Polytechnique - Montréal

INF4410 – Systèmes répartis et infonuagique

TP1 – Appels de méthodes à distance

Auteurs :

Jérémy Wimsingues *1860682*,
Robin Royer *1860715*

5 octobre 2016

Table des matières

1 Partie 1 **2**

1.1 Introduction . . . 2

1.2 Impact de la taille des arguments . . . 2

1.2.1 Appel local . . . 2

1.2.2 Appel RMI local . . . 3

1.2.3 Appel RMI distant . . . 4

1.2.4 Synthèse . . . 4

1.3 Interaction entre les différents acteurs . . . 5

2 Partie 2 **6**

2.1 Choix de conception . . . 6

2.1.1 La classe File . . . 7

2.1.2 Exceptions . . . 7

2.1.3 Procédure interne de validation - Valeur de retour . . . 7

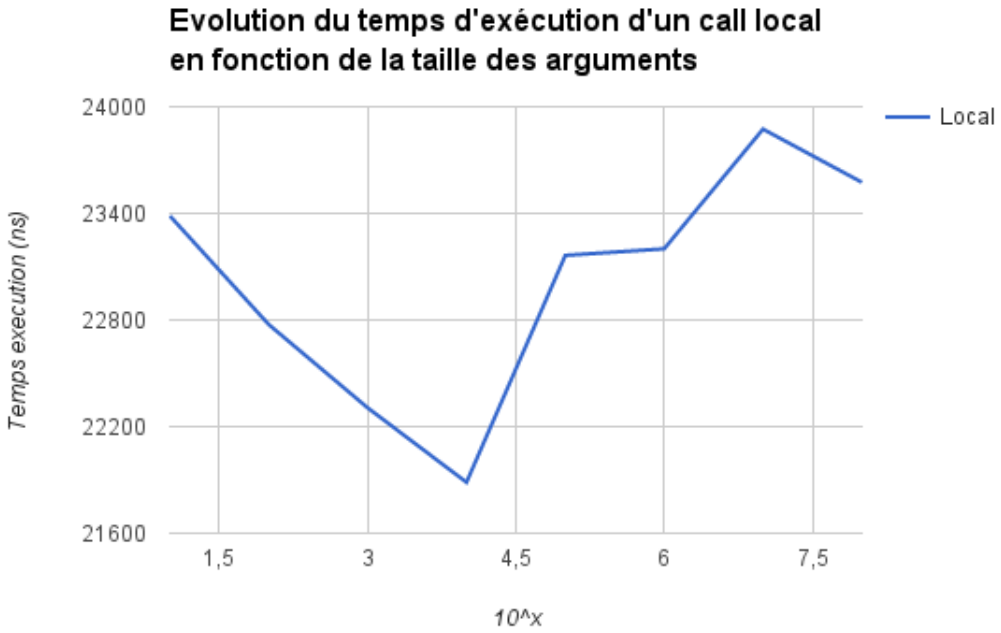
1 Partie 1

1.1 Introduction

Afin de pouvoir proposer des éléments de comparaison entre les différents types d'appels (locaux, RMI locaux ou RMI distant), nous avons comparé les temps moyens pour dix appels pour chacune des puissances de dix (paramètre de plus en plus important). Nous avons remarqué que les performances de la JVM [1] étaient variables. En effet on peut voir que effectuer dix exécutions consécutives impacte les temps de réponse. Ceci s'explique par l'utilisation de ce que les experts qualifient parfois de "Java à chaud" (adaptation de la JVM à traiter certaines instructions et donc devient plus performante) et de "Java à froid" (utilisation sans adaptation). Dans le cadre de ce travail pratique nous ne nous attarderons pas sur ces performances variables et afin d'atténuer l'importance de ces derniers nous avons pris parti de garder la moyenne du temps de réponses moyen sur dix exécutions.

1.2 Impact de la taille des arguments

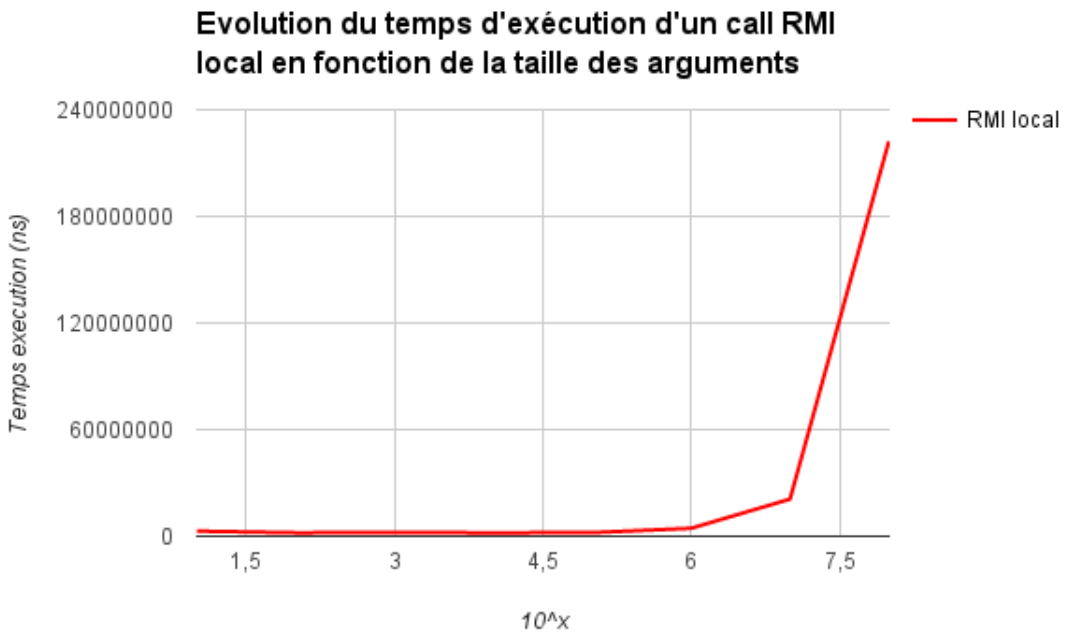
1.2.1 Appel local



Pour les appels locaux, on constate que la courbe fluctue autour de sa valeur de départ, l'impact de l'augmentation de la taille du tableau de bytes est donc peu visible. En effet, bien que la courbe effectue une chute sur les quatre premières puissance de dix, cela n'est pas forcément significatif au vue de l'échelle utilisée en ordonnée. La différence est **de moins de 3 000 nanosecondes** ce qui demeure négligeable. On se rappelle des ordres de grandeur traditionnel de l'informatique : les temps d'accès **d'un disque dur** sont de l'ordre de **la dizaine de millisecondes** alors que le temps d'accès à une **mémoire RAM** moderne est de l'ordre de **quelques nanosecondes** (soit environ un million de fois plus rapide).

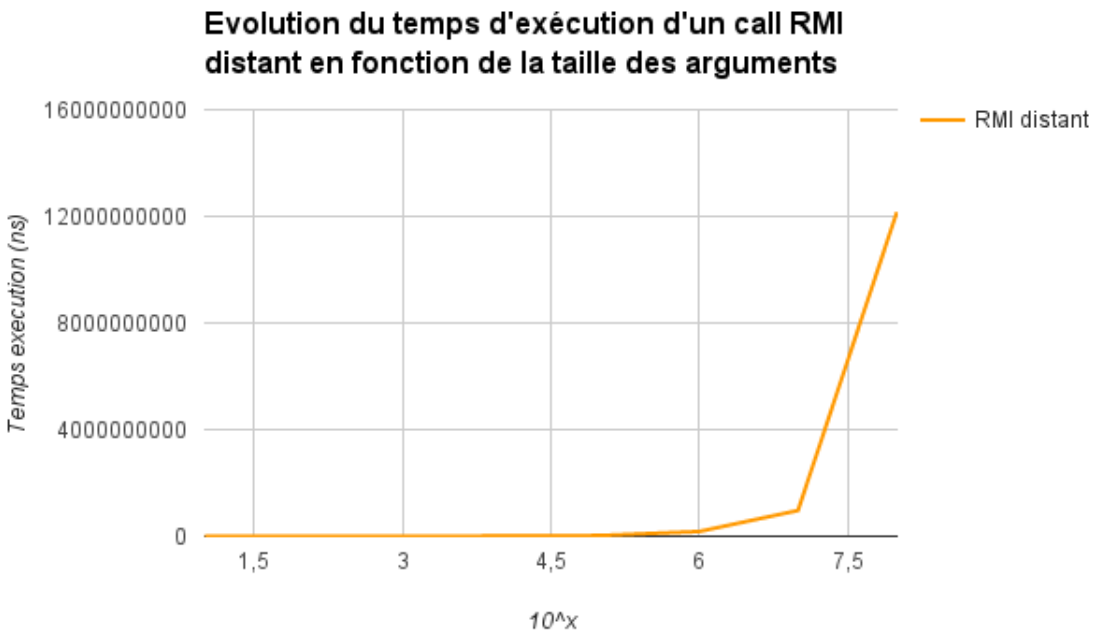
1. Java Virtual Machine

1.2.2 Appel RMI local



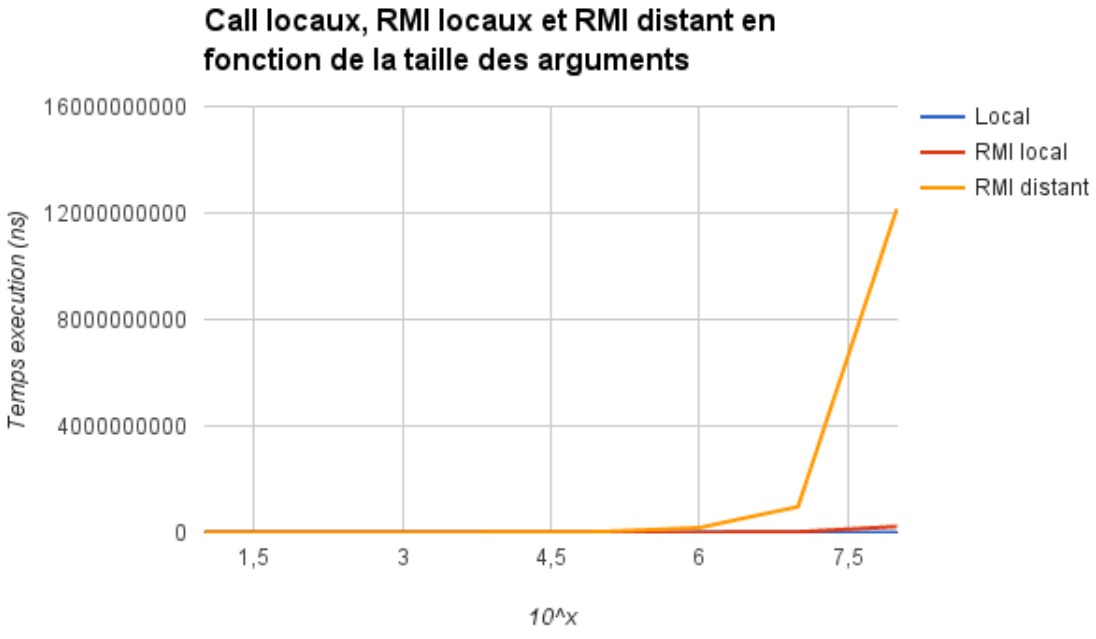
Pour les appels RMI locaux, on constate que l'allure de la courbe est exponentielle en échelle logarithmique. L'impact de l'augmentation de la taille du tableau de bytes est donc bien visible : on observe **un facteur 100** entre un appel avec 10^1 et un appel avec 10^8 bytes. Autrement dit, bien que nous ayons émuler un serveur RMI local, le transfert RMI évolue de façon exponentielle ce qui diffère énormément de la courbe précédente. Le temps maximal de réponse pour un tableau de bits d'une taille de 10^8 s'élève à **222 228 404 nanosecondes**, soit environ 0,2 seconde. Ces valeurs sont déjà largement au dessus de l'analyse précédente effectué en local. Bien que l'évolution soit exponentielle, le temps de réponse reste relativement correct compte tenu des ordres de grandeur énonce dans l'analyse précédente. Cependant, nous pouvons donc déjà émettre l'hypothèse que **RMI semble peu approprié lorsque la taille des arguments devient trop importante** au vu de l'évolution exponentielle de la courbe.

1.2.3 Appel RMI distant



Pour les appels RMI distant, on constate que l'allure de la courbe est exponentielle en échelle logarithmique. L'impact du réseaux et de l'augmentation de la taille du tableau de bytes est donc bien visible : on observe **un facteur 2 000** entre un appel avec 10^1 et un appel avec 10^8. Le temps maximal de réponse pour un tableau de bits d'une taille de 10^8 s'élève à **19 226 447 770 nanosecondes**, soit environ **19,2 secondes**. Cette fois ci, l'impact réseau est tel que le temps de réponse est **beaucoup trop élevé**. Cela vient donc confirmer l'hypothèse avancée lors du résultat obtenu sur les appels en RMI local : **RMI semble peu approprié lorsque la taille des paramètres devient trop importante**.

1.2.4 Synthèse



La comparaison à la même échelle des trois cas vu précédemment est très révélatrice. L'impact du réseaux est non négligeable lorsque l'on doit échanger de gros arguments : on observe facteur multiplicatif 76, dû au réseau, entre le RMI distant et local pour 10^8 bytes.

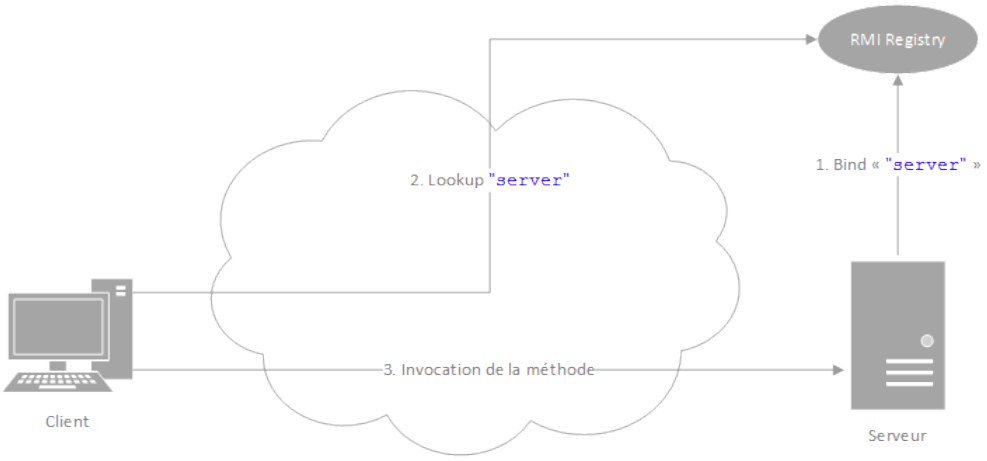
Java RMI permet donc **une implémentation relativement facile** notamment grâce au partage de la même interface pour le client et le serveur, à l'abstraction d'un format d'échange tel que json ou xml, à la gestion centralisé d'un garbage collector ainsi que la gestion de la sécurité. Nous n'avons rien eu à faire de spécifique pour la partie réseau : l'ouverture et la fermeture des sockets sont gérées de façon complément transparente par Java.

Cependant, il montre des limites : l'implémentation doit **obligatoirement** être faite en java du côté client comme du côté serveur. RMI reste également la propriété de SUN, il est, comme nous avons pu le voir, sensible à la taille des arguments et reste plus lent que les implémentations CORBA.

1.3 Interaction entre les différents acteurs

L'un des avantages du RMI est que l'implémentation est très simple que ce soit pour le code côté client ou le code côté serveur. Cependant il faut bien comprendre son fonctionnement afin de ne pas faire d'erreur. Deux machines virtuelles java (JVM) tournent au minimum (cas un seul client et un serveur). Le principe de RMI se base sur le fait que la JVM cliente va envoyer un message par le réseau à la JVM serveur qui va alors exécuter ses méthodes sur ses objets qui lui sont propres. Nous retrouvons bien par cette explication le nom du protocole ainsi choisi Remote Method Invocation.

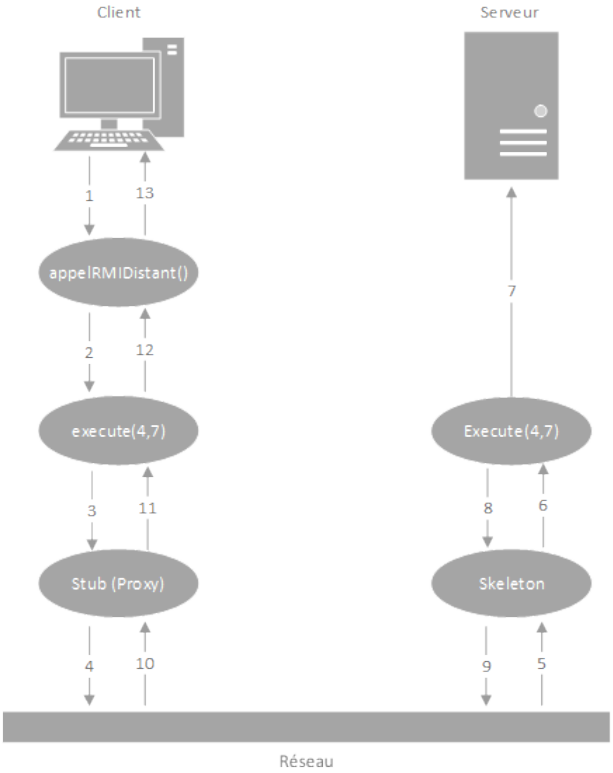
Le schéma ci après montre comment fonctionne l'architecture générale d'un système RMI.



Les trois phases sont les suivantes :

1. Phase de "bind ou rebind" : Le serveur RMI demande au service RMI registry de créer une nouvelle entrée dans son annuaire afin de rendre les méthodes visibles aux clients RMI. La nouvelle entrée de l'annuaire associera un nom au serveur RMI. Dans le code fourni, le nom est **"server"**.
2. Phase de "lookup" : Le client RMI demande à la RMI registry de lui donner le serveur RMI associé à un certain nom dans son annuaire. Il est donc nécessaire que le client connaisse le nom sous lequel le serveur a été inscrit dans l'annuaire de la registry. Dans notre cas, la demande dans le code se fait via le nom **"server"**.
3. Phase d'invocation de méthodes distantes : maintenant le client peut invoquer les méthodes du serveur. Dans notre cas, la méthode ***execute(4,7)***.

Le diagramme ci-après détaille les différents appels effectués.



1. Nous sommes dans la partie gauche du digramme. Toutes les bulles alignées à gauche font partie de la JVM cliente. Le client effectue la commande ./client ce qui lance le **main** du programme.
2. Le code se déroulant logiquement, nous nous retrouvons ensuite dans la méthode ***execute(4,7)***.
3. Dés lors, nous allons consulter ce que l'on appelle le **stub** (autrement dit notre proxy) afin de consulter le serveur.
4. Le **stub** va alors nous indiquer notre destination. Ainsi cette étape ne constitue qu'un passage dans le réseau. Ce dernier pouvant être un LAN (cas du serveur RMI émulé en LAN) ou internet (cas RMI distant).
5. La réception des paquets a lieu dans la partie droite de notre diagramme. Il s'agit de la zone de la JVM serveur. Le **Skeleton** est consulté, bien que cela soit transparent nous nous.
6. Ce dernier permet alors l'exécution distant de la méthode ***execute(4,7)***.
7. Cette étape est sans importance et consiste en l'affichage de log/traces sur le serveur en cas de besoin. Cela n'est pas le cas dans cette partie du TP.
8. La réponse ainsi calculée peut faire le chemin inverse et suivre les étapes **8, 9, 10, 11 et 12** pour retourner à la méthode appelante.
13. Cette étape est l'affichage du résultat sur le terminal de l'utilisateur.

2 Partie 2

2.1 Choix de conception

D'un point de vue conception, nous avons suivi la conception classique d'implémentation de méthodes en RMI. Nous avons pris parti de définir des structures de données différentes afin de répondre au mieux aux commandes définies dans le sujet. Afin de rendre cela possible, il est évident que avons modifié légèrement les signatures préconisées par l'énoncé.

Aussi, nous avons essayé de respecter au mieux les Best Practices Java : utilisation des exceptions Java, code d'erreur lors de sortie sous fin d'exécution non normale du programme (facilite les futurs scripts possibles autour du programme), définition de toutes les constantes en haut de fichiers afin de pouvoir les externaliser facilement, implémentation de serial UID générique pour les objets sérialisables, ainsi de suite. Nous avons aussi rendu notre code robuste : toutes les erreurs

qu'il est possible de gérées ont été gérer et tous les contrôles simples qu'il était possible de faire on était fait. Parmi les choix de conception forts, deux d'entre eux sont les suivants : la création d'une classe File et la gestion des exceptions.

2.1.1 La classe File

Nous avons choisi d'implémenter toute la logique de fichier du projet sous la forme d'une classe *File* propre à nous. Cet objet est composé de deux sous objet : un objet *Header* et un objet *Content*. L'objet *Header* a les propriétés suivantes : l'identifiant du propriétaire (owner) du fichier si le dernier est verrouillé, le nom du fichier et son état (verrouillé ou disponible). L'objet *Content* contient quand à lui deux propriété : le contenu et le checksum relatif à ce contenu.

2.1.2 Exceptions

Afin d'être plus granulaire et de respecter la philosophie objet de Java dans la gestion des différents cas, plusieurs classes d'exceptions ont été crées. En effet, afin de traiter au mieux les retours du serveur, le langage Java met à dispositions un système d'exception que nous avons utilisé. Nous avons ainsi créé deux nouvelles classes d'exception UnlockableFileException et UnpushableFileException qui nous permettent de gérer certains cas afin d'informer au mieux l'utilisateur de la raison pour laquelle son action est invalide ou impossible.

2.1.3 Procédure interne de validation - Valeur de retour

En cas de sortie du programme sur une exception ou une erreur, nous avons défini un certain nombre de code de retour distinct afin de rendre plus facile la création de scripts futur autour de notre système de fichiers client/serveur. Par exemple, dans un script, le fait de tester la valeur de retour peut permettre de faire une action plutôt qu'une autre. Voici la liste exhaustive de ces codes de retours :

- 0 : valeur de retour par défaut lors de l'exécution réussit d'un programme Java.
- -10 : valeur de retour en cas d'exception lors de la commande **Push**.
- -20 : valeur de retour en cas d'erreur lors de la vérification des arguments.
- -30 : valeur de retour en cas d'exception lors de la commande **Create**.
- -40 : valeur de retour en cas d'exception lors de la commande **Lock**.
- -50 : valeur de retour en cas d'erreur liée à RMI.
- -60 : valeur de retour en cas d'erreur de **Bound**.
- -70 : valeur de retour en cas d'exception lors de l'accès à un fichier.
- -80 : valeur de retour en cas d'exception lors des entrées / sorties (IO).
- 10 : valeur de retour en cas d'exception lors de la commande **Get**.