R6.B.05
« Optimisation de services complexes »
TP 1 : Load Balancing

L'objectif de ce TP est d'implémenter plusieurs algorithmes d'équilibre de charge pour un système réseau vues en cours :

— Aléatoire
— Round Robin
— Round Robin Pondéré
— Aléatoire Pondéré
— Least Connection

Tout ce dont vous avez besoin peut être téléchargé de Moodle. À la fin du TP, rendez l'ensemble de votre code sous forme d'une archive « .zip » sur l'espace de rendu Moodle.

Prise en Main

Dans un premier temps, télécharger le code depuis Moodle. Vous devez donc avoir ceci :

— `Client.jar` - Le code du client
— `Server.jar` - Le code du serveur
— `LoadBalancer.zip` - Le code du système d'équilibre de charge

Le code contenu dans les fichiers `.jar` n'a pas besoin d'être modifié et vous donne une base d'une interaction client - serveur. Le client envoie un nombre spécifié de messages à un serveur, espacés d'un intervalle de temps. Le serveur quant à lui, reçoit ces messages, effectue un traitement sur une durée de temps, avant de répondre au client. Le serveur possède également la file d'attente de messages, où il peut stocker les messages en attente de traitement. Cependant, si la file est plein, alors il répond directement avec un refus au client. Ces informations sont affichées, à la fois sur la sortie du serveur, mais également sur la sortie du client.

Votre travail intervient au niveau du système d'équilibre de charge. Vous avez donc quatre fichiers `.java` :

— `LoadBalancer.java` - Le code principal du Load Balancer.
— `AlgorithmTypes.java` - Le code concernant les types d'algorithmes implémentés
— `BalancingAlgorithm.java` - Le code de la classe abstraite définissant le format des algorithmies
— `NoAlgorithm.java` - Le code d'un algorithme basique en exemple

Prenez du temps pour étudier le code et comprendre son fonctionnement. Votre travail consistera à implémenter de nouvelles classes pour chacun des algorithmes mentionnés plus haut. Bien sûr, certaines modifications seront nécessaires au niveau du `LoadBalancer.java`, mais l'ensemble de la logique et les actions de vos algorithmes devraient s'effectuer à l'intérieur de leur propre classe. Vous pouvez vous inspirer du code de `NoAlgorithm.java` et `BalancingAlgorithm.java`. Ces deux classes **n'ont pas besoin d'être modifiés**.

Essayer d'exécuter le code Client et Serveur (`java -jar Client.jar`) pour s'envoyer des messages entre eux. Vous pouvez quitter les programmes à tout moment avec un `ctrl+C`. Les trois parties du TP (Client, Serveur et LoadBalancer) peuvent chacun prendre un certain nombre d'arguments. Ces arguments sont disponibles en utilisant la commande « *help* » comme premier argument. L'ensemble de ces arguments sont optionnels et possèdent des valeurs par défaut donc n'ont pas besoin d'être spécifiés. Cependant, l'ordre des arguments est obligatoire donc si vous souhaitez modifier le 2e argument, vous devez également spécifier le 1er, mais vous n'avez pas besoin du 3e.

Une fois que vous avez assuré que votre code fonctionne en mode *Client – Serveur*, vous pouvez commencer le TP.

1 Quel est le problème ?

Dans un premier temps, nous allons observer la problématique que nous cherchons à traiter. Commencez par lancer le Serveur et le Client de nouveau. Vous observez donc que le Client envoie ses messages à 1 seconde d'intervalle, et le Serveur prend également 1 seconde à les traiter. Dans tous les cas, le Serveur est assez rapide pour traiter chaque message à temps avant de recevoir la suivante. Il est important de noter aussi qu'ici, le Serveur possède une file d'attente avec une seule place, mais en vérité une place est réservée pour le message en cours de traitement par le serveur. Attention lorsque vous changez la taille de la file dans les prochains exercices.

Maintenant, relancez le Client, mais en réduisant l'intervalle de temps à 0.5 secondes (*attention, le temps se précise en millisecondes*) et observez le comportement du réseau. Essayez de jouer sur la taille de la file d'attente du côté Serveur.

Vous devez observer que dans tous les cas, à moins d'avoir une file d'attente très très grande, il y a toujours des messages qui sont malheureusement jetés. Votre objectif à travers les cinq algorithmes, est de réduire au maximum le nombre de messages jetés.

2 Aléatoirement aléatoire

Avant de commencer, lancez trois serveurs avec les paramètres de base. Notez les numéros de ports qui leur sont attribués. Modifiez le code du `LoadBalancer` pour identifier ces trois serveurs. **Attention par la suite à toujours relancer les serveurs dans le même ordre pour garder les numéros de port**. Lancez ensuite le `LoadBalancer` et notez également le numero de port associé. Enfin, lancez de nouveau le Client, pour envoyer 10 messages avec un intervalle de 0.5 s au `LoadBalancer`.

Vous devez observer le même comportement que précédemment puisqu'aucun algorithme n'a été implémenté pour l'instant. Commencez donc par créer la classe `LoadBalancer.Algorithms.RandomAlgorithm` et implémentez l'approche `aléatoire`. N'oubliez pas de modifier le code dans `AlgorithmTypes.java` pour ajouter votre implémentation.

Pour rappel, l'approche aléatoire choisie le serveur destinataire, justement de manière aléatoire parmi ceux disponibles. Testez votre approche et observez. N'oubliez pas de changer l'algorithme utilisé lorsque vous lancez le `LoadBalancer`. Relancez plusieurs fois le client pour observer des changements de comportement.

Comme attendue, l'approche aléatoire réduit le nombre de messages perdus, mais à cause de sa nature aléatoire, nous n'aurons pas toujours le même résultat, et un serveur peut se retrouver submergé.

3 Et on fait tourner les messages ...

Pour combattre ce problème, vous allez implémenter l'approche « **Tourniquet** », autrement appelé **Round Robin**. L'approche Round Robin est un fonctionnement une chaîne, ou chacun est choisi un après l'autre, toujours en tournant sur l'ensemble des possibilités. Créez donc la classe `LoadBalancer.Algorithms.RoundRobin` et ajoutez-le à `AlgorithmTypes.java`. Lancez ensuite votre algorithme et observez le comportement, de nouveau avec 10 messages et 0.5 secondes d'intervalle. Relancez le Client plusieurs fois de nouveau et comparez avec l'approche aléatoire.

Vous devez donc observer que les 10 messages reviennent au Client avec aucune perte. On voit donc que le LoadBalancer a fait son travail et a distribué les messages de manière efficace. Cependant, il faut noter ici que l'ensemble des Serveurs possèdent la même capacité de traitement, représenté par l'intervalle de traitement, jusqu'ici fixé à 1 seconde chacun. Dans un vrai déploiement, on peut avoir plusieurs types de systèmes cohabitant le même écosystème, possédant donc différentes capacités. Maintenant, relancez vos serveurs pour avoir cette distribution, pour simuler

trois machines avec trois capacités différentes :

— **Serveur 1** - 0.75s d'intervalle
— **Serveur 2** - 1.5s d'intervalle
— **Serveur 3** - 3s d'intervalle

À partir de maintenant, assurez-vous de l'ordre des serveurs si vous les relancez. Cela pourrait impacter votre algorithme.

Relancez votre Client et observez le comportement. Maintenant, nous perdons malheureusement des paquets de nouveaux ...

4 Pondération tournante

Une façon de faire est d'utiliser l'approche **Round Robin Pondéré**, qui vise à étendre l'approche précédente en appliquant un poids à chaque serveur qui correspond directement a sa capacité de traitement. Par exemple, si nous avons deux serveurs, un avec un poids à 2 et l'autre avec un poids à 1, pour chaque message envoyé au serveur 2, 2 seront envoyés au serveur 1. Cependant, la configuration des poids est un acte manuel, et donc l'administrateur doit effectuer les bons calculs afin de déterminer le meilleur poids.

Créez la classe `LoadBalancer.Algorithms.RoundRobinWeighted` qui implémente cette approche. Comme mentionné, vous devez calculer le poids des trois serveurs par rapport à leur intervalle de traitement des messages. Ces poids seront ensuite associés aux serveurs lors de leur configuration dans `LoadBalancer.java`. Mettez donc à jour le contenu de la fonction `setServeurs()`, afin de pouvoir ajouter le poids. Vous aurez à modifier plusieurs fonctions et des classes dans ce fichier. **Attention, les deux algorithmes précédents devraient toujours être capables de fonctionner malgré vos modifications**.

Une fois implémenté, testez avec la même configuration du Client et vos trois Serveurs. Vous aurez peut-être besoin d'augmenter la taille des files d'attente des Serveurs pour compenser. Choisissez une taille de file appropriée pour minimiser la mémoire utilisée, toutes en permettant le fonctionnement de l'algorithme.

5 Pondération ... Partout

L'approche Round Robin Pondéré est une approche couramment utilisée. Cependant, est-ce qu'elle est meilleure que l'approche l'aléatoire pondéré ?

Implémentez l'approche **aléatoire pondéré** `LoadBalancer.Algorithms.RandomWeighted` pour apporter l'avantage des poids dans l'approche aléatoire. Gardez la même configuration du Client et des Serveurs que précédemment et comparez avec l'approche Round Robin Pondéré.

Bien que plus « efficace » que l'approche aléatoire simple, on remarque que cette approche est toujours imprévisible, et on se retrouve parfois dans une situation ou un serveur se trouve surchargé de messages.

6 Soyons un peu plus intelligent

On a donc vu les avantages de l'approche pondérée par rapport aux approches simples. Cependant, comme vous avez remarqué, ces approches nécessitent une étape de configuration manuelle. Bien que cela paraît trivial dans cette situation, lorsque vous avez beaucoup plus de serveurs, avec des capacités variées ... Cela devient très compliqué. On peut donc utiliser une approche visant à distribuer la charge de manière équilibrée sur l'ensemble des serveurs. Cette approche s'appelle l'algorithme des « **Least Connections** ».

Concrètement, l'algorithme choisit le serveur selon l'état de sa file d'attente, privilégiant ceux avec moins de messages. Implémentez cet algorithme dans la classe `LoadBalancer.Algorithms.LeastConnections`. Testez ensuite votre approche et comparez-le aux précédents.

7 Robuste ? Ha, on va voir ...

Jusqu'ici, nous avons utilisé seulement 1 Client et 3 Serveurs. Cependant, dans la vraie vie on peut avoir beaucoup plus de Clients et des Serveurs plus performants. L'implémentation du `LoadBalancer` fourni est capable de supporter des connexions venant de plusieurs Clients en simultané. Testez les différents algorithmes en jouant sur les différents paramètres des Clients, Serveurs et LoadBalancer. **Attention cependant, seul UN Load Balancer peut exister dans ce contexte**.

Amusez-vous bien !