Département de génie informatique et génie logiciel

INF8480

Travail pratique 1

Par:
Félix Boulet (1788287)
Erica Bugden (1748542)

Remis à:
Anas Balboul

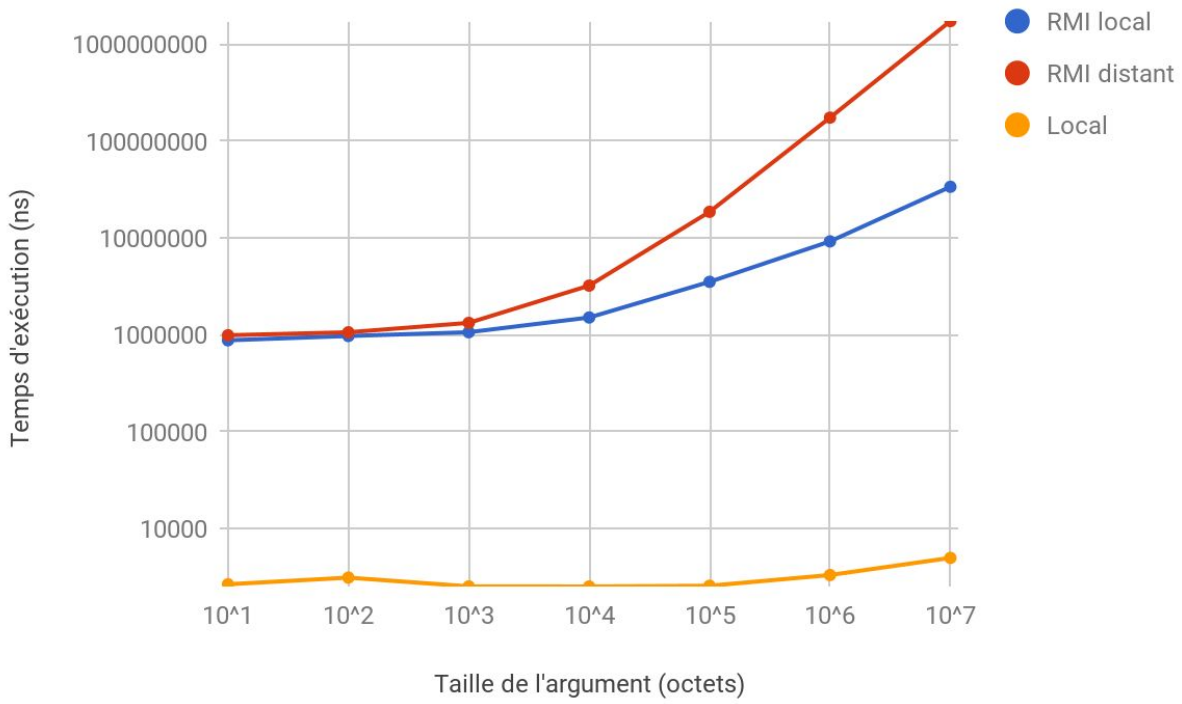
Février 2018

N.B. Le code source utilisé pour effectuer les tests de la première question se trouve dans le dossier `ResponseTime_Analyzer/src/ca/polymtl/inf8480/tp1`

Partie 1

Question 1

Dans le cadre de ce travail pratique, nous avons effectué des expériences pour déterminer l'effet de la taille des arguments sur le temps d'exécution d'un appel de fonction RMI. Le graphique qui suit illustre cet effet sur des appels RMI locaux (un appel dirigé vers un serveur exécuté sur la machine locale) et distants (un appel dirigé vers une machine virtuelle distante déployée sur OpenStack), ainsi que sur des appels locaux "standards" à la même fonction (via une classe *FakeServer* implémentant de façon locale la fonction définie par *ServerInterface*). La taille variable de l'argument est implémenté à l'aide d'un tableau de caractères (*char[]*) vide, chaque case de tableau faisant ainsi 1 octet.



Graphique 1 **:** Effet de la taille de l'argument sur le temps d'exécution d'un appel de fonction

Premièrement, on observe que le temps d'exécution d'un appel de fonction local ne varie pas vraiment en fonction de la taille de l'argument. Ceci est normal, car Java passe les arguments de type "objet" (que ce soit un objet issu d'une classe ou un tableau de primitifs comme dans notre cas) par valeur, mais cette valeur n'est en fait qu'une référence vers l'objet en mémoire. Ainsi, puisqu'il s'agit toujours d'une simple adresse mémoire qui indique où se trouve le tableau

passé à la fonction, et que cette dernière ne fait rien, le temps d'exécution ne varie sensiblement pas. On pourrait probablement attribuer la variation pour les arguments de grande taille à du simple "bruit" (occupation du CPU, délai, etc.), puisqu'on observe également des pics à des tailles de 10^1 et 10^2.

Pour ce qui est des appels RMI, le temps d'exécution de l'appel augmente exponentiellement même pour des arguments de taille relativement réduite. Cette augmentation est probablement dûe au fait que les appels doivent passer par toutes les étapes de préparation pour l'envoi sur le réseau, ("marshalling") qui peuvent prendre une quantité non-négligeable de temps. Nous pouvons observer que le délai relié au "marshalling" devient de plus en plus important et que les temps d'exécution des appels RMI locaux augmente rapidement passé des arguments de taille 10^3. Même si le tableau de caractères est passé comme une "référence" (on dira *pass-by-value-of-reference*), le processus de "marshalling" doit tout de même envoyer tout le contenu de celui-ci (donc l'ensemble de l'espace mémoire qui lui est alloué) via le réseau; en effet, l'adresse mémoire seule n'indique rien à la machine distante, qui ne partage pas le même espace mémoire. Les appels RMI distants prennent encore davantage de temps, puisqu'en plus d'avoir à préparer les paquets de réseau pour l'appel, le temps de transfert et de propagation de ces paquets sur le réseau prend beaucoup de temps par rapport à un appel RMI local (qui n'utilisera que la fonction de *loopback* de l'équipement réseau sur la machine locale).

Un des avantages intéressants des appels Java RMI est que le client peut télécharger la définition du "stub" nécessaire pour les communications sur réseau s'il n'a pas déjà été défini dans sa machine virtuelle (JVM) locale, donc si l'interface présentée par le serveur (comme *ServerInterface*) ne se trouve pas sur la machine locale. De plus, puisque le langage utilisé pour implémenter ces fonctionnalités est le Java, les programmes qui utilisent cette technologie peuvent être exécutés sur plusieurs plateformes dotées d'architectures différentes (x86, ARM…). Cependant, c'est aussi un désavantage puisque les appels RMI peuvent seulement être utilisés pour la communication entre deux programmes écrits en Java, et aucun autre langage.

Question 2

Avant que le client puisse invoquer un appel RMI, le serveur doit enregistrer dans le registre RMI une instance du "stub" de l'objet qui va traiter les appels du client. Le client doit ensuite récupérer une référence à cet objet.

L'enregistrement du stub se passe dans la fonction *run()* de la classe *Server*. Après avoir instancié le stub, le serveur récupère une référence vers le registre RMI (avec la fonction *getRegistry()*) et enregistre l'association de ce stub avec le nom "server" (fonction *rebind()*).

Le client peut maintenant récupérer ce stub dans la fonction *loadServerStub()* de la classe Client. Le client trouve le registre RMI avec la fonction *getRegistry()* et ensuite communique avec ce registre pour chercher le stub associé avec le nom "server" (fonction *lookup()*). Après

avoir téléchargé la classe correspondant au stub si nécessaire (ce qu'il n'a pas à faire ici puisqu'il a accès à la classe partagée *ServerInterface*), le client peut maintenant exécuter des fonctions sur le serveur à travers le stub (par exemple *distantServerStub.execute()*).