Rapport TP1

INF8480
Systèmes répartis et infonuagique

Groupe 2

Présenté à : Houssem Daoud

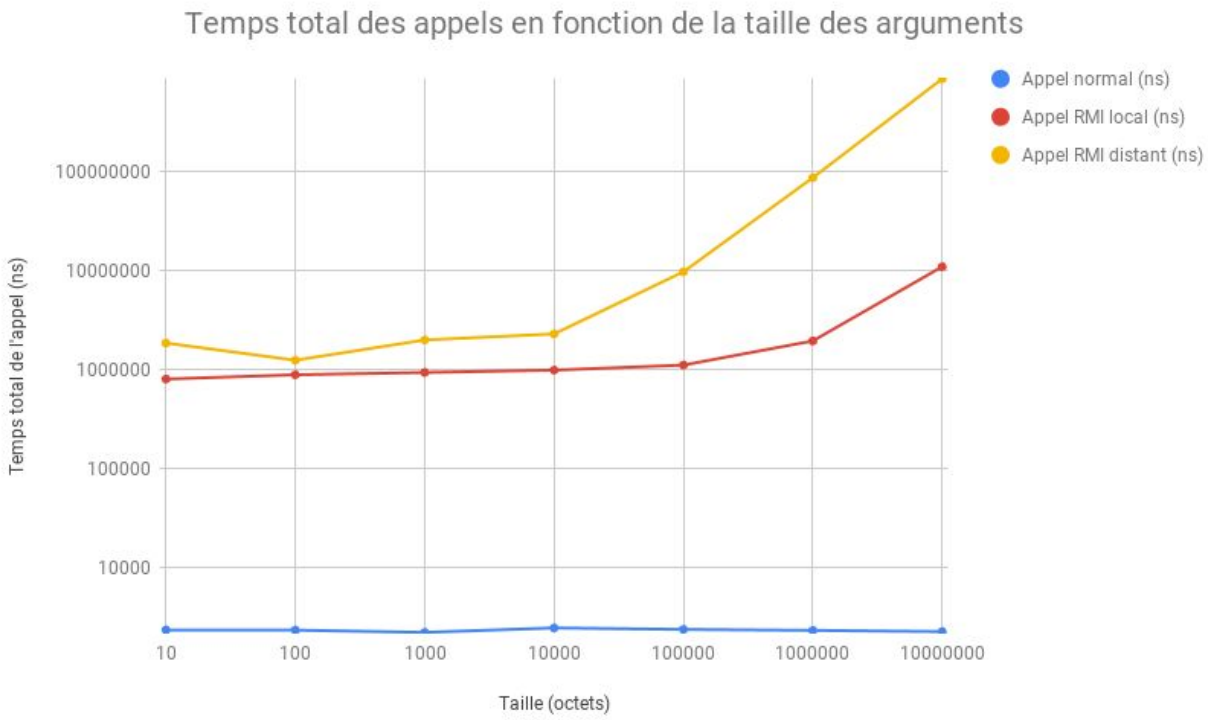
Soumis par :
Vincent Chassé 1795836
Quoc-Hao Tran 1972967

2 octobre 2018

Question 1

Pour effectuer l'expérience demandée, nous avons premièrement créer une fonction RMI vide qui prend en paramètre deux tableau d'octets (byte[]). On passe ensuite le facteur de 10 en paramètre à client.sh pour avoir les différentes valeurs de taille des tableaux. Nous avons aussi créé un script runEmptyTests.sh pour lancer les tests avec chaque taille demandée séquentiellement.

Voici les résultats que nous avons obtenus lors de l'expérience sous forme d'un graphique avec une échelle logarithmique (en base 10) :



Les données exactes sont disponibles sous forme de tableau dans le document *RéponseQuestion1.xlsx.*

Premièrement, on remarque que le temps pris pour l'appel local non RMI ne change pas selon la taille du tableau passé en paramètre, il reste presque parfaitement constant. Cela est normal puisqu'en local, Java peut simplement passer les paramètres par référence (l'adresse mémoire du paramètre). Cela est très rapide et ce temps n'est pas plus ou moins grand si la taille du paramètre varie. Les petites variations sont probablement causées par le fonctionnement du CPU au moment de chaque test (délais différents, usage du CPU, etc.).

Deuxièmement, on voit que le temps pour l'appel RMI local augmente lentement pour les tailles <=100000, puis augmente environ d'un facteur de 10 pour les deux dernières tailles. Puisque l'appel utilise RMI, les paramètres doivent être sérialisés pour être envoyé au serveur qui, dans

ce cas, est sur la même machine. Il n'y a donc pas de délai de transmission de paquets sur le réseau. Selon le graphique, on voit que faire un appel RMI local prend un temps d'environ 0.001s et que le temps pour sérialiser les tableaux d'octets devient assez grand pour augmenter ce temps d'un facteur 10 lorsque la taille est suffisamment grande.

Troisièmement, pour un appel RMI distant, la même logique s'applique que pour l'appel RMI local sauf qu'en plus, les paquets doivent être transmis sur le réseau jusqu'à la machine distante. Ce délai est plus important que la sérialisation. Le temps pour un tel appel varie grandement selon la vitesse de la connection internet des deux machines. Nous constatons bien dans le graphique que le temps écoulé pour l'appel distant augmente beaucoup plus rapidement que les deux autres types d'appel. Pour la taille la plus élevée, le temps est à un facteur 100 plus élevé que pour l'appel RMI local équivalent.

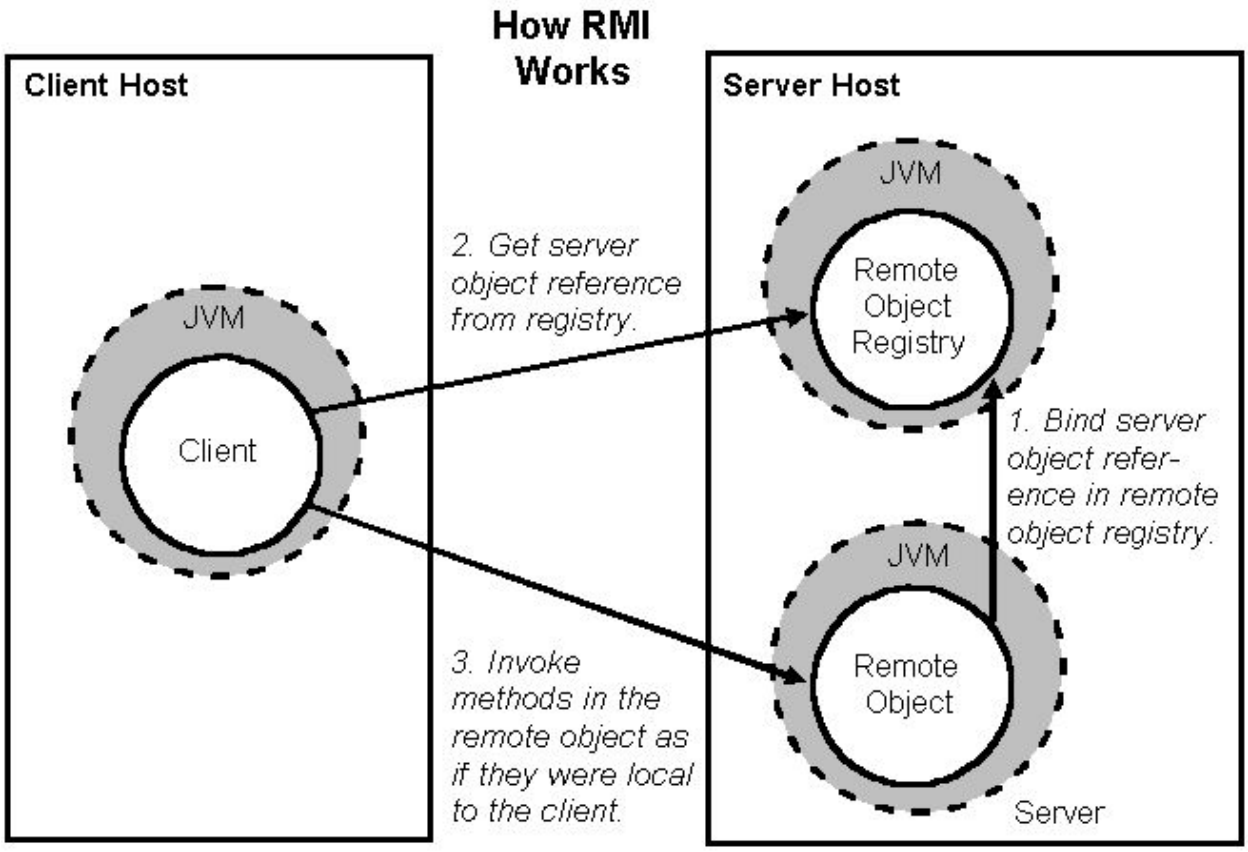
Un avantage de Java RMI est que le langage utilisé est le Java, les programmes qui l'utilisent peuvent être lancés sur plusieurs plateformes et architectures différentes. L'envers de cette médaille est que Java RMI peut seulement communiquer avec d'autres programmes utilisant aussi Java RMI, ce qui limite les possibilités.

Un autre avantage serait qu'il est pratique de pouvoir considérer les objets distants comme étant locaux grâce à l'abstraction faite par Java RMi. Par contre, il ne faut pas oublier de prendre en compte les possibilités d'erreur sur le réseau dont les *RemoteException* entre autre, ce qui peut diminuer la fiabilité du serveur.

Finalement, Java RMI est capable de gérer des appels effectués par plusieurs clients sans que le développeur ait à modifier le code. Tout fonctionne *out-of-the-box*.

Question 2

Un client appele une méthode de l'objet au serveur à travers le runtime RMI.
Voici le schéma avec des étapes qui décrit cet interaction:



Le runtime RMI détient un registre pour référencer les objets à distance. Au lancement du serveur, on instancie un stub qui représente l'objet qui va traiter les appels du client à distance. Cela est fait dans la fonction *run()* de la classe *Server* à la ligne 28 :

```
ServerInterface stub =
(ServerInterface)UnicastRemoteObject.exportObject(this, 0);
```

Le serveur enregistre ensuite ce stub dans le registre RMI. Pour ce faire il récupère une référence faire le registre à l'aide de la fonction *LocateRegistry.getRegistry()*. Le serveur associe ensuite un nom avec le stub précédemment créé en utilisant la fonction *registry.rebind(nom, stub)*. Le runtime RMI ensuite ouvre un socket et écoute sur un port anonyme pour toutes les requêtes d'invocation.

À la première connexion, le client demande au runtime RMI la référence des objets au serveur qui sont associe au registre RMI à l'aide de la fonction *LocateRegistry.getregistry(hostname)* dans la classe *Client*. Il faut ensuite que le client récupère le stub nécessaire à la

communication entre lui et le serveur. Cela est fait dans la fonction *loadServerStub()* dans la classe *Client* :

```
stub = (ServerInterface) registry.lookup("server");
```

Il faut s'assurer que le nom donné ici est le même qui a été utilisé pour le *registry.rebind(nom, stub)* du côté serveur.

Une fois tout cela fait, le client peut maintenant appeler des fonctions sur le serveur à partir du stub récupéré. Les appels locaux sont interceptés par le runtime RMI, qui envoie ces appels à travers un socket au serveur.

De coté serveur, le runtime RMI reçoit les appels et retourne les résultats au client.