TP1 – Appels de méthodes à distance

INF4410 - Systèmes répartis et infonuagique - Michel Dagenais

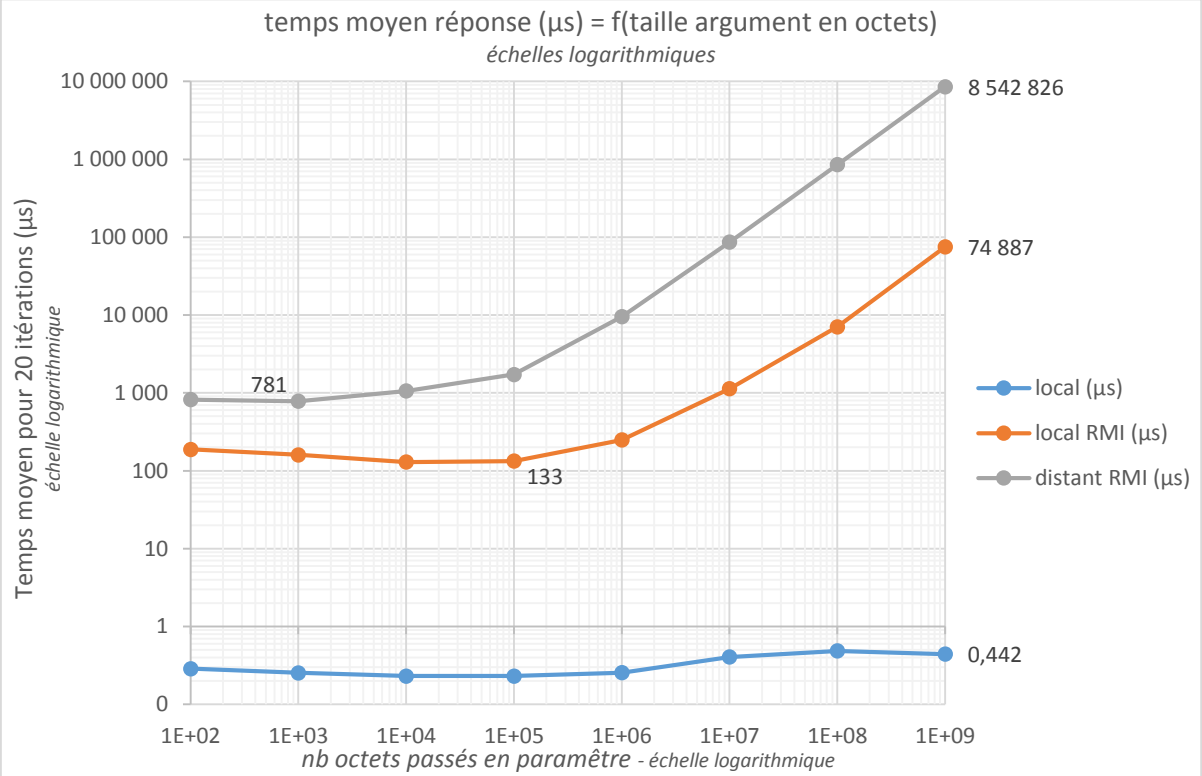
POLYTECHNIQUE MONTRÉAL
LE GÉNIE EN PREMIÈRE CLASSE

Antoine Giraud – #1761581

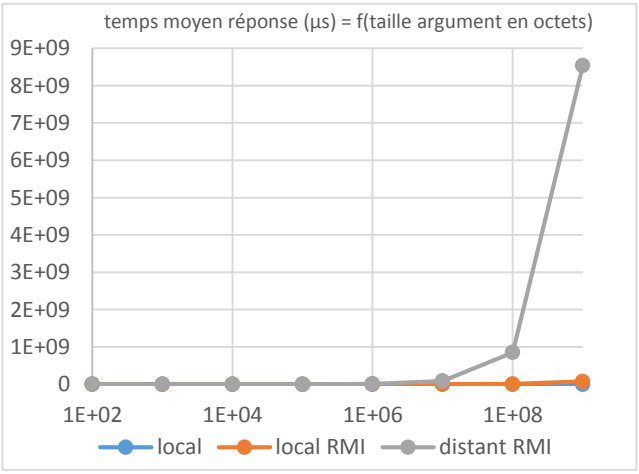
Le 04 octobre 2014

Partie 1 – Etude performance Java RMI

Question 1



nb octets	local (µs)	local RMI (µs)	distant RMI (µs)
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 en µs en fonction de taille de l'argument envoyé du client au serveur en octets.

Nous avions trois types de connexion client-serveur :

1. En local, nous avons en moyenne 0.324 µs. Nous pouvons noter tout de même que le système prend 1.75 plus de temps quand l'on passe un argument dès 10Mo, et ce jusque 1Go.
2. En local via RMI, nous avons en moyenne avant de passer à plus de 1e6 octets (1Mo) une moyenne de 172 µs pour le temps de traitement. Ensuite celui-ci croit exponentiellement (de base 10).
3. à un serveur distant via RMI, nous avions un temps de traitement plutôt peu stable d'environ 900 µs jusque 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 sur 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 apparait aux alentours de 1Mo.

Au final, 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 croit 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).

Donc au final, 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 croiserons à vitesse exponentielle (de base 10). Il n'est donc pas conseiller d'utiliser Java RMI pour échanger beaucoup d'informations, ou de gros objets dans le cas RMI. 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 ne pas qu'ils soient trop grands.

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 server 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 différentes 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 server que l'on voudra utilizer 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 extends de l'interface ServerInterface qui extends elle-même de la classe Remote de Java RMI, cela se fera en tout 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