École Polytechnique Montréal

Département de Génie Informatique et Génie Logiciel

INF8480

Systèmes Répartis et Infonuagique

Travail pratique 2

Services distribués et gestion des pannes

Soumis par

Ibrahima Séga Sangaré (1788085)

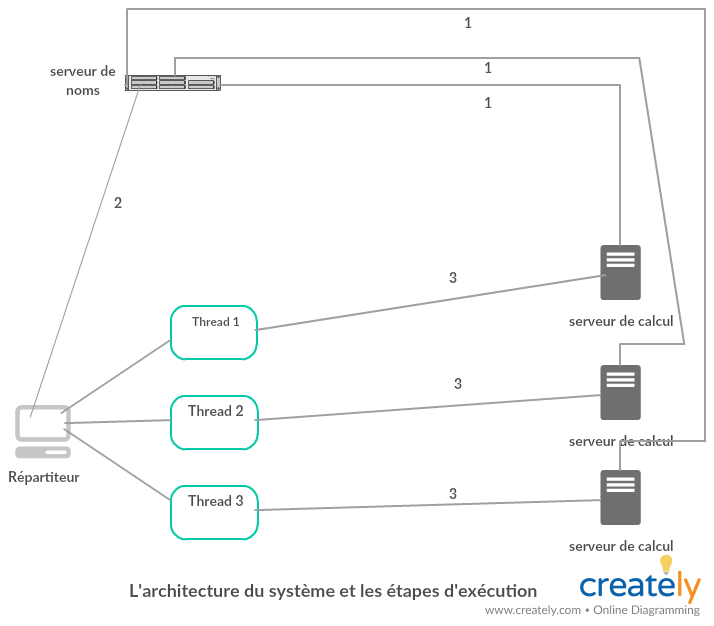
Mohamed Djemai (1911666)

Le 6 novembre 2018

**Description du projet :**

Ce projet permet de se familiariser avec les systèmes distribués en termes de la collaboration entre les différents acteurs, et comment gérer les pannes et les défaillances. Par exemple comment diviser les taches de calculs sur les différents serveurs de calculs disponibles selon leurs capacités, détecter le surcharge et l’arrêt brusque d’un serveur. En concluant, concevoir un système qui peut gérer n’importe quelle situation peut arriver au système pour accomplir les taches demandées.

Précisément, dans ce projet l’objectif est de distribuer une tache de calcul sur plusieurs serveurs de calculs afin d’avoir le résultat dans les meilleurs délais, en simulant les différents états qui peuvent l’arriver.



Le système est constitué de 3 programmes, le répartiteur, les serveurs de calculs, et le serveur de noms. Chaque programme a des fonctionnalités et des taches spécifiques indispensable pour le fonctionnement général du système.

La figure au-dessus montre l’architecture générale du système, les programmes s’exécutent sur des machines différentes et communiquent entre eux avec Java RMI à travers le réseau. Le répartiteur communique avec le serveur de noms pour récupérer les informations des serveurs de calculs comme l’adresse ip et la capacité de chaque serveur, et aussi avec les serveurs de calculs de donner des taches de calcul à faire. Le serveur de noms stocke les informations sur les serveurs de calculs. Les serveurs de calculs reçoivent les requêtes de calcul du répartiteur, pour les exécuter et retourner les résultats.

L’ordre du lancement des programmes est montré sur la figure. Premièrement en lance le serveur de noms pour que les serveurs de calculs puissent s’enregistrer auprès de lui. Deuxièmement, on lance les serveurs de calculs qui vont premièrement s’enregistrer sur le serveur de noms pour permettre au répartiteur de les trouver. Dernièrement, on lance le répartiteur qui va d’abord récupérer les adresses ip et les capacités des serveurs.

**Réponses aux questions :**

**I - Tests de performance – mode sécurisé**

2 serveurs = 3380728669 ns

3 serveurs = 2280181139ns

4 serveurs = 1947918089ns

* ***Présentez le graphique du temps d'exécution en fonction du nombre de serveurs. Expliquez les résultats en faisant des liens avec vos choix d'implémentation.***

On voit dans le graphe que le temps d’exécution diminue en augmentant le nombre de serveurs. L’augmentation du nombre de serveurs implique la division du travail sur plusieurs morceau plus petits donc la tache de calcul donnée à chaque serveur diminue et puisque dans notre architecture on a utilisé les threads, le calcul se fera en parallèle dans tous les serveurs.

**I - Tests de performance – mode non-sécurisé**

3 serveurs de bonne foi = 7665605495 ns

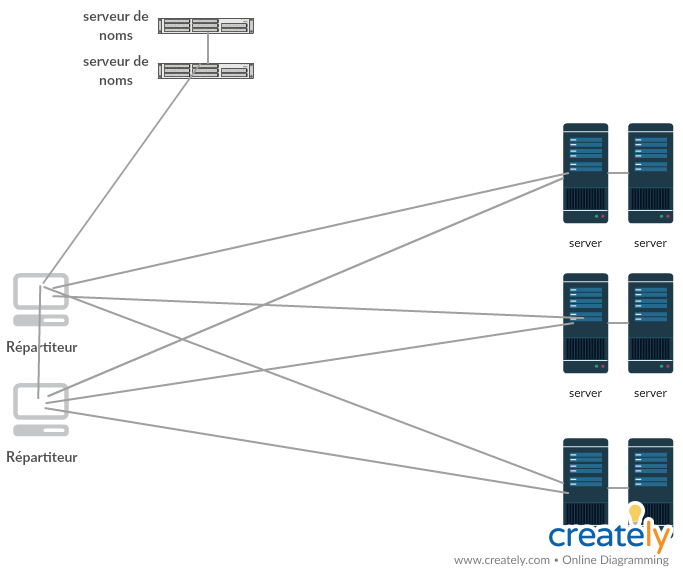
1 serveur malicieux 50% du temps, 2 serveurs de bonne foi = 6318615668 ns

1 serveur malicieux 75% du temps, 2 serveurs de bonne foi = 6236068321 ns

* ***Présentez et expliquez les temps obtenus.***

Le temps de calcul dans ce cas reste relativement constant, puisque dans notre architecture pour contourner le problème des serveurs malicieux, on envoie la tache a au moins 3 serveurs et donc surement on trouve le bon résultat dans 2 serveurs parmi 3 donc on n’est pas obligé de renvoyer la tache encore une fois. En comparant par rapport le mode sécurisé on voit que le temps est triplé parce que dans le mode non-sécurisé on envoie les données a plusieurs serveurs pour éviter les résultats erronés du serveur malicieux.

* ***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 est de faire la redondance dans le système, on double par exemple les serveurs de calculs, serveur de noms et le répartiteur. Dans ce cas on aura moins de temps un serveur malicieux ou une intempestive dans le système. Malgré ça cette solution est très chère et consomme beaucoup de ressource. Une catastrophe naturelle peut tomber le système.

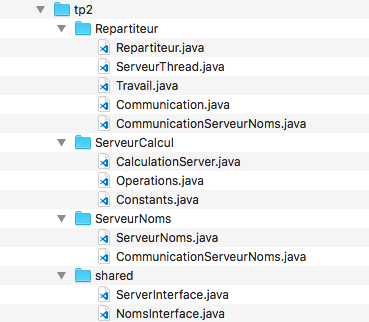
**Description générale de l’implémentation:**

Cette partie va expliquer brièvement les choix de conception que nous avons pour obtenir un programme fonctionnel. Pour le répartiteur, nous avons choisi de diviser les tâches entre les serveurs de manière équitable selon leurs capacités. Nous avons fait ce choix, car il nous semblait plus efficace de profiter du nombre de serveurs pour améliorer le temps d’exécution et de ne pas déséquilibrer les serveurs en termes de nombres d’opérations. En visant un taux refus d’environ 10%, nous avons choisi de relancer la demande au serveur de calcul en cas de refus, car cela ne devrait pas arriver aussi souvent. Le nombre d’opérations est donc légèrement supérieur à la capacité du serveur de calcul à chaque tentative.

Pour identifier les pannes, nous avons utilisé un *shutdown hook* qui permet de détecter une fermeture de programme, par exemple. Le serveur de calcul qui tombe en panne notifie le service de noms pour qu’il soit retiré de la liste des serveurs disponibles pour effectuer les calculs. De cette manière, les opérations assignées à ce serveur vont être redistribuer sur les autres serveurs.

**L’implémentation en détail**:

Dans cette partie on explique l’implémentation du système et les classes utilisées. On voit dans la figure suivante les classes des trois programmes utilisés pour déployer le système. On voit 4 dossiers, les trois premiers sont les classes des programmes à exécuter (répartiteur, serveur de calcul, serveur de noms), et le 4ème dossier est pour les interfaces de java RMI.



**Le répartiteur :**

On trouve dans le répartiteur 4 classes. La classe Repartiteur contient la méthode main(), dans cette classe on récupère les arguments de l’utilisateur, comme le fichier de calcul (pell, prime), le mode du système (SECURISE, NONSECURISE), et l’adresse ip du serveur de nom pour récupérer les adresses des serveurs de calculs. Cette classe est responsable pour instancier les autres classes. Le répartiteur est comme le créateur dans le modèle GRASP. On utilise ce modèle pour ce projet, pour que le projet soit clair, et bien organisé, et flexible pour éventuelle modification. Le répartiteur est divisé sur 3 parties. D’abord, il se communique avec le serveur de noms avec la classe CommunicationServeurNoms pour récupérer les informations des serveurs de calculs et pour s’enregistrer. Deuxièmement, et avec la classe Travail il récupère le fichier de calcul, et rend les données sous forme de Map<String,String> pour chaque ligne du fichier. Après il divise le travail à faire selon les capacités des serveurs, c’est-à-dire crée d’autres données Map<String, String> pour chaque serveur de calcul. La classe ServeurThread joue un rôle très important. Elle crée des threads selon le nombre de serveurs de calcul disponibles (la figure qui montre l’architecture de système), et instancie la classe Communication pour envoyer les taches au serveur de calculs à travers java RMI. Elle est aussi responsable de détecter les pannes et les défaillances dans le système, comme par exemple un serveur malicieux et la manque ressources.

Le trouve dans le répartiteur 2 méthodes pour retourner les résultats ( resultSecirise() pour le mode sécurisé, resultNonSecurise() pour le mode non sécurisé). Dans le mode sécurise le répartiteur détecte si un serveur de calcul accepte ou refuse une requête de calcul, pour ne pas perdre les données de calcul. Dans le mode non sécurisé il envoie les données a au moins 3 serveurs de calculs et compare les résultats partiels pour connaitre le serveur malicieux.

**Le serveur de calcul :**

Le serveur de calcul est responsable pour faire les calculs. La classe CalculationServer contient la méthode main() et l’implémentation des méthodes remote de java RMI (calculate() qui faire les calculs et retourné le résultat, acceptRequest() pour simuler le manque de ressources dans le serveur et refuse la tache). On trouve plusieurs méthodes pour simuler le comportement d’un serveur réel (isMalicious() pour simuler un serveur malicieux). Les classes Operations et Constants sont utilisées par le serveur pour faire les calculs de pell et prime, et aussi les constants pour simuler le comportement du serveur réel.

**Serveur de noms :**

La classe ServeurNoms permet de recevoir les informations des serveurs de calcul et les stocke dans une List<String> pour les adresses ip List<Integer> pour les capacités, et aussi elle permet d’enregistrer le répartiteur. Elle permet de détecter une panne intempestive lors de l’exécution des taches. Dans ce cas le répartiteur peut récupérer cette information pour ne pas envoyer les tachés au serveur tué.