Chloé Constantineau – 1720146

Véronique Demers - 1783901

Question 1 :

|  |  |  |  |
| --- | --- | --- | --- |
| Payload Size(10x) | Appel local | RMI local | RMI distant |
|  |  |  |  |
| 1 | 210,946 | 864,175 | 1,297,597 |
| 2 | 180,640 | 954,596 | 1,345,955 |
| 3 | 204,022 | 958,319 | 1,698,735 |
| 4 | 179,697 | 932,457 | 1,948,195 |
| 5 | 211,677 | 943,243 | 9,762,439 |
| 6 | 311,425 | 2,319,707 | 87,120,317 |
| 7 | 2,667,122 | 13,523,159 | 886,522,213 |

*Les valeurs des temps d’appel sont en nanosecondes.*

Nous avons ajouté une fonction appelée EmptyFunc() dans notre interface ServerInterface. Cette fonction prend en paramètre deux arguments, chacun de type *byte[].* Cela veut donc dire que deux bytes array doivent être envoyé à chaque appel de fonction. Nous initialisons chaque byte array avec 10x ‘0’ où x varie entre [1,7]. Nous avons choisi d’envoyer deux bytes array au lieu d’un pour bien démontrer la lenteur d’exécution quand la taille des arguments augmente. Comme on peut le remarquer dans le graphique ci-haut, plus les arguments sont grands plus le temps augmente. On peut aussi remarquer que plus l’appel est distant, plus son temps augmente et ce pour un même ordre de grandeur de x. On peut donc en déduire que l’appel local est le plus rapide. Cela s’explique par le fait que la transmission des arguments se fait selon la vitesse d’exécution de la copie en mémoire de l’ordinateur (Temps copie mémoire par programme). Il s’agit du transfert de données le plus léger possible. Il est aussi très constant comme on en constate avec le graphique. L’appel avec un RMI local est le deuxième appel le plus léger. On y va tout de même une augmentation significative par rapport à l’appel local. Comme il s’agit d’un serveur local, la transmission des données se fait à même l’ordinateur avec une communication interprocessus. Le dernier appel, l’appel RMI distant, est définitivement le plus lent en termes de temps d’exécution. EN effet, contrairement aux deux derniers appels, ce dernier doit passer par une communication réseau. Cela implique donc de nombreux paquets à transmettre en réseau. Du fait de la nature de cette communication distante, le temps d’exécution s’allonge inévitablement et ce par un gros facteur.

Java RMI peut être un outil puissant. En effet, il permet de faire des appels de fonction sur des machines distantes. Cela permet par exemple de libérer du CPU sur la machine locale, ou bien de faire des calculs demandant beaucoup de capacité de *processing* sur une machine distante qui possède justement cette capacité. Cependant, Java RMI devient contreproductif du moment où les arguments passés sont de grandes tailles puisque le gain de temps de complétion de la fonction sur la machine distante est contrecarré par le temps de transmission des données. Il faut donc toujours garder cette optique en tête.

Question 2 :

Le premier processus lancé est celui du serveur (Server.java) :

1. Exécute le code du serveur sur une machine distante ou sur une machine locale, comme désiré.
2. Le code crée en premier un stub de notre objet ServerInterface et ce avec la fonction UnicastRemoteObject.exportObject fournit dans le package Java RMI.
3. On récupère la table du RMI registry avec la fonction getRegistry du package LocateRegistry
4. On ajoute une entrée à cette table ; le nom ‘server’ comme clé et le stub comme valeur. Cet ajout et ce binding se font avec la fonction *rebind.*

Ensuite, on exécute le client (Client.java) :

1. Le client commence par loader le registry. Pour ce faire, il utilise la fonction getRegistry() de LocateRegistry en lui donnant l’adresse du serveur : 127.0.0.1 pour un serveur local, ou bien l’adresse distante s’il s’agit d’un serveur distant.
2. Le client récupère ensuite un stub du serveur en appelant registry.lookup(‘server’). Le nom doit correspondre au nom donné dans le code du serveur ; la clé insérée dans la table.
3. Si le client veut faire un appel de fonction au serveur, il utilise le stub pour se faire. Lors de l’appel, les arguments envoyés en paramètre seront packagés (*marshalling*). L’appel de la fonction doit être fait selon la signature définit dans l’interface *ServerInterface* (interface commune au Server.java et Client.java)
4. Par la suite le serveur reçoit l’appel, de compacte les arguments, exécute la fonction demandée et renvoie la réponse au client.
5. Le client reçoit le résultat de son appel de fonction.