Environnement Client/Serveur

Cours 6
Appel de méthode à distance

kn@lri.fr
http://www.lri.fr/~kn

UNIVERSITÉ PARIS SUD
Comprendre le monde, construire l'avenir

université PARIS-SACLAY

Plan

RPC ?

RPC (*Remote Procedure Call*) : système permettant à un programme d'appeler une procédure où une fonction se trouvant dans un espace d'adressage séparé. Typiquement, la procédure appelée se trouve dans un autre processus qui peut lui-même se trouver sur une autre machine.

Le système se charge de prendre en charge les aspects bas-niveau (communication par des fichiers partagés ou le réseau, (dé)-sérialisation des données, ...). Du point de vue du programmeur, c'est un simple appel de fonction.

Dans les langages objets on parle de RMI (*Remote Method Invocation*).

Intérêt

Technique permettant d'implémenter facilement des applications client/serveur en les imaginant comme des applications multi-composant/multi-modules. Le système se charge en suite de fournir des opérations pour démarrer les composants dans différents processus et les faire communiquer.

Inconvénients

- Certains aspects bas-niveau viennent quand même polluer le code (fait qu'une méthode peut être interrompue par une erreur réseau par exemple)
- Les systèmes étant génériques (marchent pour tous les domaines d'application) ils génèrent souvent du code peu efficace (protocole lourd, sérialisation coûteuse en temps/espace/bande passante, insertion de multiples verrous, ...)
- Pleins de standard différents, tous incompatibles entre eux :
  - D-Bus (linux)
  - CORBA (multi-systèmes, mais très lent et complexe)
  - COM (Windows uniquement)
  - XML-RPC (arguments sérialisés en XML, utilise le protocole HTTP, très utilisé pour les services Web)
  - Java RMI

Donc tous les composants de l'application doivent être écrit avec le même framework, souvent dans le même langage.

Principe du RMI en Java

Une application RMI en Java utilisera trois composants :

1. Une (ou plusieurs) classe serveur, qui doivent implémenter l'interface `java.rmi.Remote`
2. Une (ou plusieurs) classe client qui appellent des méthodes de la classe serveur. Les arguments et types de retours doivent implémenter l'interface `java.io.Serializable`
3. Un programme spécial `rmiregistry` fourni avec l'environnement Java qui s'occupe de gérer les connexions et auprès duquel :
   - Le serveur peut enregistrer des objets appelables depuis le réseau
   - Le client peut récupérer des objets dont le code se trouve de l'autre côté du réseau, sur le serveur

Définitions

Proxy : Objet P possédant les même méthodes qu'un Objet O et pour lesquelles chaque appel à une méthode de P fait appel à la méthode correspondante sur O.

Dans le cadre de Java RMI :

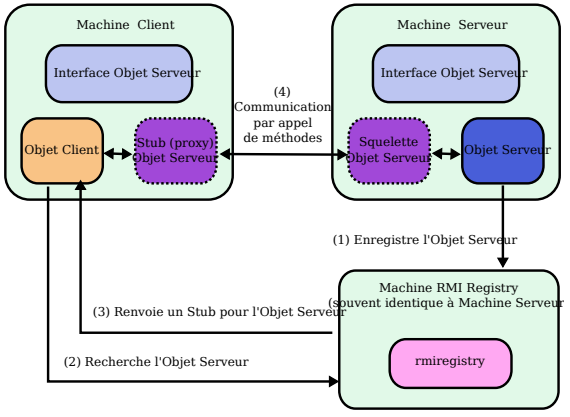
Stub : Objet proxy crée côté client, ayant la même interface que l'Objet Serveur distant et qui transmet les appels de méthodes sur le réseau

Skeleton (squelette) : Objet crée côté serveur qui reçoit les données du réseau et appel les méthodes de l'Objet Serveur avec les paramètres dé-sérialisées

Étapes de la mise en place d'un système RMI

1. Démarrage de la `rmiregistry`.
2. Démarrage du code serveur. Ce dernier instancie l'Objet Serveur et l'enregistre auprès de la `rmiregistry`. Un squelette est crée et placé en attente de connexions.
3. Démarrage du code client. Ce dernier contacte la `rmiregistry` pour demander une implémentation de l'Objet Serveur
4. La `rmiregistry` renvoie un stub de l'Objet Serveur
5. Le client appelle une méthode du stub. Le stub sérialise les arguments et les envoie par le réseau au squelette. Le squelette dé-sérialise les arguments et appelle la méthode de l'Objet Serveur avec.
6. S'il y a un résultat, il suit le chemin inverse (Objet Serveur → Squelette → Stub → Code client). Si le code Serveur lève une exception elle est sérialisée et renvoyée comme un résultat, puis relancée par le stub.

Récapitulatif



RMI en Java (interface)

On prend l'exemple d'un Objet serveur permettant de faire `+ 42` sur un entier donné en argument.

Il faut d'abord spécifier le code du serveur dans une interface, qui doit être connue du client et du serveur :

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

interface IAdd42 extends Remote {
  public Integer add42(Integer i) throws RemoteException;
}
```

RMI en Java (code serveur)

L'objet serveur doit implémenter l'interface ainsi que du code permettant de s'enregistrer auprès de la `rmiregistry`

```
public class Add42 implements IAdd42 {

public Integer add42(Integer i) { return i + 42; }

public static void main(String[] args) {

  if (System.getSecurityManager() == null)
   System.setSecurityManager(new SecurityManager());
  try {
   String name = "MonObjetPlus42";
   IAdd42 adder = new Add42();
   IAdd42 stub = (IAdd42) UnicastRemoteObject.exportObject(adder, 0);
   Registry registry = LocateRegistry.getRegistry();
   registry.rebind(name, stub);
   System.out.println("Ok");
   } catch (Exception e) { … };
  }
}
```

RMI en Java (code serveur)

L'objet doit implémenter l'interface définie

```
public class Add42 implements IAdd42 {
```

Il faut qu'un `SecurityManager` soit installé (il n'y en a pas par défaut). Sans cela, les objets ne peuvent pas faire d'appels réseau

```
if (System.getSecurityManager() == null)
  System.setSecurityManager(new SecurityManager());
```

On crée un nom ainsi qu'un instance de l'objet réel

```
String name = "MonObjetPlus42";   //Création d'un nom
IAdd42 adder = new Add42();         //Création de l'objet réel
```

On crée l'objet stub qui sera envoyé au client, on récupère une référence vers la `rmiregistry` et on y enregistre le stub :

```
IAdd42 stub = (IAdd42) UnicastRemoteObject.exportObject(adder, 0);
Registry registry = LocateRegistry.getRegistry();
registry.rebind(name, stub);
```

RMI en Java (code client)

Le client récupère (auprès de la `rmiregistry`) le `stub` de l'objet serveur dont il connaît l'interface :

```
import …
public class Client {

public static void main(String [] args) {
   if (System.getSecurityManager() == null)
   System.setSecurityManager(new SecurityManager());

   try {
    Registry registry = LocateRegistry.getRegistry("machine.foo.com");
    String name = "MonObjetPlus42";
    IAdd42 addstub = (IAdd42) registry.lookup(name);

    System.out.println( "Un et quarante deux font : ");
    System.out.println(addstub.add42(new Integer(1)));
    } catch (Exception e) { … };
  }
}
```

RMI en Java (code client)

Le client doit aussi mettre en place un *security manager* :

```
if (System.getSecurityManager() == null)
System.setSecurityManager(new SecurityManager());
```

Le client récupère une référence vers la `rmiregistry` se trouvant sur une machine distante :

```
Registry registry = LocateRegistry.getRegistry("machine.foo.com");
```

Le client utilise le nom pour récupérer le stub de l'objet serveur

```
String name = "MonObjetPlus42";
IAdd42 addstub = (IAdd42) registry.lookup(name);
```

Le client peut ensuite utiliser le stub comme si c'était l'objet réel

```
System.out.println(addstub.add42(new Integer(1)));
```

Lancement des applications

Le programme `rmiregistry` doit avoir accès au code de l'objet serveur ainsi qu'à l'interface.

```
$ rmiregistry -J-Djava.rmi.server.codebase=file:/chemin/vers/les/classes
```

On peut spécifier un numéro de port particulier pour les communications (par défaut 1099) ainsi qu'un autre protocole pour les classes (`http` par exemple). Attention, le programme `rmiregistry` ne rend pas la main et doit être lancé avant le démarrage du serveur.

Les codes client et le serveur doivent être exécuté avec une certaine politique de sécurité :

```
java -Djava.security.policy=file.policy Add42
  …
java -Djava.security.policy=file.policy Client
```

Dans notre cas, le fichier `file.policy` contiendra :

```
grant { permission java.security.AllPermission; };
```

Politique de sécurité

Java implémente un mécanisme complexe de politique de sécurité. Par défaut, une application « normale » lancée par un utilisateur a tous les droits. Certaines classes d'application n'ont cependant aucune permissions par défaut (impossibilité d'accéder au réseau, aux périphériques, aux fichiers, ...).

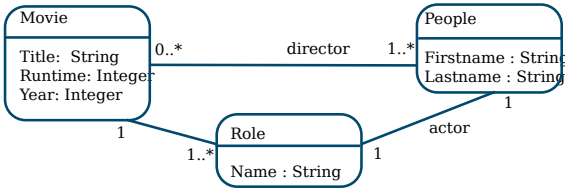
→ C'est le cas des applets Java (même si de moins en moins utilisés)

→ Des applications, client et serveur, utilisant les RMI. Le serveur reçoit des objets du clients (donc potentiellement dangereux). Le client reçoit un stub du serveur (potentiellement dangereux aussi).

Plan

Une base de données de films

On se place dans le cadre d'une application permettant d'accéder à une base de données de films. Le schéma *logique* de la base est représenté par le diagramme UML suivant :



- Un film peut avoir plusieurs réalisateurs, et au moins un;
- Un film a au moins un « acteur ». Un acteur est composé d'un rôle *pour ce film* et d'une personne. Attention, le *Role* de Luke Skywalker est différent pour les films Starwars IV, V, VI (même si c'est le même acteur et le même nom de rôle).

Encodage « naturel » en SQL (1/2)


```
CREATE TABLE PEOPLE (pid INTEGER, firstname VARCHAR(30),
             lastname VARCHAR(30),
             PRIMARY KEY(pid));

CREATE TABLE MOVIE (mid INTEGER, title VARCHAR(90) NOT NULL,
            year INTEGER NOT NULL,
            runtime INTEGER NOT NULL, rank INTEGER NOT NULL,
            PRIMARY KEY (mid));

CREATE TABLE ROLE (mid INTEGER, pid INTEGER, name VARCHAR(70),
           PRIMARY KEY(mid, pid, name),
           FOREIGN KEY (mid) REFERENCES MOVIE,
           FOREIGN KEY (pid) REFERENCES PEOPLE);

CREATE TABLE DIRECTOR (mid INTEGER, pid INTEGER,
           PRIMARY KEY (mid, pid),
           FOREIGN KEY (mid) REFERENCES MOVIE,
           FOREIGN KEY (pid) REFERENCES PEOPLE);
```

Encodage « naturel » en SQL (2/2)

- entités $\Longrightarrow$ tables de données
- relations $\Longrightarrow$ *tables de jointure & contraintes de clé*
- (économie d'une table de données pour ROLE.name)

Remarque

On utilisera ce schéma à partir du TP 7 pour et le TP noté de la séance 10.

Ici on veut faire une mini-application simplifiée pour rechercher toutes les personnes qui contiennent une chaîne donnée dans leur nom, trié par ordre alphabétique.

Création du modèle

On va créer deux classes Java pour le modèle :

`Person` : Une classe représentant une personne, avec son nom et son prénom

`PersonDB` : Une classe encapsulant la connexion à la base et permettant de renvoyer l'ensemble (Java) de toutes les personnes de la base ayant une certaine chaîne dans son nom.

```
public class Person {
   private final String firstname;
   private final String lastname;

   public Person(String f, String l) {
       this.firstname = f;
       this.lastname = l;
   }
   public String getFirstname() { return firstname; }
   public String getLastname() { return lastname; }
}
```

Classe `Person`

- Doit être publique
- Doit posséder des *getter* publiques pour les attributs qu'on veut afficher dans la vue

Le *getter* pour une propriété `foo` est une méthode publique `getFoo()`

Classe PersonDB et rappels JDBC


```
public class PersonDB {
  Connection cnx;
  public PersonDB() {
    Class.forName("org.postgresql.Driver");
    cnx = DriverManager.getConnection(
                                    "jdbc:postgresql://host:port/base",
                                    "username", "password");
  }
  public Vector<Person> getPersons (String s) throws SQLException {
      Vector<Person> res = new Vector<>();
      Statement st = cnx.createStatement();
      ResultSet r = st.executeQuery("SELECT * FROM PEOPLE "
                                     + " WHERE LASTNAME LIKE "
                                     + " '%" + s + "%'");
      while (r.next())
         res.add(new Person<>(r.getString("FIRSTNAME"),
                                    r.getString("LASTNAME")));
      return res;
  }
 }
```

Connexion à une base


```
Class.forName("org.postgresql.Driver");
connection =
DriverManager.getConnection("jdbc:postgresql://host:port/" + base,
      username, password);
```

◆ On importe dans la JVM courante le classe qui code le driver vers une base de donnée (ici Postgresql)

◆ Le code de cette classe doit se trouver dans un *.jar* ou *.class* accessible depuis le `CLASSPATH`

◆ La classe *DriverManager* maintient une *Map* entre chaîne de caractères (`"jdbc:postgresql"`) et classe (`org.postgresql.Driver`)

◆ La méthode `getConnection` utilise le préfixe de l'URL de connexion pour savoir quel driver utiliser.

Exécution de requêtes

- On crée un objet *Statement*
- L'évaluation d'une requête se fait via *executeQuery* sur le *Statement*.
- Un *ResultSet* implémente une interface d'itérateur, initialement positionné *avant* la première ligne de résultats.
- La méthode *next* avance dans l'itérateur et renvoie vrai tant qu'on est sur un résultat.
- On accède à la colonne voulue avec *getType*. On doit donner le type Java correspondant au type SQL de la colonne. On peut accéder aux colonnes par numéro (à partir de 1) ou par nom.

Plan

Étude de cas

On utilise ici comme support Java et JDBC, mais les concepts sont applicables à d'autres langages

Considérons un site Web réaliste diffusant des petites annonces. Une annonce a un titre, un auteur, une catégorie et une description. Les annonces peuvent être stockées dans une table de la manière suivante :

```
CREATE TABLE ADS (INTEGER id PRIMARY KEY,
                  VARCHAR(50) title,
                  VARCHAR(50) author,
                  INTEGER cid REFERENCES CATEGORY(id),
                  TEXT description);
```

Recherche par titre

On souhaite rechercher toutes les annonces dont les titres contiennent une sous-chaîne donnée (fournie par l'utilisateur)

```
Statement stmt = conn.createStatement();
ResultSet res =
      stmt.executeQuery ( "SELECT id FROM ADS WHERE title LIKE '%"
                          + pattern + "%'");
…
```

Le code ci-dessus est fragile (`pattern` pourrait contenir des caractères non échappés tels que `'`) et vulnérable. En effet `pattern` pourrait être `"x'; DROP TABLE ADS; --"` :

```
SELECT * FROM ADS WHERE title LIKE '%x';
DROP TABLE ADS;
-- %'
```

Solution simple : `PreparedStatement`

On peut pré-compiler un ordre SQL en y laissant des paramètres à remplir plus tard :

```
PreparedStatement stmt = conn.prepareStatement(
           "SELECT id FROM ADS WHERE title LIKE ?");

stmt.setString(1, pattern);
ResultSet res = stmt.executeQuery ()
  …
```

L'ordre est envoyé à la base et compilé au moment de la création. Les valeurs scalaires manquantes sont insérées manuellement avant évaluation.
Cette technique est plus efficace si on réutilise plusieurs fois la même requête avec des paramètres différents.

Recherche multi-critères

L'utilisateur a maintenant accès à un formulaire lui permettant de sélectionner les attributs de l'annonce dans lesquels chercher :

Chercher le terme :

Chercher le terme dans : titre auteur catégorie description

Rechercher

La requête est maintenant de la forme :

```
SELECT id FROM ADS WHERE col1 LIKE … OR … OR coln LIKE …
```

On peut avoir jusqu'à 16 configurations différentes !

Problème des `PreparedStatement`

- non modifiable une fois créés
- requête de taille et forme fixe (on ne peut pas mettre une sous-requête en paramètre)
- créer un `PreparedStatement` pour ne pas le réutiliser est plus coûteux que faire un `Statement` simple

Dans le cas présent, on est en plus dans la situation simple où c'est le même paramètre à insérer à chaque fois. Dans des cas plus complexes, on peut avoir un type par colonne (par exemple `INTEGER` dans la colonne `age` et `VARCHAR` dans la colonne `name`). Mais dans ce cas :

```
…                                   -- pas de condition
… WHERE age = ?                     -- seulement age
… WHERE name = ?                    -- seulement nom
… WHERE name = ? OR age = ?         -- les deux conditions
```

dans le premier cas, pas de paramètre. Dans le deuxième `setInt(1, …)`, dans le troisième `setString(1, …)` dans le quatrième `setString(1, …)` et `setInt(2, …)`.
Le code de génération de requête devient très complexe.

En pratique ?

- Requêtes figées ⟹ `PreparedStatement`
- Requêtes variables ⟹ `Statement` plus validation des entrées

Remarque : les *framework* du type ORM dans des langages compilés souffrent de la lourdeur des encodages de requête (cf. cours 9 requêtes Hibernate/JPA).

Les langages dynamiques + une écriture fonctionnelle du code rendent les choses supportables

```
#pseudo Ruby
        cond = TRUE;
        checkboxes.each |cb| do
             if (cb.isChecked())
              cond = cond.or(like(cb.name, "%" + prefix + "%"))
        end
        result = ads.where(cond);
        …
```