École Polytechnique de Montréal

TP2 - Répartiteur de tâches

Benoît Paradis, 1677688

Laurent Tremblay,1642342

11 novembre 2016

INF4410

Systèmes répartis et infonuagique

POLYTECHNIQUE
MONTRÉAL

Table des matières

1 Introduction

Une des idées principales d'un système répartis est d'être résilient, c'est à dire fortement tolérant aux pannes. Il serait en effet impossible, pour n'importe quel système de taille non-triviale, de garantir le fonctionnement simultané de toute les machines le composant. Google a estimé, dans une étude publiée en 2007 [1], que la probabilité qu'un disque dur brise pendant une année d'utilisation normale est de 1.7% pour un disque neuf et de 8.6% pour un disque de trois ans.

Ces valeurs, multipliées par la quantité de disques fonctionnant en même temps dans une infrastructure moderne, indique qu'un système doit être capable de tolérer les pannes afin d'être utilisé de façon efficace.

Le fait de répartir un système sur un ensemble de machines reliés entres-elles par un réseau implique aussi de devoir séparer un ensemble de tâches en sous-unités plus simples et de répartir ces tâches équitablement entre les nœuds du réseau afin d'obtenir une performance optimale.

Notre répartiteur est un seul programme s'exécutant sur une machine et distribuant du travail à plusieurs nœuds de calculs. C'est une architecture maître-esclave classique, tel qu'illustré à la figure 1.

2 Présentation de l'architecture

Si le fait d'avoir plusieurs nœuds de calcul permet une redondance du système, le répartiteur en soi est un *"singleton"* et ne comprend aucun mécanisme de redondance et représente un point unique de défaillance. Notre implémentation se doit d'être robuste et a donc été développé avec un style défensif. Certaines des *"10 falacies of*

1. `https://static.googleusercontent.com/external_content/untrusted_dlcp/research.google.com/en//archive/disk_failures.pdf`

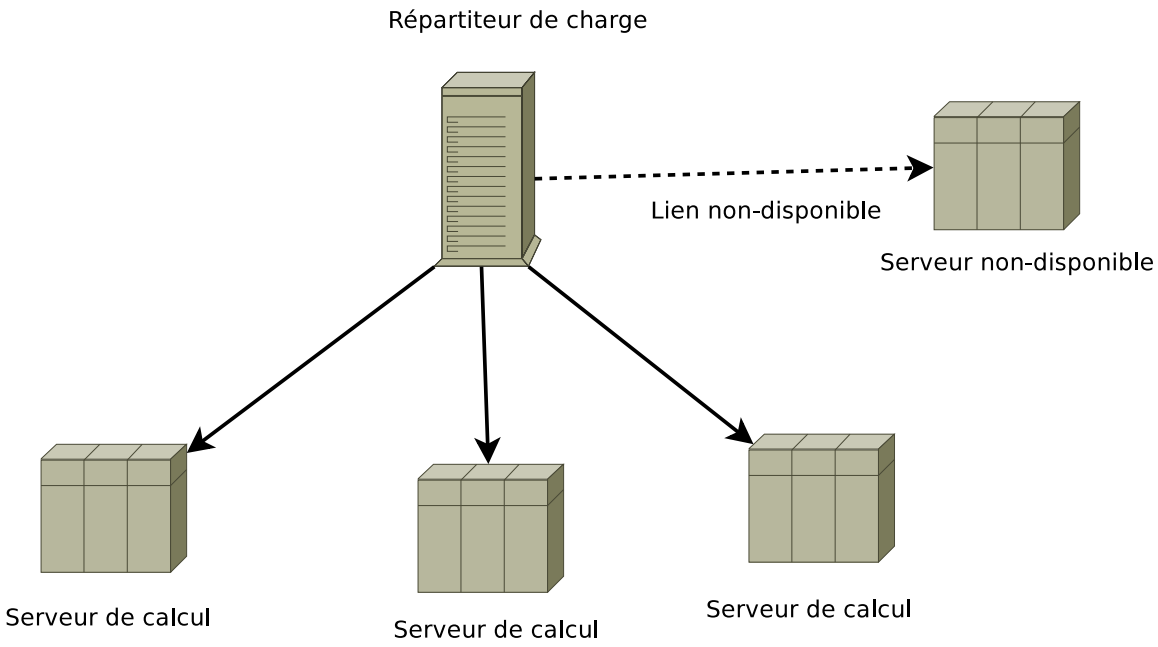

FIGURE 1 – L'architecture du répartiteur de charge

distributed computing" ont été prise en compte lors de l'élaboration et implémentation de notre architecture.

Le serveur lui-même est écrit en utilisant un *thread-pool* afin de gérer les serveurs de calcul. C'est une solution plus élégante et simple que d'attribuer un *thread* par nœud de calcul, en raison du grand nombre potentiel de nœuds et du fait que les serveurs ne sont pas utilisés de façon intensive (on attribut un lot de travail et on attend de recevoir le résultat).

Java RMI a été utilisé afin d'implémenter l'exécution de méthodes à distances. Ceci évite d'avoir à implémenter sois-même la couche réseau et permet de profiter du mécanisme d'exceptions de Java ainsi que pouvoir isoler les erreurs de réseau lors du développement de l'application. L'utilisation de Java permet aussi de faire appel à des fonctions du langages, comme les *Futures*, simplifiant grandement la réduction effectuée par le répartiteur. Celui-ci attend en effet simplement que les *Futures* soient disponibles. La communication entre les nœuds et le répartiteur est donc asynchrone par nature.

L'implémentation non-sécuritaire repose sur un consensus où une majorité de machines doivent donner le même résultat pour une série d'opération données afin d'être considéré comme valide.

2.1 Gestion des serveurs morts

Puisque nous utilisons Java RMI pour exécuter les opérations sur les serveurs de calcul, il est possible d'utiliser les mécanismes d'exceptions de Java et de RMI afin de détecter si un serveur de calcul est indisponible. Le répartiteur recevra simplement une exception de type *RemoteException* quand il tentera d'exécuter un lot de travail sur un nœud n'étant plus disponible. Aucun mécanisme de *keep-alive* n'est implémenté par le répartiteur, toute la communication entre le répartiteur et les nœuds étant encapsulé dans la couche de Java RMI. Le répartiteur peut ensuite donner le lot de travail à un autre nœud de calcul et périodiquement réessayer d'attribuer du travail au nœud ne répondant pas. Si la panne persiste, le répartiteur retirera simplement le nœud de calcul de sa liste de nœuds valides. La panne peut se produire n'importe-quand durant l'exécution du lot de travail, puisque l'exécution d'un lot de travail est une opération atomique du point de vue du répartiteur (elle est soit exécutée au complet, refusée ou une exception est levée par Java RMI signalant qu'il est impossible de l'exécuter).

2.2 Déterminer la capacité de chaque serveur

Chaque serveur est vu comme une entité unique ayant des propriétés distinctes par le répartiteur. Le répartiteur ignorant le facteur q de chaque serveur, ce dernier doit l'estimer. La stratégie pour estimer le facteur q est d'envoyer un lot de travail d'une taille donnée. Si ce dernier est accepté, on conserve la taille du lot de travail. Si ce dernier est refusé, on diminue la taille du lot de travail d'une opération et on tente de la re-soumettre. L'opération retirée est replacée dans la liste des opérations

en attente et sera ajoutés à un prochain lot de travail. Une taille de 15 comme lot de travail initial a été choisie de façon empirique à partir des valeurs de q typiques données dans l'énoncé. Le fait de choisir un q légèrement plus gros que les valeurs de l'énoncé permet de profiter du fait que certaines opérations où $q > u$ pourraient quand-même être exécutées avant que, statistiquement, une d'entre-elles échoue et force le répartiteur à diminuer le nombre d'opérations dans sa requête.

3 Test de performance - mode sécurisé

4 Test de performance - Mode non-sécurisé

5 Réponse Question 1

Question 1 : Le système distribué tel que présenté dans cet énoncé devrait être résilient aux pannes des serveurs de calcul. Cependant, le répartiteur demeure un maillon faible. 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 ?

Notre approche serait de faire fonctionner plusieurs répartiteurs en parallèle, afin de permettre à un répartiteur de tomber en panne sans arrêter le système au complet. Dans un scénario idéal, les répartiteurs communiqueraient entre-eux afin de se distribuer un sous-ensemble des tâches à effectuer et confirmer aux autres répartiteurs les tâches ayant été données au nœuds de calculs et ayant été complétés, pour éviter qu'une même tâche ne soit exécuté deux fois et comptabilisé deux fois. Ce n'est pas exceptionnellement grave si une tâche est exécuté deux fois, en autant que cette dernière ne sois pas comptabilisé deux fois lors de la réduction. Cette architecture est illustrée à la figure 2.

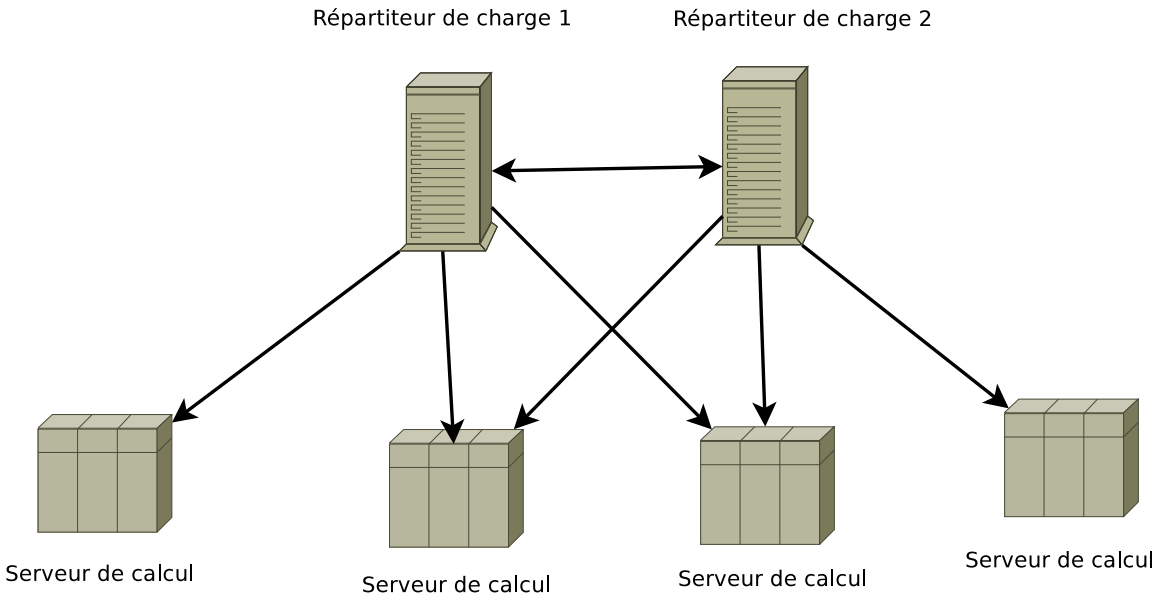

FIGURE 2 – Architecture éliminant le point unique de défaillance

Utiliser plusieurs répartiteurs indépendants pose toutefois un problème de taille : Une mauvaise configuration ou un problème de réseau peut maintenant partitonner notre infrastructure en deux, tel que les répartiteurs ne se "voient" plus, tel qu'illustré à la figure 3.

On aurais le problème du P du théorème CAP, c'est à dire que le système peut devenir partitionnée et que les répartiteurs peuvent essayer d'assigner les mêmes tâches à deux serveurs sans se coordoner. Notre système doit donc faire le choix entre rester disponible ou être consistant.

Une solution serait de donner une copie de l'ensemble des tâches à réaliser à chaque répartiteur et d'utiliser des messages de synchronisation pour s'assurer que les tâches ne soient exécutées qu'une seule fois et comptabilisé une seule fois. Dans le cas d'un partitionnement, un seul des deux serveurs devrait continuer d'opérer normalement, un serveur dit "chef", l'autre se mettant en attente du premier serveur afin de se faire renvoyer la liste des tâches effectuées depuis le partitionnement par le serveur afin de pouvoir continuer l'exécution à deux serveurs en parallèle, tel qu'illustré à la figure 4.

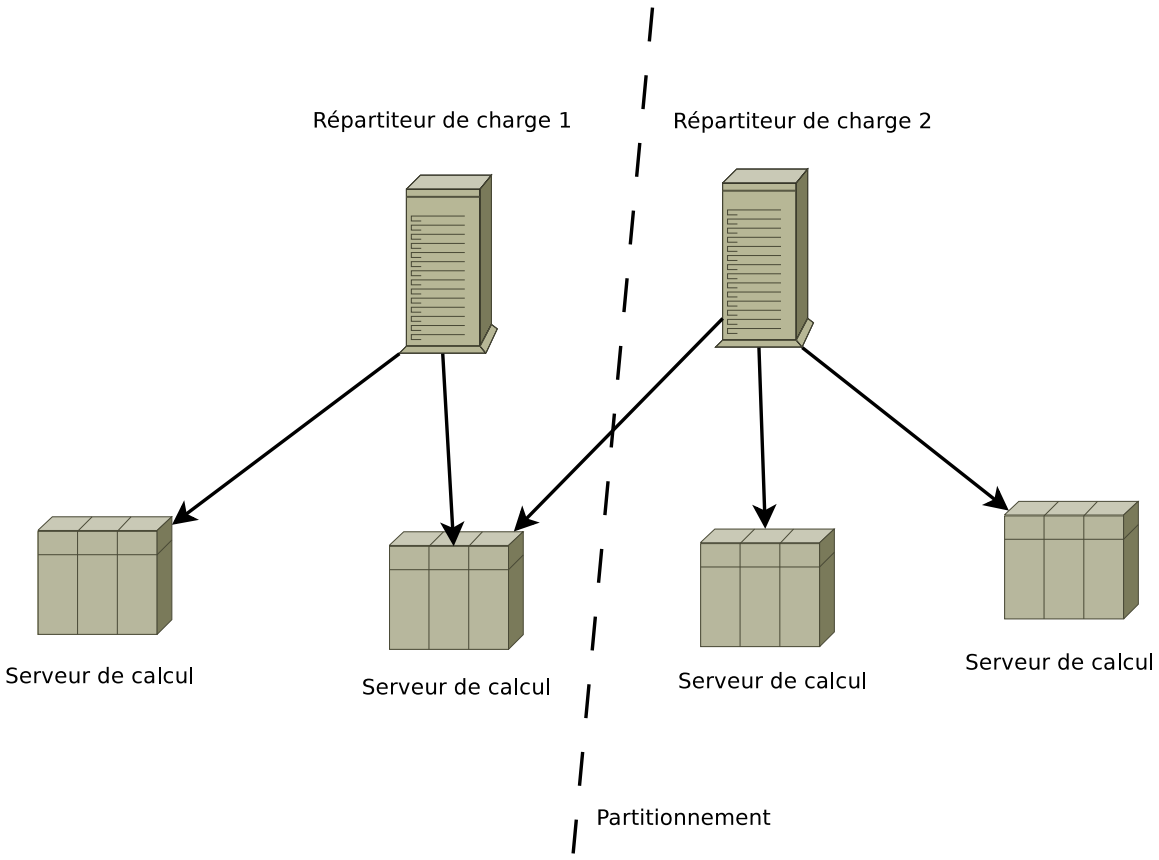

FIGURE 3 – Partitionnement possible de l'architecture redondante

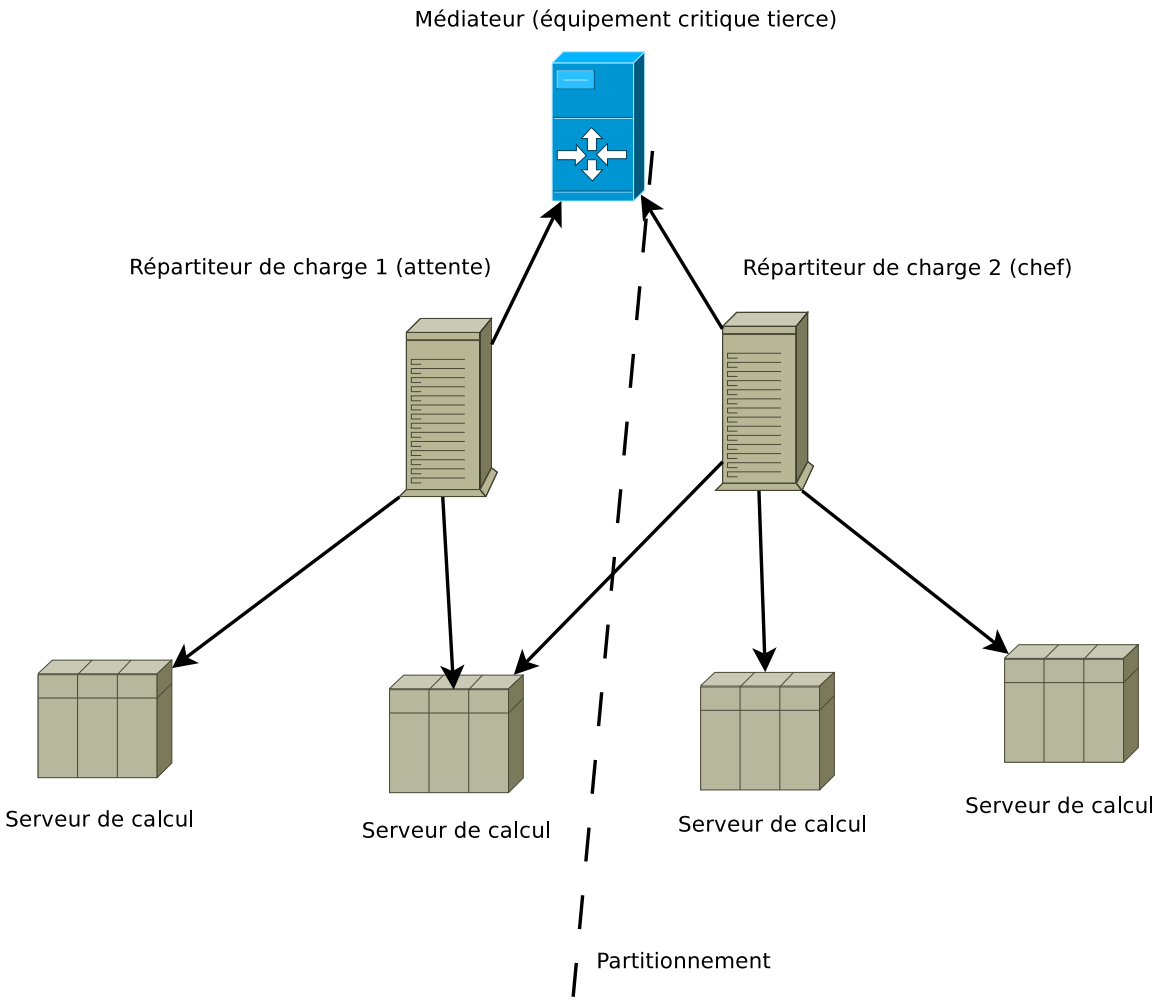

FIGURE 4 – Médiation du partitionnement par un arbitre externe

Cette approche permet une consistance des données (on retombe dans le cas du maître-esclave traditionnel et du singleton) mais le système sera plus lent et moins disponible.

Un problème de cette approche est toutefois d'identifier le partitionnement lorsque ce dernier se produit et de déterminer quel répartiteur doit agir comme "chef". Le cas trivial d'un serveur totalement déconnecté du réseau est évident à traiter, puisque ce dernier ne peut plus voir aucun autre nœud ou répartiteur, mais le cas où la partition isole les répartiteurs l'un de l'autre mais où des nœuds de calculs sont toujours accessibles, déterminer un chef est un problème de taille en sois. Une solution serait de choisir une machine tierce comme point de référence pour notre système, comme une *switch* réseau centrale où un serveur particulièrement robuste. En cas de partitionnement, tout serveur étant capable de rejoindre cette pièce d'équipement sera le répartiteur "chef". Cette solution permet même à cette pièce d'équipement, de signaler qu'elle a déjà donné le contrôle à un autre répartiteur, si deux répartiteurs sont capable de la contacter. Cette solution est toutefois vulnérable à d'autres scénarios de partitionnement.

Une autre approche est de faire travailler les répartiteurs comme si de rien était mais sans effectuer la réduction finale sur les résultats des calculs. Quand un autre répartiteur reviendra accessible, ces derniers communiqueront quelles opérations ils ont effectués et s'assureront d'effectuer la réduction uniquement une fois sur chaque opération.

Une dernière approche serait de donner des états aux serveurs de calcul, mais cette approche comporte aussi son lot de problèmes additionnels. Nous préférons grandement une implémentation où les serveurs de calculs agissent sans conserver d'états, comme dans une architecture REST traditionnelle.

6 Conclusion et implémentations alternatives.

En terme d'optimisations, notre répartiteur repose sur plusieurs heuristiques (choix du nombre de requêtes initial) n'ayant pas été étudiés. Un profilage détaillé de l'application ainsi que la variation de ces paramètres pourrait permettre de les fixer à des valeurs optimales en fonction du type de charge de travail à effectuer. De plus, notre implémentation se base sur un réseau relativement fiable et constant en terme de latence et de débit. Aucune optimisation n'est effectués pour se protéger d'un réseau fiable mais au performances aléatoires. À titre d'exemple, le choix des serveurs ne se fait pas en prenant en compte la performance du réseau. Un serveur inoccupé mais connecté à un réseau très lent sera favorisé par rapport à un serveur très occupé mais connecté à un réseau très rapide et offrant, au final, de meilleures performances. Une façon de remédier à ce problème serait de faire mesurer le temps de réponse d'un serveur pour une requête donnée par le répartiteur et la comparer à une valeur heuristique correspondant à l'exécution locale de ce lot de travail afin de déterminer la pénalité imposé par le réseau pour utiliser ce nœud en particulier.

Il serait aussi possible d'obtenir le facteur d'utilisation des serveurs directement en les interrogeant plutôt qu'en envoyant des lots de travail de taille aléatoire. Théoriquement, rien n'empêche le serveur de renvoyer des informations sur sa charge de travail actuelle.