INF4410 – Systèmes répartis et infonuagique

TP1 – Appels de méthodes à distance

Chargé de laboratoire :

Houssem Daoud

12 Septembre 2016

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. Dans la première partie, on va analyser le temps de réponse des appels RMI en fonction de la taille des arguments. La deuxième partie consiste à implémenter un système de fichiers partagés simple basé sur l'invocation de méthodes à distance. Ce TP a aussi pour but de vous familiariser avec l'infrastructure Openstack.

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 07 Octobre 2016, 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.

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

Le but de cette partie est 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.

Pour effectuer les tests de performance, un code de base vous est fourni (Référez-vous à l'annexe pour savoir comment l'utiliser). On doit lancer deux instances du serveur : une instance **locale** qui s'exécute sur la même machine du client et une instance **distante** qui s'exécute sur une machine différente.

Une machine virtuelle déployée dans un nuage informatique va être utilisée comme étant un serveur distant.
- Connectez vous à l'interface graphique de OpenStack en utilisant votre compte utilisateur. Créez une machine virtuelle avec l'image ***INF4410-Ubuntu-trusty-mini*** et la flavor ***INF4410-mini*** et associez une adresse IP flottante à cette machine***.*** *La section Annexe explique en détails les étapes nécessaires à la réalisation de cette partie.*

- Créez un groupe de sécurité permettant les accès TCP et SSH à la machine virtuelle depuis l'extérieur du nuage. (Projet → Accès et Sécurité → Groupes de sécurité)

- Démarrez les deux instances du serveur sur la machine virtuelle et sur la machine du laboratoire, puis lancez le client depuis la machine du laboratoire en utilisant la commande suivante : ***./client ip_flottante*** , le paramètre *ip_flottante* est l'adresse IP flottante de la machine virtuelle créée.

Question 1: On se propose d'étudier l'impact de la taille des arguments sur la performance des appels RMI.

On veut comparer le temps d'exécution d'une fonction vide qui prend en paramètre des arguments de tailles 10^x octets, x variant entre 1 et 7. Adaptez le code source fourni à ce besoin. Le client doit prendre le paramètre x comme second argument.

Placez le code que vous avez utilisé pour faire vos tests et les tableaux de résultats dans l'archive que vous remettez.

Faites un graphique (échelle logarithmique) illustrant le temps total des appels pour les trois méthodes [*appelNormal*, *appelRMILocal*, *appelRMIDistant*] en fonction de la taille des arguments passés. Recommencez l'expérience si vous jugez que les valeurs trouvées sont anormales.

Commentez et expliquez les résultats obtenus.

Discutez les avantages et les inconvénients de Java RMI

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. Il faut citer les classes et les méthodes responsables de chaque interaction.

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 sept opérations décrites à la page suivante.

Le client peut lister les fichiers du serveur en utilisant la commande **list**. Pour récupérer le contenu d'un fichier donné, il peut utiliser la commande **get** qui crée une copie locale du fichier distant. La commande **create** permet à l'utilisateur de créer nouveau fichier sur le serveur.

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.

Le transfert du contenu d'un fichier sur le réseau est très coûteux en terme de ressources. Pour diminuer les transferts inutiles, on va utiliser les sommes de contrôle (*checksum)* afin de comparer le contenu des fichiers avant de les envoyer.
Le client doit garder une somme de contrôle MD5 pour chaque fichier reçu et la fournir au serveur lors de l'appel des commandes **get** et **lock.** Si le client possède la même somme de contrôle que celle du serveur, ce dernier retourne une valeur nulle pour indiquer que le fichier du client est à jour.

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 et l'identifiant du client possédant le verrou (le cas échéant) est retourné.
`syncLocalDir()`	Permet de récupérer les noms et les contenus de tous les fichiers du serveur. Le client appelle cette fonction pour synchroniser son répertoire local avec celui du serveur. Les fichiers existants seront écrasés et remplacés par les versions du le serveur.
`get(nom, checksum)`	Demande au serveur d'envoyer la dernière version du fichier spécifié. Le client passe également la somme de contrôle du fichier qu'il possède. Si le client possède la même somme de contrôle que celle qui est présente sur le serveur, celui-ci doit retourner une valeur nulle au client pour lui indiquer que son fichier est à jour et éviter un transfert inutile. Si le client ne possède pas encore ce fichier, il doit spécifier une somme de contrôle de -1 pour forcer le serveur à lui envoyer le fichier. Le fichier est écrit dans le répertoire local courant.

`lock(nom, clientid, checksum)`	Demande au serveur de verrouiller le fichier spécifié. La dernière version du fichier est écrite dans le répertoire local courant (la somme de contrôle est aussi utilisée pour éviter un transfert inutile) . 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, syncLocalDir, 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 fichier1
fichier1 ajouté.
client$ ./client create fichier2
fichier2 ajouté.
client$ ./client list
* fichier1   non verrouillé
* fichier2   non verrouillé
2 fichier(s)
client$ ./client get fichier1
fichier1 synchronisé
client$ ./client push fichier1
opération refusée : vous devez verrouiller d'abord verrouiller le fichier.
Client$ ./client lock fichier1
fichier1 verrouillé
client$ ./client list
* fichier1   verrouillé par client 1
* fichier2   non verrouillé
client$ ./client push fichier1
fichier1 a été envoyé au serveur
client$ ./client list
* fichier1   non verrouillé
* fichier2   non verrouillé
client$ ls
* fichier1
client$ syncLocalDir
client$ ls
* fichier1
* fichier2
```

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

Description du code fourni

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
            └── inf4410
                └── 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.

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* 54.153.69.164

Accès au nuage

Toutes les interactions avec le nuage se font via une interface Web.

URL: http://pingouin.info.polymtl.ca/dashboard

Chaque équipe dispose d'un compte d'utilisateur. Les noms d'utilisateur et mots de passe seront distribués lors de votre première séance de TP. Vous pouvez aussi envoyer un courriel au chargé de laboratoire en spécifiant les matricules de tous les membres de votre équipe.

Création d'une machine virtuelle

La création d'une nouvelle machine virtuelle peut se faire directement depuis l'interface web de Openstack (Calcul → Instances → Lancer une instance). Il faut choisir une image et un gabarit. Dans ce TP, on va utiliser l'image **INF4410-Ubuntu-trusty-mini** avec le gabarit **INF4410-mini**.

Toutes les instances créées pour ce travail pratique doivent être connectées au réseau « reseau-pour-tous »

Pour accéder à une machine virtuelle, il faut la configurer avec une paire de clés ssh au moment de la création. (Accès et sécurité – Paires de clés – créer une paire de clés)

Pour se connecter à une machine virtuelle
ssh -i (clé privée) ubuntu@ip-flottante

Pour envoyer un fichier vers une machine virtuelle :
scp -i (clé privée) nom_du_fichier ubuntu@ip-flottante:

Associer une ip flottante à une machine virtuelle:

Pour associer une ip flottante à une machine virtuelle, il faut suivre les étapes suivantes:

- Récupérer le fichier rc dans le dashboard, dans Projet, «Accès et sécurité», «Accès API» et l'exécuter:
  source ./[LOGIN]-projet-openrc.sh
- S'allouer une ip externe:
  nova floating-ip-create ext-net
- Ou vérifier celle qu'on a si on se l'est déjà allouée:
  nova floating-ip-list
- Associer notre ip externe à une instance (Il faut que l'instance soit connectée dans un subnet interconnecté avec ext-net, comme «réseau-pour-tous»):
  nova add-floating-ip [Nom ou ID de l'instance] [Mon_IP_externe]