Simulation d'algorithmes d'équilibrage de charge dans un environnement distribué

Memoire final

Kevin Barreau Guillaume Marques Corentin Salingue

7 avril 2015

Résumé

Ce document, mémoire final de notre Projet de Programmation, décrit l'ensemble du projet et les algorithmes utilisés. Il contient aussi les besoins fonctionnels et non fonctionnels ainsi que l'architecture du produit final. Enfin, il récapitule point par point le travail effectué ainsi que sa mise en oeuvre. Le document présente les tests effectués et leurs analyses. Enfin, il conclue sur les améliorations possibles du produit.

Sommaire

Table des figures

1 Présentation du projet

Dans le présent document, nous considérons que le lecteur possède des notions en informatique et que chaque mot est défini par son sens commun. Cependant, si un terme qui est utilisé présentant une définition différente que celle admise par tous, nous ne manquerons pas de le préciser et de le définir.

1.1 Utilisation d'une base de données par un client

Une *base de données* est un outil permettant de stocker et récupérer des *données*, qui sont des informations binaires utilisant des structures propres au système de base de données.

Dans un premier temps, le client se connecte à la base de données. Le client interagit avec celle-ci en lui envoyant des *requêtes*, messages, dont la forme dépend de la base de données et permettant de stocker, récupérer ou modifier des données.

Selon les requêtes émises par le client, la base de données lui renvoie des résultats (voir la figure 1).

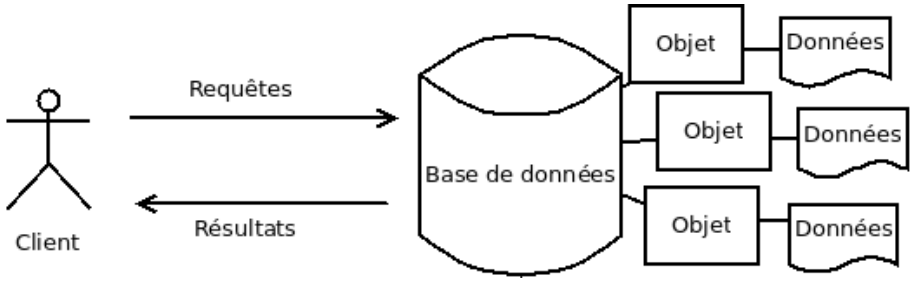


FIGURE 1 – Interactions client/base de données

On distingue deux types de requêtes :

— Les requêtes de **lecture** : requêtes ne modifiant pas les données contenues dans la base de données. Il s'agit de récupérer des objets contenus dans la base de données.

— Les requêtes d' **écriture** : requêtes modifiant les données contenues dans la base de données.

Le client peut être une personne physique ou un logiciel. Dans notre cas, il s'agit d'un logiciel permettant l'importation de fichiers contenant des requêtes ou de générer des requêtes pseudo-aléatoirement.

1.2 Base de données distribuée

La base de données utilisée par le client est plus précisément une base de données dite *distribuée*. Le client ne voit pas de différence, lorsqu'il l'utilise, entre une base de données classique et une base de données distribuée. On dit qu'une base de données est distribuée lorsque les données qu'elle stocke sont réparties sur plusieurs machines ou emplacements physiques, appelés *noeuds*. Les noeuds sont capables de communiquer entre eux afin de s'échanger des informations.

On peut rassembler des noeuds pour former un *data center*. Un rassemblement de data center correspond à un *cluster* (voir la figure 9). Dans ce projet, nous nous intéressons seulement au cas où un cluster est composé d'un seul data center.

La base de données va stocker les données sous forme d'*objets*. Un *objet* est composé d'une clé d'identification appelée *token* et d'un ensemble de *données*.

En positionnant les noeuds suivant leur token, on obtient alors une forme d'anneau (ou de *ring*), qui est donc la forme d'un data center.

Pour savoir quel noeud doit stocker quelle donnée, on utilise une méthode de *partitionnement*. Cette méthode se base sur les *tokens*. Chaque noeud a un token qui lui est attribué. Un noeud prend en charge des objets dont le token est compris entre celui que le noeud possède et celui de son "prédécesseur" (si on imagine un anneau orienté dans le sens du plus petit au plus grand token, sauf pour les extrêmes) dans l'anneau (voir la figure 10). Ainsi dans cet exemple, le noeud 2 a le token 25 qui lui est attribué. Il s'occupe donc des objets dont le token est compris entre 25 et 0 (qui est le token le plus grand dans ses prédécesseurs). On parle alors de l'*intervalle* de tokens dont s'occupe le noeud.

Afin de garantir une meilleure disponibilité, chaque objet possède des copies, appelées *réplicas*, disposées sur d'autres noeuds que le noeud initial (le noeud qui s'occupe du token de cet objet). La méthode pour choisir l'emplacement des copies d'un objet est variable. C'est ce que l'on appelle la *stratégie de réplication*, qui est abordée plus loin dans ce document.

1.3 Gestion des requêtes dans la base de données distribuée

1.3.1 Requêtes de lecture

Il est possible de réaliser des requêtes de lecture sur un objet, ce qui consiste à vouloir récupérer une donnée contenue dans un objet. Pour expliquer le cheminement d'une requête de lecture dans la base de données, nous allons prendre un exemple (voir la figure 12).

Un client réalise une requête de lecture R. Il envoie la requête à n'importe quel noeud du réseau. On appelle alors ce noeud : le noeud *coordinateur* pour cette requête. Ce noeud ne contient pas forcément l'objet de la requête, mais il va faire la liaison entre le réseau et le client.

Le noeud coordinateur va avoir cette requête dans une file d'attente dédiée aux requêtes des clients. Il les traite les unes à la suite des autres. Lorsque le noeud commence à traiter cette requête, il va d'abord identifier les noeuds responsables de l'objet de la requête. Cela inclut le noeud possédant l'objet *original* (dont le token est géré par ce noeud) ainsi que les noeuds possédant un réplica. Cette étape exige une connaissance complète du réseau sur chaque noeud et une connaissance de la stratégie de réplication mise en place.

Dès que les noeuds sont identifiés, le noeud coordinateur leur envoie un message pour traiter la requête de lecture (les flèches rouges sur le schéma entre le noeud coordinateur et les autres noeuds). Ce message est mis dans la file d'attente des requêtes de lecture de ces noeuds.

A un moment, l'un des noeuds qui possède cette requête dans sa file d'attente va la défiler et la traiter. Ce noeud s'*affecte* la requête. Il avertit les autres noeuds possédant cette même requête dans leur file d'attente (c.à.d tous les autres noeuds possédant une copie de l'objet de la requête) qu'ils n'auront pas besoin de la traiter, et qu'ils peuvent la supprimer de leur file d'attente (les flèches oranges sur le schéma). Si la requête à supprimer est déjà en cours d'exécution, on la laisse se dérouler normalement. Le noeud qui s'est affecté la requête la traite et renvoie le résultat au noeud coordinateur, qui peut transmettre le résultat obtenu au client (les flèches vertes sur le schéma).

1.3.2 Requêtes d'écriture

Il est possible de réaliser des requêtes d'écriture d'un objet, ce qui consiste à stocker des données dans la base de données, sous forme d'objet. Pour expliquer le cheminement d'une requête d'écriture dans la base de données, nous allons prendre un exemple (voir la figure 13). Le cheminement est plus simple que pour une requête de lecture car il n'y a pas le mécanisme d'affectation.

Un client réalise une requête d'écriture R. Il envoie la requête à n'importe quel noeud du réseau. On appelle alors ce noeud le noeud *coordinateur* pour cette requête. Ce noeud n'est pas forcément celui qui va stocker les données, mais il va faire la liaison entre le réseau et le client.

Le noeud coordinateur va avoir cette requête dans une file d'attente dédiée aux requêtes des clients. Il les traite les unes à la suite des autres. Lorsque le noeud commence à traiter cette requête, il va d'abord identifier les noeuds responsables de l'objet de la requête. Cela inclut le noeud qui se charge de l'objet *original* (dont le token est géré par ce noeud) ainsi que les noeuds devant posséder un réplica. Cette étape exige une connaissance complète du réseau sur chaque noeud et une connaissance de la stratégie de réplication mise en place.

Dès que les noeuds sont identifiés, le noeud coordinateur leur envoie un message à tous pour traiter la requête d'écriture (les flèches rouges sur le schéma entre le noeud coordinateur et les autres noeuds). Ce message est mis dans la file d'attente des requêtes d'écriture de ces noeuds.

Tous les noeuds recevant le message vont alors stocker les données envoyées par la requête. Le noeud coordinateur peut demander un certain nombre de messages de retour pour s'assurer que les requêtes d'écritures se sont bien déroulées. Dans l'exemple, le noeud coordinateur demande **1** retour. L'un des messages envoyés aux noeuds contiendra donc une demande d'un message de retour pour confirmer que l'écriture s'est bien passée (la flèche verte entre les noeuds sur le schéma). Dès que le noeud coordinateur reçoit le message, il indique au client que sa requête s'est terminée et bien passée.

1.4 Stockage des données

Chaque base de données possède sa propre manière de stocker les données dans un espace de stockage. Pour le projet, la méthode de stockage n'est pas un problème sur

lequel nous allons travailler. La seule contrainte imposée pour la base de données est qu'elle stocke les données sous la forme d'objet. C'est à dire qu'un objet est identifiable par son token, une clé d'identification générée le plus souvent par une fonction de hachage.

Le token est généré par la base de donnée à partir de la *clé primaire* d'une table. Une clé primaire est, comme le token, une donnée permettant d'identifier un objet. Sauf que que la clé primaire est une donnée choisit par le client. Elle peut être un entier, une chaîne de caractères, toutes les représentations possibles d'une donnée au sein de la base de données (voir la figure 14).

1.5 Protocoles de réaffectation des requêtes de lecture

Lorsqu'une requête de lecture est envoyée par un client, on a vu précédemment que cette requête était transmise à tous les noeuds possédant une copie de l'objet à lire. Si un nombre important de requêtes de lecture arrivent en même temps, les files d'attentes dans les noeuds pour les requêtes de lecture vont commencer à se remplir plus vite que les requêtes ne sont traitées. Le nombre de requêtes dans une file d'attente est appelée la *charge*. Les charges des files d'attentes ne seront pas forcément uniformes entre les noeuds, certains pouvant avoir plus de requêtes à traiter que d'autre.

C'est pourquoi on met en place un système de *réaffectation* des requêtes de lecture, afin de rééquilibrer la charge des noeuds. La réaffectation consiste, après chaque modification locale (une modification locale sera le traitement d'une requête de lecture ou de suppression dans notre cas, mais on peut imaginer d'autres moments aussi), à enclencher un processus permettant de décider de l'affectation des requêtes suivant l'état actuel du réseau. Le nombre de requêtes affectées (donc assignées à ce noeud et avec un message de suppression pour ces requêtes envoyé aux autres noeuds) est appelé la *charge effective*. Une requête affectée ne peut pas être supprimée.

Les algorithmes de réaffectation à implémenter, **SLVO** et **AverageDegree**, ont un comportement similaire qui se base sur la connaissance des charges de chaque noeud du réseau. L'algorithme consiste à comparer, pour tous les noeuds, sa propre charge par rapport à une certaine valeur.

Pour SLVO, la valeur est la charge minimale sur le réseau. Pour AverageDegree, la valeur est la charge moyenne sur le réseau.

Si la valeur est inférieure ou égale (strictement égale dans le cas de SLVO), alors le noeud s'affecte toutes les requêtes de sa file d'attente et avertit tous les autres noeuds. Les noeuds possédant les requêtes qui ont été affectées les suppriment de leur file d'attente, modifiant ainsi leur charge (voir la figure 15 pour un exemple avec SLVO).

1.6 Gestion de la popularité des objets

Pour mieux équilibrer la charge du réseau, nous nous intéressons à la *popularité* des objets. En effet, plus un objet va recevoir de requêtes, plus il sera populaire et occasionnera une grande charge pour les noeuds qui s'en occupent. Afin de répartir cette charge, il faudra alors augmenter ou diminuer le nombre de réplicas. Si un objet est populaire, il suffira de créer de nouveaux réplicas, ce qui permettra d'envoyer une partie de la charge sur d'autres noeuds. A l'inverse, si un objet n'est pas populaire,

diminuer le nombre de copies fera gagner de l'espace mémoire et du temps (quand on a besoin de contacter tous les noeuds qui gèrent un objet, le nombre de noeuds influe sur le temps nécessaire à réaliser l'action...).

Il y a plusieurs méthodes pour calculer la popularité des objets durant un intervalle de temps T défini par l'utilisateur :

— La première consiste à ce que chaque noeud possède un vecteur de la taille du nombre d'objets dont il a la gestion. A chaque nouvelle requête, la case de l'objet est incrémentée. Au début de chaque période T, les noeuds envoient la popularité aux autres noeuds et décident du nombre de copies à faire.

— La seconde méthode est une variante visant à réduire la taille du vecteur d'objets et est défini par le Space-Saving Algorithm [ADA05]. On choisit un vecteur de la taille du nombre de noeuds du réseau qui contient des structures de la forme *(identifiant de l'objet ; nombre de requête)*. Soit n le nombre de noeuds dans le reseau, l'algorithme permet de connaitre les n objets les plus populaires.

  Soit une requête sur un objet o.
  Si o est présent dans le vecteur, on augmente son nombre de requêtes de 1.
  Si o n'est pas présent dans le vecteur et qu'il reste de la place (la taille du vecteur est inférieure à n), on l'ajoute au vecteur avec un nombre de requête de 1.
  Si o n'est pas présent et que le vecteur est plein, on cherche l'endroit qui contient l'objet le moins populaire du vecteur et on le remplace par o. Cependant, on garde la popularité de l'ancien objet et on l'incrémente de 1.
  Il est possible de consulter le pseudo code sur la figure 16

Soient les paramètres suivants :
r = Nombre de requêtes total effectuées durant l'intervalle de temps T ;
n = Nombre de noeuds dans le réseau ;
p = Popularité d'un objet ;
k = Nombre de copies de l'objet.

On augmente le nombre de copies d'un objet quand la formule suivante est respectée :
$2 \times \frac{r}{n} \geq \frac{p}{k}$

On diminue le nombre de copies d'un objet quand la formule suivante est respectée :
$\frac{r}{2n} \leq \frac{p}{k}$

1.7 Gestion des copies d'un objet

Une *fonction de hachage* est une fonction mathématique qui possède les propriétés suivantes :

— Ensemble d'entrée : une clé primaire ;

— Ensemble d'arrivée : Un entier ;

On lui associe souvent d'autres propriétés pour équilibrer la répartition des données (cf paragraphe suivant).

Pour fabriquer un token, qui sert à placer les données sur le réseau, une clé primaire est hachée avec une fonction de hachage. Nous obtenons une valeur (ici un entier). Chaque noeud du réseau est responsable d'un intervalle d'entiers de l'ensemble des valeurs du domaine. Avec une bonne fonction de hachage, les hash des clés primaires seront distribués uniformément dans l'ensemble des entiers.

Placer les copies sur le même noeud revient à n'avoir théoriquement l'équivalent d'aucune copie puisque la charge reste sur le même noeud. Il existe plusieurs stratégies de placement de copies des données et nous n'en développerons que deux ici. La première permet de comprendre les mécanismes de base d'une stratégie de placement. La seconde décrit celle que nous allons développer.

On le rappelle, chaque noeud possède la gestion d'un intervalle de *tokens*. La base de données se base sur des intervalles d'entiers. Si on organise ces noeuds selon l'ordre croissant des intervalles, on obtient un cercle.
La plus simple stratégie consiste à prendre les noeuds qui suivent sur ce cercle. Prenons comme exemple la figure 17. Nous souhaitons stocker une donnée. La fonction de hachage de la clé primaire de notre donnée retourne 123. La donnée originale est donc placée sur le noeud 2.
De plus, notre stratégie va créer des réplicas qu'elle disposera sur les noeuds suivants. Ici, on a décidé que 3 réplicas suffisaient, les copies sont donc placées sur les noeuds 3, 4 et 5.

La seconde consiste à utiliser les fonctions de hachage. En effet, dans la stratégie précédente, une seule fonction de hachage était définie pour placer la donnée et la position des réplicas était ensuite déterminée à partir de l'emplacement de la première. Supposons qu'on a au maximum n fois la donnée présente dans le réseau (c'est à dire n = nombre de copies d'une donnée + 1 pour le placement initial). Nous avons besoin de n fonctions de hachage, toutes numérotées de 0 à $n-1$. La fonction de hachage numéro 0 sert à placer la donnée. La fonction numéro 1 sert à placer le premier réplica, la fonction numéro 2 sert à placer le second réplica et ainsi de suite...

1.8 Visualisation des statistiques de fonctionnement de la base de données distribuée

Le but est de visualiser les statistiques de fonctionnement de la base de données pour permettre une comparaison de l'efficacité des algorithmes d'équilibrage de charge.

On souhaite récupérer :

— la charge effective de chaque noeud ou taille de la file d'attente des requêtes de lecture.

— une représentation de la file d'attente des requêtes de lecture

— la popularité de chaque objet

— la requête en cours de traitement

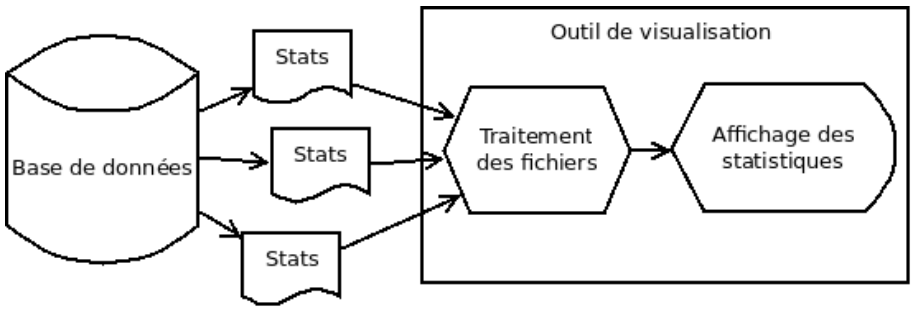


FIGURE 2 – Processus pour la visualisation des statistiques

On enregistre les statistiques de fonctionnement de la base de données distribuée dans des fichiers. Un outil de visualisation traite ces fichiers et affiche ensuite les statistiques (voir la figure 2).

1.9 Pour en savoir plus...

Pour nous aider dans le projet, nous avons rechercher des écrits dans plusieurs domaines différents que nous devons aborder. Cette bibliographie initiale est une étude de l'existant. D'autres références sont ensuite disponibles tout au long de ce document, que nous avons trouvé au fil de l'avancement du projet.

Système distribué

— Le livre [ÖV11] apporte les principes sur les bases de données distribuée. On peut notamment y retrouver des principes sur la réplication ainsi que sur les communications dans le base de données.

Algorithmes d'équilibrage de charges

— L'article [ABKU99] met en exergue des algorithmes d'équilibrage, sources d'inspiration pour les clients dans l'élaboration de leurs algorithmes.

— Le papier [MRS05], de là même manière que le précédent, s'intéresse de plus près à des algorithmes d'allocation aléatoire. Les deux articles permettent de comprendre l'intérêt des modifications à apporter pour les stratégies de placement et de réplication dans la base de données distribuées.

— L'article [FKSS14] traite des algorithmes d'équilibrage des charges dans un réseau pair à pair.

Cassandra

— Le livre [Hew10] renseigne sur la compréhension du fonctionnement de Cassandra, ainsi que les structures particulières de ce système de base de données.

— Le lien [Fou09] permet d'accéder à l'ensemble de la documentation technique et au dépôt des sources de la base de données Cassandra. On peut y retrouver la documentation côté développeur, pour modifier Cassandra suivant des règles prédéfinies. Les informations sur la compilation du projet et la gestion des tests est aussi présente si l'on se dirige vers le wiki.

— Le papier [LA09] apporte les explications des créateurs de Cassandra, sur les spécifications de Cassandra par rapport aux autres bases de données du même genre. On y retrouve des informations sur le partitionnement, la réplication et le modèle de données utilisé par Cassandra.

— Le lien [Dat15b] possède la documentation la plus complète sur la dernière version de Cassandra, et sur les manières de l'utiliser. Les nombreux exemples et les approfondissements sont des points d'appuis importants pour le projet. Tous les principes de la base de données sont expliqués de manière globale, sans entrer dans les détails du code source.

Visualisation

— Le lien [Dat14] apporte un logiciel, OpsCenter, capable de montrer graphiquement des statistiques sur une base de données Cassandra opérationnelle. Il peut être utilisé dans le projet, ou apporté des pistes pour la visualisation des données.

— L'article [AAB+12] s'intéresse à un logiciel de représentation de graphe développé au Labri. Une visualisation sous la forme d'un graphe des données à afficher est tout à fait envisageable. Cet outil nous permettrait de répondre à ce besoin. De plus, ce logiciel est développé au Labri. Nous avons donc des personnes aptes à nous aider à proximité dans le cas de l'utilisation de ce logiciel.

2 Ordonnancement des besoins

Nous avons dégagé une liste de besoins fonctionnels et non-fonctionnels. Pour mieux les comparer, nous les avons ordonnés en fonction de leur priorité.

La priorité est un indicateur de l'ordre dans lequel nous devrons implémenter les fonctionnalités afin de satisfaire les besoins du client.

Valeur	Signification
1	Priorité haute
2	Priorité moyenne
3	Priorité faible

3 Besoins fonctionnels

3.1 Communication entre les noeuds

Tous les besoins concernent un seul noeud. Tous les noeuds du réseau doivent répondre à ces besoins.

- Envoyer les informations du noeud à n'importe quel autre noeud (*Priorité*:1)
- Recevoir les informations provenant d'un autre noeud (*Priorité*:1)
- Stocker les informations de tous les noeuds du réseau (*Priorité*:1) Cela concerne tous les noeuds, y compris soi-même.

Les informations d'un noeud doivent permettre d'envoyer un message à ce dernier.

3.2 Gestion des requêtes

Tous les besoins concernent un seul noeud. Tous les noeuds du réseau doivent répondre à ces besoins.

3.2.1 Requêtes client

- Créer une file d'attente des requêtes client (*Priorité*:1)
- Ajouter une requête à la file d'attente des requêtes client (*Priorité*:1)
- Défiler une requête de la file d'attente des requêtes client (*Priorité*:1)
- Traiter une requête client (*Priorité*:1)

- Identifier les noeuds responsables d'un objet (*Priorité*:1) Cela nécessite de connaitre plusieurs informations :
  - la stratégie de réplication
  - la token de chaque noeud
  - le nombre de copie de chaque objet

- Créer une requête de lecture (*Priorité*:1) Les requêtes de lecture doivent être identifiable, ceci afin de pouvoir les supprimer. Il faut donc générer un identifiant pour chaque requête de lecture lors de sa création.
- Envoyer une requête de lecture (*Priorité*:1)
- Créer une requête d'écriture (*Priorité*:1)
- Envoyer une requête d'écriture (*Priorité*:1)

3.2.2 Requêtes de lecture

- Créer une file d'attente des requêtes de lecture (*Priorité*:1)
- Recevoir une requête de lecture (*Priorité*:1)
- Ajouter une requête à la file d'attente des requêtes de lecture (*Priorité*:1)
- Supprimer une requête de la file d'attente des requêtes de lecture (*Priorité*:1)
- Défiler une requête de la file d'attente des requêtes de lecture (*Priorité*:1)
- Traiter une requête de lecture (*Priorité*:1)

— Créer un message de suppression de requête de lecture (*Priorité*:1)
— Envoyer un message de suppression de requête de lecture (*Priorité*:1)
— Recevoir un message de suppression de requête de lecture (*Priorité*:1)
— Traiter un message de suppression de requête de lecture (*Priorité*:1)

— Créer un message de résultat (*Priorité*:1)
— Envoyer un message de résultat au noeud coordinateur (*Priorité*:1)
— Recevoir un message de résultat (*Priorité*:1)
— Transmettre un message de résultat au client (*Priorité*:1)

3.2.3 Requêtes d'écriture

— Créer une file d'attente des requêtes d'écriture (*Priorité*:1)
— Recevoir une requête d'écriture (*Priorité*:1)
— Ajouter une requête à la file d'attente des requêtes d'écriture (*Priorité*:1)
— Défiler une requête de la file d'attente des requêtes d'écriture (*Priorité*:1)
— Traiter une requête d'écriture (*Priorité*:1)

— Créer un message de résultat (*Priorité*:1)
— Envoyer un message de résultat au noeud coordinateur (*Priorité*:1)
— Recevoir un message de résultat (*Priorité*:1)
— Transmettre un message de résultat au client (*Priorité*:1)

3.3 Réaffectation des requêtes de lecture

Les besoins sur la réaffectation des requêtes de lecture se recoupent avec ceux de gestion des requêtes de lecture en ce qui concerne les messages envoyés entre les noeuds.

— Connaître la charge des files d'attentes de requêtes de lecture de chaque noeud du réseau (*Priorité*:1) Cette information fait partie des informations de chaque noeud, communiqué entre eux comme vu précédemment dans la partie *Communication entre les noeuds.*
— Définir un protocole de réaffectation (*Priorité*:1)
— Modifier le protocole de réaffectation par une configuration (*Priorité*:3) La configuration est accessible par l'utilisateur, et le protocole de réaffectation doit être la même pour tous les noeuds du réseau
— Exécuter le code d'un protocole de réaffectation défini (*Priorité*:1)

3.4 Gestion d'un réseau

3.4.1 Popularité d'un objet

Stockage de la popularité

Tous les besoins concernent un seul noeud. Tous les noeuds du réseau doivent répondre à ces besoins.

— Créer un vecteur d'entiers comptabilisant le nombre de requêtes (*Priorité*:1)
— Augmenter la taille du vecteur (*Priorité*:1) d ans le cas où on a l'algorithme qui calcule la popularité de tous les objets
— Créer un identifiant permettant de relier la popularité à un objet (*Priorité*:1)

Calcul de la popularité

Tous les besoins concernent un seul noeud. Tous les noeuds du réseau doivent répondre à ces besoins.

— Incrémenter la popularité de l'objet requêté dans le vecteur à chaque requête sur celui-ci (*Priorité*:1)

Communication de la popularité

Tous les besoins concernent un seul noeud. Tous les noeuds du réseau doivent répondre à ces besoins.

— Identifier le noeud responsable d'un objet (*Priorité*:1)
— Créer un message de popularité (*Priorité*:1)
— Envoyer un message de popularité au noeud responsable de l'objet (*Priorité*:1)
— Recevoir un message de popularité (*Priorité*:1)
— Traiter un message de popularité (cf paragraphe suivant) (*Priorité*:1)

Traitement d'un message de popularité

Tous les besoins concernent un seul noeud. Tous les noeuds du réseau doivent répondre à ces besoins.

— Stocker la popularité du message dans le vecteur du noeud traitant le message (*Priorité*:1)
— Vérifier avoir reçu tous les messages concernant les objets dont le noeud a la gestion (*Priorité*:1)
— Décider de créer ou non de nouveau réplicas (*Priorité*:1)
— Décider de supprimer ou non des réplicas (*Priorité*:1)
— Réinitialiser le vecteur après la création des nouveaux objets les plus populaires (*Priorité*:1)

3.4.2 Réplication d'un objet

Tous les besoins concernent un seul noeud. Tous les noeuds du réseau doivent répondre à ces besoins.

— Créer une nouvelle stratégie de réplication (*Priorité*:1)

— Permettre la définition par l'utilisateur de fonctions de hachage et leur ordre d'utilisation (*Priorité*:1)

— Stocker chaque fonction de hachage et son ordre (*Priorité*:1)

— Définir un ordre dans les réplicas (*Priorité*:1)

— Utiliser la première fonction de hachage pour placer le premier réplica (*Priorité*:1) l a seconde pour le second, et ainsi de suite...

— Retrouver les réplicas en fonction des fonctions de hachage (*Priorité*:1)

3.5 Application cliente

3.5.1 Interactions avec Cassandra

— Se connecter à Cassandra (*Priorité*:1)

— Se déconnecter de Cassandra (*Priorité*:1)

3.5.2 Initialisation des données

L'initialisation des données consiste à créer un Keyspace et à enregistrer des données dans celui-ci.

— Créer un Keyspace (*Priorité*:1)

— Importer des données (*Priorité*:1)

— Initialiser les données (*Priorité*:1) S i les données sont modifiées, le client doit pouvoir les initialiser pour revenir aux données d'origine.

3.5.3 Gestion de requêtes

Pour tester la validité des algorithmes, l'application devra posséder une fonction de génération de requêtes. Si l'utilisateur ne détient pas de suites de requêtes prêtes, il pourra demander à l'application d'en créer pour lui. L'application, ne connaissant pas la nature des données, ne pourra qu'effectuer un nombre restreint de requêtes différentes. Elle pourra par exemple, compter le nombre de données sauvegardées, chercher si une donnée existe réellement, mais ne pourra pas en modifier une.

— Récupérer le nom des tables (*Priorité*:1) A fin de pouvoir générer des requêtes, nous devons connaître le nom des tables contenues dans un keyspace.

— Générer un jeu de données pseudo-aléatoirement (*Priorité*:2) I l s'agit de créer une fonction f, qui aura pour ensemble des antécédents des suites de caractères alpha-numériques (par exemple : 5832fg4gh52) et pour image un jeu de données.

— Importer un jeu de requêtes (*Priorité*:1) L 'utilisateur peut importer son propre jeu de requêtes.

3.6 Visualisation des données

3.6.1 Sauvegarde des données

Afin de suivre l'évolution des charges de chaque noeud lors de l'exécution des algorithmes, on enregistre les données locales de chaque noeud à chaque modification de celles-ci.

— Accéder aux données locales de chaque noeud (*Priorité*:1)

— Sauvegarder les données locale de chaque noeud (*Priorité*:1) L a sauvegarde peut avoir lieu dans un fichier, dans une base de données, dans n'importe quel support nous permettant ensuite de pouvoir récupérer les données.

3.6.2 Affichage des données

L'affichage des données locales permet à l'utilisateur de suivre le film de l'exécution des jeux de requête.

— Lire les données locales sauvegardées (*Priorité*:1) S elon le format dans lequel elles ont été sauvegardées, un analyseur syntaxique pourrait être utile.

— Afficher les données locales (*Priorité*:1) L e format d'affichage importe peu. Il peut s'agir d'un graphique, d'un réseau...

4 Besoins non fonctionnels

4.1 Cassandra

Cassandra est une base de données distribuée. Nous créons notre environnement distribué à partir de la dernière version stable de Cassandra.

Le choix de cette solution nous a été fortement recommandé par le client. En effet, celui-ci dispose de connaissances sur cette application et pourra donc plus facilement intervenir s'il souhaite faire évoluer le projet en implémentant par exemple de nouveaux algorithmes.

4.2 Maintenabilité du projet

L'envergure du projet fait qu'il est possible que d'autres personnes travaillent sur la finalité de ce projet, peu importe son état d'avancement. Afin de faciliter la compréhension, nous avons défini quelques normes pour que le projet puisse être repris :

— documentation dans le code source suivant la norme du langage utilisé ;

— document externe spécifiant les fichiers modifiés par rapport au code source original ;

— guide d'installation pour utiliser le projet et pour modifier le projet.

4.3 Visualisation des données

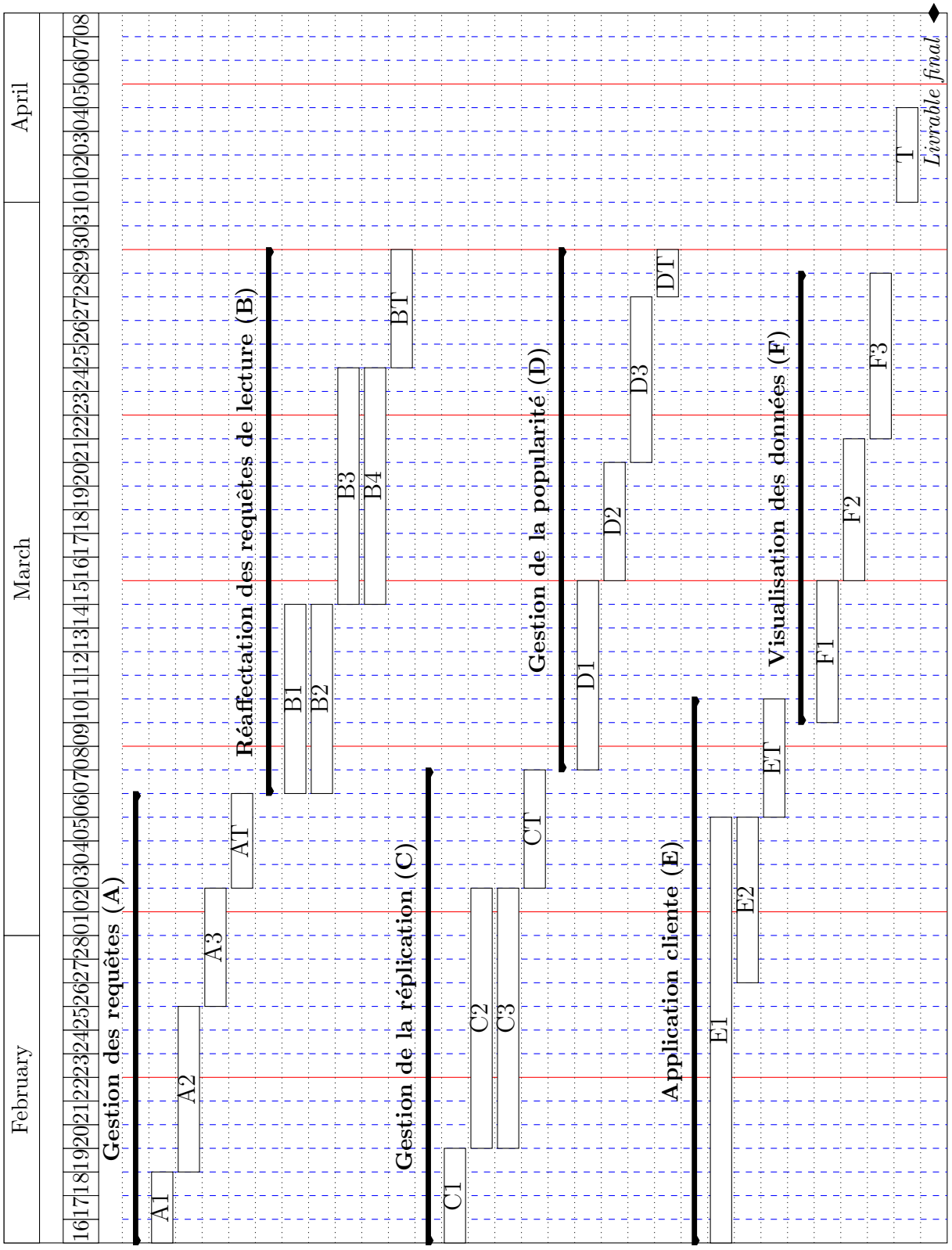
L'état du réseau doit être visible en temps réel.

La vue peut donc être actualisée toutes les 0.5 secondes. Un délai plus faible risquerait de ralentir le système, étant donné que l'obtention des données nécessaires à la visualisation se fait sur la même base de données que celle qui est testée.

5 Gestion de projet

5.1 Répartition des tâches

5.1.1 Diagramme de Gantt



5.1.2 Affectation des tâches

Fct	Description	Développeur(s)	Commentaire
A1	Envoi des requêtes de lecture	Kévin	Court-circuitage du système de digest de Cassandra
A2	Suppression des requêtes de lecture	Kévin	
A3	Gestion des messages de suppression d'une requête	Kévin	Envoi/réception/traitement
AT	Tests groupe A	Kévin	Vérification, tests, mémoire
B1	Système d'assignation d'une requête	Kévin	Ne pas envoyer les messages de suppression plusieurs fois
B2	Communication charge file d'attente	Kévin	Voir Gossip
B3	Algorithme SLVO	Kévin	Base modulaire
B4	Algorithme AvergeDegree	Kévin	Base modulaire
BT	Tests groupe B	Kévin	Vérification, tests, mémoire
C1	Stratégie de réplication	Corentin	Nouvelle stratégie
C2	1 fonction de hachage par niveau de réplica	Corentin	
C3	Retrouver les réplicas grâce aux fonction de hachage	Corentin	
CT	Tests groupe C	Corentin	Vérification, tests, mémoire
D1	Calcul de la popularité d'un objet	Corentin	Temps de recherche et exploration des mécanismes
D2	SpaceSaving Algorithm	Corentin	Temps de recherche et exploration des mécanismes
D3	Simulateur	Corentin	Développement Simulateur
DT	Tests groupe D	Corentin	Sur Simulateur
E1	Gestion de Cassandra	Guillaume	Développement de l'application et utilisation pilote
E2	Générateur de requêtes	Guillaume	
ET	Tests groupe E	Guillaume	Vérification, tests et mémoire
F1	Prise en main de Graphite	Guillaume	Installation, tests
F2	Ajout de metrics	Kévin et Guillaume	Mesure de la charge d'un noeud
F3	Paramétrage	Guillaume	Paramétrage de la web app
T	Tests finaux	*	Vérification, tests, mémoire

5.2 Outils utilisés

5.2.1 Flowdock

Flowdock est un outil de travail d'équipe. Il permet un dialogue entre les membres grâce au Chat intégré, un partage facile de fichiers et il affiche les dernières modifications réalisées sur des outils annexes (comme les derniers commits sur le dépôt GitHub ou les derniers post-it de Trello). Il est également disponible sur mobile avec une application dédiée.

Bien pratique, Flowdock a permis de concentrer en un unique endroit les avancées du projet et permet un énorme gain de temps, sur la recherche et la gestion des informations.

5.2.2 Trello

Trello est une sorte de grand mur à post-it. L'organisation des tâches, des rendez-vous, le partage des documents, tout est facilité. Classés en différentes catégories, les fiches de Trello permettent en un clin d'oeil de voir le travail effectué, en cours ou restant à faire. Il possède une gestion de label qui permet de chercher rapidement ce que l'on souhaite.

Nous avons utilisé Trello pour la répartition du travail et le découpage des tâches. Disponible également sur mobile, nous avons délégué à la plateforme la gestion des plannings du projet.

5.2.3 Git - Svn

Git et Svn sont deux gestionnaires de version largement connus. Nous avons utilisé Git pour sa facilité de mise en oeuvre (avec GitHub) et sa possibilité de travailler en local. Les liens ont été faits à chaque fois entre le dépôt Git et Svn (à chaque rendez avec le chargé de TD par exemple).

6 Architecture

6.1 Application cliente

Pour communiquer avec la base de données, nous utilisons un *pilote*, programme informatique permettant à un programme d'intéragir avec un autre programme.

6.1.1 Présentation du pilote

Le pilote que nous utilisons est Java Driver 2.0 for Apache Cassandra développé par l'entreprise Datastax, distributeur de Cassandra. Le pilote est sous licence Apache.

L'entreprise Datastax étant très spécialisé dans le développement de logiciels centrés sur Cassandra, son pilote est le plus complet. Il propose toutes les fonctionnalités dont nous avons besoin.

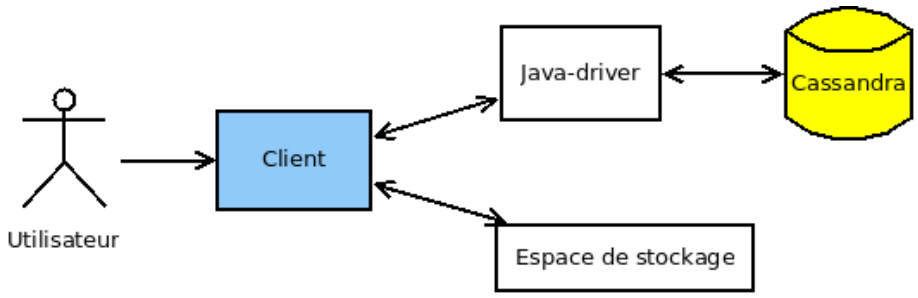


FIGURE 3 – Interactions du client

Sur la figure 3, l'application cliente est représentée par le rectangle bleu nommé client. Le client exécute les actions souhaitées par l'utilisateur. Les communications entre l'application cliente et la base de données Cassandra passent par le java-driver. Le client peut lire et écrire des fichiers dans l'espace de stockage.

6.1.2 Architecture de l'application cliente

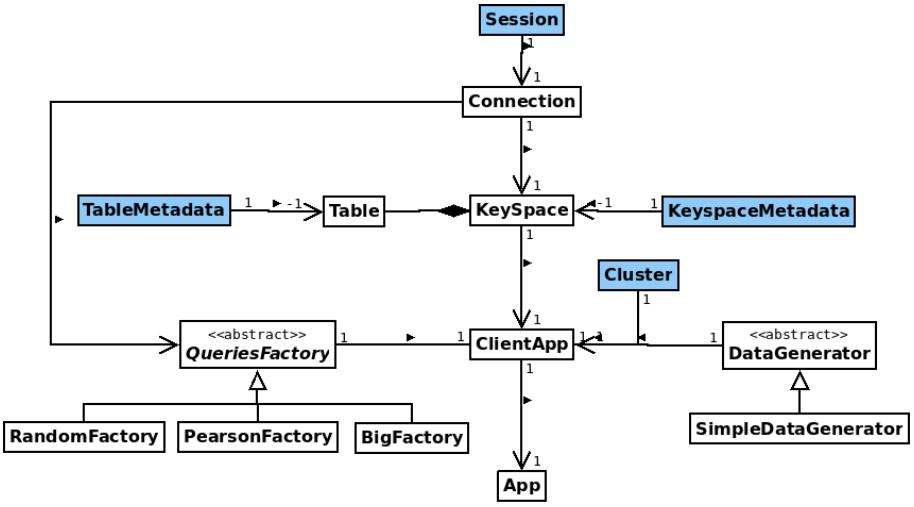


FIGURE 4 – Diagramme de classes simplifié de l'application cliente

Les classes du java-driver sont bleues dans la figure 4.

— La classe `Cluster` crée des objets permettant la connexion à la base de données Cassandra.

— La classe `Session` crée des objets permettant d'envoyer des requêtes au format Cassandra Query Language (CQL) à Cassandra.

— La classe `KeyspaceMetadata` crée des objets permettant de récupérer les informations principales concernant un keyspace. Par exemple, le nom du keyspace, les tables stockées ou bien la classe pour la stratégie de réplication utilisée.

— La classe `TableMetadata` crée des objets permettant de récupérer les informations principales concernant une table. Par exemple, le nom de la table ou les colonnes composant la table.

A partir de ces quatres classes, nous avons développé l'architecture de l'application cliente.

— La classe `App` contient le `main`.

— L'objet `ClientApp` est le coeur de l'application. C'est l'objet qui communique et exécute les actions que souhaite effectuer l'utilisateur.

— La classe `Table` crée un objet contenant les informations propre à une table donnée.

— La classe `KeySpace` crée un objet contenant les informations propre à un keyspace donné.

— L'objet `Connection` permet d'exécuter des requêtes sur la base de données.

— Les objets héritant de `QueriesFactory` sont des générateurs de requêtes, ils envoient des suites de requêtes à Cassandra.

— Les objets héritant de `DataGenerator` sont des générateurs de données, ils permettent de génerer des données pour effectuer les simulations.

6.2 Visualisation des métriques

Pour visualiser les métriques, nous utilisons l'application nommée `Graphite` écrite en Python.

Graphite est l'union de trois logiciels :

— **carbon**, un logiciel permettant de traiter des données temporelles, comme l'évolution de la charge d'un noeud en fonction du temps.

— **whisper**, une base de données stockant toutes ces données

— **graphite webapp**, une application web permettant la visualisation de ces données.

Les métriques sont directement envoyées à Graphite en créant une connexion entre le metrics de Cassandra et Graphite, voir figure 5 Pour cela, on utilise l'objet `GraphiteReporter` dans le module `metrics-graphite`.

Ces données sont ensuite traitées par Carbon puis sauvegardées dans la base de données Whisper. Enfin, la web-app de Graphite affiche ces informations sous forme de graphes (courbes, histogrammes...) dans un navigateur web.

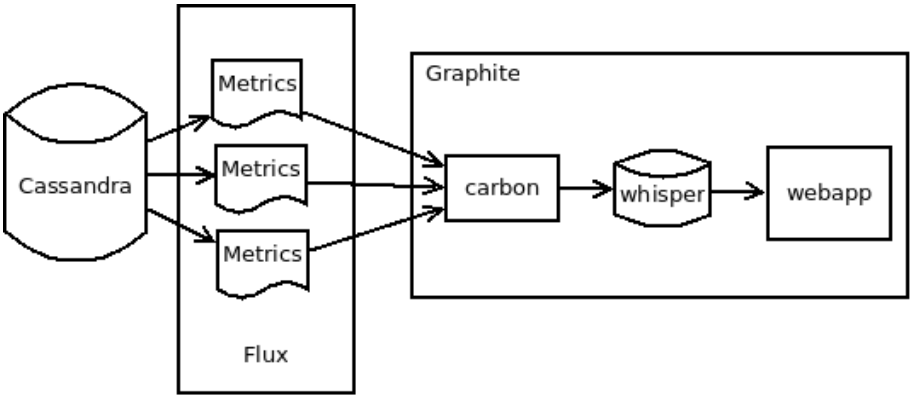


FIGURE 5 – Graphite, union de trois logiciels

L'utilisation de cette application permet de visualiser un grand nombre de métrics déjà disponible dans la version originale de Cassandra. On peut citer par exemple, le volume total de données stockées sur un cluster, le temps de lecture des données, la taille totale du cache... En ajoutant un metric pour mesurer la charge d'un noeud, on peut visualiser celle-ci au cours du temps grâce au divers formats de graphiques proposés par Graphite.

7 Réalisation du projet

7.1 Modification de Cassandra

Une partie importante du projet consiste à adapter le comportement de la base de données Cassandra afin de répondre aux besoins. Nous avons décidé de travailler sur la version **2.1.2**, dernière version stable au début du projet. La modification du code source de Cassandra demande, outre la création de code, des compétences de lecture et de compréhension de code. En effet, il n'existe pas de documentation officielle concernant le code source de Cassandra. On peut tout de même retrouver des explications sur l'architecture interne dans le wiki officiel hébergé par la fondation Apache [Fou13] ainsi que d'autres détails sur la documentation réalisée par Datastax [Dat15a].

Dans la mesure du possible, nous avons respecté les conventions de codage de Cassandra [Fou14], elles mêmes découlant des conventions de codage Java créées par Sun.

La majeur partie du projet, pour la modification de Cassandra, a consisté en sa compréhension, car pour modifier un comportement, il faut d'abord connaître celui qui existe. C'est pourquoi l'architecture du projet s'est construite en même temps que son implémentation (l'architecture représentant bien 4 ou 5 fois plus de temps que l'implémentation). Pour trouver les points d'intérêt dans le code (cheminement d'une requête de lecture par exemple), l'utilisation des *logs* est primordiale. Couplé à un environnement de travail (IDE) permettant l'usage du *"Go To Definition"* (permet de retrouver instantanément, à partir d'un endroit où une fonction est utilisée, l'endroit où elle est définie), nous avons alors pu retracer efficacement différentes parties de Cassandra sur lesquels nous avons travaillé. Les diagrammes de classes et de séquences de l'architecture ont été créé à partir de cette méthode (mais ils sont fortement simplifiés pour faciliter leur compréhension).

7.1.1 Outils utilisés

CCM

CCM (Cassandra Cluster Management) est un script/bibliothèque écrit en Python permettant de gérer facilement un cluster de Cassandra sur une machine locale. Il a été développé par Sylvain Lebresne, développeur chez Datastax (entreprise spécialisée dans Cassandra), et est dynamique dans son évolution avec des mises à jour constantes depuis plusieurs années. C'est pourquoi nous avons choisi cet outil pour nous aider dans notre développement.

Il faut savoir que chaque instance de Cassandra correspond à un noeud, et qu'une instance est faite pour fonctionner sur chaque machine du réseau (un noeud équivaut à une machine). Mais en phase de développement et de tests, on souhaite pouvoir lancer un cluster entier sur une seule machine, ce qui n'est pas aisé à faire. C'est ici que CCM intervient en automatisant la création/gestion/suppression d'un cluster Cassandra en local.

Son installation est peu triviale et son utilisation n'est pas dénuée de bugs, mais dans l'ensemble, CCM nous a apporté un gain de temps très important dans la réalisation de ce projet.

Ainsi, la création, le lancement et la destruction d'un cluster se déroule de la manière suivante. Tout d'abord, on crée un cluster à partir des fichiers sources de Cassandra.

```
# py ccm create nom_cluster --install-dir=<chemin_de_cassandra>
```

Ensuite, on ajoute des noeuds à ce cluster. Ici on décide d'en créer 3. C'est à ce moment que toute la configuration de noeuds se fait de manière automatique, une étape longue et critique lorsqu'elle est réalisée à la main.

```
# py ccm populate -n 3
```

On peut alors lancer le cluster, qui va se charger des créer les instances de Cassandra pour chaque noeud du réseau.

```
# py ccm start
```

Lorsqu'on a fini de réaliser nos tests, on peut ensuite arrêter les instances de Cassandra tournant en fond de tâche.

```
# py ccm stop
```

Il est à noté que si l'on veut relancer le même cluster, il suffit de refaire la commande de lancement sans repasser par les étapes précédentes. Le cluster ainsi que les données qu'il possède ne sont pas supprimé. Mais si on veut le faire, alors il suffit d'une commande.

```
# py ccm remove
```

Class Visualizer

Class Visualizer est un outil de visualisation de code Java sous la forme d'un diagramme de classes UML. Il est très utile pour comprendre les relations entre les classes dans un projet. Cependant, la structure particulière de Cassandra (nombreux singletons statiques) ne permet pas d'utiliser toute la puissance de cet outil car il n'arrive pas à retrouver les liens de dépendances entre les classes. Nous nous sommes donc peu servi de cet outil.

7.1.2 Affectation des requêtes de lecture

Solution implémentée

Lors d'une requête de lecture, le comportement par défaut de Cassandra est d'envoyer la requête de lecture au noeud possédant l'objet *"le plus proche"* (défini suivant la configuration) qui doit renvoyer la donnée, et d'envoyer la requête de lecture à plusieurs autres noeuds possédant une copie de l'objet qui doivent renvoyer un *digest*, correspondant à un hash MD5 de la donnée demandée. Le nombre de noeuds auquel on envoie une requête de type *digest* est paramétrable grâce au niveau de consistance. Ainsi, une consistance de 1 ne demandera pas de *digest* et sera donc plus rapide, mais la donnée ne sera pas forcément la plus récente. Car l'intérêt du *digest* est de pouvoir comparer la valeur de l'objet demandé sur plusieurs noeuds pour essayer au maximum de donner la valeur la plus récente. Mais envoyer la donnée est plus gourmand sur le réseau qu'envoyer un valeur de hash, c'est pourquoi cette technique est utilisée. On attend une réponse de chaque noeud concerné pour pouvoir répondre à la requête.

Dans notre cas, nous voulons comme comportement envoyer une requête de lecture à **tous** les noeuds possédant une copie, quelque soit le niveau de consistance, et en leur

demandant de renvoyer la **donnée complète**, donc sans *digest*. On répond ensuite à la requête dès qu'**un** noeud à répondu.

Pour implémenter ce comportement, nous avons modifier la classe `AbstractReadExecutor` pour qu'elle nous renvoie une instance qui n'utilise pas de *digest*, instance de la classe `NoDigestReadExecutor` que nous avons créé (voir figure 20). De plus, elle envoie une requête à tous les noeuds, peu importe le niveau de consistance. Pour créer des `NoDigestReadExecutor`, il a fallu empêcher la création des autres classes (ce sont toutes des classes privées statiques contenues dans `AbstractReadExecutor`), créées à partir de la méthode `getReadExecutor`. Seule notre nouvelle classe peut maintenant être instanciée.

Un autre point important concerne la suppression d'une requête de lecture. Par défaut, c'est un comportement qui n'existe pas dans Cassandra, et qui n'a certainement pas été imaginé possible au vu de l'implémentation. La première chose que nous avons réalisé est d'envoyer des requêtes pouvant être supprimées dans les files d'attentes, avec des méthodes différentes de celles pouvant être supprimées. Si l'on reprend la figure 20, on peut remarquer un changement dans les messages envoyés au stage de lecture. On a réutilisé les fonctions précédentes mais en ajoutant "Removable" à la fin de leur nom pour les différentier.

Pour comprendre comment supprimer une requête de la file d'attente, il faut comprendre ce qu'il y a dans une file d'attente. Si l'on regarde l'architecture d'un stage (voir figure 18), on s'aperçoit qu'une requête dans la file d'attente sera représenté par une instance de `FutureTask`. Or, ce que l'on souhaite supprimer correspond à une instance de `ReadCommand` (la classe gérant la requête de lecture). Il a donc fallu trouver un système pour comparer ces deux classes. La première chose a été d'ajouter une classe `RemovableFutureTask` pour gérer les requêtes pouvant être supprimées de la file d'attente (voir figure 21). Cette classe possède la particularité de laisser la vérification de l'égalité de la classe à son attribut *runnable* (qui correspond au *callable* de `FutureTask` avant sa transformation). Ainsi, si l'on cherche à supprimer une requête de la file d'attente en envoyant une instance de `ReadCommand`, on compare cette instance au objet de la file d'attente, qui eux-même passeront par leur attribut *runnable*, et ainsi de suite jusqu'à retrouver l'objet de `ReadCommand` original (voir figure 6).

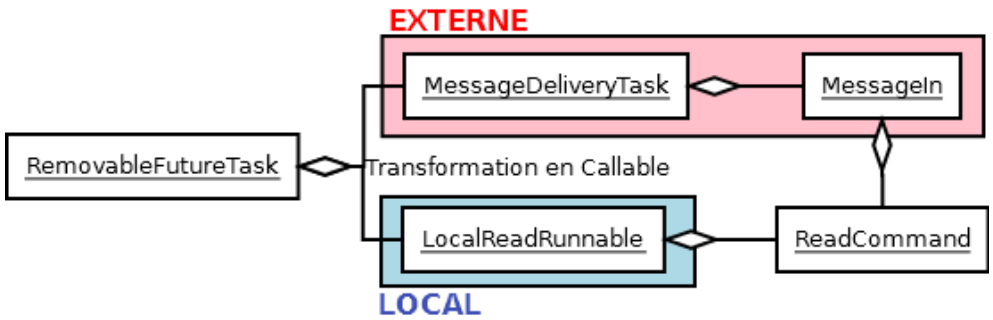


FIGURE 6 – Classes redéfinissant la méthode *equals* pour pouvoir comparer et supprimer des requêtes de lectures

Les instances de `ReadCommand` sont identifiées et comparées grâce à leur attribut "timestamp" et "key".

Maintenant qu'il est possible de supprimer une requête de lecture d'une file d'attente, il faut s'intéresser à l'envoi/réception d'un message de suppression, ainsi qu'à son traitement. Nous avons donc créer un nouveau stage pour gérer ces messages : le stage **READ_REMOVE**. Il est mono-threadé, contrairement au stage de lecture qui est multi-threadé (32 threads par défaut).

Les messages qu'il traite sont les verbes **READ_REMOVE** dont le traitement se fait dans la classe `ReadRemoveVerbHandler`. La création d'un message se fait à partir de `ReadCommand` avec la méthode `createRemoveMessage`, qu'il suffit alors d'envoyer aux autres noeuds par le biais du `MessagingService`. Cette tâche est donnée à la classe `AsbtractReadExecutor` et sa méthode `makeRemoveRequests`.

Problèmes et limites rencontrés

La suppression d'une requête de la file d'attente de lecture a été une partie compliquée, car les relations entre les objets dans une file d'attente et les objets que l'on cherche à comparer ne sont pas évidentes à retrouver.

Il a aussi fallu faire attention à la concurrence étant donné que nous travaillons dans un environnement multi-threadé. C'est pourquoi nous utilisons la méthode native de suppression d'une file (représentée en interne par une `ConcurrentLinkedQueue`) pour assurer une meilleure sécurité. C'est ce qui nous a conduit à choisir comme solution de redéfinir les méthodes *equals*.

Le dernier point qu'il a fallu prendre en compte est qu'il y a deux manières de faire une requête : local et externe. Il faut donc s'assurer que les deux manières de faire sont traitées. Le fait d'avoir ces deux manières est un gain de temps en production, mais complexifie cependant la compréhension et la lisibilité du code.

Travail restant à réaliser

Cette partie du projet a été réalisé entièrement. On peut malgré tout s'interroger sur une méthode plus efficace (en terme de lisibilité) pour traiter la suppression d'une requête. Mais cela nécessiterait de chambouler un peu plus le code original, et un travail beaucoup plus important pour s'assurer que le comportement de Cassandra n'est pas altéré.

Des tests sur le nombre de threads accordés au stage de suppression de requête seraient aussi intéressant. L'augmentation de threads provoquerait une diminution du traitement des autres stages, mais augmenterait la vitesse de suppression des requêtes, permettant alors d'en traiter moins en plusieurs exemplaires.

Les attributs permettant d'identifier les requêtes sont aussi à revoir. Actuellement, le *timestamp* n'est pas la valeur la plus sûre car il est toujours possible que deux requêtes identiques soient lancées au même moment par deux clients différents. Une possibilité seraient de générer un identifiant aléatoirement pour chaque instance de `ReadCommand`. Le risque de collision avec cet identifiant couplé au timestamp serait alors quasiment nul. Cela demanderait alors de modifier les *serializer* pour être capable de transmettre cette donnée dans un message.

7.1.3 Réaffectation des requêtes de lecture

Solution implémentée

Charge des files d'attente des requêtes de lecture (avec gestion de la décrémentation lorsqu'une requête est traitée) grâce à un compteur. Communication de la charge à tous les autres noeuds du réseau (Gossiper, ApplicationState, LoadReadBroadcaster, versionedValue).

Dans le projet, nous devons implémenter un système permettant, pour chaque noeud, de s'affecter toutes les requêtes de lecture de sa file d'attente. Cette décision se prend suivant un protocole de réaffectation, qui doit être modifiable facilement.

Pour prendre cette décision, il est nécessaire d'avoir comme information la *charge* de chaque noeud du réseau pour la file d'attente des requêtes de lecture. Cela implique d'avoir cette donnée localement, de pouvoir la modifier, et de pouvoir la transmettre aux autres noeuds.

Cette donnée se situe dans la classe `SEPExecutor` sous l'attribut `effectiveLoad`. Chaque fin de traitement de requête de lecture entraîne la décrémentation du compteur (si la valeur actuelle du compteur est supérieure à 0). Le point important est d'utiliser un type permettant de gérer la concurrence (environnement multi-threadé). C'est pourquoi nous utilisons le type `AtomicLong` spécialement prévu pour cet effet.

Maintenant que nous possédons un compteur du nombre de tâches affectées, il est facile de retrouver le nombre de tâches non affectées (un compteur de tâches en attentes existe déjà). Il faut alors communiquer cette donnée. Pour ça, nous avons ajouté cette information à celle déjà présente dans les communications Gossip (voir la section 6) entre les noeuds.

Cet ajout se déroule en plusieurs étapes. Tout d'abord, nous avons ajouté la donnée **READ_LOAD** dans `ApplicationState`, permettant de dire ce qu'on transmet entre les noeuds. Le `Gossiper` transmettra l'information contenu dans **READ_LOAD**.

Ensuite, nous avons créé une classe qui gère la relation entre le `Gossiper` et le reste de l'application : `LoadReadBroadcaster`. Son rôle est, à intervalle régulier, de mettre à jour l'information de **READ_LOAD** à partir de l'information du `SEPExecutor` du stage de lecture. C'est aussi à partir de lui que l'on peut récupérer les informations de la charge de tous les autres noeuds du réseau à partir de sa méthode `getLoadInfo`.

Problèmes et limites rencontrés

La solution actuelle se base sur un compteur pour déterminer les requêtes qui ont été affectées dans la file d'attente. Cependant, l'aspect multi-threadé peut, plus tard, poser des problèmes. En effet, il se peut que le compteur ne soit pas parfaitement en adéquation avec les requêtes que nous nous sommes assignées, et pour lesquels nous avons envoyé des requêtes de suppression aux autres noeuds. On enverrai alors des requêtes de suppression lors du traitement de la requête de lecture alors qu'on avait déjà envoyé ces requêtes auparavant lors de l'affectation.

Le risque est alors d'envoyer des messages inutiles dans le réseau. Le contraire n'est pas possible si on modifie la charge avant de parcourir la file d'attente pour envoyer les messages de suppressions (cela aurait été plus problématique car il n'y aurait pas eu de réponse à la requête du client du coup).

Une autre solution, plus couteuse en terme de temps de développement, serait de créer une nouvelle file d'attente pour les requêtes affectées, et de basculer les requêtes dans cette file lorsque l'on veut l'assigner à ce noeud.

Travail restant à réaliser

Nous n'avons pas réalisé tous les objectifs dans cette partie par manque de temps. Il reste de nombreux points à aborder pour achever ce travail, mais ils ont été tout de même réfléchi.

Il faut encore exécuter le protocole de réaffectation, qui doit être lancé dès que l'on traite une requête de lecture ou de suppression. Ce protocole doit vérifier les conditions d'exécution qui lui sont propres (SLVO ou AverageDegree ne possèdent pas les mêmes conditions), et si elles passent, doit modifier la charge et parcourir la file d'attente pour envoyer un message de suppression à tous les autres noeuds possédant ces mêmes requêtes.

Une requête affectée ne doit pas entrainer de message de suppression lors de son traitement car ils ont été réalisés plus tôt par le système de réaffectation. L'idée serait de marquer les requêtes (un booléen par exemple) pour indiquer si elles sont affectées ou non.

Le dernier point concerne la configuration du protocole de réaffectation. Il est nécessaire de pouvoir le modifier facilement afin de comparer les résultats sur les performances des différents protocoles. La solution envisagée est d'utiliser le même système qui est actuellement en place pour configurer la réplication : un système de réflexion paramétrable pour chaque Keyspace, où il suffit de rentrer le nom de la classe du protocole de réaffectation lors de la création d'un Keyspace.

7.1.4 Réplication des objets

Solution implémentée

Après discussion avec la communauté de développeurs de Cassandra, une solution de placement a été implémentée. En effet, pour rappeler le besoin sur le placement des copies, il a été souhaité que le placement de l'objet initial se fasse avec une fonction de hachage $H_0(c)$ avec c la clé primaire de l'objet. Le placement des réplicas devait s'effectuer à l'aide de la fonction $H_1(c)$ pour la première copie, $H_2(c)$ pour la deuxième copie et ainsi de suite.

La méthode implémentée consiste à utiliser la fonction $H_0(c)$ pour le placement initial. Le premier réplica est placé grâce à $H_1(H_0(c))$, le second grâce à $H_2(H_0(c))$ et ainsi de suite.

La création de cette implémentation a nécessité la création d'une nouvelle stratégie de réplication nommée `MultiHashStrategy`. Elle peut être utilisée comme n'importe quelle stratégie de réplication, c'est à dire qu'il suffit d'indiquer son nom lors de la création d'une keyspace. Il est a noté que cette stratégie ne fonctionne correctement qu'avec la méthode de répartition en **Murmur3**, car cette stratégie utilise des fonctions de hachages Murmur3. Si on utilise une autre méthode de répartition, la distribution des copies ne sera plus uniforme.

Problèmes et limites rencontrés

Outre la difficulté à comprendre les mécanismes de Cassandra, plusieurs problèmes se sont posés à nous. En effet, la communication de la clé primaire d'un objet n'est pas possible immédiatement dans le développement de nouvelles stratégies. Il faudrait modifier le comportement de plusieurs autres classes, ce qui nous était difficile durant le temps imparti. Le comportement a légèrement été modifié comme décrit ci-dessus.

Un autre aspect (mineur dans notre projet) est que cette fonctionnalité entraîne une perte de la capacité de Cassandra à s'adapter à son réseau. En effet, Cassandra est conçu pour se reconfigurer facilement lors d'ajout ou de suppresion de noeuds dans son réseau. Cependant, dans notre projet, nous nous intéressons à un réseau de taille fixe qui n'évoluera pas dans le temps.

Et enfin, il est important de mesurer les différences entre les deux solutions. Effectivement, si elles se ressemblent sur leur principe, un point important est à noter : une différence forte au niveau des collisions. Une *collision* est lorsqu'une fonction de hachage retourne la même valeur pour deux entrées différentes.
Prenons par exemple, le cas de deux objets ayant pour clé primaire $c1$ et $c2$ et avec $H_0(c1) = k$ et $H_0(c2) = k$ (nous avons donc une collision).

- Solution initiale :
  Placement initial : $H_0(c1) = k$ et $H_0(c2) = k$
  Premier réplica : $H_1(c1) = k1$ et $H_1(c2) = k2$ avec $k1 \neq k2$ sauf si $H_1(c1) = H_1(c2)$
  Avec une collision sur le placement de la donnée initiale, la probabilité d'avoir les premiers réplicas sur le même noeud dépend du domaine de collision de la fonction H_1.
- Solution implémentée :
  Placement initial : $H_0(c1) = k$ et $H_0(c2) = k$
  Premier réplica : $H_1(H_0(c1)) = H_1(k) = k3$ et $H_1(H_0(c2)) = H_1(k) = k3$
  Second réplica : $H_2(H_0(c1)) = H_2(k) = k4$ et $H_2(H_0(c2)) = H_2(k) = k4$
  Avec une collision sur le placement de la donnée initiale, toutes les copies seront placées sur le même noeud deux à deux.
  Autrement dit, toutes les données qui entrent en collision pour la première fonction de hachage, auront leur premier réplica sur le même noeud Y et ainsi de suite.

On a une donc une répartition des données moins efficace car si on avait une collision, on souhaitait que les réplicas ne soit pas sur les mêmes noeuds pour ces données. Toutefois, le risque de collision reste assez faible car la fonction de hachage Murmur3 donne des résultats codés sur 64 bits (le type *Long* en Java), soit $2^{64} - 1$ solutions possibles.

Travail restant à réaliser

Bien que l'ensemble de la partie ait été réalisée, il n'est pas aisé de changer de fonction de hachage. En effet, si l'utilisateur souhaite utiliser de nouvelles fonctions de hachage, il doit recréer une nouvelle stratégie de réplication et recompiler l'ensemble. On pourrait alors imaginer une stratégie de réplication qui charge des fonctions de hachage grâce à des paramètres fournis à l'exécution.

7.1.5 Popularité des objets

Solution implémentée

Aucune solution technique n'a été implémentée. Cependant, les réflexions sur le développement ont avancé. Le point est fait dans la partie *Travail restant à réaliser*.

Problèmes et limites rencontrés

Créer un mécanisme de popularité est un gros ajout dans Cassandra qui nécessite la modification (et donc la compréhension) d'un ensemble de mécanismes. En effet, nous intervenons au niveau des requêtes pour compter leur nombre et savoir quel objet elles vont utiliser. S'ajoutent ensuite la communication réseau pour transmettre le nombre total de requêtes et la décision de créer ou non ainsi que de supprimer ou non des copies.

Travail restant à réaliser

Il faut premièrement ajouter au niveau du traitement des requêtes, un compteur du nombre de requêtes totale reçues (déjà existant dans Cassandra). Chaque noeud ajoutera n compteurs (s'il possède la gestion de n données ou n noeuds selon l'algorithme utilisé) pour compter les requêtes de chaque objet.

Puis, un mécanisme à part, qui à chaque période de temps prédéfini par l'utilisateur, enverrait le nombre total de requêtes reçues à ses voisins. La réflexion n'est pas claire ici car on a besoin de créer une synchronisation entre tous les noeuds pour le calcul du nombre.

Les noeuds choisissent de créer, de supprimer ou de ne rien faire à propos des objets dont ils ont la gestion.

Il faut penser aussi que les stratégies de réplication (et d'autres mécanismes) vont chercher le nombre de copies des objets dans les fichiers de configuration de Cassandra. Ici, le nombre de copies variant en fonction de la popularité, il faut ajouter en plus, un mécanisme de transmission du nombre de copies. L'étendu de cette modification n'est pas encore au point car cela peut éventuellement fausser d'autres mécanismes de Cassandra (comme la répartition des requêtes aux réplicas...).

A la fin de chaque période de temps défini, il faut remettre à zéro les calculs de popularité. Cela doit s'effectuer aussi par un message commun au noeud.

Après la création d'une copie, celle-ci doit être attribuée à un noeud, cela nécessite donc de comprendre comment un noeud assimile de nouveaux objets. De même, lors de la suppression, que se passe-t-il des requêtes présentes dans la file d'attente du noeud ?

7.2 Création d'une application cliente

L'application cliente permet à l'utilisateur d'interagir avec la base de données Cassandra.

7.2.1 Gestion de Cassandra

Solution implémentée

Les constantes : nom des commandes, temps avant un timeout... sont configurables dans `simulassandra.client.Config`.

L'application cliente fonctionne grâce à l'objet `App` créant un objet `ClientApp`. Cet objet réceptionne les commandes saisies par l'utilisateur dans une console (ou terminal) et exécute les actions qui correspondent à chaque commande.

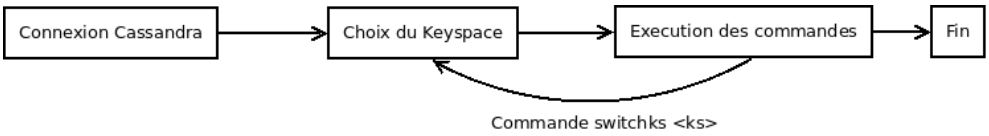


FIGURE 7 – États de l'application cliente

Le schéma 7 décrit les différents états de l'application.

Connexion Cassandra Dans un premier temps, l'utilisateur doit entrer l'adresse IP à laquelle se situe la base de données Cassandra à laquelle il souhaite se connecter. L'utilisateur a trois essais, sinon l'application se ferme. Le nombre d'essais est configurable. L'adresse entrée étant valide, l'objet `ClientApp` crée un objet `Cluster` initialisant la connexion à la base de données.

Choix du keyspace Une fois connecté, l'utilisateur est invité à entrer le keyspace dans lequel il souhaite travailler. Si le keyspace entré n'existe pas, l'application lui propose de le créer. Il doit alors choisir une stratégie de réplication et un facteur de réplication. L'objet `ClientApp` crée alors un objet `KeySpace` contenant les informations du keyspace et l'objet `Session` permettant d'envoyer des requêtes à la base de données.

Exécution des commandes Le keyspace étant choisi, l'utilisateur est libre d'exécuter les commandes proposées par l'application.

On distingue deux types de commandes. Les commandes permettant de consulter l'état du logiciel et de la base de données. C'est-à-dire, les données du keyspace courant (celui auquel nous sommes connectés), les tables stockées dans le keyspace...

`# help`
Liste les commandes proposées par l'application accompagnée d'une brève description de chaque commande.

`# showksdata`
Affiche les méta-données du keyspace courant : le nom du keyspace, le facteur de réplication et la stratégie de réplication utilisée.

`# showtabledata <table>`
Affiche les méta-données de la table `<table>` : le nombre d'enregistrements, le nom des

colonnes de la table et leur type.

`# lstable`

Liste les tables stockées dans le keyspace courant.

`# quit`

Quitte l'application cliente, on arrive donc sur l'état *Fin*.

Ainsi que les commandes permettant de modifier le contenu de la base de données, d'effectuer des requêtes, de modifier le keyspace courant.

`# import <file>`

Exécute les requête au format CQL contenues dans le fichier `<file>`.

`# switch <ks>`

Change de keyspace. Passage sur le keyspace `<ks>` avec création du keyspace s'il n'existe pas. On retourne sur l'état *Choix du Keyspace*.

`# queries <qf> <s> <ns> <nq>`

Exécute une suite de requêtes générées aléatoirement selon le seed `<s>` et la classe `<qf>`. Le nombre de simulation est définie par `<ns>` et le nombre de requêtes par simulation par `<nq>`. Cette commande est détaillée dans la partie suivante.

`# createdatafile <file> <nb_tables> <nb_rows> <data_length>`

Écrit dans un fichier `<file>` la suite de requêtes CQL permettant de créer `<nb_tables>` et d'insérer `<nb_rows>` enregistrements par table. Les données crées sont de longueur `<data_length>`.

Ajout de commandes Une commande est une suite de caractères saisie par l'utilisateur pour exécuter une action. Elle est au format :

`# nom_commande <argument_1> <argument_2> ... <argument_n>`

Il est aisé d'ajouter des commandes à l'application cliente.

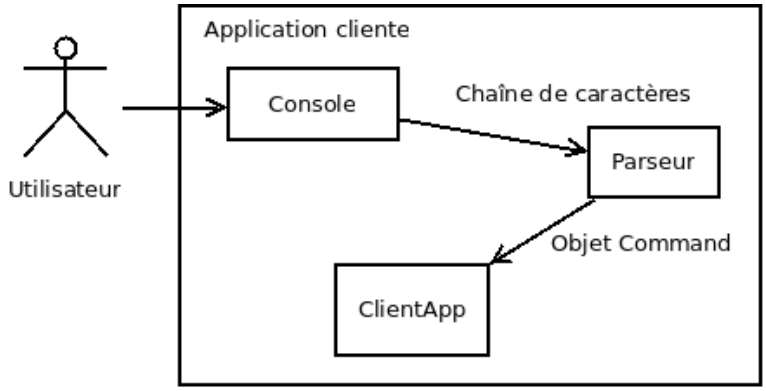


FIGURE 8 – Traitement d'une commande

Un objet de type `Command` est la représentation d'une commande. Ces attributs sont l'identifiant de `nom_comande` et un tableau de chaîne de caractères contenant tous les arguments de la commande.

La commande tapée par l'utilisateur est d'abord traitée par les méthodes de la classe `InputCommandParser`. Le but est de transformer la chaîne de caractères en un objet de type `Command`. L'objet est ensuite récupéré dans la méthode `run` de l'objet `ClientApp` pour être exécutée par la méthode `execute` du même objet.

Pour ajouter une commande, il faut donc ajouter les constantes définissant le nom de la commande, l'identifiant de la commande ainsi que son nombre d'arguments dans la classe `Config`. Il faut ensuite modifier la méthode `whichAction` de `InputCommandParser` pour attribuer au nom de la commande le bon identifiant et ajouter le traitement correspondant à la commande dans la méthode `execute` de `ClientApp`.

Problèmes et limites rencontrés

Travail restant à réaliser

7.2.2 Gestion des requêtes

Solution implémentée

Problèmes et limites rencontrés

Travail restant à réaliser

7.3 Création d'une application de visualisation de données

La visualisation des métrics de Cassandra n'étant pas une priorité du client.

8 Tests et résultats

Pour les sections suivantes, les tests se sont déroulés sur un réseau d'Amazon composé de 10 noeuds reliés entre eux par Internet. Il faut relativiser ces résultats, car l'expérience se déroule dans des conditions idéales d'utilisation : les données sont en cache de la base de données distribuée.

A noter : pour des raisons de coûts de location des machines, il ne nous a pas été possible d'effectuer plus de tests en environnement *réel*, c'est à dire non simulé sur la même machine grâce à l'outil *CCM*.

8.1 Différence de temps d'exécution avec la nouvelle stratégie de placement

Pour comparer les performances de notre nouvelle stratégie de placement des données, nous allons effectuer deux tests :

— On compare les temps d'exécution de différents jeux de requêtes sur un facteur de réplication de 1 (sans copie) et un facteur de 2 (avec une copie) (voir figure 23)

— On réitère l'expérience avec une version de Cassandra non modifiée par nos soins (voir figure 24)

On remarque un gain d'environ 3 secondes sur de grands jeux de requêtes entre un facteur de réplication de 1 et de 2. On peut expliquer ce gain de performance qui est sûrement du au facteur de réplication.

8.2 Différence de temps d'exécution pour comparer le travail effectué sur les requêtes

Pour comparer les performances de notre traval effectué au niveau du traitement des requêtes, nous allons comparer toujours les mêmes tests (figures 23 et 24). Pour

effectuer notre analyse, nous devons étudier les courbes rouges ensembles et les bleues ensembles, afin de mettre en évidence lors d'un même facteur de réplication, le gain obtenu.

La différence entre notre version et celle de Cassandra non modifiée est d'environ 1 seconde sur de grands jeux de requêtes. Les résultats sur de petits jeux de requêtes s'expliquent nottamment par un sur-coût lors des communications entre les noeuds et sur la taille des données qui est relativement petite. Les courbes de la figure 22 montrent d'ailleurs les différences de temps d'exécution entre une donnée de 10 Ko et de 1 Mo.

9 Annexe - Un simulateur

9.1 Introduction

Derrière ce nom peu évocateur, se cache un prototype d'une base de données distribuée. Il porte le nom de simulateur car, bien évidemment, durant le temps imparti, nous n'aurions pas pu coder une base de données distribuée complète avec l'ensemble des mécanismes de traitements des données. Nous avons repris les fonctions les plus essentielles :

— Le stockage des données de manière distribuée selon différentes stratégies.

— Le concept de distribution : Les données sont réparties et traitées par différents noeuds.

— La possibilité de faire des requêtes qui ont un coût de traitement.

— Les requêtes sont stockées sous forme de files. Il suffit de prendre la tête pour pouvoir la traiter.

— Un système de communication intra-noeuds.

Le simulateur a pour objectif d'afficher des résultats et d'affiner la première approche des fonctions à implémenter (en particulier pour la popularité, fonction complexe à créer dans Cassandra du fait de sa distributivité).

9.2 Etat des lieux

Le simulateur est écrit en C. Il fait environs 1300 lignes de code (commentaires Doxygen compris) pour 19 fichiers. La compilation du logiciel s'effectue dans *src* en tapant la commande :

```
# make
```

La compilation de la documentation Doxygen s'effectue dans *data* et en tapant la commande :

```
# make
```

Le logiciel est capable de créer des grappes de *noeuds* (nommées *clusters*). Il est capable de créer des *requêtes*. Chaque noeud est capable de traiter des requêtes qui sont dans sa *file d'exécution*. Il est possible de changer de *stratégie de réplication et de positionnement* des données. Actuellement, deux stratégies ont été implémentées :

— On hash l'identifiant de la donnée, on obtient un noeud et on prend les 3 suivants.

— Même que précédemment avec augmentation en fonction de la popularité. Les 2 objets les plus populaires sont augmentés.

Le schéma de la figure 19 permet de mieux comprendre son fonctionnement. Nous allons le détailler.
Tout d'abord, nous créons des données. Ensuite, nous créons des requêtes sur des données. On demande ensuite au Cluster de transmettre ces requêtes au noeud souhaité. Quand le noeud aura du temps, il traitera la requête. Si celle-ci est sur une donnée dont il a la gestion, il la traitera. Sinon, il demandera à la stratégie de lui fournir les noeuds responsables, et il transmettra.

9.3 Inconvénients de cette approche

Premièrement, l'architecture telle qu'elle est pensée n'est pas distribuée : elle est centralisée. En effet, l'objet cluster, est un élément central du réseau. Mais dans une première approche, cela suffisait.

Les noeuds sont synchrones. En effet, cette partie est omise dans la description précédente. Afin de pouvoir "simuler" totalement tous les paramètres, le cluster dispose d'une horloge appelée *pas_de_calcul* qui consiste à donner du crédit temps à intervalle régulier aux noeuds. Les effets sont donc différents sur une base de données totalement distribuée et non synchrone que sur notre simulateur.

9.4 Améliorations possibles

Pour obtenir un bon simulateur, il faudrait commencer par corriger les problèmes ci-dessus. Pour enlever le problème de centralisation, les noeuds devraient connaître une partie de leurs voisins, qu'il faudrait réactualiser de temps en temps. La stratégie de réplication devrait être répliquée sur tous les noeuds. Pour corriger le problème de synchronisation, et devenir asynchrone, on pourrait jouer avec un thread / noeud. Le départ serait donné par un mutex et la fin par un signal par exemple. Cependant, on perd la possiblité d'exécuter pas à pas le calcul.

D'autres fonctionnalités et besoins pourraient ou devraient être implémentés :

— Possibilité de générer des requêtes sur des données aléatoires (relativement facile, créer un générateur à objets requête...)

— Placer les requêtes sur les noeuds de façon aléatoire (facile aussi, choisir aléatoirement les noeuds sur lequels seront poussées les requêtes dans le générateur)

— Travailler sur les besoins concernant les requêtes. (modification du traitement au niveau d'un noeud, moins trivial...)

10 Conclusion et Remerciements

TODO : C'est la fin.

Références

[AAB+12] D. Auber, D. Archambault, R. Bourqui, A. Lambert, M. Mathiaut, P. Mary, M. Delest, J. Dubois, and G. Melançon. The tulip 3 framework : A scalable software library for information visualization applications based on relational data. [Research Report] RR-7860, hal-00659880, 2012.

[ABKU99] Y. Azar, A. Broder, A. Karlin, and E. Upfal. Balanced allocations. *SIAM Journal on Computing*, 29(1) :180–200, 1999.

[ADA05] Metwally A, Agrawal D, and El Abbadi A. Efficient computation of frequent and top-k elements in data streams. 2005.

[Dat14] DataStax. Datastax opscenter : Datastax. <http://www.datastax.com/what-we-offer/products-services/datastax-opscenter>, 2014. [Accessed 15 January 2015].

[Dat15a] DataStax. About apache cassandra — datastax cassandra 2.0 documentation. <http://docs.datastax.com/en/cassandra/2.0/cassandra/gettingStartedCassandraIntro.html>, 2015. [Accessed 04 April 2014].

[Dat15b] DataStax. Datastax cassandra 2.1 documentation. <http://www.datastax.com/documentation/cassandra/2.1/cassandra/gettingStartedCassandraIntro.html>, 2015. [Accessed 15 January 2015].

[FKSS14] P. Felber, P. Kropf, E. Schiller, and S. Serbu. Survey on load balancing in peer-to-peer distributed hash tables. *IEEE Communications Surveys and Tutorials*, 16(1) :473–492, 2014.

[Fou09] The Apache Software Foundation. The apache cassandra project. <http://cassandra.apache.org>, 2009. [Accessed 15 January 2015].

[Fou13] The Apache Software Foundation. Architectureinternals - cassandra wiki. <http://wiki.apache.org/cassandra/ArchitectureInternals>, 2013. [Accessed 04 April 2015].

[Fou14] The Apache Software Foundation. Codestyle - cassandra wiki. <http://wiki.apache.org/cassandra/CodeStyle>, 2014. [Accessed 04 April 2015].

[GC15] Marios Hadjieleftheriou Graham Cormode. Finding the frequent items in streams of data. <http://dimacs.rutgers.edu/~graham/pubs/papers/freqcacm.pdf>, 2015. [Accessed 1 Avril 2015].

[Hew10] E. Hewitt. *Cassandra : The Definitive Guide*. O'Reilly Media, 2010.

[LA09] M. Prashant L. Avinash. Cassandra - a decentralized structured storage system. 2009.

[MRS05] M. Mitzenmacher, A. Richa, and R. Sitaraman. The power of two random choices : A survey of techniques and results. 2005.

[ÖV11] T. Özsu and P. Valduriez. *Principles of Distributed Database Systems*. Computer science. Springer, 2011.

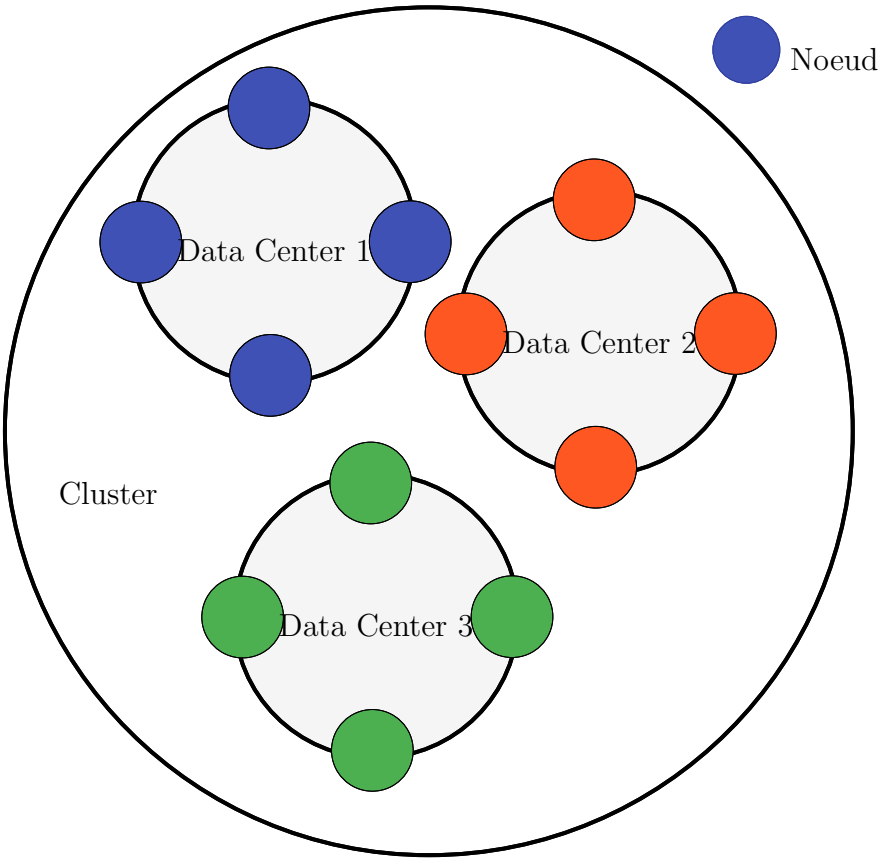


FIGURE 9 – Visualisation d'une base de données distribuée sous forme de cluster possédant trois data center

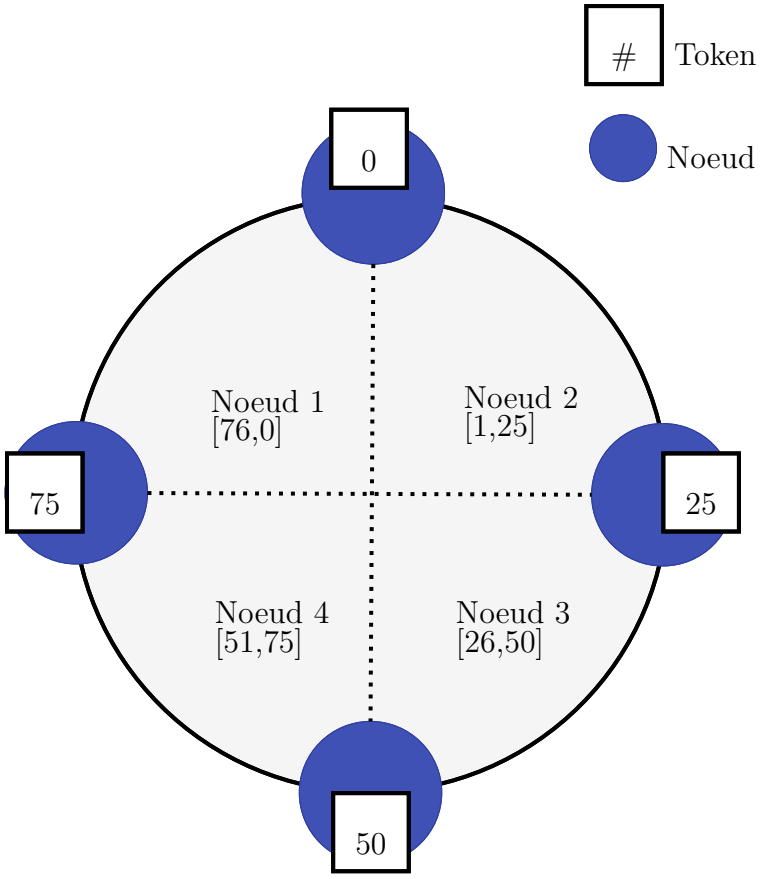


FIGURE 10 – Exemple de partitionnement des données dans une base de données distribuée

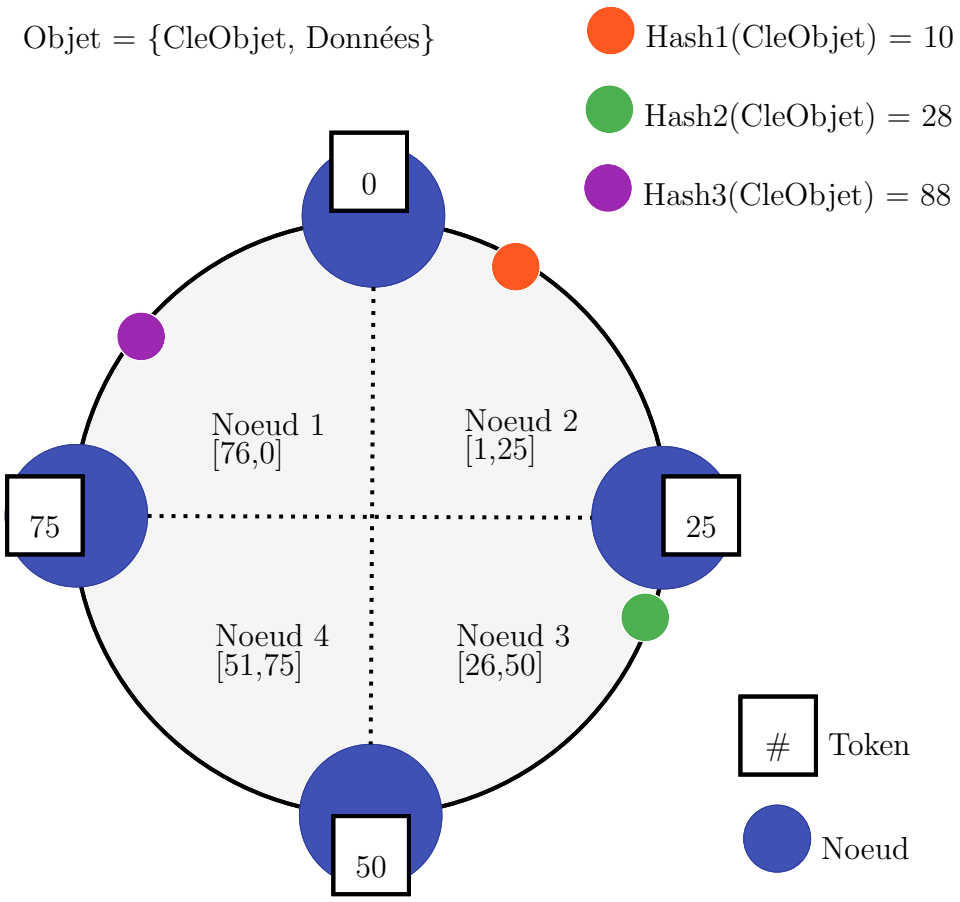


FIGURE 11 – Partitionnement des réplicas d'un objet avec une fonction de hachage pour chaque réplica

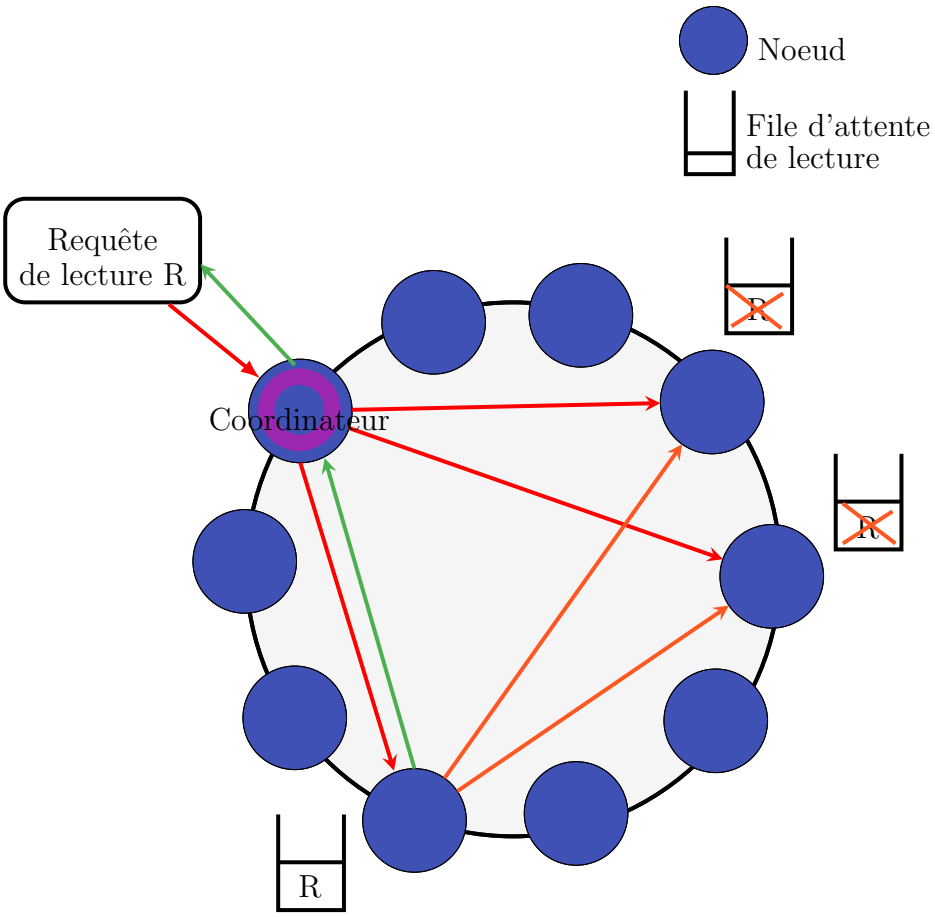


FIGURE 12 – Cheminement d'une requête de lecture dans une base de données distribuée avec la prise en charge de l'affectation (un seul noeud traite la requête)

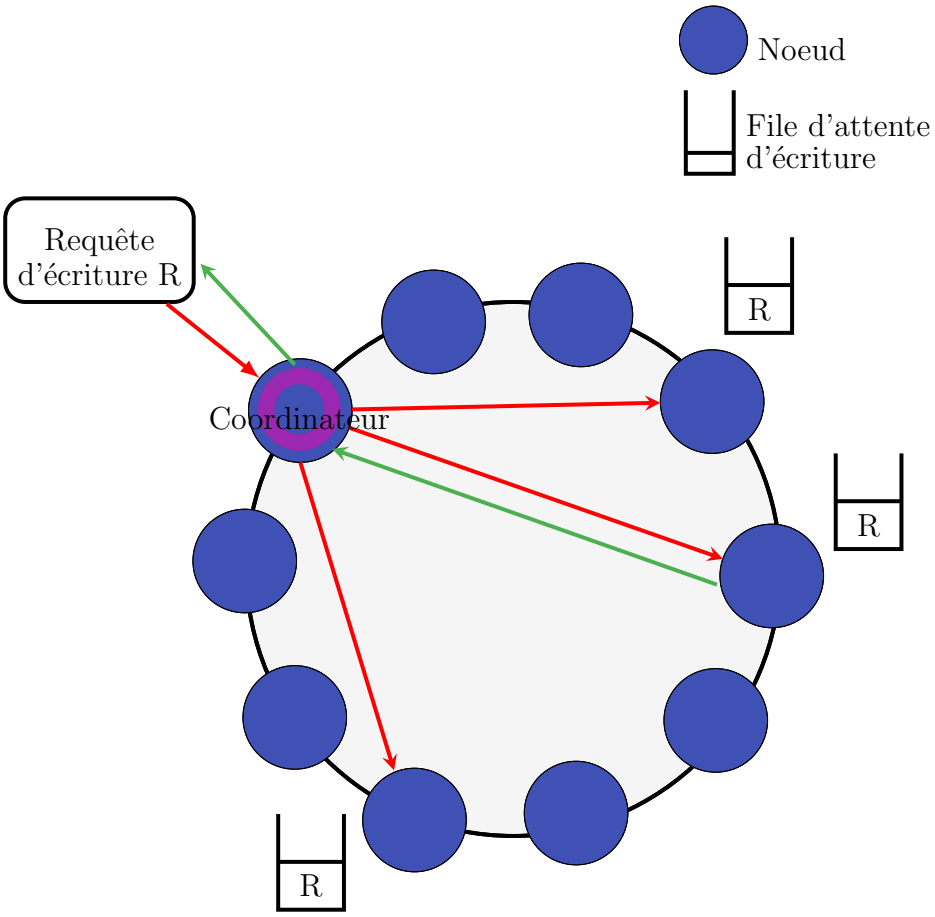


FIGURE 13 – Cheminement d'une requête d'écriture dans une base de données distribuée

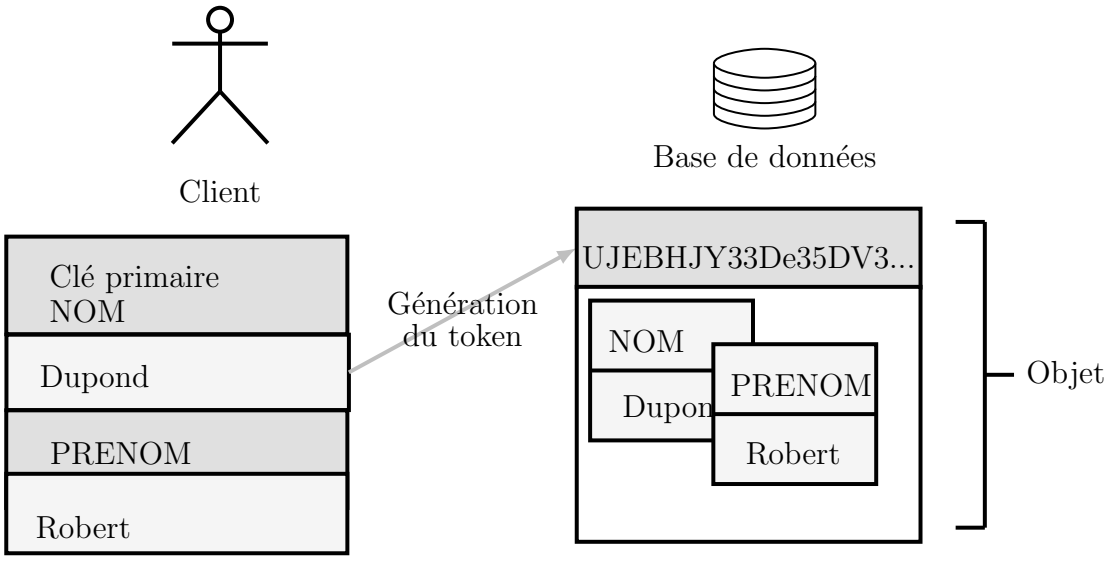


FIGURE 14 – Passage d'une répresentation des données pour le client à une représentation pour la base de données

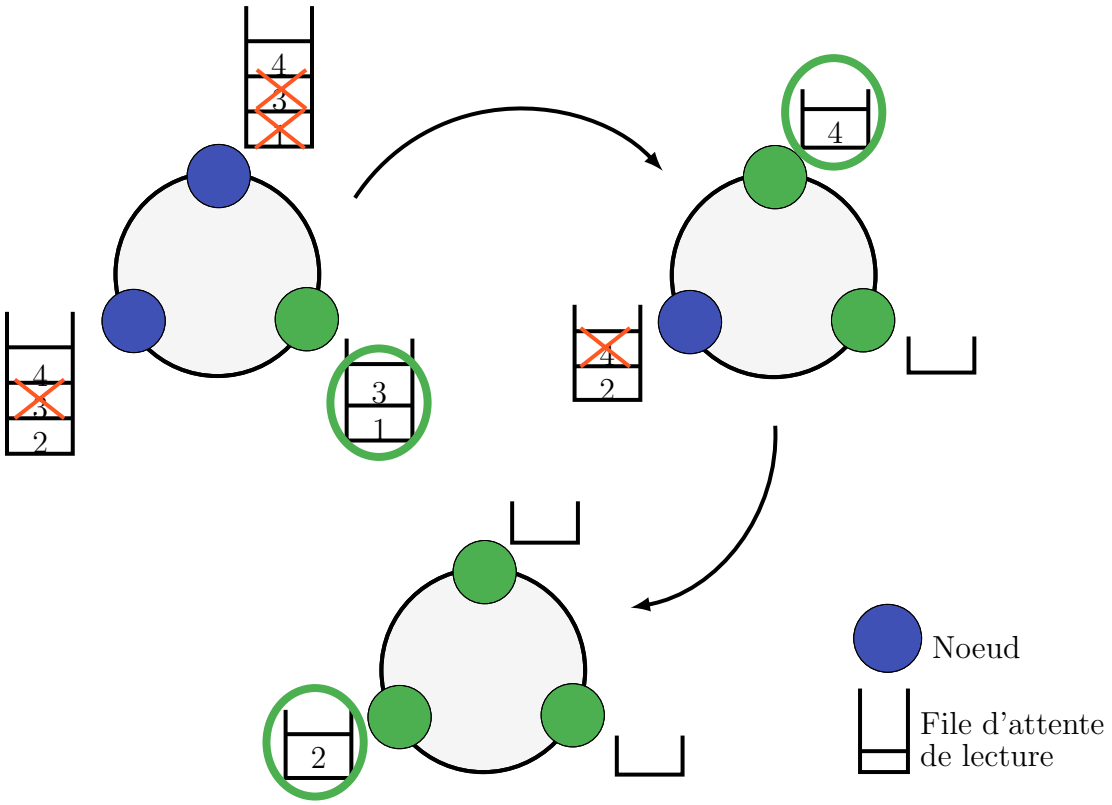


FIGURE 15 – Fonctionnement de l'algorithme de réaffectation des requêtes de lecture SLVO

Algorithm 3: SPACESAVING(k)

$n \leftarrow 0$;
$T \leftarrow \emptyset$;
foreach i **do**
  $n \leftarrow n + 1$;
  if $i \in T$ **then** $c_i \leftarrow c_i + 1$;
  else if $|T| < k$ **then**
    $T \leftarrow T \cup \{i\}$;
    $c_i \leftarrow 1$;
  else
    $j \leftarrow \arg\min_{j \in T} c_j$;
    $c_i \leftarrow c_j + 1$;
    $T \leftarrow T \cup \{i\} \setminus \{j\}$;

FIGURE 16 – Pseudo-code de l'algorithme SpaceSaving (Source : voir [GC15])

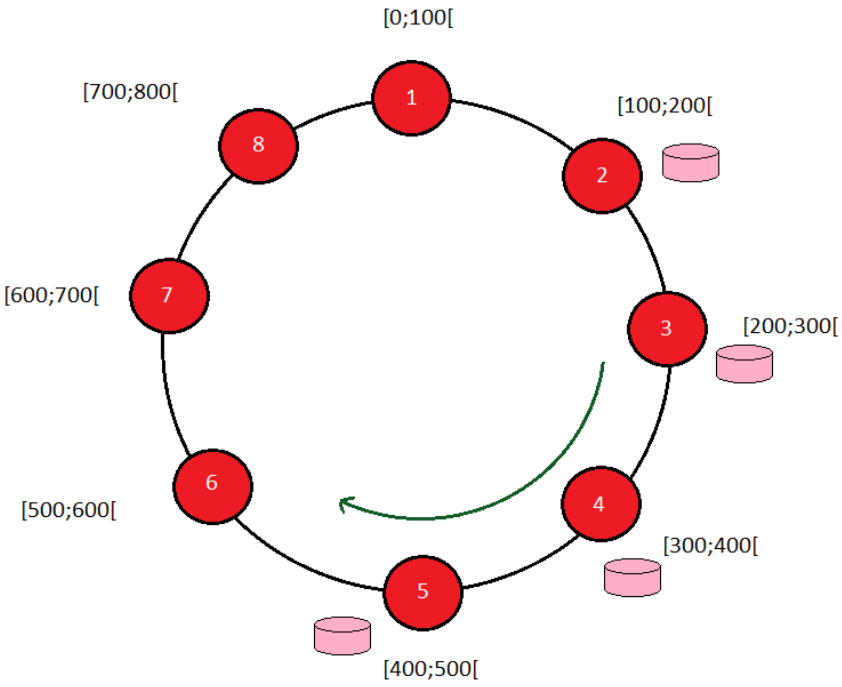


FIGURE 17 – Répartition des copies sur les noeuds dans une base de données distribuée

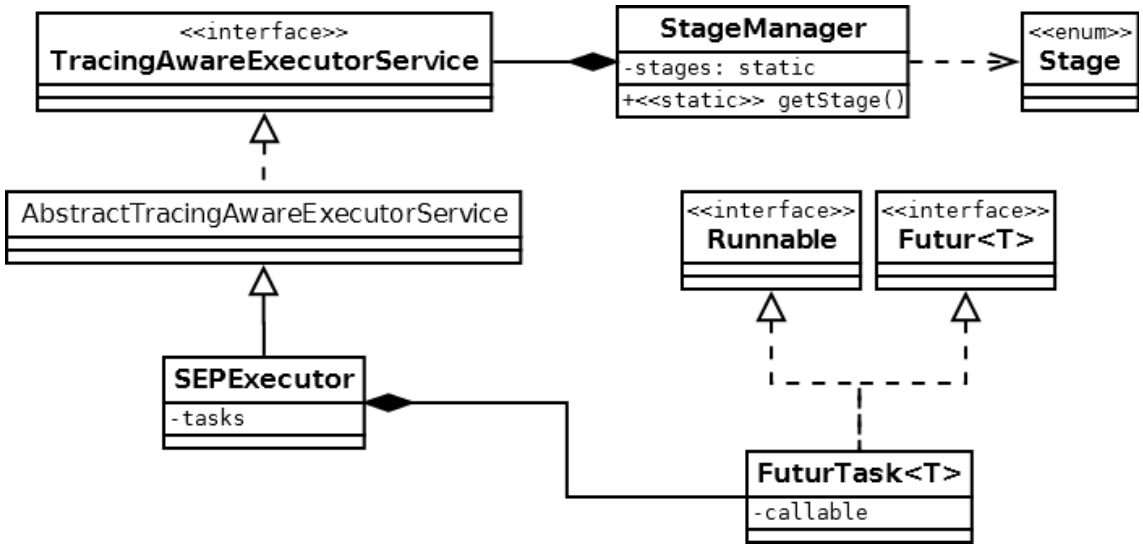


FIGURE 18 – Diagramme de classe simplifié de l'architecture SEDA dans Cassandra

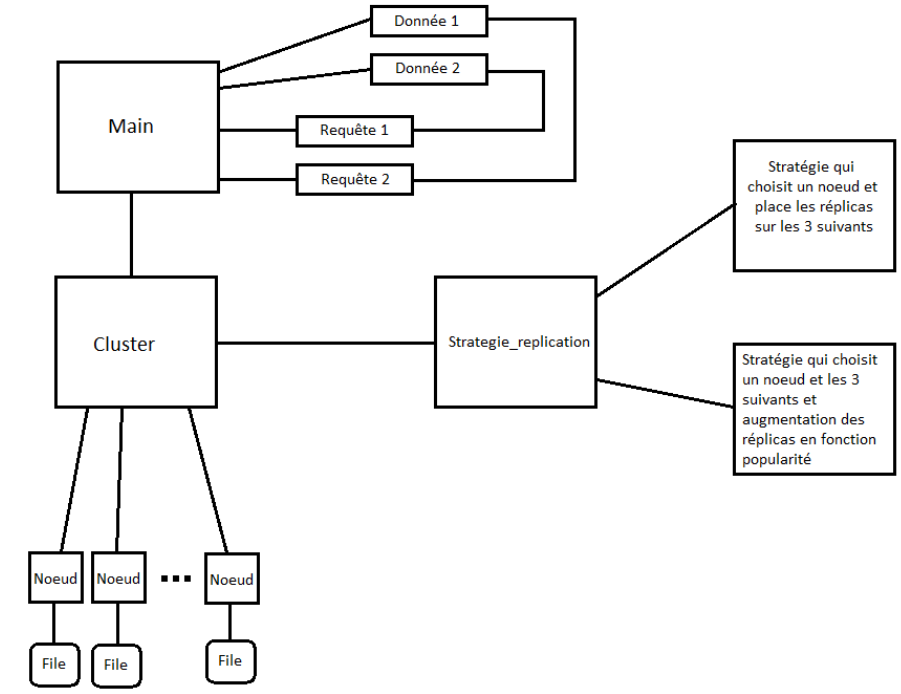


FIGURE 19 – Architecture des objets du simulateur

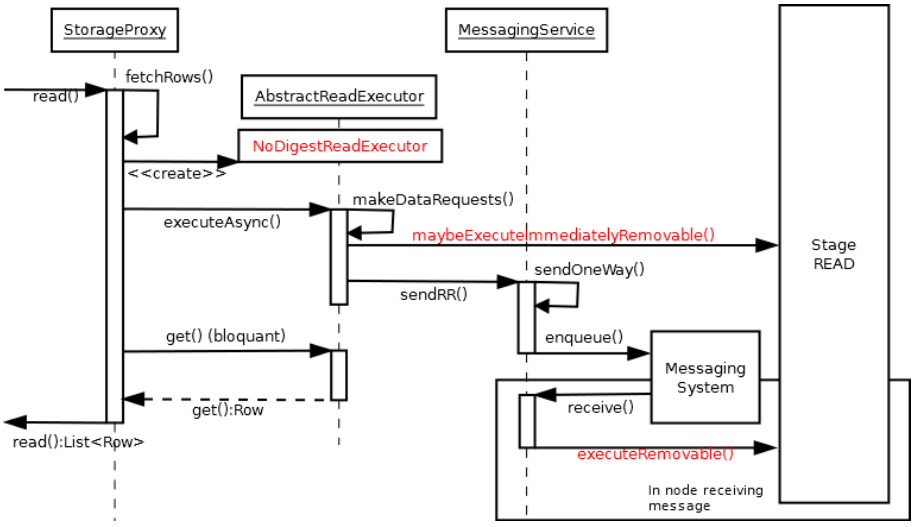


FIGURE 20 – Diagramme de séquence simplifié d'une requête de lecture dans Cassandra après modification

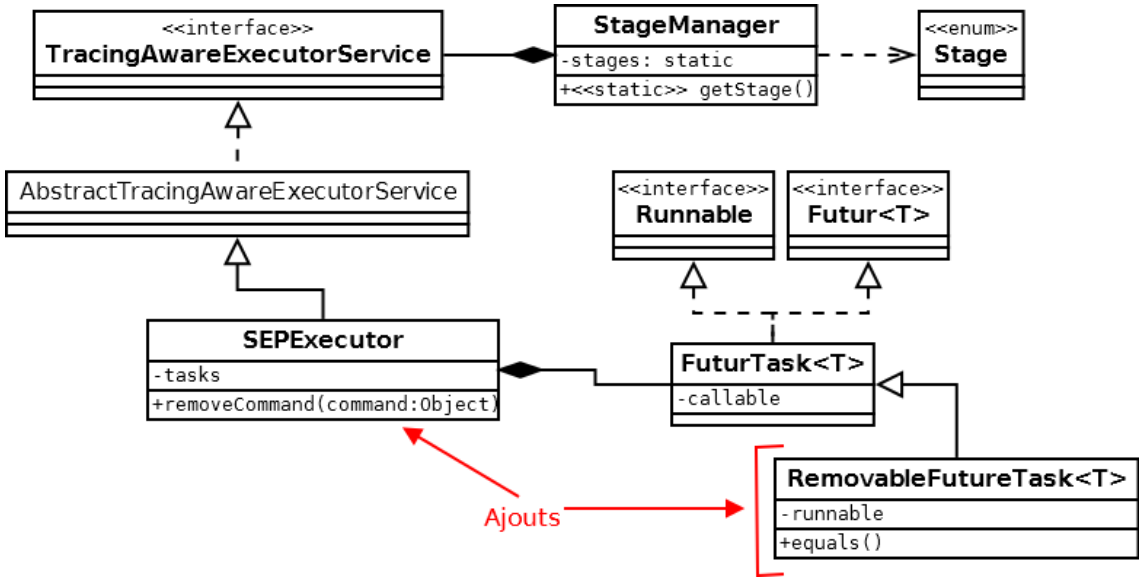


FIGURE 21 – Diagramme de classe simplifié de l'architecture SEDA dans Cassandra après modification

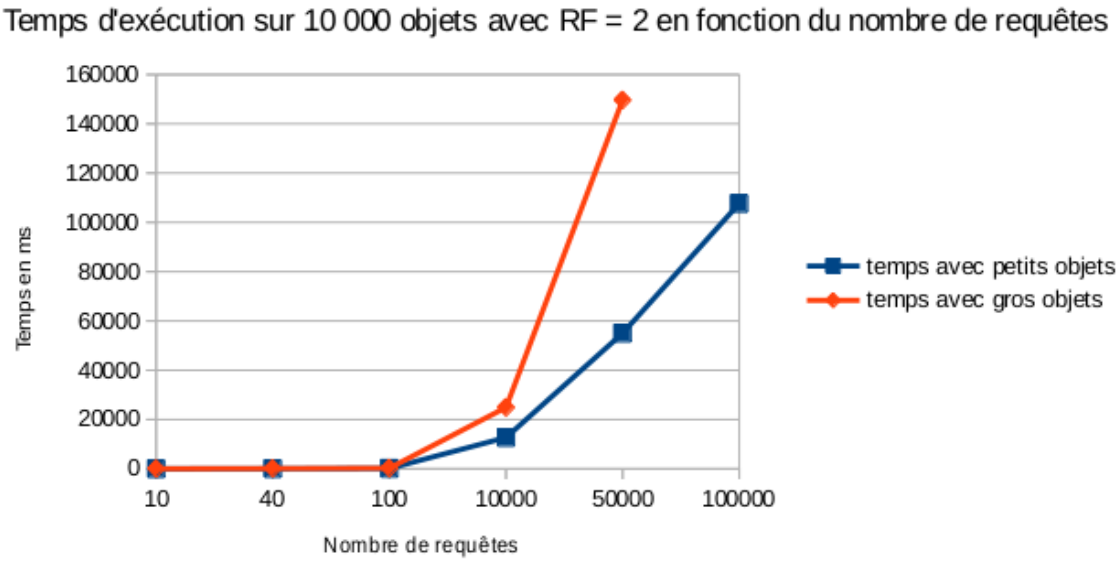


FIGURE 22 – Comparatif des temps d'exécution de requêtes en fonction de leur nombre et de la taille des objets

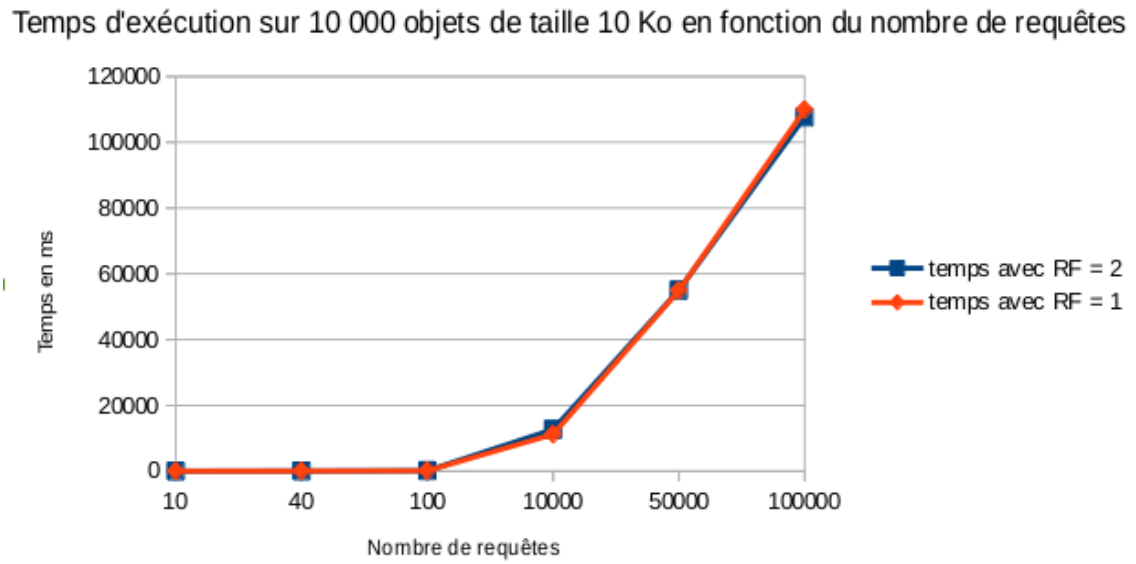


FIGURE 23 – Comparatif des temps d'exécution de requêtes en fonction du facteur de réplication et du nombre de requêtes sur des petits objets

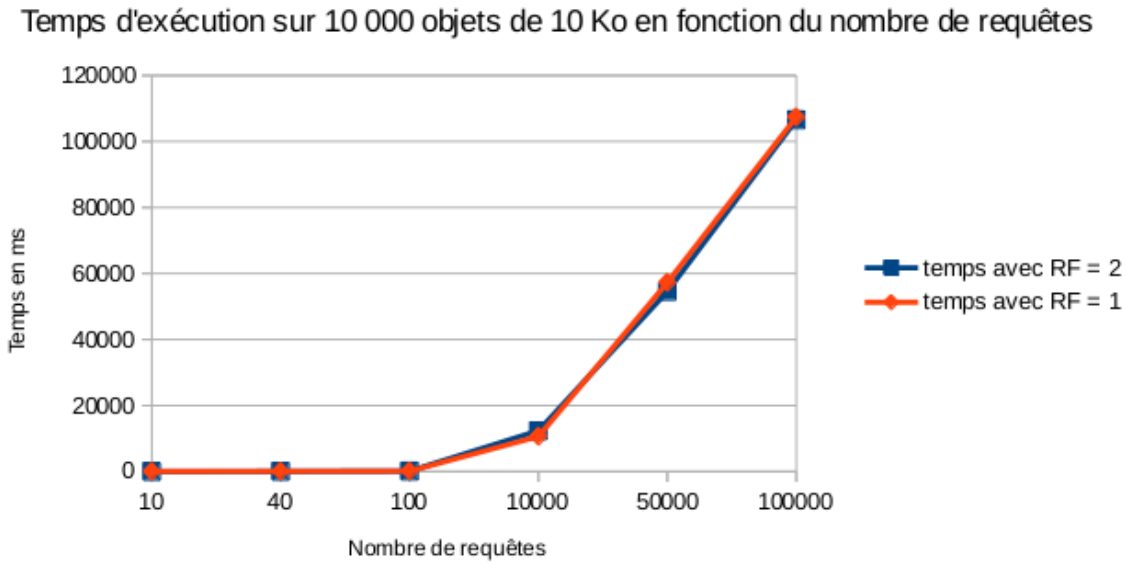


FIGURE 24 – Comparatif des temps d'exécution des requêtes en fonction du facteur de réplication et du nombre de requêtes sur des petits objets sur une base de Cassandra non modifiée