INF4410 – Travail pratique 2

Automne 2017

Services distribués et gestion de pannes

[1740216] - Ulric Villeneuve

[1739351] - Simon-Pierre Desjardins

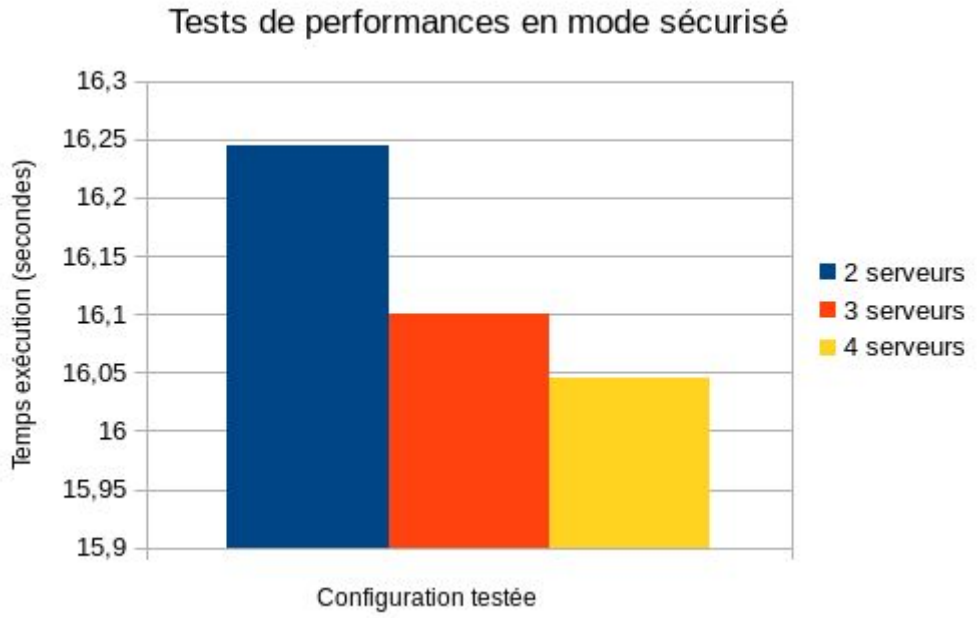
13 novembre 2017

Explication de nos choix de conception

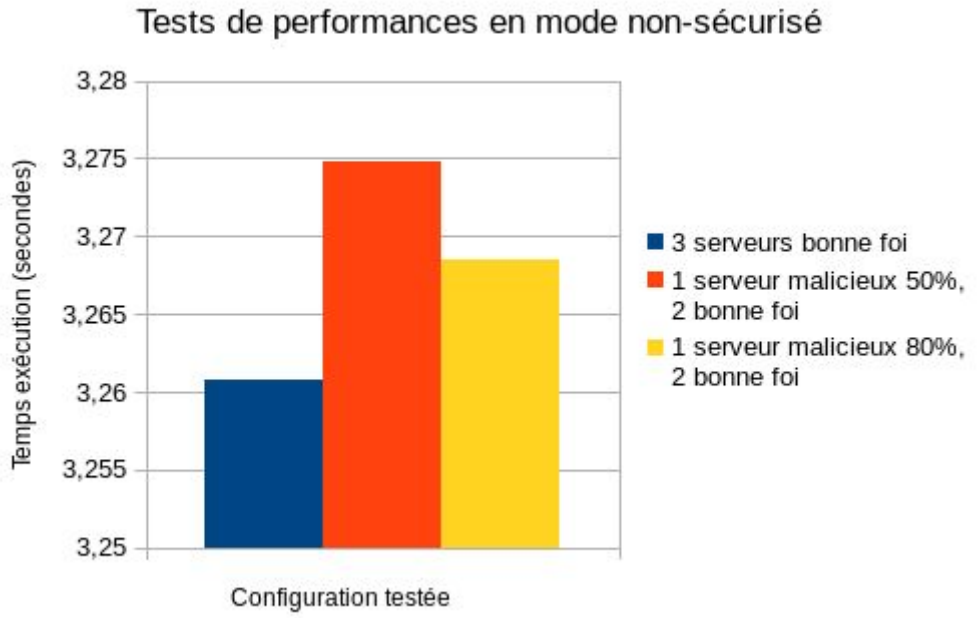
Pour le mode sécurisé, notre répartiteur divise les tâches avec un formule pour trouver le nombre d'opérations à envoyer qui correspond au nombre d'opérations précédent + nombre d'opération précédent * 0,2 + 1. Cela permet d'augmenter le nombre d'opération tant que le serveur accepte nos demandes afin de maximiser cette valeur. Lors que notre demande est refusée, nous diminuons le nombre d'opérations de 1 jusqu'à ce qu'elle soit acceptée. Lorsque celle-ci est accepté, nous lançons le thread pour le serveur en envoyant un vecteur contenant le nombre d'opérations en question. Après avoir lancé le thread, nous allons enlever les opérations que le serveur exécute de la liste des opérations. Par la suite, nous allons faire le même processus pour tous les serveurs. Lorsque tous les serveurs ont retournés une réponse, nous recommençons la boucle du début. Si un des serveurs se déconnecte pendant l'exécution, le thread effectue un callback au répartiteur en spécifiant que le serveur n'est plus disponible et en remettant les opérations qu'il effectuait dans la liste. Le reste de l'exécution se continue , mais en ignorant le serveur qui n'est plus disponible. Nous avons choisi cette méthode, car elle est plus efficace qu'en faisant les calculs de manière synchrone. En utilisant un thread pour chacun des serveurs, cela nous permet d'exécuter plusieurs calculs en même temps. Notre méthode n'est pas optimale, mais nous considérons que celle-ci nous permettrait d'obtenir un gain suffisant en performance.

Pour le mode non-sécurisé, notre implémentation est assez similaire en général. La première différence est que le nombre d'opération sera toujours de la taille du petit q des serveurs. Cela s'explique par le fait que nous aurons à confirmer nos résultats entre ceux-ci et nous ne voudrions pas qu'une demande soit refusée. Nous allons ensuite faire le premier bloc d'opération sur tous les serveurs avec un thread par serveur et nous allons ensuite comparer les résultats afin de vérifier si nous avons au moins deux résultats pareils. Si c'est le cas, nous allons passer au prochain bloc d'opération. Sinon, nous allons refaire le calcul sur tous les serveurs jusqu'à ce que nous trouvons deux valeurs pareils. Nous avons choisi de prendre cette méthode au lieu de seulement partir deux serveurs et lancer un troisième fil d'exécution si les valeurs ne sont pas pareils, car dans le cas où nous avons un taux malicieux élevé, une méthode qui lance un fil d'exécution tant que nous avons pas deux valeurs pareils sera beaucoup plus lente. En lançant 1 fil d'exécution sur chaque serveur pour chaque bloc nous permet d'augmenter les chances de trouver le bon résultat en une seule itération. Dans le cas où un serveur n'est plus disponible (ne fonctionne plus), nous allons l'enlever de la liste des serveurs disponibles et nous allons faire la vérification des résultats avec les serveurs restants.

Dans les serveurs (appelés Calculator dans notre travail), nous avons déterminé si le serveur va renvoyer un résultat malicieux en choisissant un nombre aléatoire entre 0 et 100, si celui-ci est plus petit que le m, alors le serveur va retourner un faux résultat. Pour simuler qu'il effectue les calculs, nous avons effectué toutes les opérations qu'il auraient normalement dû faire, mais notre résultat final sera un nombre au hasard entre 0 et 499999.



Comme nous pouvons le constater, l'ajout d'un serveur permet de diminuer le temps d'exécution mais de manière assez minimale. En effet, nous avons effectué de nombreux tests avec un fichier contenant 2000 opérations afin de valider nos temps pour chacune des configurations, mais ceux-ci pouvaient varier considérablement (par exemple un temps de 13,682 secondes a été enregistré dans un des tests pour la configuration à deux serveurs). Cela peut s'expliquer par notre fonction qui calcule le nombre d'opérations qu'on tente d'envoyer au serveur en question ainsi que par les probabilités que le serveur accepte ou non la demande. En effet, notre fonction n'est fort probablement pas optimale en terme de rapidité ce qui pourrait être amélioré pour le futur. De plus, dans la deuxième phase de demande d'opération, nous diminuons notre u de 1 à chaque demande jusqu'à ce qu'il accepte. Cette méthode peut également être non optimal et devrait probablement avoir une correspondance avec notre fonction d'augmentation du nombre d'opération. Bref, nous remarquons quand même que les trois temps sont très similaires et c'est majoritairement la taille de l'argument q qui influence le temps d'exécution, car nos calculs sont assez rapide à faire comparativement au temps qu'il faut pour faire une demande de calculs.



Voici la façon dont nous procédons pour les calculs en mode non-sécurisé. Nous séparons tous les opérations en tâches de la taille du plus petit paramètre q des serveurs de calculs. Nous faisons la demande, qui sera toujours acceptée, car le nombre d'opérations u que nous envoyons est inférieur ou égal au paramètre q de chaque serveur. Ensuite, nous envoyons la tâche à chacun des serveurs. Après avoir reçu la réponse de chacun d'eux, nous vérifions que nous avons reçu au moins deux fois le même résultat, si oui, nous l'additionnons au résultat et nous envoyons une autre tâche aux serveurs, si non, nous renvoyons la même tâche, jusqu'à ce que ce résultat soit vérifié.

De cette façon, lorsque nous avons au moins deux serveurs de bonne foi, en envoyant la tâche qu'une seule fois à chaque serveur nous nous assurons d'avoir au moins 2 bonnes réponses et donc que le résultat pour cette tâche soit vérifié. Donc les résultats obtenus avec 3 serveurs de bonne foi, 2 serveurs de bonne foi et 1 de mauvaise foi (peu importe la valeur de son paramètre m) ne devrait pas changer le temps d'exécution selon notre implémentation. C'est ce qu'on peut observer dans les résultats obtenus; on voit que les résultats sont très similaires peu importe la configuration utilisée, tant que nous ayons 2 serveurs de bonne foi. La différence qu'on observe peut être causée par une latence dans le réseau ou par des processus sur les machines qu'on utilise pour tester.

Question 1: Présentez une architecture qui permette d'améliorer la résilience du répartiteur. Quels sont les avantages et les inconvénients de votre solution? Quels sont les scénarios qui causeraient quand même une panne du système?

Une solution possible serait de posséder un ou plusieurs répartiteurs de secours. Le répartiteur principale serait ainsi constamment en train de transmettre ses informations aux serveurs de secours en cas de panne. Lors d'une panne, les serveurs pourraient se connecter au prochain répartiteur disponible qui aurait toutes les informations nécessaires afin de continuer le processus comme si rien ne s'était produit. Cela a l'avantage d'avoir aucun temps d'arrêt ou très peu dans l'exécution ce qui permet de compléter les tâche sans interruption. L'inconvénient est qu'il est plus coûteux en terme de ressource et d'argent d'avoir à garder constamment plusieurs répartiteurs dans le cas d'une panne qui pourrait techniquement ne jamais se produire. Il pourrait quand même y avoir une panne dans le cas où tous les répartiteurs de secours ne fonctionnent pas ce qui devrait quand même être plus rare. Il pourrait également avoir une panne avec une coupure de l'électricité, mais cela pourrait être contourné avec des répartiteur à différentes localisations ou même en ayant quelques-uns connectés à des génératrices. Il pourrait quand même y avoir une panne dans le cas où le réseau devient indisponible pendant un certain temps, car il faudrait repartir les serveurs de calculs afin de les rendre disponible à nouveau.