

TP1 – Appels de méthodes à distance

INF4410 - Systèmes répartis et infonuagique - Michel Dagenais



POLYTECHNIQUE
MONTREAL

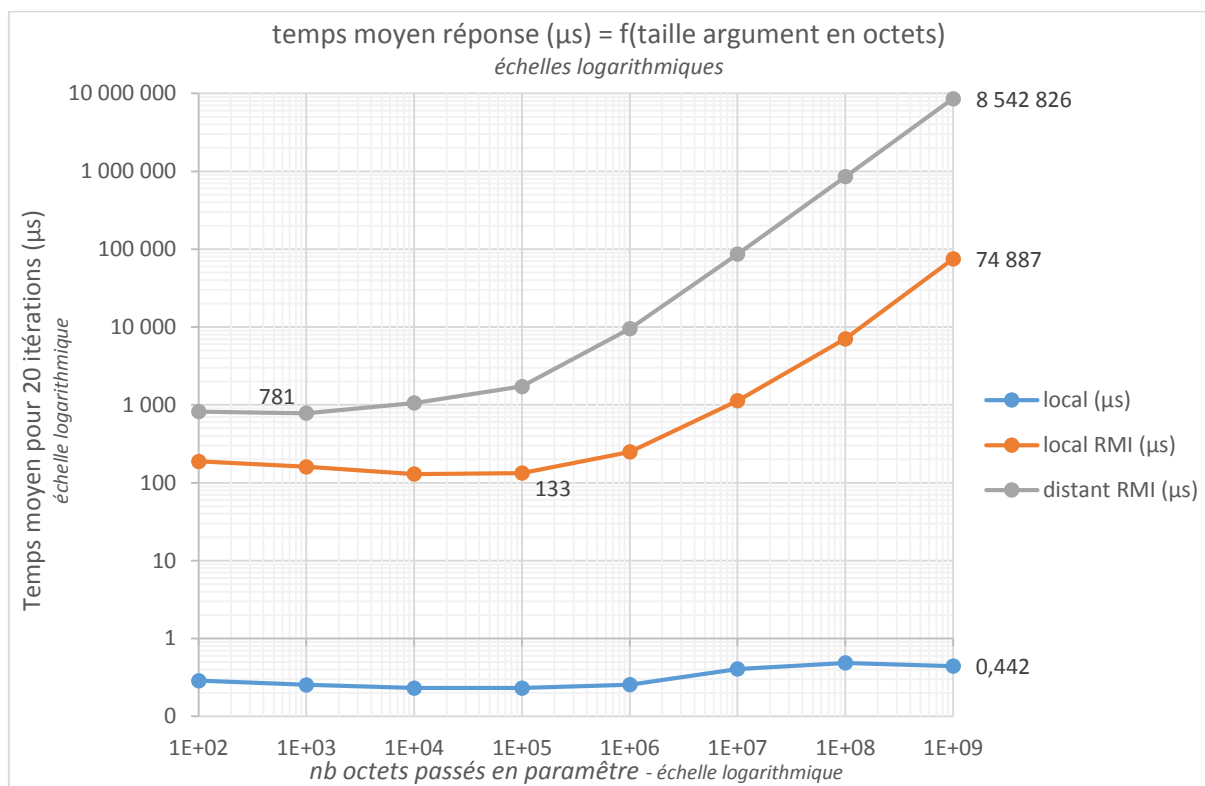
LE GÉNIE
EN PREMIÈRE CLASSE

Antoine Giraud – #1761581

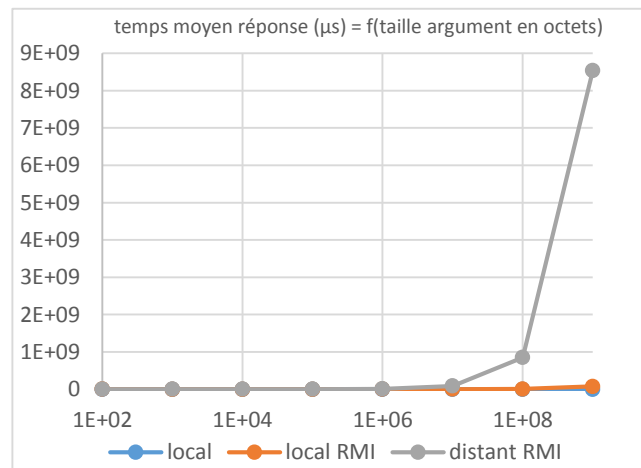
Le 05 octobre 2014

Partie 1 – Etude performance Java RMI

Question 1



nb octets	local (μ s)	local RMI (μ s)	distant RMI (μ s)
1E+02	0,288	188	821
1E+03	0,254	160	781
1E+04	0,231	130	1061
1E+05	0,231	133	1726
1E+06	0,255	249	9516
1E+07	0,406	1130	86209
1E+08	0,485	7050	853101
1E+09	0,442	74887	8542826
moyenne	0,324	10491	1187005



Exploitation des résultats

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).

Nous avons représenté dans un repère le temps moyen réponse d'une méthode de communication client-server (local, RMI local, RMI distant) en μ s en fonction de taille de l'argument envoyé du client au serveur en octets / bytes.

Nous avons donc trois méthodes de connexion client-serveur. Le serveur était soit :

1. Local, nous avons en moyenne 0.324 μ s. Nous pouvons noter tout de même que le système prend 1.75 fois plus de temps quand l'on passe un argument dès 10Mo, et ce jusqu'à 1Go.
2. Local utilisant RMI, nous avons en moyenne avant de passer à plus de 1e6 octets (1Mo) une moyenne de 172 μ s pour le temps de traitement. Ensuite celui-ci croît exponentiellement (de base 10).
3. Distant utilisant RMI, nous avons un temps de traitement plutôt peu stable d'environ 900 μ s jusqu'à 1ko. Ensuite, de même que pour le RMI en local, nous commençons une croissance exponentielle (de base 10) marquée mais cette fois plus tôt dès que nous passons les 100ko.

Au final, si nous restons en local, nous pouvons apercevoir que le système ne bronche pas plus que cela à un argument de très grosse taille. Cela reste bien sûr relatif car on ne peut tester des arguments de plus de 1Go mais nous n'avons pas de croissance exponentielle de base 10 comparé à l'utilisation du RMI pour passer cet argument du client au serveur qui apparaît aux alentours de 1Mo.

De plus, on remarque que Java RMI à partir d'une certaine taille entre 0.1Mo et 1Mo commence à montrer ses limites car le temps de traitement croît alors exponentiellement dans une base 10. On peut voir sur le graphique logarithmique que nous avons deux droites d'un coefficient directeur de 1 (dans cette même échelle). Pour le RMI distant, le débit du réseau pourrait expliquer ces courbes dans le sens où l'on souhaite envoyer un gros fichier à travers notre réseau. Cela se fera à une certaine vitesse (5Mo/s par exemple).

Pour résumer, on peut en conclure que l'utilisation de Java RMI ou un autre système de RPC semblable sera adapté pour des fichiers restant en dessous de 1ko si l'on veut avoir un temps de traitement qui soit stable. Si on met des fichiers de taille supérieure, nous aurons comme expliqué ci-dessus des temps qui croîtront à vitesse exponentielle (de base 10). Il n'est donc pas conseillé d'utiliser Java RMI pour échanger de gros objets. Il faudrait alors utiliser d'autres méthodes pour envoyer nos fichiers / arguments de lourds poids, ou revoir la manière dont sont codés nos objets pour que leur poids ne soit pas trop important.

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.

Nous avons ci-dessous le schéma d'interaction entre nos différents acteurs.



Entre le client et le serveur se trouve notre interface de programmation RMI (Remote method invocation) qui fait le lien entre eux deux. Notre interface de programmation RMI se présente en deux classes stub et skeleton qui sont respectivement chez le client et le serveur. Ces deux classes sont générées automatiquement par l'outil `rmic` du JDK. Elles se chargent de gérer les mécanismes (appel, communication, exécution, renvoi / réception de résultat) pour permettre à nos deux acteurs de communiquer entre eux.

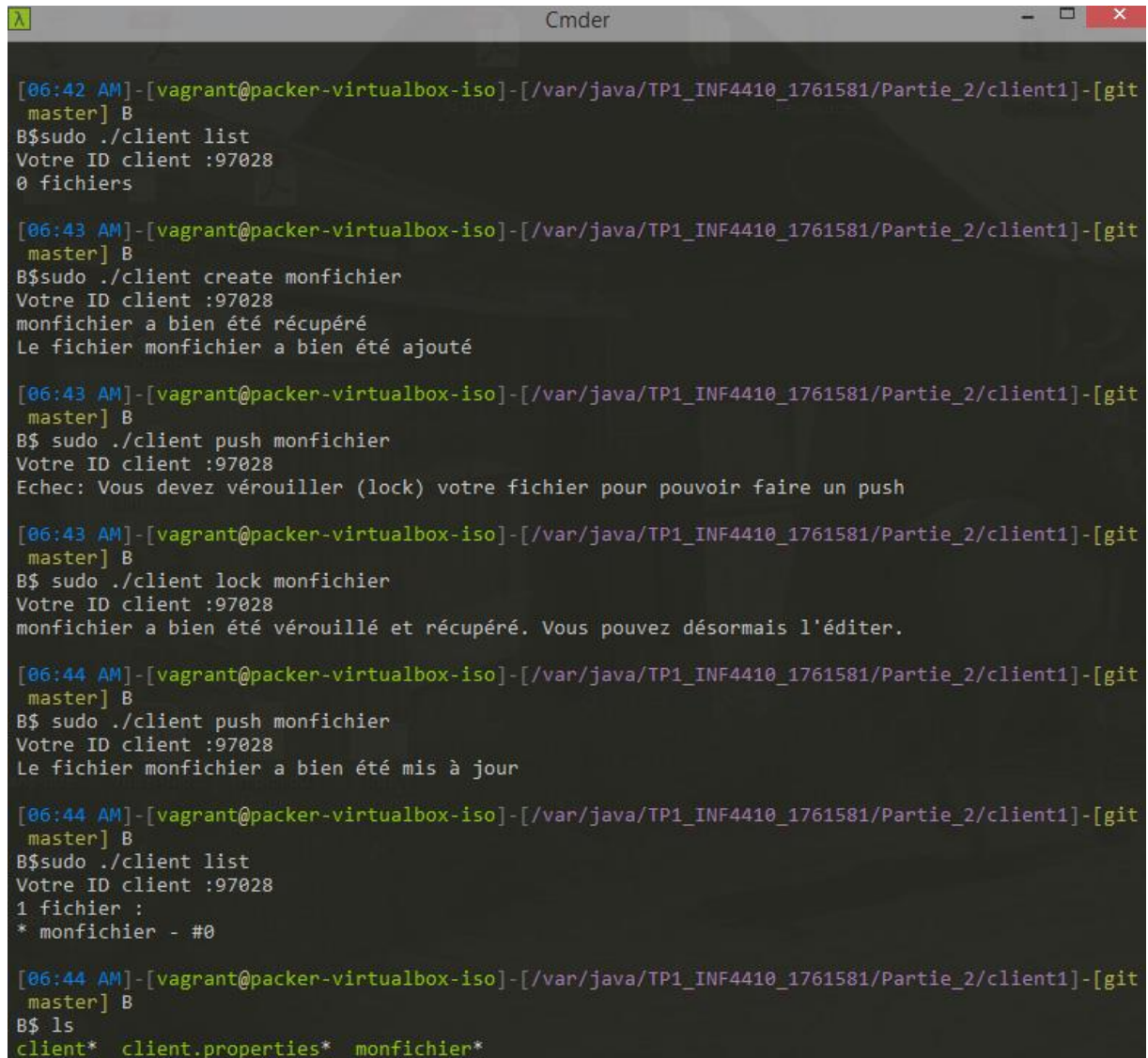
1. On compile le projet avec la commande `ant`
2. Lancer le registre de nom RMI depuis le répertoire `bin` du projet. On enregistrera dans celui-ci une instance de l'objet que l'on souhaite appeler à distance.
3. On lance notre serveur (commande `./server`). (classe `Server.java`)
 - Celui-ci crée un objet serveur `Server server = new Server();` (l.14) qui hérite de notre interface `ServerInterface`. On le lance ensuite avec la commande `run()` définie juste après le main.
 - On va dans un premier temps mettre en place un security manager pour pouvoir charger dynamiquement certaines classes. (l.24)
 - On essaie de se connecter au registre de nom RMI que l'on vient de lancer. (l.28)
 - On va ensuite enregistrer notre classe serveur dans le registre RMI avec la fonction `registry.rebind("server", stub);` (l.32)
 - On en a alors fini côté serveur. On peut passer au client.
4. On lance notre client (commande `./client`). (classe `Client.java`)
 - On va avoir dans la déclaration de notre client différents objets correspondant à nos futurs serveurs que nous contacterons. (`private ServerInterface localServerStub, ...` l.23..26)
 - Lors de l'instanciation du client, on peut au besoin se connecter à un serveur distant dont l'on aura récupéré l'ip en paramètre. Dans tous les cas nous nous connectons au serveur local RMI (`localServerStub`) que l'on a lancé et au faux serveur RMI (`FakeServer`). Nous utilisons pour nous connecter aux « vrais » serveurs RMI la méthode `loadServerStub` qui prend en paramètre l'IP du serveur qui possède un registre RMI lancé. Celle-ci résume les opérations à faire pour se connecter au serveur RMI :
 - `Registry registry = LocateRegistry.getRegistry(hostname);` (l.75)
On va chercher à se connecter au serveur qui a le Registre RMI de lance.
 - `stub = (ServerInterface) registry.lookup("server");` (l.76)
On va une fois connecté à ce serveur chercher dans le registre RMI l'objet serveur que l'on voudra utiliser par la suite.
 - On est maintenant connecté ! On a plus qu'à utiliser notre objet. Par exemple : `var retour = localServerStub.execute(2, 1);` On souhaite exécuter la requête `execute` sur le serveur – que nous avons défini dans le serveur – qui va nous retourner la somme de nos deux arguments. Comme nous avons côté client cet objet `localServerStub` qui étends de l'interface `ServerInterface` qui étends elle-même de la classe `Remote` de Java RMI, cela se fera en toute transparence à travers le stub créé qui se chargera de tout faire pour que l'on exécute la requête côté client.

Partie 2 – système de fichiers à distance via RMI

Démonstrations

Exemple Simple :

Côté client :



```
[06:42 AM]-[vagrant@packer-virtualbox-iso]-[/var/java/TP1_INF4410_1761581/Partie_2/client1]-[git master] B
B$ sudo ./client list
Votre ID client :97028
0 fichiers

[06:43 AM]-[vagrant@packer-virtualbox-iso]-[/var/java/TP1_INF4410_1761581/Partie_2/client1]-[git master] B
B$ sudo ./client create monfichier
Votre ID client :97028
monfichier a bien été récupéré
Le fichier monfichier a bien été ajouté

[06:43 AM]-[vagrant@packer-virtualbox-iso]-[/var/java/TP1_INF4410_1761581/Partie_2/client1]-[git master] B
B$ sudo ./client push monfichier
Votre ID client :97028
Echec: Vous devez verrouiller (lock) votre fichier pour pouvoir faire un push

[06:43 AM]-[vagrant@packer-virtualbox-iso]-[/var/java/TP1_INF4410_1761581/Partie_2/client1]-[git master] B
B$ sudo ./client lock monfichier
Votre ID client :97028
monfichier a bien été verrouillé et récupéré. Vous pouvez désormais l'éditer.

[06:44 AM]-[vagrant@packer-virtualbox-iso]-[/var/java/TP1_INF4410_1761581/Partie_2/client1]-[git master] B
B$ sudo ./client push monfichier
Votre ID client :97028
Le fichier monfichier a bien été mis à jour

[06:44 AM]-[vagrant@packer-virtualbox-iso]-[/var/java/TP1_INF4410_1761581/Partie_2/client1]-[git master] B
B$ sudo ./client list
Votre ID client :97028
1 fichier :
* monfichier - #0

[06:44 AM]-[vagrant@packer-virtualbox-iso]-[/var/java/TP1_INF4410_1761581/Partie_2/client1]-[git master] B
B$ ls
client* client.properties* monfichier*
```

Côté Serveur



```
[06:42 AM]-[vagrant@packer-virtualbox-iso]-[/var/java/TP1_INF4410_1761581/Partie_2]-[git master] B
B$ sudo ./server
Server ready.
Liste des fichiers :
Fichier monfichier ajouté !
Fichier monfichier locked par #97028
Fichier monfichier MAJ & unlocked par #97028
Liste des fichiers :
* monfichier - #0
```


Exemple conflit :

Côté client 1

```
[06:58 AM]-[vagrant@packer-virtualbox-iso]-[/var/java/TP1_INF4410_1761581/Partie_2/client1]-[git master] B
B$ sudo ./client create monfichier
Votre ID client :89866
monfichier a bien été récupéré
Le fichier monfichier a bien été ajouté

[06:58 AM]-[vagrant@packer-virtualbox-iso]-[/var/java/TP1_INF4410_1761581/Partie_2/client1]-[git master] B
B$ sudo ./client lock monfichier
Votre ID client :89866
monfichier a bien été verrouillé et récupéré. Vous pouvez désormais l'éditer.

[06:59 AM]-[vagrant@packer-virtualbox-iso]-[/var/java/TP1_INF4410_1761581/Partie_2/client1]-[git master] B
B$ sudo ./client push monfichier
Votre ID client :89866
Le fichier monfichier a bien été mis à jour

[06:59 AM]-[vagrant@packer-virtualbox-iso]-[/var/java/TP1_INF4410_1761581/Partie_2/client1]-[git master] B
B$
```

Côté client 2

```
[06:58 AM]-[vagrant@packer-virtualbox-iso]-[/var/java/TP1_INF4410_1761581/Partie_2/client2]-[git master] B
B$ sudo ./client create monfichier
Votre ID client :72907
Echec: le fichier monfichier existe déjà

[06:59 AM]-[vagrant@packer-virtualbox-iso]-[/var/java/TP1_INF4410_1761581/Partie_2/client2]-[git master] B
B$ sudo ./client lock monfichier
Votre ID client :72907
Echec: le fichier monfichier appartient au client #89866

[06:59 AM]-[vagrant@packer-virtualbox-iso]-[/var/java/TP1_INF4410_1761581/Partie_2/client2]-[git master] B
B$ sudo ./client lock monfichier
Votre ID client :72907
monfichier a bien été verrouillé et récupéré. Vous pouvez désormais l'éditer.

[06:59 AM]-[vagrant@packer-virtualbox-iso]-[/var/java/TP1_INF4410_1761581/Partie_2/client2]-[git master] B
B$ sudo ./client push monfichier
Votre ID client :72907
Le fichier monfichier a bien été mis à jour

[06:59 AM]-[vagrant@packer-virtualbox-iso]-[/var/java/TP1_INF4410_1761581/Partie_2/client2]-[git master] B
B$
```

Côté Serveur

```
[06:58 AM]-[vagrant@packer-virtualbox-iso]-[/var/java/TP1_INF4410_1761581/Partie_2]-[git master] B
B$ sudo ./server
Server ready.
Fichier monfichier ajouté !
Fichier monfichier locked par #89866
Echec: le fichier monfichier appartient au client #89866
Fichier monfichier MAJ & unlocked par #89866
Fichier monfichier locked par #72907
Fichier monfichier MAJ & unlocked par #72907
|
```

Retour sur le développement :

Comme vous pouvez voir avec l'architecture des fichiers à droite, nous avons gardé la même structure que dans la première partie.

Nous avons deux classes partagées à tous :

ServerInterface.java

Même principe que pour la première partie du TP. Cette interface va permettre d'exposer au client des fonctions qu'il pourra appeler et faire exécuter sur le serveur. Elles sont précisées en détail dans l'énoncé du TP. Voici leur déclaration avec les types de variables prises par celles-ci :

- generateclientid()
- generateclientid(int clientId) - non exigée mais nous simplifie la vie pour la gestion de l'identifiant du client.
- create(String nom)
- Hashtable list()
- Fichier get(String nom)
- Fichier lock(String nom, int clientId)
- boolean push(String nom, byte[] contenu, int clientId)

Fichier.java

Cette classe nous est très utile car elle nous permet d'avoir une classe pour gérer nos fichiers et leurs propriétés.

On pourra même écrire sur le disque dur le contenu du fichier pour le conserver autrement qu'en mémoire.

La classe n'est pas faite pour gérer l'arborescence.

Cependant, pour un souci de clarté dans l'organisation des fichiers sur notre serveur, on va pouvoir indiquer vers quel répertoire l'on souhaite enregistrer les fichiers.

C'est au programme qui va utiliser les objets Fichiers de passer en argument un chemin vers le dossier dans lequel placer les fichiers.

Server.java

Basée de nouveau dans sa structure sur la classe du même nom dans la partie 1 du TP, nous avons implémenté les méthodes exposées dans l'interface ServerInterface.

On ajoutera juste deux fonctions getAllFile(File file), pour l'initialisation et la mise en mémoire des fichiers présent au cas où sur le serveur créés lors d'une dernière session, et et getRandomNumber(), pour la génération des identifiants des clients.

```
- Partie_2
  - client1
    client
    client.properties
    pomme.txt
  - client2
    client
    client.properties
    poire.txt
  - server_files
    poire.txt
    pomme.txt
  - src
    - ca
    - polymtl
    - inf4402
    - tp1
      - client
        Client.java
      - server
        Server.java
      - shared
        Fichier.java
        ServerInterface.java
    build.xml
    client
    client.jar
    policy
    README.md
    server
    server.jar
    shared.jar
```

Client.java

Comme précisé dans l'énoncé du TP, voici les exigences concernant cette classe :

*« 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. »*

Cette classe, de même que pour la classe Server est basée sur le code de la classe du même nom de la partie 1 du TP.

On aura rajouté les fonctions *list*, *get*, *create*, *lock* et *push* pour répondre aux exigences demandées. On aura aussi rajouté deux fonctions *readClientIdFromFile()* et *writeClientIdInFile()* qui vont nous permettre de gérer l'écriture dans un fichier de l'identifiant client attribué par le serveur.

Commandes disponibles comme demandé au client :

./client list : Permet d'afficher la liste des fichiers présents sur le serveur de fichier

./client create monSuperFichier : Permet de créer le fichier monSuperFichier sur le serveur. Il sera récupéré dans la foulée sur le répertoire du client

./client get monSuperFichier : récupère le fichier monSuperFichier si présent sur le serveur

./client lock monSuperFichier : verrouille le fichier monSuperFichier à ce client pour qu'il puisse l'éditer et le mettre à jour sans crainte de voir sa MAJ écrasée.

./client push monSuperFichier : Envoie la mise à jour au serveur et libère le fichier

Exécuter le projet 2 :

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.
3. Démarrez le serveur avec le script server (``./server`` ou `bash server`).
4. dans une nouvelle console, déplacez vous dans le répertoire ``client1`` et exécutez la commande ``./client XX YY`` voulue
5. réalisez de même que 4. dans le répertoire ``client2`` pour contrôler un deuxième client au serveur.