POLYTECHNIQUE MONTRÉAL

LE GÉNIE EN PREMIÈRE CLASSE

06 NOVEMBRE 2018

INF8480 – SYSTÈMES RÉPARTIS ET INFONUAGIQUE

TP2 – SERVICES DISTRIBUÉS ET GESTION DES PANNES

AURORE LOISEAU 1973902
ALEXANDRE GOURGAUD 1973924

Table des matières

Choix d'implémentation

L'idée générale était de pouvoir répartir et paralléliser un ensemble de calculs entre différents serveurs. La répartition restant à la charge d'une entité unique qui est le répartiteur. Le mécanisme adjacent à l'authentification et au serveur de nom était imposé. Ainsi les choix commencent à partir du moment où le répartiteur a en sa possession une liste de calculs à effectuer et une liste de serveurs vers lesquels les dispatcher.

Premier constat et premier choix, l'ordre d'exécution des calculs n'a pas d'importance, les termes d'une somme étant permutables. Nous avons donc opté pour la queue comme structure de données pour stocker les calculs. De même, c'est la queue que nous avons choisie pour le stockage centralisé des stubs serveurs. La notion d'ordre des serveurs n'était, selon nous, pas prioritaire. Ils ne sont pas triés par capacité décroissante, car cela aurait conduit à sursolliciter un même serveur sans vraiment pousser à sélectionner ceux de moindre capacité, mais appelés en parallèle.

En Java, gérer le Thread à la main est fastidieux, il faut ouvrir les threads, les stocker puis enfin les attendre. Dans le cas où un serveur appelé en premier a plus de calculs qu'un second appelé ensuite (C différents), on perdra du temps. Les waits de fin de thread seront effectués dans l'ordre de création des threads et le deuxième thread devra attendre que le premier soit récupéré pour être récupéré à son tour. Il perd du temps, car il a déjà reçu depuis longtemps le résultat de son stub. On pourrait déjà avoir relancé un nouveau thread avec le même stub pour un autre jeu de calculs. Le cas d'un serveur refusant le calcul est aussi très important, car on souhaite le replacer rapidement dans la queue des serveurs de calculs disponibles. Il en va de même pour crash réseau. Afin de pallier ce problème, nous avons choisi d'utiliser deux abstractions très puissantes de Java.

La première est l'utilisation du ExecutorCompletionService. Cette classe est un wrapper surpuissant pour la gestion d'un pool de threads ou plus exactement ici de Callable. On donne à une instance de ExecutorCompletionService des Callable dans l'ordre que l'on veut. Il est ensuite capable de nous retourner les résultats des Callable s'exécutant en son sein dans l'ordre de terminaison de ces derniers. Ainsi le thread principal pourra récupérer les résultats partiels le plus tôt possible et relancer des nouveaux calculs plus tôt. L'alternance entre envoie et réception de Callables pour un ExecutorCompletionService est native.

La deuxième est l'utilisation des Callables. Contrairement aux Threads et aux Runnables pour qui la valeur de retour n'est pas gérée nativement, les Callables possèdent by design cette fonctionnalité. Ainsi pour chaque Thread et donc pour chaque appel à un serveur de calcul, les résultats sont acheminés jusqu'au thread principal du répartiteur. Il est ainsi le seul à ajouter où à supprimer des stubs de la queue des stubs disponibles. Il est également le seul à ajouter ou à supprimer des calculs dans la queue des calculs de départ ou celle des résultats.

Ainsi les structures de stockage n'ont pas besoin d'être Thread safe et synchronisées. Elles sont donc plus rapides à utiliser.

Envoyer à chaque serveur un nombre de calculs tel qu'il y ait 15% de risque de se voir refuser la tâche par le serveur pour raison de surcharge nous a semblé pertinent d'après une étude empirique sur différents jeux de test.

Pour le cas du mode non sécurisé, nous avons fait l'hypothèse que, comme dans les tests de performance, les serveurs de calcul avaient des capacités similaires. Ainsi nous envoyons à deux serveurs différents simultanément le même sous ensemble de calculs. Un Callable intermédiaire a la charge de ces appels et renvoie au Thread principal les résultats validés et ceux non validés. On choisit comme nombre de calculs le minimum de capacité entre les deux serveurs sélectionnés multiplié par la constante correspondant au 15% de risque de refus. Dans le cas où les résultats sont différents, ils sont replacés dans la queue des calculs de départ.

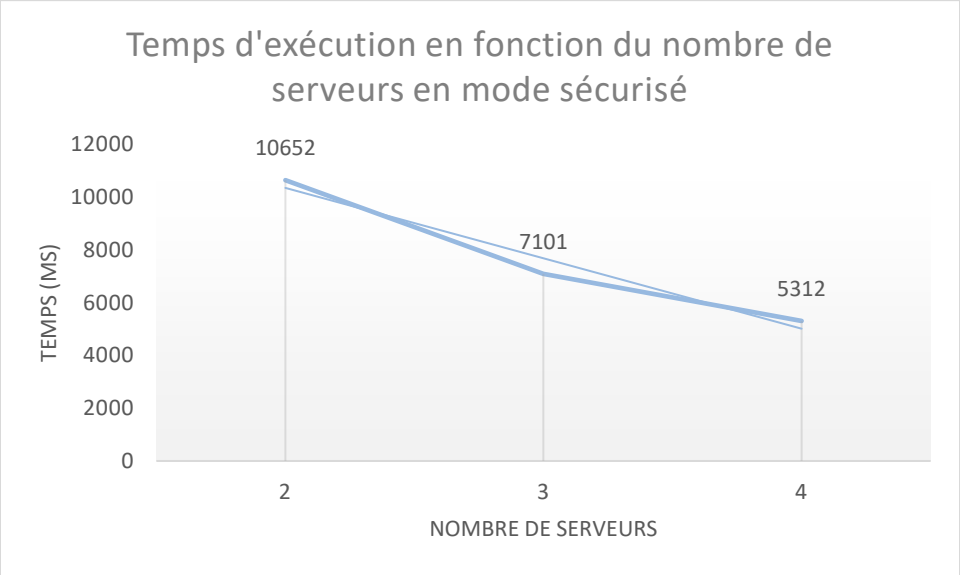
Tests de performance

Mode sécurisé

Une instance du système en mode sécurisé a été créée afin de réaliser les tests de performances en mode sécurisé. Pour ces tests, nous avons utilisé le fichier « operations-800 » comprenant 6000 calculs à réaliser, permettant d'obtenir des temps d'exécution de quelques secondes.
Le même calcul a été exécuté avec un nombre différent de serveurs disponibles :

- 2 serveurs disponibles avec C_1=4, C_2=4
- 3 serveurs disponibles avec C_1=4, C_2=4, C_3=4
- 4 serveurs disponibles avec C_1=4, C_2=4, C_3=4, C_4=4

Les temps d'exécution obtenus sont présentés dans le graphique ci-dessous :



On observe une diminution quasi linéaire des temps d'exécution avec l'augmentation du nombre de serveurs disponibles. Cela s'explique en partie par nos choix d'implémentation : le système d'abstraction de gestion des threads nous permet de paralléliser l'exécution des tâches envoyées aux différents serveurs de calcul. Cependant, dans ces tests de performance, les serveurs ont tous la même capacité, ce qui ne permet pas de voir l'avantage tiré d'avoir utilisé un ExecutorCompletionService à la place d'un système de thread classique retournant le résultat des threads dans l'ordre de lancement de ces derniers.
La diminution n'est pas totalement linéaire en raison de l'augmentation des appels réseau liée à l'augmentation du nombre de serveurs. En effet, une tâche de 10 calculs envoyée à un seul serveur sera plus rapidement réalisée que deux tâches de 5 calculs envoyées à deux serveurs successivement. Cette raison n'est pas totalement visible dans les résultats obtenus étant donné le faible nombre de serveurs, mais le serait en augmentant progressivement et plus largement le nombre de serveurs.

Mode non sécurisé

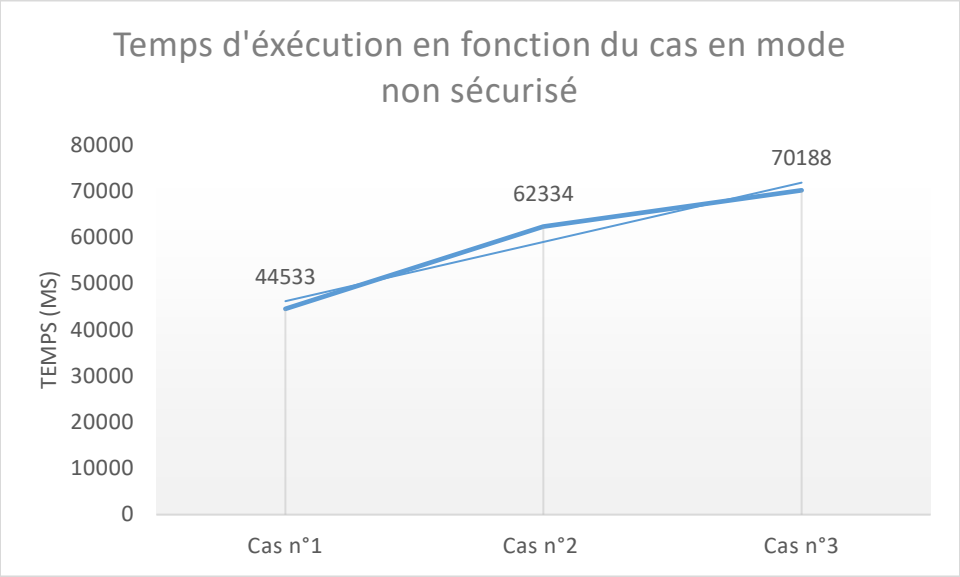
Une instance du système en mode non sécurisé a été créée afin de réaliser les tests de performances en mode non sécurisé. Pour ces tests, nous avons utilisé le même fichier « operations-800 » que précédemment comprenant 6000 calculs à réaliser.

Le même calcul a été exécuté dans trois différents cas:

- Cas n°1 : Les trois serveurs sont de bonne foi (taux de mauvaises réponses du 0%).
- Cas n°2 : Un serveur est malicieux 50% du temps, et deux serveurs sont de bonne foi.
- Cas n°3 : Un serveur est malicieux 75% du temps, et deux serveurs sont de bonne foi.

Dans les trois cas, la capacité des serveurs était de 5.

Les temps d'exécution obtenus sont présentés dans le graphique ci-dessous :



Le premier constat que l'on peut effectuer est la lenteur en ayant que des serveurs de bonne foi comparé au même nombre de serveurs en mode sécurisé. En mode non sécurisé, le calcul est 6 fois plus lent. Premièrement, cela s'explique par le fait que chaque calcul est effectué deux fois. Également, nous n'avons qu'un nombre pair de serveurs qui tournent en mode non sécurisé. C'est comme si l'on avait un seul serveur en mode sécurisé.

Aussi la lenteur s'explique par le fait que deux serveurs doivent simultanément accepter une même tâche. Au final un calcul ne sera effectué que 72% du temps.

Questions de réflexion

Architecture avec répartiteur plus résilient

L'idée dans cette partie est proposer un système avec un répartiteur tolérant aux pannes. Dans l'architecture actuelle, le répartiteur n'est pas redondé, il représente donc un point unique de défaillance (*single point of failure*). L'idée générale, si l'on s'abstrait des contraintes techniques liées au réseau support des serveurs, est de multiplier le nombre de répartiteurs. On parle alors de répartiteurs redondés. Cela implique bien sûr que les serveurs de calcul puissent recevoir simultanément des tâches de plus d'un répartiteur à la fois. En conséquence, ces derniers devront avoir une méthode moins naïve pour déterminer leur acceptation ou non d'une tâche. Ce calcul sera basé sur le taux d'utilisation du serveur de calcul en question.

Bien que ce ne soit pas implémenté actuellement, l'idée est d'avoir un client qui envoie sa liste de calculs à l'un des répartiteurs redondés. Le répartiteur ayant reçu la liste agira comme il le fait actuellement en distribuant la charge aux serveurs de calculs. Le répartiteur renverra au client le résultat final. Pour la communication entre les clients et les répartiteurs, on choisit une communication sous la forme d'appels HTTP en mode REST comme avec n'importe quelle API. L'avantage de ce paradigme est de pouvoir utiliser un grand nombre de clients différents basés sur le web, mais également de fournir aux développeurs une approche simple et générique pour soumettre des tâches à notre système.

Pour répondre à la question épineuse de l'équilibrage de la charge entre les différents répartiteurs redondés et la haute disponibilité, on choisit de mettre en place plusieurs serveurs nginx accessibles au travers d'une même IP de service virtuelle. Le service de haute disponibilité fera une gestion automatisée du « failover/failback » d'une instance nginx. En amont, le routeur central enverra les requêtes vers l'IP de service. Ci-dessous le schéma de notre système, on montre ici le nombre minimal de machines pour la mise en place de la solution décrite au-dessus.

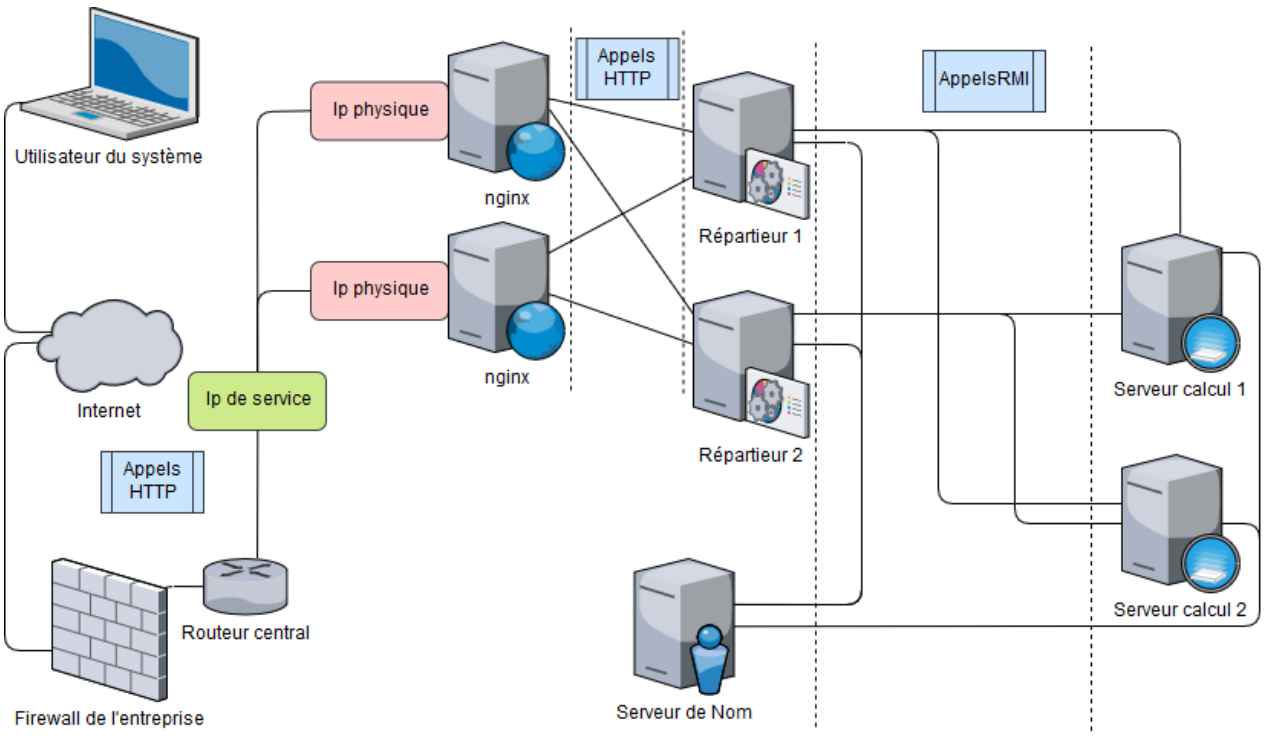


Schéma de l'architecture plus résiliente avec un nombre minimal de serveurs

Avantages et inconvénients de notre solution

L'avantage principal de cette solution est la haute disponibilité multi niveau qu'elle propose. En effet, le répartiteur n'est pas le seul redondé, le load balancer nginx l'est aussi. L'implémentation de la communication jusqu'aux répartiteurs sous la forme d'appels HTTP est aussi un réel avantage, car il standardise le client en faisant abstraction du backend codé en Java. Pensée de la sorte, chaque couche (nginx, répartieur, serveur de calcul) peut être mise à l'échelle de manière très flexible afin de s'adapter au nombre d'utilisateurs.

Il existe néanmoins des inconvénients liés aux compromis de notre architecture. Le plus gros étant son coût. En effet, on ajoute une couche logique complète de load balancer web nginx. Si ce service est d'ordinaire léger, il peut devenir gourmand lors de montées en charge. Le service logique gérant l'IP de service a lui aussi un coût. En outre, les serveurs de nom et les serveurs de calculs doivent aussi être adaptés à la gestion de plus d'un répartiteur.

Scénarios pouvant poser problème

Malgré les améliorations apportées par notre architecture, certains points restent critiques.
Dans le cas du mode de calcul non sécurisé, aucun système n'assure qu'au moins deux serveurs de calculs soient démarrés. De la même manière, il manque un système meta au-dessus de nos couches pour monitorer et assurer un nombre minimal d'instances de machines sur chaque couche. Un tel système aurait aussi la tâche d'adapter (augmenter ou diminuer) le nombre de machines par couche en fonction du taux d'utilisation du système.
Également, le serveur de nom n'est pas redondé, c'est un problème, car il reste un élément central du système entre les répartiteurs et les serveurs de calculs.
Pour être encore plus critiques, on peut noter que dans le schéma présenté, le routeur central n'est pas non plus redondé.