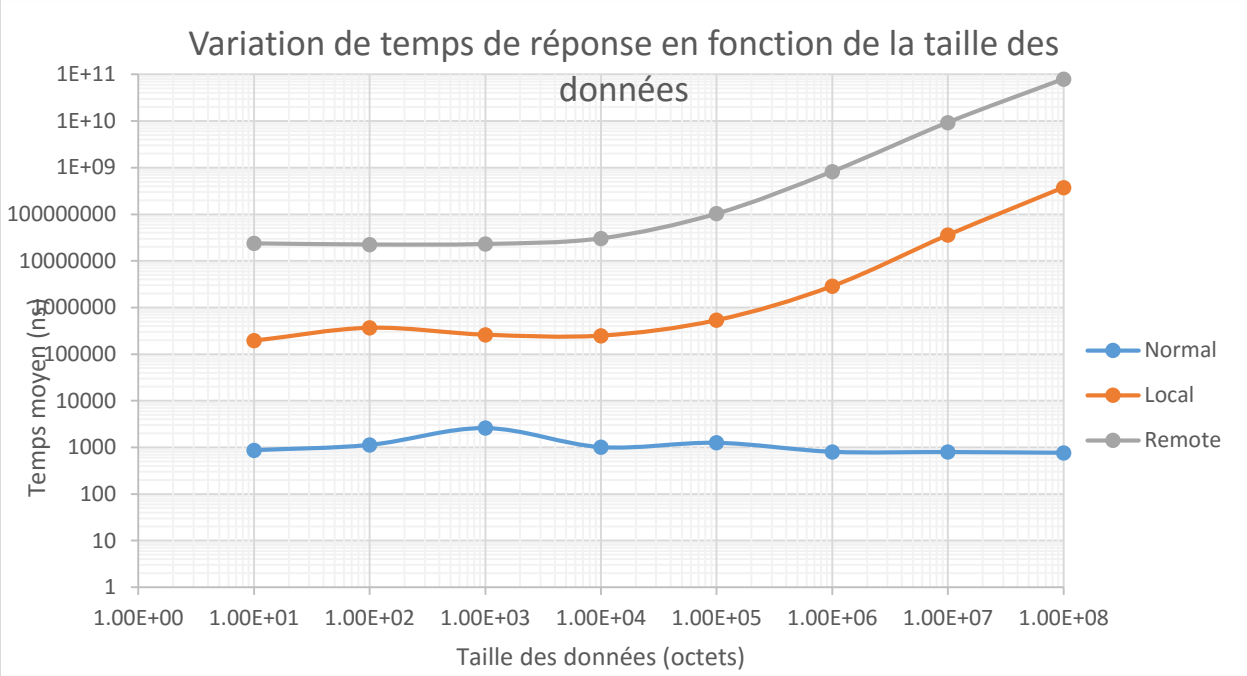
Guide d'utilisation

Pour utiliser le client, vous devez d'abord configurer quel serveur vous voulez utiliser. Pour ce faire exécutez la commande *client setServer [option]* avec pour option *local* pour un serveur RMI local ou le nom de l'hôte à utiliser.

Pour afficher l'aide, utilisez la commande *client*.

Pour exécuter le benchmark, utilisez la commande *client benchmark*.

Question 1



Notre méthode de test pour cette question consistait à appeler la méthode *execute* 100 fois avec en paramètre un tableau de valeur aléatoires. Nous avons roulé la simulation donc 100 fois avec chacune des tailles d'arguments variant entre 10^1 et 10^8, puis nous avons calculé la moyenne de temps de chaque appel. Ceci signifie environ 10Go de données, ce qui n'est pas très long avec une connexion décente.

Le code pour exécuter le benchmark se trouve avec le code du client, il suffit de l'appeler avec *client benchmark*. Une limite de 100Mo a été implémentée pour chaque test pour éviter d'envoyer trop de données.

Le code de ce test est en charge d'écrire tous les résultats moyens sous format .csv dans le dossier *results* et de les afficher dans la console. Les résultats peuvent ensuite être lus dans Excel pour créer un graphique.

Analyse des résultats

Comme nous pouvons le constater, le temps moyen de l'appel normal ne varie pas en fonction de la taille de l'argument donnée.

L'appel RMI local à un coût fixe de communication entre les processus, puis à partir d'un certain volume de données, la latence commence à augmenter. Dépassé 1Ko, les appels commencent à être de plus en plus lents, comme en témoigne la courbe orange sur le graphique.

Pour un serveur RMI distant, le résultat est le même, toutefois le coût fixe est encore plus grand. Ce coût fixe représente le temps d'envoi de la commande et de *marshalling* de la réponse. De plus, lorsqu'on dépasse un 1Ko le temps de réponse commence aussi à augmenter.

Il est important de noter qu'à un point dans la courbe, la bande passante de la connexion entre le client et le serveur contribuera à ralentir les appels.

Un bon exemple d'utilisation d'appel RMI serait pour effectuer des appels peu fréquents ou contenant peu de données, mais qui nécessitent beaucoup de temps de calcul. Par exemple, déchiffrer des mots de passe. Ainsi, la performance du logiciel ne serait pas compromise par la latence des appels RMI.

Un contre-exemple serait dans le cadre d'une application temps réel. Exemple, dans un jeu, on ne pourrait pas calculer le déplacement d'objets sans compromettre la fluidité du jeu.

Question 2

En imaginant que le registre RMI s'exécute déjà sur la machine serveur, il se passe une suite d'évènement lorsqu'on lance l'application serveur. D'abord, l'application localise le registre RMI dans son environnement (server.java#54). Le registre et le serveur n'ont pas besoin de s'exécuter dans la même JVM.

Ensuite, le serveur crée un stub de lui-même, lequel s'attache à un port qui servira à créer un socket RMI (server.java#51). Le serveur tente d'attacher son stub à une clé dans le registre. Cette opération s'effectue à l'aide de la commande *bind* ou *rebind* (pour remplacer) (server.java#55). Ceci fera en sorte que les clients pourront localiser le serveur en le demandant par sa clé au registre RMI.

Au niveau du client maintenant, l'application tente de localiser le registre RMI contenant le serveur. Pour ce faire on utilise la commande *LocateRegistry.getRegistry* à laquelle on passe en paramètre le nom de l'hôte contenant le registre en question (Client.java#227). Ceci ouvrira une connexion à un socket RMI du serveur contenant le registre. Le port par défaut pour cette communication est 1099 et la communication est basée sur le protocole TCP.

Si le registre est trouvé, on lui demande alors de nous fournir le stub du serveur qu'on demande grâce à sa clé à l'aide de la méthode *lookup* (Client.java#228). Ceci transférera un stub à notre client et ouvrira une autre connexion à un socket RMI, mais cette fois connecté au port défini par l'objet serveur. C'est par cette deuxième connexion que le client et le serveur communiqueront.

Finalement, à chaque appel fait par le client sur le stub du serveur, les appels et les arguments subiront le marshalling selon le protocole RMI puis s'exécuteront sur la machine distante, puis le résultat sera retourné au client de la même manière.