Nous avons déterminé qu'un taux de refus de 25% était optimal, de façon expérimentale, c'est-à-dire que nous avons répété les appels en variant le taux de refus d'une tache, de façon à obtenir le temps d'exécution le plus cours pour un fichier donné. Cependant, il faut tenir en compte le taux d'utilisation du réseau qui peut avoir fausser les résultats pour d'autres valeurs. Nous allons cependant maintenir le taux de refus à 25% pour l'entièreté des résultats présentés dans ce rapport.

Test de performance – mode sécurisé

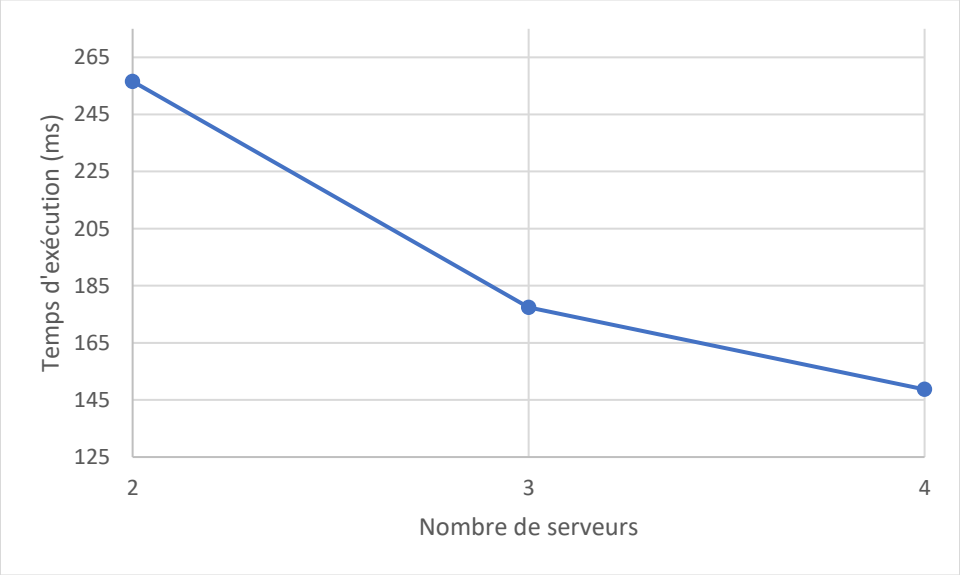

Figure 1 : Graphique du temps d'exécution (en ms) en fonction du nombre de serveurs

Pour recueillir ces résultats, nous avons répété les tests 10 fois, rejeté les données extrêmes de façon à garder 6 résultats pour chaque nombre de serveurs.

Il serait logique de croire qu'augmenter le nombre de serveurs diminue le temps d'exécution, et c'est ce qu'on observe dans ces résultats. Notre implémentation commence d'abord par diviser l'ensemble des tâches à calculé en blocs; la taille des blocs est déterminée par le taux de refus maximal choisi lors de l'exécution du répartiteur. Par la suite, le code boucle jusqu'à ce que l'entièreté des blocs aient été assignés : le premier serveur disponible se voit assigné le prochain bloc. Lorsque tous les serveurs sont occupés, le répartiteur « cherche » un serveur disponible, et lui assigne le prochain bloc. De cette façon, plus on ajoute de serveurs, plus les blocs peuvent être distribuées rapidement, et donc la réponse finale est obtenu plus rapidement. Dans tous les cas, le *bottleneck* est le rassemblement des multiples résultats par le répartiteur.

Test de performance – mode non-sécurisé

Dans le cas de trois serveurs de bonne foi, on obtient un temps d'exécution (selon la méthodologie présentée plus haut) de 311 millisecondes; un temps raisonnable et similaire aux autres résultats du mode sécurisé.

Dans le cas d'un serveur malicieux 50% du temps, et de deux serveurs de bonne foi, il était nécessaire de diminuer la taille de l'échantillon en raison des temps d'exécutions élevés. Nous avons donc un temps d'exécution moyen de 584 secondes (3 exécutions au total).

Dans le cas d'un serveur malicieux 75% du temps et de deux serveurs de bonne foi, on obtient un temps d'exécution moyen (en usant de la même méthodologie que le cas précédant) de 785 secondes.

Les temps sont causés par notre implémentation. En effet, pour faciliter la gestion lors de résultats incorrect, nous répétons le même calcul avec les mêmes deux serveurs jusqu'à l'obtention d'un résultat acceptable. Évidemment, il ne serait pas souhaitable d'utiliser une telle gestion lorsqu'il est possible qu'un serveur soit malicieux 100% du temps. De façon similaire, il aurait peut-être été préférable, lors du refus d'un résultat d'un bloc (les deux réponses ne sont pas la même), de remettre ce bloc dans la liste des tâches à complété de façon à les assignées à d'autres serveurs, et ainsi espérer avoir une autre paire de serveurs. De façon complémentaire, il aurait été nécessaire de randomiser le choix des deux serveurs, plutôt que de choisir des serveurs de façon séquentielle.

Bref, le choix de l'implémentation (répétition jusqu'à une réponse acceptable) cause les délais d'exécution élevés lorsqu'un serveur est malicieux, et plus ce serveur est malicieux, plus le temps d'exécution augmente. De plus, si un serveur peut être malicieux à 100% du temps, l'exécution ne terminera pas.

Questions de réflexion

L'utilisation d'une adresse IP pour le répartiteur est un maillon faible. Il faudrait utiliser un identifiant qui permettrait de trouver l'adresse IP : un nom de domaine avec un *lookup* effectué à un serveur DNS qui fournit l'adresse IP du répartiteur. Par la suite, il serait pertinent d'assurer une redondance du répartiteur : au moins 2 serveurs, supposons les serveurs *A* et *B*. Le A rempli le rôle de répétiteur. Pendant ce temps, le serveur B est en attente, il écoute pour une panne du serveur A de quelconque façon. Lorsque A tombe en panne, B met à jour le serveur DNS et redirige le trafic vers lui. Les requêtes associées au serveur A sont perdu, et les clients auront à gérer le renvoie vers B. Lorsque le serveur A est de nouveau en état de fonctionner de façon adéquate, il se met en attente et écoute pour une panne du serveur B afin qu'il alterne et redevienne le répartiteur. Et ainsi de suite. Cependant, pour que cette solution soit efficace, il est nécessaire d'avoir les répétiteurs sur des sites distants l'un de l'autre (redondance géographique) de façon que les sites ne soient pas affectés par une même panne (coupure d'électricité, ouragan, etc). De plus, cette solution présente également des contraintes économiques plus élevés puisqu'il faut maintenir plus d'un répartiteur à tout moment : ils doivent avoir des ressources similaires afin que le trafic ne soit pas impacté de façon majeure; par exemple A sur un mainframe et B sur un Raspberry Pi.

Malgré qu'une redondance des serveurs procure des avantages dans le cas d'une panne d'un répartiteur, en fonction de la nature de la panne et de la configuration de la redondance, un problème avec le serveur DNS peut causer une panne globale de tous les répartiteurs. Similairement, un DDoS causera tout de même une panne du système.