages

  - Le Client envoie une requête ;
  - Le Serveur retourne une requête.

- **Les clients** sont les programmes qui sollicitent des services disponibles sur une machine distante.
- **Le serveur** est le programme qui fournit un service à un ensemble de clients.

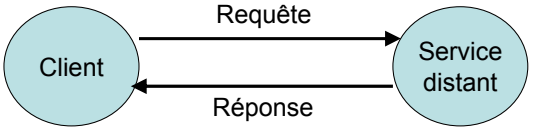
Du modèle client/serveur …

- Selon le modèle client/serveur, deux messages au moins sont échangés :

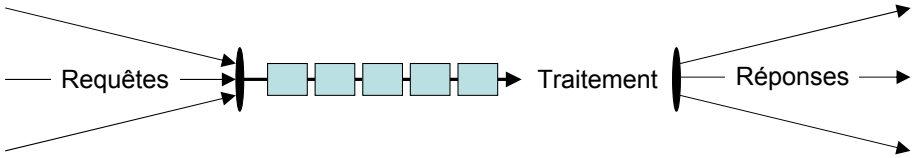


Du modèle client/serveur …

- Vue du client



- Vue du serveur
  - Gestion des requêtes (priorité)
  - Exécution du service (séquentiel, concurrent)
  - Mémorisation ou non de l'état du client



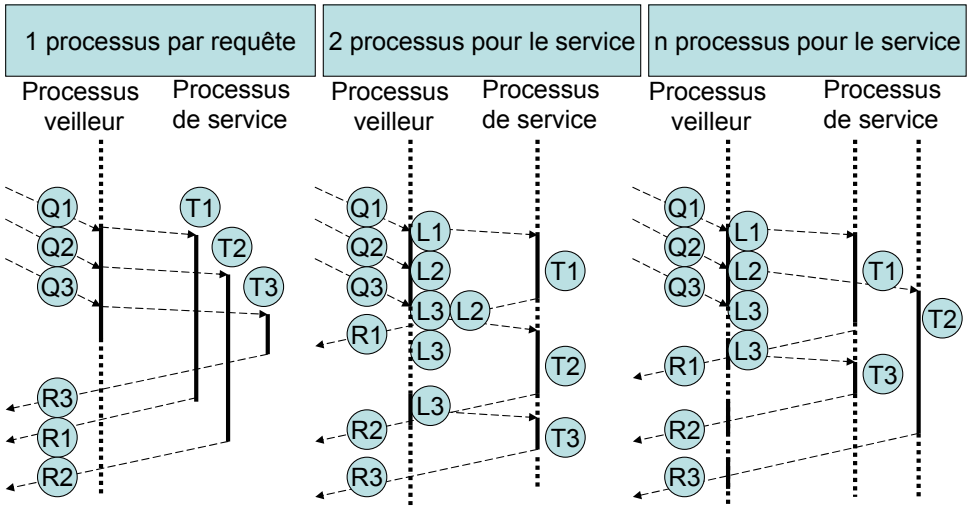
Du modèle client/serveur …

Exemples d'application client/serveur

- Serveur de fichiers (aufs, nfsd)
- Serveur d'impressions (lpd)
- Serveur de calcul
- Serveur de base de données
- Serveur de noms (annuaire des services)

Du modèle client/serveur …

- 1 processus par service



Du modèle client/serveur …

Service sans données persistantes :

- Situation idéale où le service s'exécute uniquement en fonction des paramètres d'entrée
- Modèle client/serveur optimal
  - Pour la tolérance aux pannes
  - Pour le contrôle de la concurrence
- Exemple :
  - le calcul scientifique

Du modèle client/serveur …

Service avec données persistantes :

- Les exécutions successives manipulent des données persistantes
  - Modification du contexte d'exécution sur le site distant
  - Problème de contrôle de concurrence
  - Difficultés en cas de panne en cours d'exécution
- Exemple :
  - Serveur de fichier réparti (lectures / écritures)

Du modèle client/serveur …

Service en mode sans état :

- Les différentes requêtes peuvent être traitées sans lien entre elles.
  - Il peut y avoir modification de données globales mais l'opération s'effectue sans lien avec celles qui l'ont précédé.
- Exemple
  - Serveur de fichier réparti : accès aléatoire

Du modèle client/serveur …

Service en mode avec état :

- Les différentes requêtes sont nécessairement traitées séquentiellement.
  - Il peut y avoir modification de données globales ou pas mais l'opération s'effectue en liaison avec celles qui l'ont précédé.
- Exemple
  - Serveur de fichier réparti : accès séquentiel

… au modèle d'appel de procédure à distance.

Outil idéal pour les applications conçues selon le modèle client / serveur.

- L'opération à réaliser est présentée sous la forme d'une procédure que le client peut appeler. Ce faisant il déclenche l'exécution du traitement associé à cette procédure, mais sur la machine distante.
  - Simplicité (en l'absence de panne)
  - Sémantique identique à celle de l'appel local
- Opération de base
  - Client
    1. `doOp(IN ServiceID s, IN Name opName, IN Msg *args, OUT Msg *result)`
  - Serveur
    1. `getRequest(OUT ServiceID s,OUT Msg *args)`
    2. `opName(IN Msg *args, OUT Msg *result)`
    3. `sendReply(IN ServiceID s,IN Msg *result)`

… au modèle d'appel de procédure à distance.

- Objectifs
  - Retrouver la sémantique habituelle de l'appel de procédure
    - Sans se préoccuper de la localisation de la procédure
    - Sans se préoccuper du traitement des défaillances
  - Objectifs très difficiles à atteindre
    - Réalisation peu conviviale
    - Sémantique différente de l'appel de procédure même en l'absence de panne.

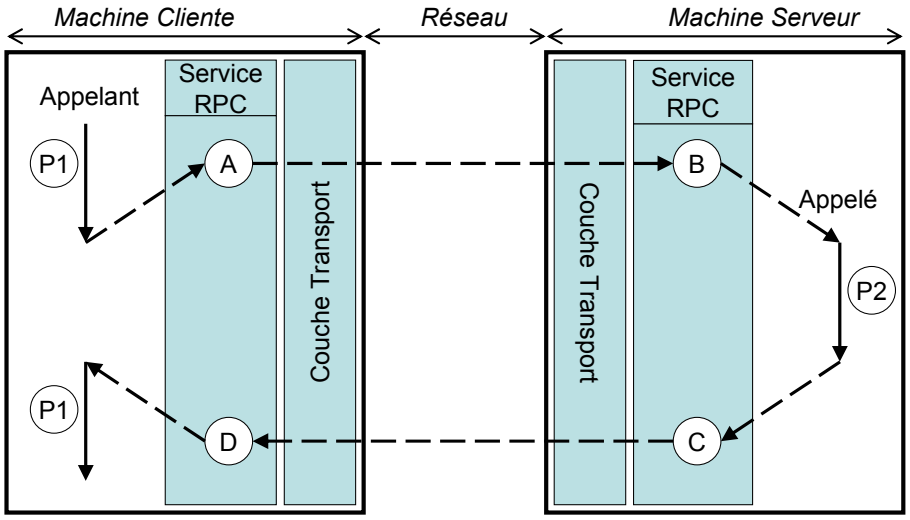
… au modèle d'appel de procédure à distance.

Les pièges des appels de procédure à distance :

Appel de procédure	Appel de procédure à distance
Appelant et appelé sont dans le même espace de travail ⇒ Environnement d'exécution de l'appelant partagé avec l'appelé. • même mode de pannes ; • appel et retour de procédure considéré comme fiable ; • temps d'appel très faible ; • concurrence des appels dans une minorité de cas.	Appelant et appelé sont dans des espaces virtuels différents : ⇒ Environnement d'exécution de l'appelant distinct de celui de l'appelé. • pannes du client et du serveur « indépendantes » • pannes du réseau de communication ⇒ appel de procédure distante considéré comme non fiable • temps d'appel non négligeable ; • concurrence des appels dans la majorité des cas.

Principe de fonctionnement d'un appel distant

- Principe de Birrel & Nelson (84)



Rôle des talons

Le talon client - Stub -

C'est la procédure d'interface du site client :

- qui reçoit l'appel local ;
- le transforme en appel distant (encodage des arguments dans un message « réseau »)
- attend réception des résultats de l'exécution distante
- décode et retourne la réponse à celui qui a appelé (localement) le Stub.

Le talon serveur - Skeleton -

C'est la procédure sur le site serveur :

- Qui reçoit l'appel sous forme de message (décode les arguments)
- Appelle « localement » la procédure serveur
- Encode la réponse fournie par la procédure serveur sous la forme d'un message « réseau ».

RPC
Les problèmes

Traitement des défaillances

- congestion du réseau ou du serveur
  - la réponse ne parvient pas en temps utile
- panne du client pendant le traitement de la requête
- panne du serveur avant ou pendant le traitement de la requête
- erreur de communication

Problèmes de sécurité

- authentification du client
- authentification du serveur
- confidentialité des échanges

Désignation et liaison

Aspect pratiques

- adaptation à des conditions multiples (protocoles, langages, matériels)
- gestion de l'hétérogénéité

RPC
Passage de paramètres

Valeur
- pas de problème particulier

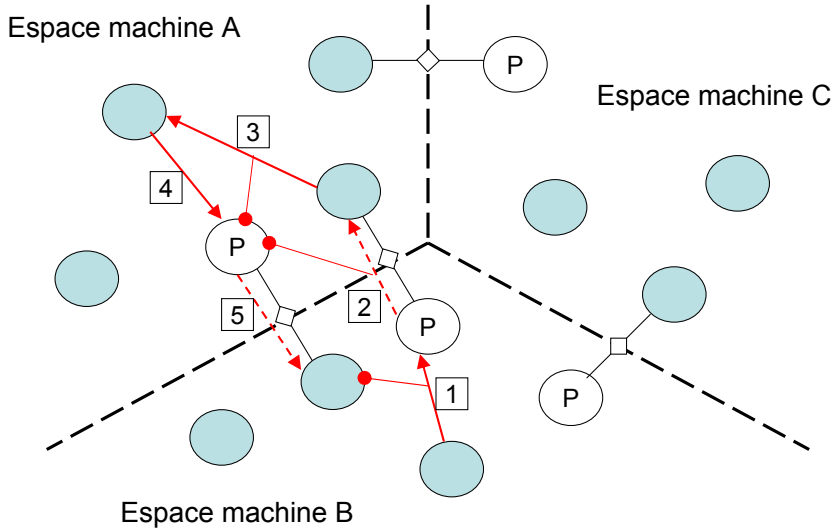
Copie / restauration
- les valeurs des paramètres sont recopiées
- Pas de difficultés majeures
- optimisation des solutions pour RPC

Références
Impossible d'utiliser « l'adresse mémoire » de l'appelant !

Solution pour les refs.
- proposer une référence de l'objet indépendante de l'adresse mémoire.
- L'objet devient lui-même une référence distante pour le serveur qui pourra l'utiliser via des appels de procédures distantes.

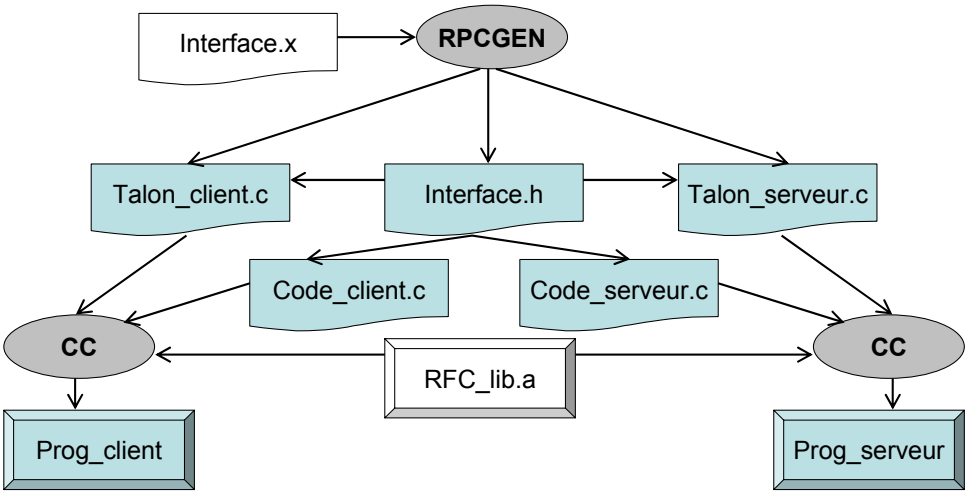
Solution insuffisante
pour représenter des références « non objet » (adresse mémoire).
⇒ Seule alternative : mémoire distribuée

Système d'objets distribués



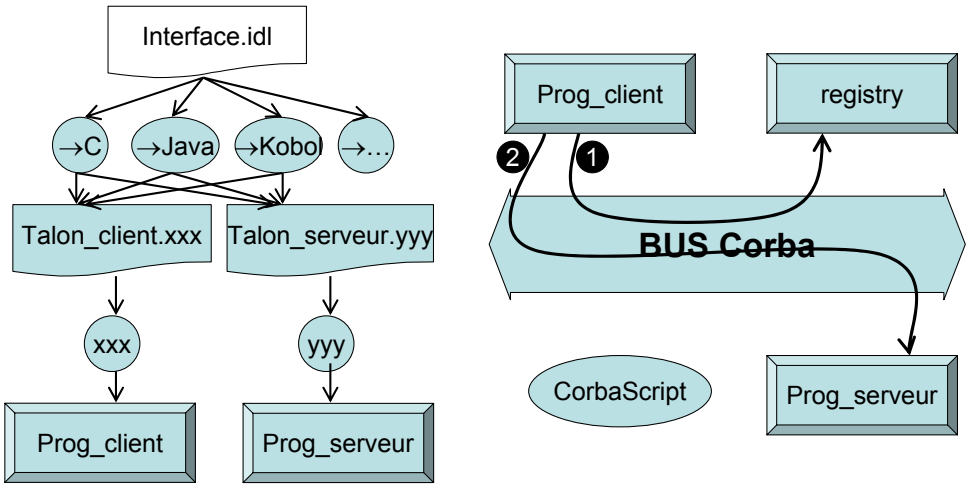
Les premières générations d'outils d'appel à distance

- RPC : langage C / Unix



Corba

Support de l'hétérogénéité des langages et des plateformes :
1 langage de description du service : IDL
1 consortium garant de la représentation du plus grand nombre : OMG

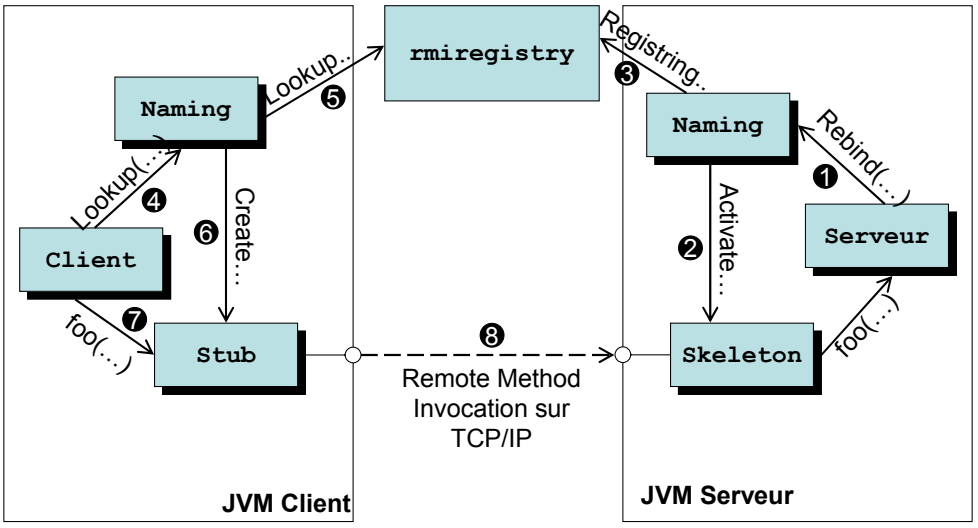


Java et les applications distribuées Java-RMI

Java possède un RPC orienté-objet intégré

- Interaction d'objets situés dans des espaces d'adressage différents sur des machines distinctes.
- Un objet distribué est un objet Java « comme les autres ». Il possède
  - Un proxy : représentant de l'objet coté client…
  - Un skeleton : coté serveur

Java RMI Architecture



Java RMI
Mode opératoire

Codage
- Description de l'interface du service
- Ecriture du code du serveur implantant l'interface
- Ecriture du client qui utilise l'interface

Compilation
- Compilation des sources (`javac`)
- Génération des stub et skeleton (`rmic`)

Activation
- Lancement du serveur de noms (`rmiregistry`)
- Lancement du serveur
- Lancement du client

Java RMI
écriture de l'interface

Simple déclaration d'une interface Java classique.

- L'interface doit être publique
- L'interface distante doit étendre l'interface `java.rmi.Remote`
- Chaque méthode doit déclarer au moins l'exception `java.rmi.RemoteException`
- Les objets passés en paramètre des méthodes doivent :
  - être une sorte d'interface `java.rmi.Remote` le paramètre est alors passé par référence ;
  - Supporter l'interface `java.io.Serializable`, dans ce cas l'objet est passé par valeur (sérialisation / desérialisation)

Exemple Java RMI
1/ création de l'interface

Hello.java

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

Exemple Java RMI
2/ création du serveur

HelloServeur.java

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

public class HelloServeur extends UnicastRemoteObject implements Hello{
  private String msg;

  public HelloServeur(String msg) throws java.rmi.RemoteException {
    super(); this.msg = msg;  }

  public String sayHello() throws java.rmi.RemoteException {
    System.out.println("Hello world: " + msg);
    return "Hello world: " + msg; }

  public static void main(String args[]) {
    try {
      HelloServeur obj = new HelloServeur("HelloServeur");
      Naming.rebind("//localhost:8080/mon_serveur",obj);
      System.out.println("HelloServer bound in registry");
    } catch(Exception e) { e.printStackTrace(); }
  }
}
```

Exemple Java RMI
2/ création du serveur

HelloClient.java

```
import java.rmi.*;

public class HelloClient {
  public static void main(String args[]) {
    try {
      Hello obj = (Hello)
  Naming.lookup("//localhost:8080/mon_serveur");
      String msg = obj.sayHello();
      System.out.println(msg);
    } catch (Exception e) {
      e.printStackTrace();
    }
  }
}
```

Exemple Java RMI
3/ génération du code

Compilation des sources :

```
javac Hello.java HelloServeur.java HelloClient.java
```

Génère les classes :

```
hello.class
HelloServeur.class
HelloClient.class
```

Production des talons associés au serveur :

```
rmic HelloServeur
```

Génère les classes :

```
HelloServeur_Stub.class
HelloServeur_Skel.class
```

Exemple Java RMI
4/ activation du système

- Trois processus à démarrer :

RMIREGISTRY

```
>
> RMIREGISTRY 8080
```

JAVA HelloServeur

```
> JAVA -Djava.rmi.server.codebase=ftp://localhost/JServ/ HelloServeur
HelloServer bound in registry
Hello world: HelloServeur
```

JAVA HelloClient

```
>
> JAVA HelloClient
Hello world: HelloServeur
>
```

Couche de transport alternative pour Java RMI

Définir une sous-classe de **`Socket`** et une sous-classe de **`ServerSocket`** adapté au support de transport visé.

Définir une classe implantant **`RMIClientSocketFactory`** et une autre pour **`RMIServerSocketFactory`** produisant des sortes de **`Socket`** et des **`ServerSocket`** qui seront utilisées par le stub et le skeleton.

Le serveur qui étend **`UnicastRemoteObject`** et qui implante l'interface RMI "remote" utilise

```
Super(0, // choix d'un port anonyme sinon numero du port
      new RMIClientSocketFactory(),
      new RMIServerSocketFactory())
```

Plutot que

```
Super()
```

N.B. : Flexibilité limitée pour la couche de transport utilisée.