

**INF8480- Systèmes répartis et infonuagique**

Groupe 01

**Travail pratique #2**

**Services distribués et gestion de pannes**

**Présenté à Adel Belkhiri**

**David TREMBLAY 1748125**

**Oswald Pichot 1921329**

Département de génie informatique et génie logiciel

Le 6 novembre 2018

École polytechnique de Montréal

# Choix de conception

Dans ce deuxième travail pratique, nous avions comme mandat la conception d’un système distribué résilient aux pannes. Par contre, une grande flexibilité nous était accordée pour réaliser la tâche au niveau de la conception et de l’implémentation. Le reste de cette section explique donc les solutions qui ont été sélectionnées ainsi que la raison derrière ces choix.

Répartition des tâches: Tout d’abord, notre programme insère les opérations du fichier de calcul ligne par ligne dans un tableau. Par la suite, nous envoyons une tâche qui correspond à un certain nombre de lignes du fichier. Ce nombre est 2 fois la capacité qui garantit l’acceptation à chaque serveur. En moyenne, 75% des serveurs accepteront donc la tâche qui leur est demandé. Par contre, si le nombres de lignes restantes dans le fichier est inférieur à 2 fois la capacité d’acceptation garantie, nous envoyons tout simplement ce qui reste au serveur. Nous avons choisi des tâches de tailles 2\*Ci de manière arbitraire. Le taux d’acceptation était élevé avec une telle taille et le nombre de tâches nécessaires étaient raisonnables pour la taille de nos fichiers de calculs de tests.

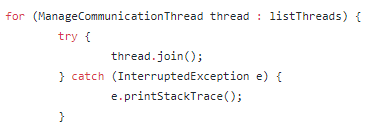
*Figure 1: Création d’une tâche pour un serveur de calcul de capacité d’attribution garantie calculator.getValue()*

Gestion des tâches échouées: Notre gestion des tâches refusées est très simple. Nous ne faisons que renvoyer la même tâche au même serveur. Avec 75% de chance d’acceptation, un même tâche devrait rarement avoir à être renvoyée plus de 2 ou 3 fois avant d’être acceptée. Cette méthode est simple puisqu’on sait que le serveur est inoccupé (il vient d’indiquer qu’il n’accepte pas la tâche) et nous possédons déjà une tâche de grandeur optimale à lui envoyer (celle refusée). Nous avons opté pour cette méthode puisque le comportement désiré pour la gestion des pannes n’était pas spécifié. Nous avons donc opté pour l’option qui nous semblait la plus simple.

# 

*Figure 2: Renvoi d’une tâche échouée*

Détection de pannes intempestives: Nos threads avec les différentes tâches en cours d’exécution sont dans un bloc try/catch qui attrappe les exceptions de type *InterruptedException*.Donc, lorsque qu’un thread en attente, en dormance ou occupé est interrompu (ce qui se passe lors d’une panne), notre répartiteur de tâche est averti. Nous retirons ce serveur de tâche de la liste des serveurs de calculs disponibles puis attribuons sa tâche au prochain serveur disponible. Nous avons opté pour cette approche puisque cette exception sert à des cas d’interruptions de threads comme son nom l’indique et est donc tout à fait adaptée et simple d’implémentation dans notre contexte de système distribué.



*Figure 3: Renvoi d’une tâche échouée*

Résultats

Tests de performance - mode sécurisé

D’après la théorie, on s’attend à observer une diminution du temps de calcul lorsque le nombre de serveurs augmente. En effet, Les opérations sont traitées en parallèles sur les différents threads du système distribué. Il est donc normal qu’un grand nombres de tâches exécutées en parallèle permettent d’obtenir une réponse finale plus rapidement puisque nos calculs sont indépendant un des autres (un calcul n’a pas besoin du calcul précédent pour s’exécuter). Par contre, la diminution du temps n’est pas linéaire (2 serveur n’est pas 2 fois plus lent que 4 serveurs). En effet, plus de serveurs de calculs entraîne plus de threads à gérer et plus de communication réseau entre le répartiteur, serveur de calcul et serveur de nom. Il y a donc un coût plus important sur le réseau qui entraîne une latence dans les réponses entre les différents composants du systèmes et c’est cette latence qui explique la non-linéarité d’une augmentation de serveurs. Les serveurs de calculs avaient tous la même capacité dans ces tests. Par contre, la théorie suggère qu’une augmentation de la capacité des serveurs de nom entraînerait forcément une diminution du temps de calcul total puisque des plus gros morceaux du fichier à calculer pourrait être envoyer, réduisant ainsi le nombre d’appels réseau nécessaire et le nombre de threads à gérer pour réaliser la tâche.

Tests de performance - mode non sécurisé

La capacité n’est pas la même en mode non sécurisé qu’en mode non sécurisé. Nous ne pouvons donc pas clairement voir l’impact mais il ne fait aucun doute que le mode non sécurisé est beaucoup plus long que le mode sécurisé. En effet, dans le meilleur des cas (tous les serveurs de bonnes foi), il faudra faire 2 fois plus de tâches que ce qui aurait été nécessaire en mode sécurisé puisque chaque tâche doit être validée par un deux serveurs validant la même réponse pour la même tâche. Ceci prend donc deux fois plus de communications entre les composants du système et on doit aussi attendre la réponse de non un mais deux serveurs de calcul avant de valider le résultat causant encore plus de délai. Maintenant, pour ce qui est des serveurs malicieux, plus nous avons de serveurs malicieux ou de serveurs malicieux avec un pourcentage de malice plus élevé, plus le temps d’exécution sera long. Ceci est tout à fait normal puisque la tâche réalisée par le serveur malicieux devra être à nouveau à deux autres serveurs ralentissant ainsi grandement l’exécution du programme. Le temps de calcul devrait toutefois être linéaire avec le pourcentage de malice des serveurs. En effet, si on ne tient pas compte du temps requis pour instancier le programme et pour retourner le résultat mais seulement le temps de calcul, chaque requête devrait être environ le même temps. Nous devons donc ajouter le temps supplémentaire pour refaire le nombre de tâches aux serveurs malicieux \* la probabilité de malices \* 2 (puisque nous devons renvoyer deux fois la tâche pour chaque réponse malicieuse du serveur). Ce raisonnement n’est toutefois valide que parce que nous n’avons qu’un serveur malicieux.

Questions de réflexion

**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 architecture qui améliorerait la résilience du répartiteur serait une architecture qui se rapproche de la gestion des pannes dans les serveurs de calculs. En effet, comme pour les serveurs de calculs, il pourrait y avoir possibilité de plusieurs hôtes pouvant accueillir le répartiteur. En cas de défaillance, un répartiteur secondaire prendrait le relai et les le programme continuera de bien fonctionner. Par contre, le répartiteur actif devrait avoir une façon d’indiquer au répartiteur secondaire les résultats intermédiaires et l’états des threads pour que le répartiteur sache d'où continuer sans manquer d’opérations ou d’avoir à tout recommencer.

Un avantage de cette solution est qu’elle est assez simple d’implémentation. Le changement de répartiteur d’a pas vraiment d’impact sur le serveur de noms et les serveurs de calculs qui devront maintenant communiquer avec le nouveau répartiteur tout simplement. Aussi, on pourrait penser à utiliser les 2 répartiteurs dans le cas d’un fichier anormalement gros pour traiter le résultat rapidement malgré que certains changements supplémentaires seraient nécessaire pour implémenter cette action.

L’inconvénient majeur de cette solution est bien entendu les ressources qui sont plus utilisées. En effet, on doit maintenant garder en mémoire l’état du répartiteur en tout temps en cas de défaillance. Ceci prend donc du temps et des ressources supplémentaires. Ceci complexifie aussi davantage notre système distribué et il faut gérer l’envoi de la liste de serveur par le serveur de nom au nouveau répartiteur et l’envoi des réponses des threads vers le nouveau répartiteur lorsqu’une tâche se termine après que le répartiteur soit tombé en panne.

Si nous n’avons que deux répartiteurs et que les deux tombent en panne (les deux sont dans le même local qui subit une inondation par exemple) alors on se retrouve avec le même problème qu’avant. Si l’état du répartiteur n’est pas à jour, qu’elle se perd ou se corrompre, alors, le programme continuerait à fonctionner mais les résultats seraient erronés, ce qui est presque aussi inutile qu’un programme qui s’arrête. Par contre, en cas de perte ou corruption, on pourrait recommencer le calcul au complet. Cette solution serait longue mais permettrait d’obtenir notre réponse avec comme seul désavantage que le programme prendrait plus de temps à s’exécuter.

Il semble pertinent de mentionner que dans une telle architecture, nous avons maintenant de la redondance partout sauf pour le serveur de nom qui est maintenant le maillon faible du système. Si celui-ci tombe en panne, le répartiteur ne pourra pas obtenir les serveurs de calculs au début. S’il tombe en panne plus tard, les serveurs de calculs ne pourront valider les requêtes du répartiteur. Dans les deux cas, le programme ne fonctionnera pas. Il serait donc pertinent d’implémenter une redondance tant au niveau du répartiteur qu’au niveau du serveur de nom.