HUC-LHUILLERY Alexia
GUEMIL Walid
RMICHE Nabil

Projet données réparties (S8)

I - CENTRALIZED LINDA

1 - Structure de données des tuples en mémoire

Les tuples ne sont plus enregistrés dans une liste mais dans une HashMap, cela permet pour la recherche d'utiliser directement les fonctions associées à cette structure de données. La HashMap prend en clef le tuple et en valeur le nombre de fois que ce tuple a été ajouté à notre espace. Si le tuple ne s'y trouve plus, sa valeur devient 0. On peut alors réintégrer ce tuple à notre espace en changeant simplement la valeur sans devoir ajouter une clef à la HashMap des tuples. On gagne donc en efficacité dans la recherche des tuples dans notre espace, les fonctions take, read et write cherchent à présent directement la clef dans le tableau mes_tuples.

2 - Abonnement

Pour les appels aux abonnements, les abonnements sont gérés dans des HashMap, une pour les take et une pour les read. Chaque tableau prend pour clef le tuple correspondant à l'abonnement, et pour valeur une liste des callback correspondant. Les tableaux d'abonnement sont directement modifiés si besoin après l'écriture d'un nouveau tuple en mémoire.

Si l'abonnement fait un take, celui-ci peut être directement trouvé si le tuple cherché correspond à celui en mémoire, et l'appel à l'abonnement est plus efficace. Sinon on cherche à ce que le tuple cherché et celui écrit correspondent en parcourant les tuples enregistrés dans le tableau d'abonnement.

3 - Parallélisme

Le parallélisme a été ajouté à la structure de donnée en utilisant un tableau concurrent pour organiser l'ensemble de tuples dans la mémoire. Cela permet la lecture en parallèle des données, l'écriture étant toujours gérée avec des moniteurs.

II - CLIENT-SERVEUR

1 - Cache

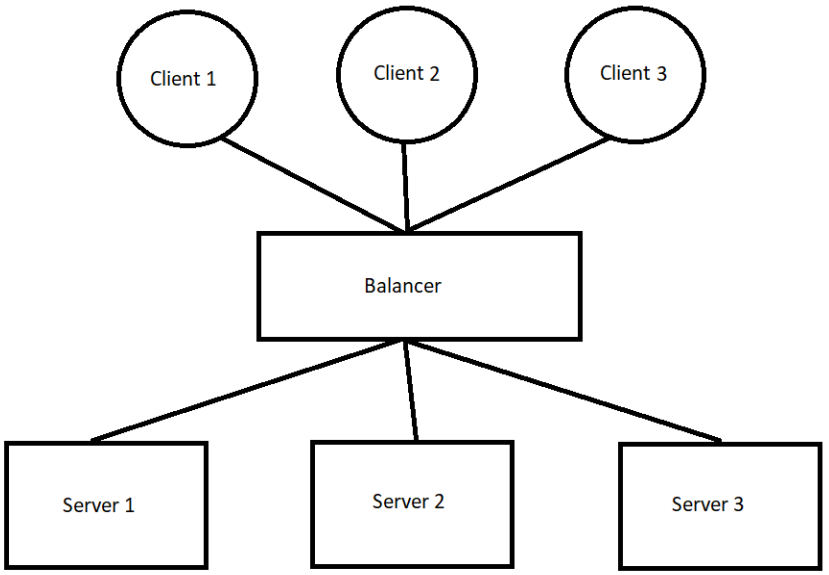
Chaque client possède désormais en local un cache de tuple. Sous la forme d'une ArrayList, à chaque fois qu'un client veut utiliser une méthode read ou tryread, le client vérifie d'abord s'il possède localement un tuple qui pourrait correspondre à sa requête et sinon il demande au serveur.

A la création du client le cache est vide et est alimenté à chaque appel de la fonction write. Cependant on aurait pu également décider, à la création d'un client, d'effectuer une copie local des tuples du serveur dans le cache client.

On a essayé d'implémenter le cache également pour la méthode take et tryTake mais on a rencontré des difficultées pour supprimer du cache de tous les clients le tuple retourné par la méthode.

2 - Multi-serveur

Pour partitionner l'espace des tuples entre les serveurs, nous avons choisi d'implanter un schéma de load balancer. Un premier serveur ServLoadBalancer est créé avec une liste de serveurs, dont on peut choisir le nombre de serveurs. L'espace des tuples est ensuite partitionné par ce LoadBalancer entre les différents serveurs. Chaque serveur est associé à des tuples ce qui permet par la suite de lire et d'écrire les tuples seulement dans le serveur correspondant. Ces tuples sont répartis dans les serveurs selon leur taille.



Architecture

Ce choix de répartition possède des limites, notamment dans des cas d'applications où tous les tuples sont de même taille donc dans ce cas la il n'y a pas de réel intérêt d'avoir cette architecture.

De plus, dans notre code actuel les serveurs sont créés sur la même machine virtuel donc il n'y a pas de gain de performances. Il faudrait alors lancer les serveurs sur différentes machines pour observer un gain.