SCIENCES SORBONNE UNIVERSITÉ

**SRCS : Systèmes Répartis Client/Serveur**

# TME 4 : Un premier intergiciel multithreadé

**Objectifs pédagogiques :**

- **Client-Serveur multi-thread**
- **Service**
- **Sensibilisation à l'appel distant**

## Introduction

L'objectif de ce TP est de construire un petit intergiciel multithreadé. Un intergiciel est une couche logicielle intermédiaire entre les gestionnaires de l'infrastructure (système d'exploitation et couches réseau) et la couche applicative. Il facilite la programmation et la communication entre deux entités applicatives. L'aspect multithreadé permet d'avoir des tâches concurrentes pour traiter plus efficacement les requêtes sur le serveur.

## Exercice 1 – Serveur multithreadé et service avec/sans état

**Question 1**

Écrire une interface `Service` qui contient une méthode `void execute(Socket connexion)` et qui définira le traitement de la requête sur ce service pour une connexion de client donnée.

**Question 2**

Définir deux annotations de classe `SansEtat` et `EtatGlobal` qui ne possèdent pas d'attribut. Ces annotations seront utilisées lors de la programmation des classes implantant `Service`.

**Question 3**

Écrire une classe `ServeurMultiThread` qui définira le code du serveur. Cette classe sera composée :

- de deux attributs déclarés `final` :
  - ◇ le port d'écoute (type `int` )
  - ◇ la classe concrète permettant de traiter une requête du service que le serveur gère, c'est-à-dire une classe qui implante `Service`. On utilisera ici la réflexivité de java.
- un constructeur permettant d'initialiser les deux attributs ci-dessus
- une méthode `listen()` qui :

1. instanciera la socket d'écoute
2. se mettra en attente sur la socket d'écoute,
3. à la réception d'une requête, appellera la méthode `getService` (voir ci-dessous) et lancera dans un nouveau thread la méthode `execute` (il est possible d'utiliser la notation lambda pour éviter de définir explicitement une classe implantant `Runnable`).
4. et reviendra à l'étape 2. (boucle infinie)

- une méthode privée `service getService()` qui renvoie un objet de la classe stockée en attribut. Si la classe est annotée `SansEtat` alors il faut envoyer à chaque requête une nouvelle instance de la classe. Si la classe est annotée `EtatGlobal` alors ça sera la même instance de la classe qui doit être renvoyée. Si aucune annotation n'est renseignée, alors une exception `IllegalStateException` est jetée.

**Question 4**

Testez votre code avec la classe de test JUnit `TestServeurMultiThread`.

# Exercice 2 – Programmation de services

Nous souhaitons définir des services distants qui seront offerts par le serveur multithreadé. Nous proposons de définir deux services :

- un service de calculatrice qui offrira 4 opérations arithmétiques sur des entiers :
  - ◇ `add` renverra un entier égal à l'addition de deux entiers donnés
  - ◇ `sous` renverra un entier égal à la soustraction de deux entiers donnés
  - ◇ `mult` renverra un entier égal à la multiplication de deux entiers donnés
  - ◇ `div` renverra un objet comprenant deux entiers qui correspondent au quotient et au reste de la division entre deux entiers donnés. La classe de cet objet sera `ResDiv` et offrira deux méthodes `getQuotient` et `getReste`.
- un service d'annuaire permettant d'associer un nom à une valeur. On considérera que le type du nom et de la valeur est une chaîne de caractères. Ce service offrira 3 opérations :
  - ◇ `lookup` qui pour un nom donné renvoie la valeur associée. Si le nom n'existe pas la chaîne vide sera renvoyée.
  - ◇ `bind` qui permet d'associer une valeur donnée à un nom donné. Cette opération ne renvoie pas de valeur.
  - ◇ `unbind` qui permet de supprimer une entrée de l'annuaire à partir de son nom. Cette opération ne renvoie pas de valeur.

**Question 1**

Définir les interfaces java `Calculatrice` et `Annuaire` pour définir les entêtes des opérations des services définis ci-dessus. Vous coderez la classe `ResDiv` en tant que membre interne statique de l'interface `Calculatrice`.

Afin de pouvoir appeler ces services depuis un client distant, nous allons définir un protocole applicatif qui fonctionne en mode requête-réponse. Ainsi une fois que le client a établi la connexion avec le serveur, il enverra 1) le nom de la méthode à appeler, 2) la valeur des arguments de l'appel 3) se mettra en attente de la réponse du serveur.

Quant au serveur, une fois avoir 1) lu le nom de la méthode voulue et 2) les arguments de l'appel, 3) il appellera la méthode pour effectuer le calcul et 4) renverra le résultat de la méthode au client en guise de réponse. Le serveur devra renvoyer quoiqu'il arrive une réponse même si la méthode appelée ne renvoie pas de résultat (on définira une classe `VoidResponse` pour modéliser ce cas). En cas de nom respect de ce protocole (ex : une méthode qui n'existe pas), un objet de type `MyProtocolException` (classe à définir) sera renvoyé au client en guise de réponse.

**Question 2**

Définir les classes `CalculatriceService` et `AnnuaireService` qui implantent à la fois l'interface `Service` définie dans l'exercice précédent et respectivement les interfaces `Calculatrice` et `Annuaire`. La méthode `execute` sera en charge d'implanter le protocole applicatif côté serveur. Ne pas oublier que vous êtes dans un contexte multithreadé, une synchronisation s'impose donc pour les services à état.

**Question 3**
Testez votre code avec la classe de test JUnit `TestService`.

# Exercice 3 – Abstraction de la couche Socket : vers l'appel distant

On a pu remarquer que l'appel de service à distance était assez fastidieux à écrire car cela nécessitait d'utiliser les flux d'entrée/sortie pour envoyer les paramètres et recevoir les résultats. Il serait intéressant de pouvoir s'abstraire de la couche réseau en permettant au client d'appeler directement les méthodes de l'interface du service qu'il souhaite utiliser tout en assurant que l'exécution de la méthode se fera sur le serveur. Il est donc nécessaire d'abstraire le protocole applicatif défini dans l'exercice précédent afin de pouvoir utiliser ce mécanisme à n'importe quelle interface `Service`.

**Question 1**
Définir une interface `AppelDistant` qui étend de l'interface `Service` et y coder une implantation par défaut de la méthode `execute` qui définit le protocole applicatif coté serveur. Utiliser l'API de reflexivité de Java.

**Question 2**
Définir les classes `AnnuaireAppelDistant` et `CalculatriceAppelDistant` qui implantent l'interface `AppelDistant` et respectivement les interfaces `Annuaire` et `Calculatrice` (copier les implantations des méthodes définis dans l'exercice précédent).

Nous allons maintenant définir la partie cliente. Pour cela il faut définir une classe qui permet de stocker les informations réseau (à savoir le nom de la machine hôte et le port du processus serveur) et une méthode permettant d'implanter le protocole côté client. Ensuite pour chaque service que l'on souhaite définir, il faudra créer une classe qui implante l'interface du service mais qui jouera un rôle de mandataire (Design Pattern *proxy*) entre le code client et le serveur.

**Question 3**
Définir une classe abstraite `ClientProxy` qui :
- possède deux attributs immuables : une chaine de caractères pour désigner le nom de la machine serveur et un entier pour le port.
- offre des getters sur les attributs
- offre une méthode `Object invokeService(String name, Object[] params)` qui ouvre une socket avec le serveur, envoie au serveur le nom puis l'ensemble des paramètres passés en arguments et renvoie l'objet résultant de l'appel. Si l'objet résultat est `MyProtocolException` alors la méthode la jettera.

**Question 4**
Définir les classes `AnnuaireProxy` et `CalculatriceProxy`qui étendent toutes les deux `ClientProxy` et qui implantent respectivement les interfaces `Annuaire` et `Calculatrice`. Les méthodes feront appel à `invokeService`.

**Question 5**
Testez votre code avec la classe de test JUnit `TestAppelDistant`.