École Polytechnique de Montréal

Département de Génie Informatique et Génie Logiciel

INF8480 : Systèmes répartis et infonuagique

TP #1 – Appels de méthodes à distance

Réalisé par :

Yacine Benouniche – 1733049

Rendu à :

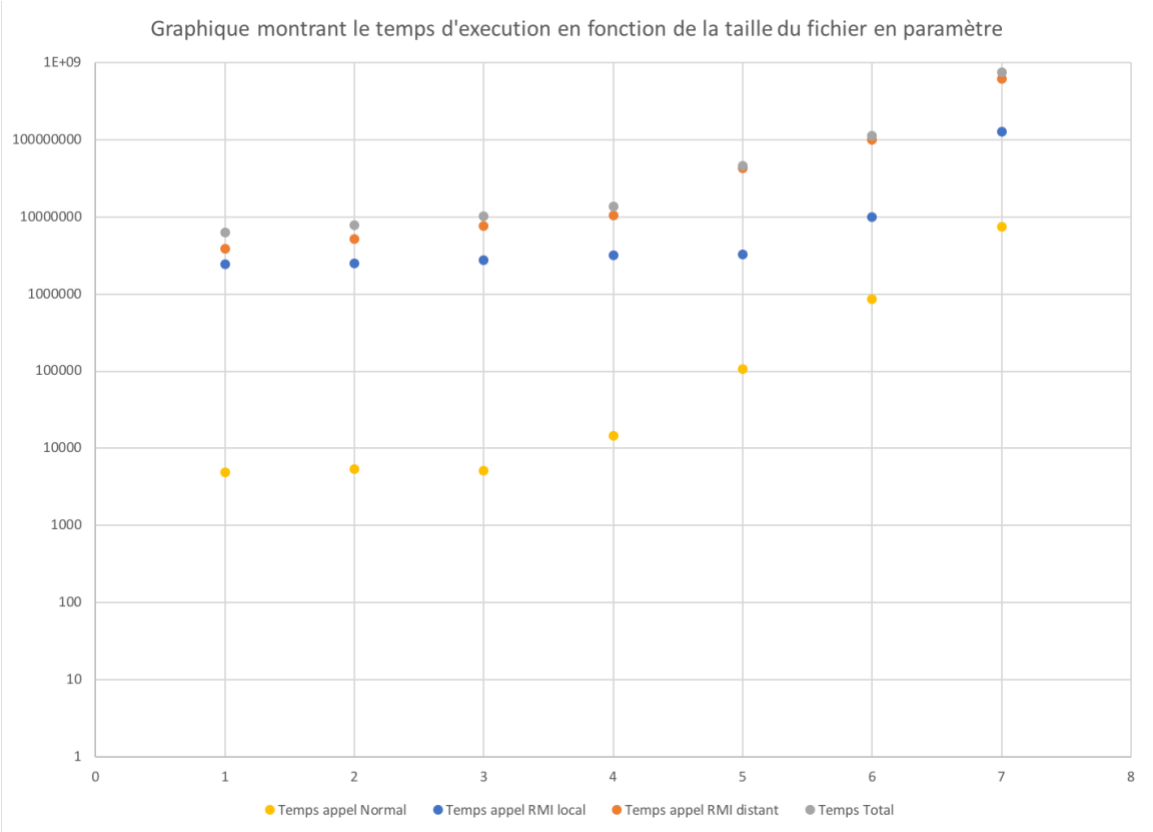
Houssem Daoud

Le 02/10/2018

Partie 1 :
Question 1 :

« On veut comparer le temps d'exécution d'une fonction vide qui prend en paramètre des arguments de tailles 10x octets, x variant entre 1 et 7. »

Afin de réaliser cela, j'envoyais en paramètres un tableau de bytes de dimension 10^x dans mon appel RMI (la fonction execute() sur le server). J'ai ensuite rapporté les valeurs affichées sur la console dans un tableau et j'ai pu finalement tracer le graphique suivant :



Le tableau des valeurs utilisées est le suivant:

Taille du fichier (octets)	10^1	10^2	10^3	10^4	10^5	10^6	10^7
Temps appel Normal (ns)	4885	5360	5156	14642	107013	860405	7535910
Temps appel RMI local (ns)	2427238	2506496	2729471	3206955	3242988	10076693	126657508
Temps appel RMI distant (ns)	3909032	5234775	7572033	10499002	42551845	101443198	612528114
Temps Total (ns)	6341155	7746631	10306660	13720599	45901846	112380296	746721532

Tableau représentant les différents temps d'exécution en fonction de la taille du fichier en paramètres

Comme on peut le remarquer sur le tableau des valeurs, le temps total d'exécution croit considérablement lorsque la taille du fichier passé en paramètres augmente. On remarque qu'a partir de 10^5 octets le temps total croit très rapidement, ainsi entre 10^4 et 10^7 octets on a une différence de temps total d'exécution de 0,73 secondes. Cependant la majeure partie de ces 0,73 secondes est représentée par le temps d'appel RMI distant avec une différence de temps d'exécution de 0,6 secondes.
On va tenter d'expliquer ces deux augmentations:

1- Augmentation du temps total d'exécution : on peut supposer qu'étant donné que le stub sérialise « marshalise » les paramètres de la fonction et qu'il les désérialise une fois du côté serveur, ainsi plus la taille du paramètre est importante plus ce processus prendra du temps à s'exécuter.

2- Augmentation très importante du temps de l'appel RMI distant : Cela peut s'expliquer très facilement. En effet, si l'appel normal et l'appel RMI local sont exécuté sur le serveur local, ainsi ils n'ont pas besoin par des protocoles http. Tout le calcul et la communication se fait localement, contrairement à l'appel distant qui transite par diffèrent routeurs. L'augmentation de la taille du paramètre est aussi un facteur expliquant cette importante augmentation du temps de l'appel RMI distant.

L'un des avantages majeurs de Java RMI est le stub. En effet, par ce biais Java RMI met à disposition du programmeur un outil très puissant et très simple qui se charge de la communication entre les différents acteurs d'un réseau. En effet nous n'avons plus à se préoccuper de la sérialisation et de la désérialisation des données envoyés et reçus par l'objet distant puisque le stub se charge de cela pour nous. L'un des inconvénients serait que Java RMI ne fonctionne que de java a java et donc manquerait de support pour des serveurs implémenter dans un autre langage.

Question 2 :

Nous allons chercher à comprendre les interactions entre les différents acteurs tout au long du processus d'exécution de notre code.

Tout d'abord on commence par exécuter la commande rmiregistry & qui va créer et lancer le remote object registry. Grace à ce dernier on peut retrouver les instances (stub) des différents serveurs qu'on a créés. Ainsi, lorsqu'on lance le serveur ce dernier va commencer par créer un objet de type UnicastRemoteObject avec la méthode exportObject(), cet objet sera par la suite « caster » avec l'interface du Server(ex : ServerInterface). Ainsi nous venons de créer notre « stub », suite à cela on va récupérer le registry avec la méthode getRegistry() de la classe LocateRegistry et lui assigné un stub qui sera relié sur le rmiregistry par un nom (exemple : « registry.rebind("server", stub); » va associer le stub en paramètre au nom « server » sur le rmiregistry). Cela va donc servir à établir la communication entre plusieurs acteurs sur le réseau.

Arrivé à cette étape, le client peut désormais charger le stub, pour pouvoir appeler notre objet distant. Ainsi avec la fonction loadServerStub() on va récupérer de la même manière le registry, ensuite on va demander au registry de nous retourner le stub souhaité. Par exemple, « stub = (ServerInterface) registry.lookup("server"); » , le lookup comme son nom l'indique va rechercher dans le registry et chercher le stub server qui aura été créer auparavant.

Suite à cela si le client veut effectuer un appel à une méthode de l'objet distant, il va passer par le stub créer qui représente une image virtuelle de l'objet distant, le serveur en l'occurrence. Le stub sérialise (ou marshalise) les arguments de la méthode lors de l'appel à l'objet distant et communique cela sous forme de flot de données en suite d'octets avec le squelette de l'objet distant. Le squelette va ensuite désérialiser les données et appelle la méthode de son objet. Suite à cela, pour la réponse de l'objet distant nous avons le même principe mais dans le sens inverse, sauf que cette fois c'est

le squelette qui va sérialiser les données et le stub chez le client qui va les désérialiser et présenter les résultats de l’appel au client.