Master 1
SD - Systèmes Distribués
Travaux Pratiques

Éric Leclercq, Annabelle Gillet
Eric.Leclercq@u-bourgogne.fr
Annabelle.Gillet@u-bourgogne.fr

Révision : 5 septembre 2022

Résumé

Ce document contient l'ensemble des exercices de TP du module de Systèmes Distribués. Tous les exercices sont normalement à réaliser et sont regroupés sous formes de chapitres thématiques s'étalant le plus souvent sur plusieurs séances. Les exercices qui illustrent les concepts du cours sont notés avec une étoile (*). Dans ces exercices les questions techniques et les questions plus générales qui leur sont associées sont essentielles pour la maîtrise des concepts. Le GitHub du cours contient des exemples de programmes de TD et TP (`https://github.com/EricLeclercq`)

Table des matières

Notes :

Plusieurs versions de J2SE (jdk) sont installées sur les serveurs et sur l'ensemble des stations Linux Debian du département IEM. Les installations de la version officielle (SUN ou ORACLE) résident dans les répertoires `/usr/local` des stations de travail ou des serveurs. Afin de pouvoir utiliser la version choisie, vous devez positionner les variables d'environnement `PATH` et `JAVA_HOME` avec la commande `export`. Ces variables ne sont actives que dans le shell courant. Il n'est pas conseillé de les mettre dans votre `.bashrc` car vous pouvez avoir besoin de les changer assez souvent. Le système d'exploitation GNU/Linux fournit une version libre du jdk (openjdk) qui peut avoir un comportement légèrement différent de celui d'ORACLE/SUN.

Pour fixer les variables utiliser les commandes shell suivantes, en prenant garde à regarder (via un `ls -al /usr/local`) le nom réel des répertoires sous `/usr/local` :

```
export PATH=/usr/local/jdk-X.Y/bin:$PATH
export JAVA_HOME=/usr/local/jdk-X.Y
```

Pour compiler un code source java dans une version spécifique du JDK, deux options du compilateurs sont proposées : `-target` génère le byte-code dans la version spécifiée de Java, `-source` permet de vérifier la compatibilité de version du code source.

```
javac -target 1.4 -source 1.4 serveurImpl.java
```

Nom local	OS/Architecture	JDK
butor.iem	Solaris 10 x64	JDK 1.6
eluard.iem	Solaris 10 T1 Sparc	JDK 1.5
MI10X et Linux	Debian 10	JDK 1.4, 1.5, 1.6, 1.7, 1.8
aragon	Debian 10	JDK 1.4, 1.5, 1.6, 1.7, 1.8

Afin d'observer et de comprendre précisément le comportement de Java RMI, ne pas utiliser d'environnement de développement mais un simple éditeur comme vi, xemacs, nedit, bluefish ou atom et une exécution en ligne de commande.

Le service `rmiregistry` écoute par défaut sur le port 1099. Si vous partagez une machine avec d'autres binômes, utilisez un autre port ou bien veillez à ne pas donner le même nom à vos objets distribués. Pour lancer le service avec un autre port `rmiregistry [port]`.

1 Java RMI

Exercice 1. Prise en main de Java-RMI (*)

Le but de ce premier exercice (application directe du cours) est de prendre en main l'environnement Java-RMI et de se familiariser avec le processus de développement d'applications objets distribués (ici uniquement la compilation et l'exécution). Réaliser le programme client serveur `HelloWorld` vu en cours.

1. Tester l'ensemble sur une seule machine.
2. Observer la génération du stub et du squelette avec les JDK 1.4, et 1.6 (ou $\geq$ 1.6), que constatez vous ? Quelles sont les évolutions d'un JDK à l'autre ? Compléter vos observation en utilisant les options du compilateurs `rmic` telle que `-keep`
3. Séparer le code du client et celui du serveur dans deux répertoires, quels sont les fichiers nécessaires à chacun pour la compilation et pour l'exécution ?
4. Recompiler client et serveur avec la même version de JDK, lancer le serveur et le client sur deux machines différentes de même architecture.
5. Répéter l'opération précédente sur deux machines d'architecture différentes en prenant garde d'activer la même version de la machine virtuelle via la variable `PATH` de votre environnement. (travail personnel)
6. Peut-on utiliser 3 machines afin d'avoir un référentiel d'interfaces séparé, que constatez-vous ?
7. Expérimenter la méthode `LocateRegistry`, résumer son fonctionnement.
8. Créer une expérience pour mesurer le temps d'invocation d'une méthode à distance.

Exercice 2. Patron de conception : `Object Factory` distribué (*)

Réaliser un programme de simulation de gestion de compte bancaires. Vous devez obligatoirement adopter une approche objet et définir une classe `compte` permettant la création d'un compte, proposant les opérations de dépôt, retrait, affichage du solde et permettant un archivage des opérations effectuées. Le programme s'organisera en plusieurs packages : client, serveur et interface.

— Lancer plusieurs clients simultanément. Que se passe-t-il ? Comment est (doit-être) gérée la concurrence ?
— Comment réaliser une application qui gère 200 000 comptes ? Expérimentez le pattern de développement *Object Factory*.
— Que se passe t-il côté client si le serveur est arrêté puis relancé ? Quelles solutions peut-on envisager pour résoudre ce problème ?

Exercice 3. Patron de conception : `Bag Of Tasks` (*)

Réaliser un programme de *bag of tasks* avec le pattern *object factory*. Vous devez obligatoirement définir une classe `Task` contenant une méthode `run` permettant d'exécuter la tâche. Le *bag of tasks* aura un méthode `addResult` qui prendre une tâche en paramètre, afin d'intégrer le résultat d'une tâche.

En guise d'implémentation de la tâche, utiliser le calcul de nombres premiers.

1. Réaliser les programmes, attention les tâches s'exécutent sur les clients.
2. Comparer la durée d'exécution et le speedup avec les autres implémentations réalisées en TD.

3. Cette solution est elle plus facilement scalable ? Qu'en est il de la transparence à la localisation, de l'autonomie des participants ? Comparer avec les autres implémentations réalisées en TD.