INF4410 – Systèmes répartis et infonuagique

TP1 – Appels de méthodes à distance

Chargés de laboratoire :

Simon Delisle

François Doray

9 septembre 2014

École Polytechnique de Montréal

Introduction

Ce premier travail pratique de INF4410 a pour but de donner à l'étudiant une expérience de première main avec les appels de procédure à distance. La technologie utilisée dans ce TP est le Java Remote Method Invocation (RMI), qui permet de faire des appels de méthodes entre deux machines virtuelles Java pouvant s'exécuter sur deux hôtes différents. En première partie, vous expérimenterez avec une paire client/serveur minimaliste qui vous est fournie. La deuxième partie consiste à implémenter un système de fichiers partagés simple basé sur l'invocation de méthodes à distance.

Remise

- Méthode: Par Moodle, un seul membre de l'équipe doit remettre le travail mais assurez-vous d'inclure les deux matricules dans le rapport et le nom du fichier remis.
- Échéance: lundi le 6 octobre 2014, avant 16h, voir le plan de cours pour les pénalités de retard.
- Format: Une archive au format .tar.gz contenant votre code, votre rapport et vos réponses aux questions. Pour le code, la remise d'un projet Eclipse (le projet seulement, pas le *workspace*) est préférable.

Vous pouvez faire le laboratoire seul, mais le travail en binôme est encouragé. On vous demande d'inclure un *README* expliquant comment exécuter votre TP. Après la remise, il se peut que le chargé de laboratoire vous demande de faire une démonstration de votre travail, conservez donc votre code.

Barème

Partie 1: 8 points (4 points par question)
Partie 2: 12 points
Respect des exigences et bon fonctionnement: 9 points
Clarté du code et commentaires: 3 points
Total: 20 points, valant pour 10% de la note finale du cours.

Jusqu'à 2 points peuvent être enlevés pour la qualité du français.

Documentation

Le tutoriel sur le site d'Oracle est sans doute la meilleure façon de débuter. Il fait pas à pas toutes les étapes pour faire un simple client/serveur avec RMI. Autrement, référez-vous au besoin aux livres mentionnés dans le plan de cours.

Oracle Trail: RMI: http://docs.oracle.com/javase/tutorial/rmi/index.html

Partie 1

On vous demande de comparer la performance d'un appel de fonction normal, d'un appel de fonction RMI vers un processus serveur roulant sur la même machine et d'un appel RMI vers un serveur distant. Évaluez également la performance pour des arguments de plusieurs tailles (10^x octets pour x=1 à 8). Faites un graphique illustrant le temps total des appels pour les trois méthodes en fonction de la taille des arguments passés. Assurez-vous de prendre un échantillonnage assez grand.

Pour vous aider, un code de base vous est fourni. Référez-vous à l'annexe pour savoir comment l'utiliser et n'hésitez pas à le modifier pour l'adapter à vos besoins. Vous aurez à démarrer une instance du serveur pour faire vos tests locaux. Pour les tests vers un serveur distant, une instance du serveur est fonctionnelle sur l'hôte *secretaire.dorsal.polymtl.ca* (132.207.72.58).

Placez le code que vous avez utilisé pour faire vos tests dans l'archive que vous remettez.

Question 1: Présentez votre graphique, commentez et expliquez les résultats obtenus. Donnez un exemple de bon et un de moins bon cas d'utilisation de Java RMI (ou un autre système de RPC semblable).

Question 2: Expliquez l'interaction entre les différents acteurs (client, serveur et registre RMI) à partir du tout début de l'exécution. Ainsi, à partir du moment où on lance le serveur jusqu'à l'appel de la fonction à distance par le client, décrivez toutes les communications qui ont lieu entre ces acteurs. Faites le lien entre vos explications et le code de l'exemple fourni.

Partie 2

Vous devez implémenter un serveur et un client qui communiqueront à l'aide d'appels RMI pour faire un petit système de fichiers à distance. Le serveur doit exposer aux clients les six opérations décrites à la page suivante.

Le client exposera les commandes `list` et `get` pour consulter les fichiers du serveur. La commande `create` permettra de créer un nouveau fichier. Pour modifier un fichier, l'utilisateur devra d'abord le verrouiller à l'aide de la commande `lock`. Il appliquera ensuite ses modifications au fichier local et les publiera sur le serveur à l'aide la commande `push`. Notons qu'un seul client peut verrouiller un fichier donné à la fois. Aussi, la commande `lock` téléchargera toujours la dernière version du fichier afin d'éviter que l'utilisateur n'applique ses modifications à une version périmée.

Quelques notes:

- Comme le serveur sera en mesure d'accepter des appels de plusieurs clients de façon simultanée, vous devez prendre les précautions nécessaires pour assurer la cohérence des données dans les structures partagées au sein du serveur.
- Tous les fichiers sont au même niveau, c'est-à-dire qu'il n'y a pas de dossiers ou d'arborescence à gérer.
- Du côté du serveur, les fichiers peuvent simplement être stockés en mémoire, par exemple dans des byte[] (les fichiers sont donc perdus lorsqu'on ferme le programme du serveur).
- Lorsque le client met à jour un fichier qu'il possédait déjà (`get`), le nouveau contenu écrase simplement l'ancien.

Interface du serveur

Les méthodes que le serveur doit exposer sont les suivantes. Vous avez la liberté de choisir les types pour les paramètres et valeurs de retour. Ceux-ci peuvent être des types primitifs Java, des classes standards ou des classes de votre conception.

Prototype	Description
`generateclientid()`	Génère un identifiant unique pour le client. Celui-ci est sauvegardé dans un fichier local et est retransmis au serveur lors de l'appel à `lock()` ou `push()`. Cette méthode est destinée à être appelée par l'application client lorsque nécessaire (il n'y a pas de commande `generateclientid` visible à l'utilisateur).
`create(nom)`	Crée un fichier vide sur le serveur avec le nom spécifié. Si un fichier portant ce nom existe déjà, l'opération échoue.
`list()`	Retourne la liste des fichiers présents sur le serveur. Pour chaque fichier, le nom du fichier et l'identifiant du client possédant le verrou (le cas échéant) sont retournés.
`get(nom)`	Demande au serveur d'envoyer la dernière version du fichier spécifié. Le fichier est écrit dans le répertoire local courant.
`lock(nom, clientid)`	Demande au serveur de verrouiller le fichier spécifié. La dernière version du fichier est écrite dans le répertoire local courant. L'opération échoue si le fichier est déjà verrouillé par un autre client.
`push(nom, contenu, clientid)`	Envoie une nouvelle version du fichier spécifié au serveur. L'opération échoue si le fichier n'avait pas été verrouillé par le client préalablement. Si le push réussit, le contenu envoyé par le client remplace le contenu qui était sur le serveur auparavant et le fichier est déverrouillé.

Interface du client

Le client doit prendre la forme d'un simple outil en ligne de commande offrant les commandes create, list, get, lock et push qui correspondent aux opérations de même nom exposées par le serveur. Voici des exemples d'utilisation:

Exemple simple:

```
client$ ./client list
0 fichier(s)
client$ ./client create monfichier
monfichier ajouté.
client$ ./client list
* monfichier   non verrouillé
1 fichier(s)
client$ ./client get monfichier
monfichier synchronisé
client$ ./client push monfichier
opération refusée : vous devez verrouiller d'abord verrouiller le fichier.
Client$ ./client lock monfichier
monfichier verrouillé
client$ ./client list
* monfichier   verrouillé par client 1
client$ ./client push monfichier
monfichier a été envoyé au serveur
client$ ./client list
* monfichier   non verrouillé

client$ ls
... monfichier ...
```

Exemple avec "conflit":

L'exemple suivant montre le comportement lorsque deux clients tentent de modifier le même fichier en même temps.

Client 1	Client 2
client1$ ./client create monfichier monfichier ajouté. **client1$ ./client lock monfichier** monfichier verrouillé (Modifs à monfichier)	
	client2$ ./client create monfichier monfichier existe déjà. **client2$ ./client lock monfichier** monfichier est déjà verrouillé par client 1
client1$ ./client push monfichier monfichier a été envoyé au serveur	
	client2$ ./client lock monfichier monfichier verrouillé (Modifs à monfichier) **client2$ ./client push monfichier** monfichier a été envoyé au serveur

Comme on invoque le client à plusieurs reprises pour effectuer les différentes opérations, il ne peut simplement stocker les fichiers en mémoire comme le fait le serveur, qui lui est un démon qu'on part et qu'on laisse rouler. Le client utilisera donc le dossier à partir duquel il est invoqué pour stocker ses fichiers.

Annexe

Cette annexe montre comment faire fonctionner le code fourni en exemple. Une fois le code extrait, l'arborescence devrait ressembler à:

```
├── build.xml
├── client
├── policy
├── server
└── src
    └── ca
        └── polymtl
            └── inf4402
                └── tp1
                    ├── client
                    │   ├── Client.java
                    │   └── FakeServer.java
                    ├── server
                    │   └── Server.java
                    └── shared
                        └── ServerInterface.java
```

1. Compilez avec la commande *ant*..
2. Démarrez le registre RMI avec la commande *rmiregistry* à partir du dossier *bin* de votre projet. Ajoutez un & pour le détacher de la console.
   Chemin complet vers rmiregistry dans les labos de l'école: `/opt/java/jdk8.x86_64/bin/rmiregistry`.

Des scripts vous sont fournis pour le serveur et le client, puisque des lignes d'une longueur significative sont nécessaires pour les démarrer.

3. Démarrez le serveur avec le script *server (./server* ou *bash server*).
4. Lancez le client avec le script *client*. Le script passe les arguments qu'il reçoit au programme Java, donc vous pouvez le réutiliser pour les commandes de la partie 2. Si un argument est donné au client, il l'utilise comme nom d'hôte pour tenter de faire un appel RMI à distance, donc vous pouvez faire *./client 132.207.72.58*.