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

Memoire final

Kevin Barreau Guillaume Marques Corentin Salingue

3 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).

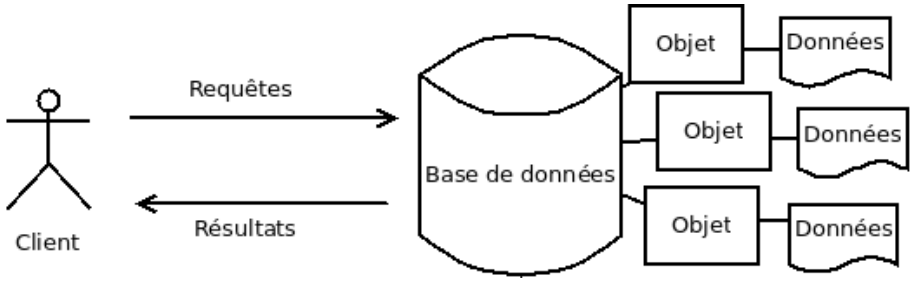


FIGURE 1 – Interactions client/base de données

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 3). 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 4). 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 6).

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 7). 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 8).

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 9 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 10

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 11. 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

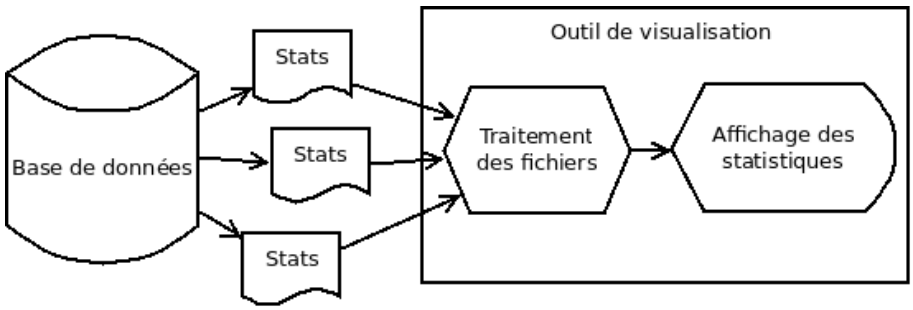


FIGURE 2 – Processus pour la visualisation des statistiques

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...

Présentations par d'autres auteurs sur le projet :

Système distribué

— Le livre [ÖV11] apporte les principes sur les bases de données distribuée.

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.

— L'article [FKSS14] est aussi important car il 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.

— Le papier [LA09] apporte les explications des créateurs de Cassandra.

— Le lien [Dat15] 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.

Visualisation

— Le lien [Dat14] est intéressant pour le projet car il apporte un logiciel 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. La proximité des créateurs peut apporter une solution viable.

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 compatabilisant 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 Intéractions 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

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 modifications de celles-ci.

3.6.1 Enregistrement des données

Ecriture dans un fichier Lorsque les données locales d'un noeud sont modifiées, on les enregistre dans un fichier. L'écriture est de la forme `itération de l'algorithme; identifiant du noeud; charge du noeud;`

3.6.2 Affichage des données

Définition Un *graphe* est un ensemble de points appelés *sommets*, dont certaines paires sont directement reliées par un (ou plusieurs) lien(s) appelé(s) *arêtes* [com15].

Noeuds L'application doit permettre la représentation de chaque noeud par un sommet.

Analyse syntaxique Lors de l'éxecution d'un algorithme, la charge de chaque noeud est enregistrée dans un fichier. Un analyseur syntaxique (un programme qui possède des règles et qui agit sur un fichier donné en entrée selon celles-ci) découpe chaque ligne du fichier pour récupérer le moment auquel a été enregistrée l'information (`itération de l'algorithme`), le noeud concerné (`identifiant du noeud`) et la charge de ce noeud à ce moment (`charge du noeud`).

Charge des noeuds A chaque sommet est associée une valeur correspondant à la charge de ce noeud. Ces données sont récupérées grâce à l'analyseur syntaxique.

Film de l'éxecution Cela consiste à afficher la charge des noeuds dans l'ordre chronologique, c'est à dire dans l'ordre des itérations croissant.

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 Protocole de test

La conformité des algorithmes implémentés est assurée par un protocole de test suivant la démarche :

— Définir un réseau R, un ensemble d'objets O et un ensemble de requêtes Q

— Faire tourner l'algorithme à la main avec R, O et Q

— Stocker l'état final du réseau

— Faire valider ce processus par le client

— Exécuter l'algorithme sur ordinateur avec R, O et Q

— Vérifier les résultats constatés avec les résultats attendus

S'il y a une différence entre les deux résultats, une vérification par le client peut être envisagée dans le cas de résultats *presque* similaires. La notion de similitude est laissée à l'appréciation de l'équipe en charge du projet, lors de la vérification.

4.4 Visualisation des données

4.4.1 Actualisation de la vue

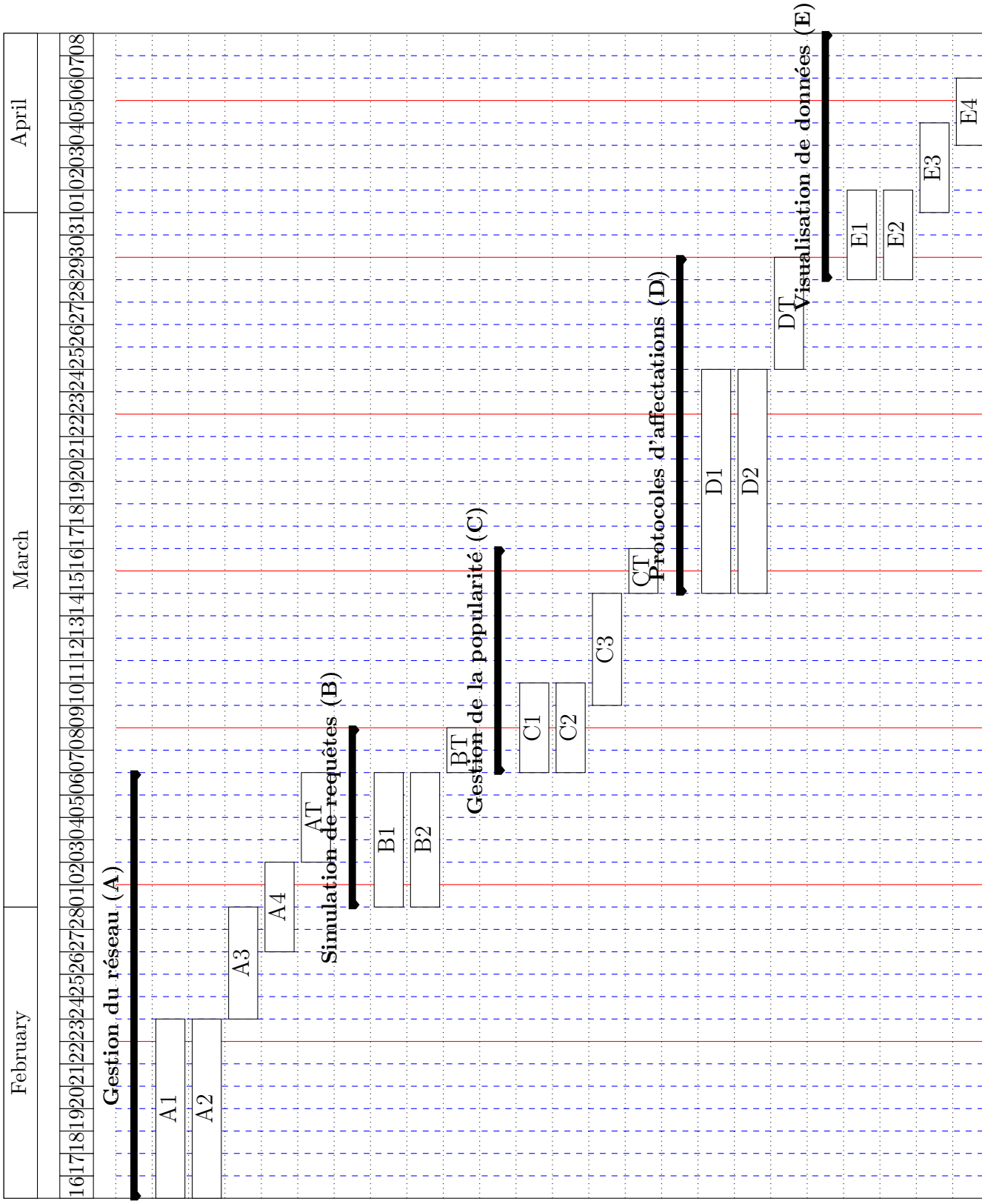
L'état du réseau doit être visible en temps réel.

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	Création des noeuds		
A2	Données locales des noeuds		Initialisation et implémentation
A3	Communication des données locales entre noeuds		
A4	Gestion des replicas		
AT	Tests groupe A		Vérification, tests, mémoire
B1	Générateur de requêtes		A détailler
B2	Importateur de jeu de requêtes		A détailler
BT	Tests groupe B		Vérification, tests, mémoire
C1	Popularité objet sur noeud		
C2	Space-Saving Algorithm		
C3	Popularité d'un objet dans le réseau		
CT	Tests groupe C		Vérification, tests, mémoire
D1	Implémentation SLVO		
D2	Implémentation AverageDegree		
DT	Tests groupe D		Avec client
E1	Prise en main Tulip		
E2	Ecriture des données dans un fichier		(+Analyseur syntaxique)
E2	Représentation réseau		
E3	Représentation données		
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

Les besoins fonctionnels sont répartis selon les 3 gros thèmes suivant :
TODO : Insérer l'architecture

7 Réalisation du projet

7.1 Modification de Cassandra

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

Reprise du diagramme de l'architecture avec les méthodes et attributs ajoutés (en couleur). Explication de AbstractReadExecutor qui envoie à tout le monde. Consistance de 1 pour s'arrêter au premier résultat. Identification d'une requête (timestamp, key, keyspace, columnFamily). Message de suppression (stage READ_REMOVE, verb READ_REMOVE, override equals, tasks removable).

Problèmes et limites rencontrés

File d'attente de Runnable : grosse complication pour la suppression. Environnement multi threadé : faire attention à la concurrence. Gérer local et distant.

Travail restant à réaliser

Amélioration de la suppression pour un code plus clair.

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).

Problèmes et limites rencontrés

Mélange dans les requêtes de lecture (slice, précis...) peut fausser peut être. Manque de temps.

Travail restant à réaliser

Exécuter le protocole de réaffectation (lorsqu'on traite une requête de lecture et lorsqu'on traite une requête de suppression). Parcours de la file d'attente pour envoyer les messages de suppression. Indiquer aux messages assignés de ne pas envoyer de message de suppression (empecher à une requête d'envoyer à nouveau une requête de suppression alors que l'on en a déjà envoyé une plus tôt avec le protocole). Configuration du choix du protocole de la même façon que pour la réplication (réflexion).

7.1.4 Réplication des objets

Solution implémentée

Problèmes et limites rencontrés

Travail restant à réaliser

7.1.5 Popularité des objets

Solution implémentée

Problèmes et limites rencontrés

Travail restant à réaliser

7.2 Création d'une application cliente

7.2.1 Gestion de Cassandra

Solution implémentée

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

8 Tests et résultats

TODO : Mettre les tests et les expliquer

9 Bonus - 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 12 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.

[com15] Wikipedia community. Théorie des graphes - wikipédia. <`http://fr.wikipedia.org/wiki/Th%C3%A9orie_des_graphes#D.C3.A9finition_de_graphe_et_vocabulaire`>, 2015. [Accessed 5 February 2015].

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

[Dat15] 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].

[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. Tthe 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.

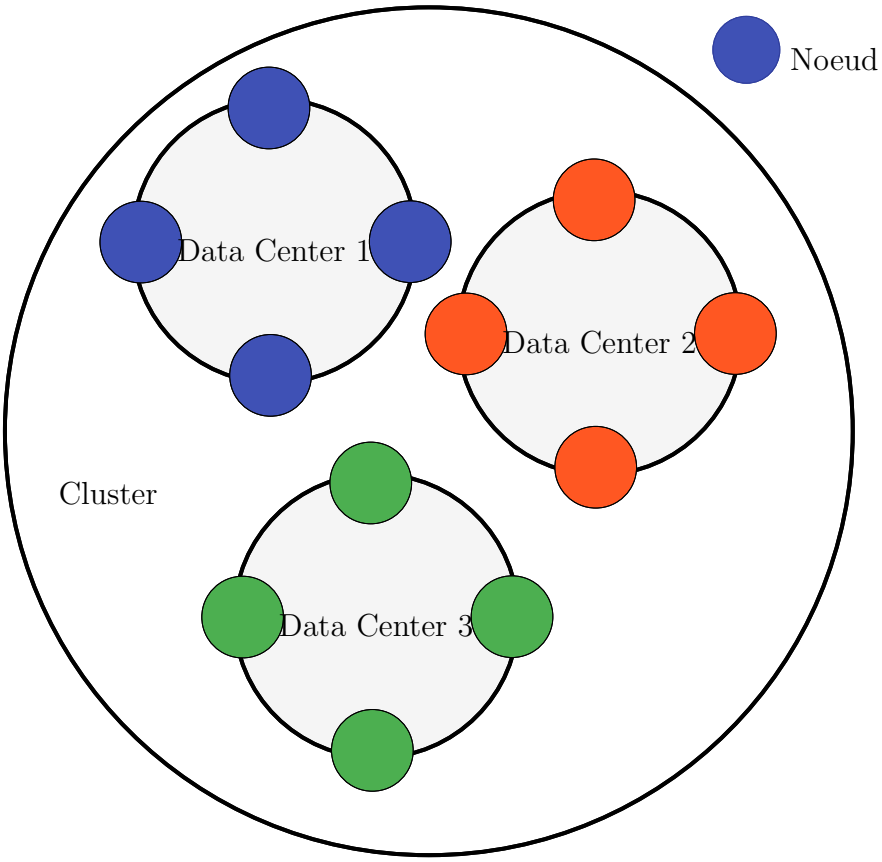


FIGURE 3 – Visualisation d'une base de données distribuée sous forme de cluster possédant trois data center

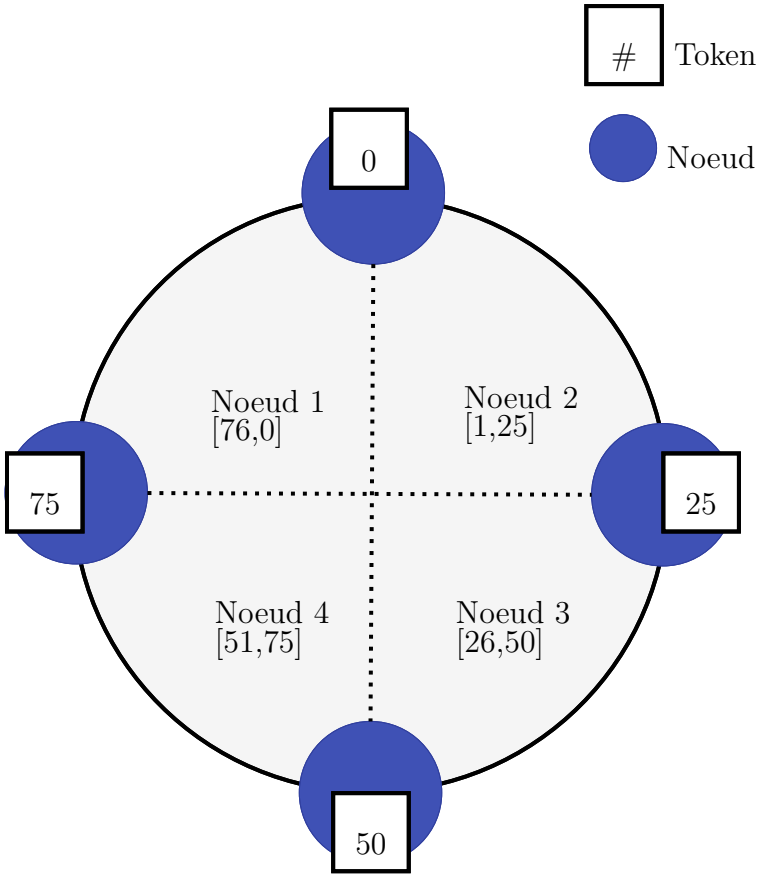


FIGURE 4 – Exemple de partitionnement des données dans une base de données distribuée

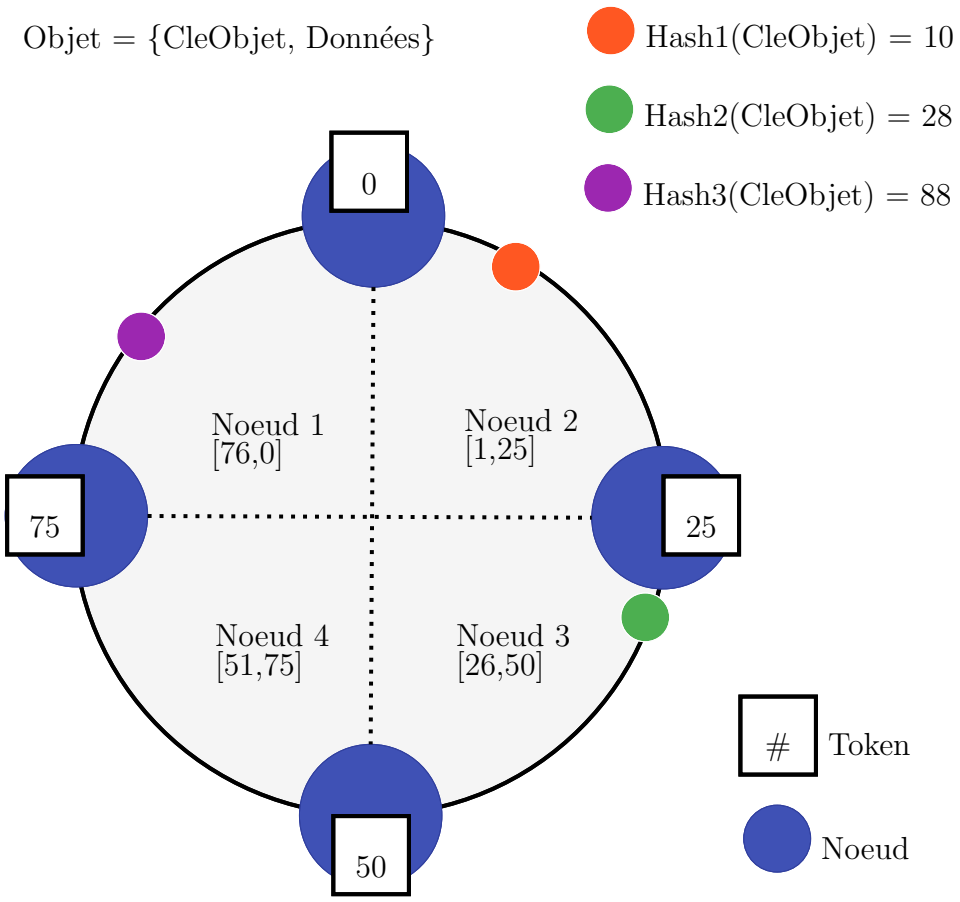


FIGURE 5 – Partitionnement des réplicas d'un objet avec une fonction de hachage pour chaque réplica

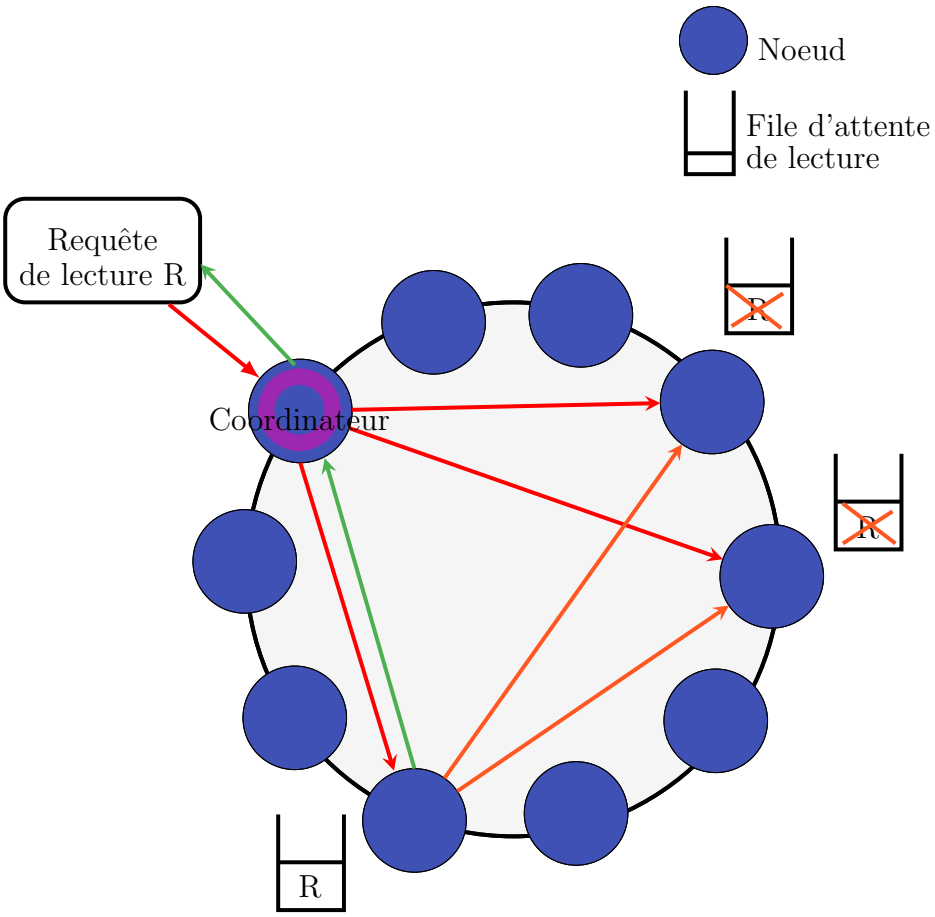


FIGURE 6 – Cheminement d'une requête de lecture dans une base de données distribuée avec la prise en charge de l'affectation (un seul noeud traite la requête)

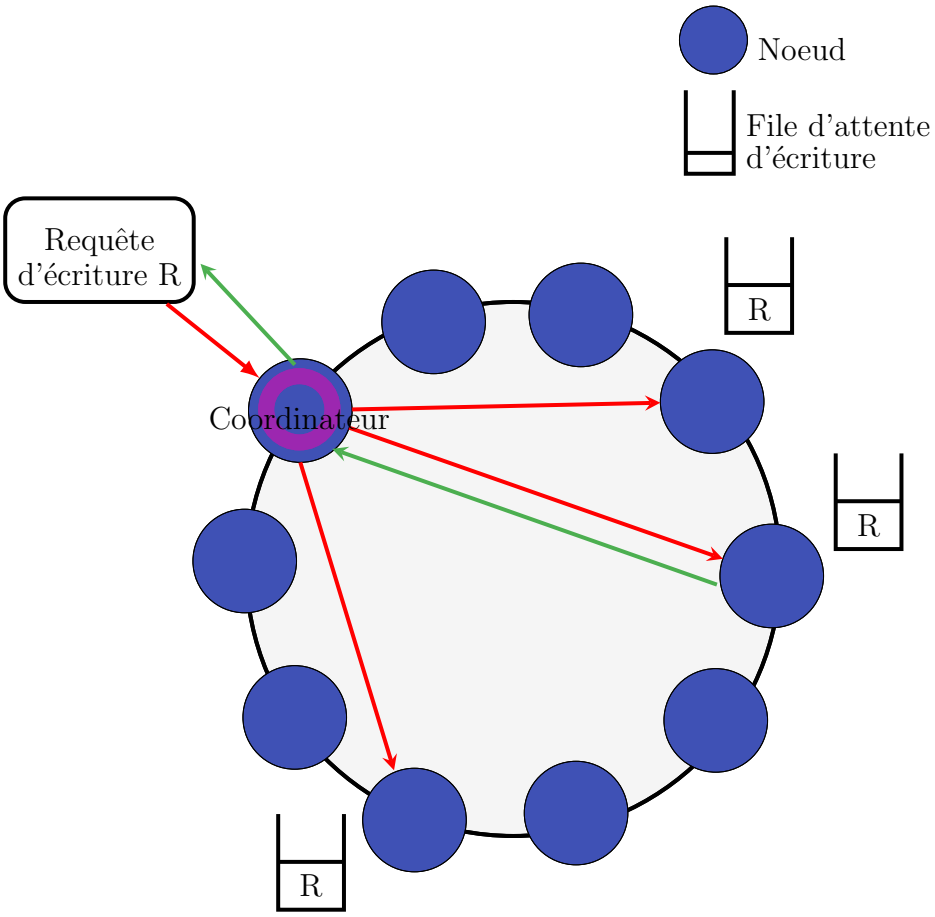


FIGURE 7 – Cheminement d'une requête d'écriture dans une base de données distribuée

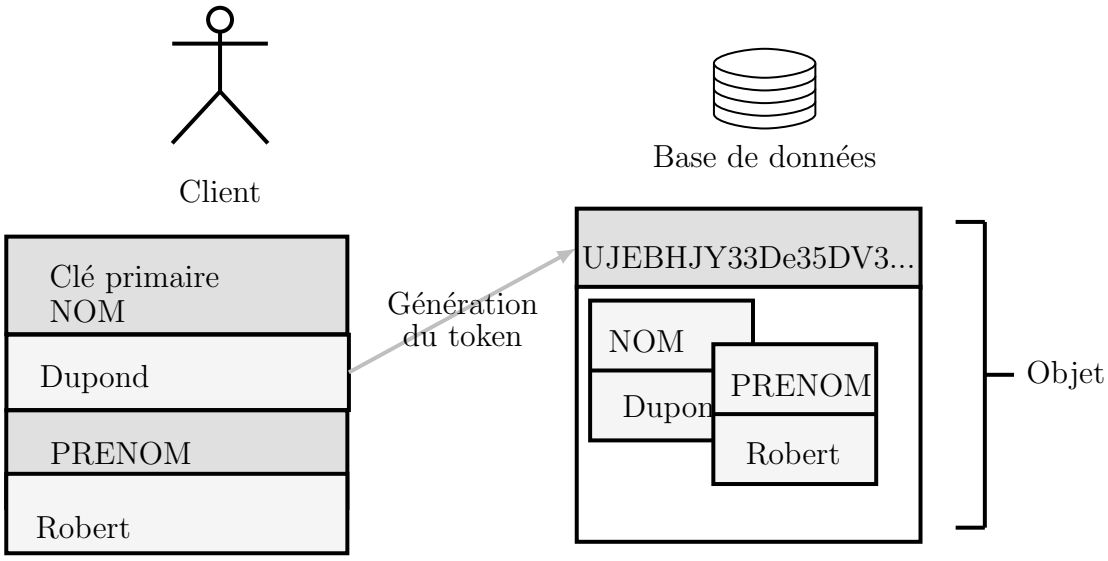


FIGURE 8 – Passage d'une répresentation des données pour le client à une représentation pour la base de données

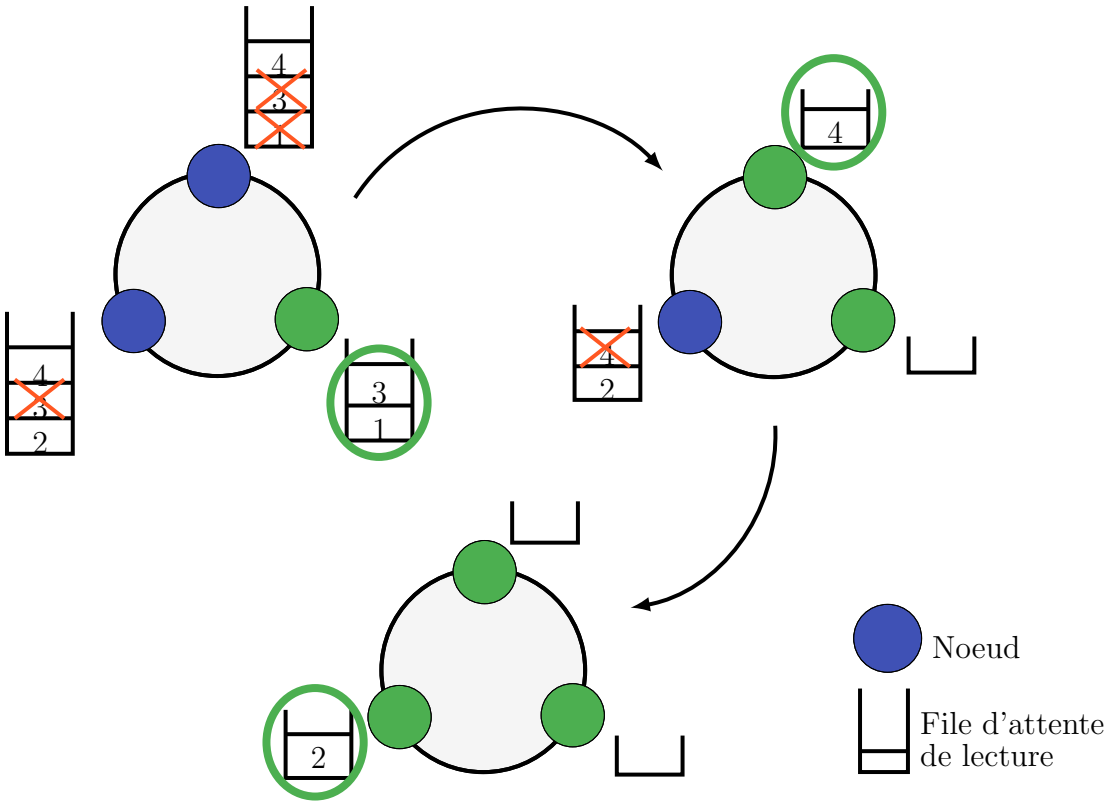


FIGURE 9 – Fonctionnement de l'algorithme de réaffectation des requêtes de lecture SLVO

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

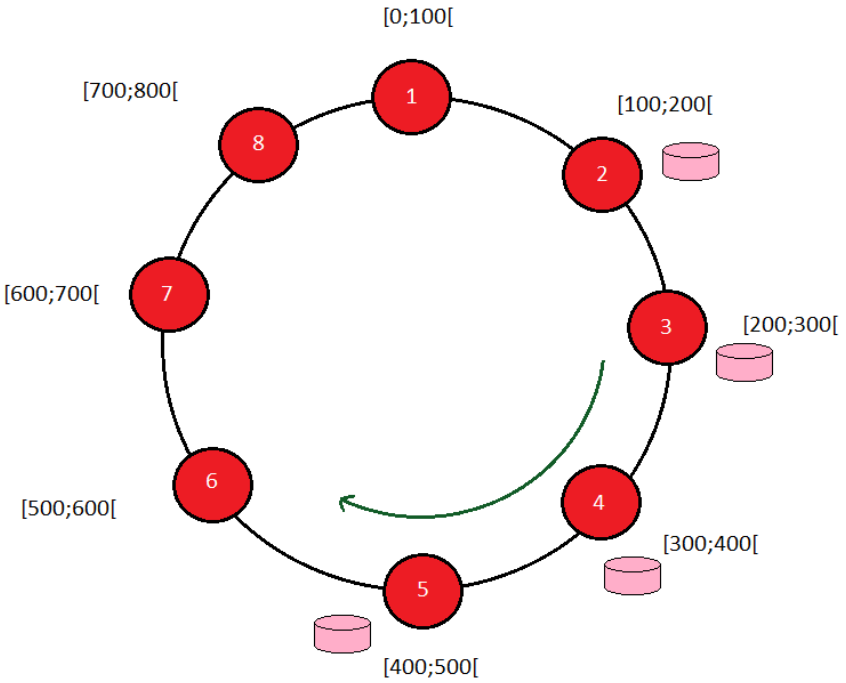


FIGURE 11 – Répartition des copies sur les noeuds dans une base de données distribuée

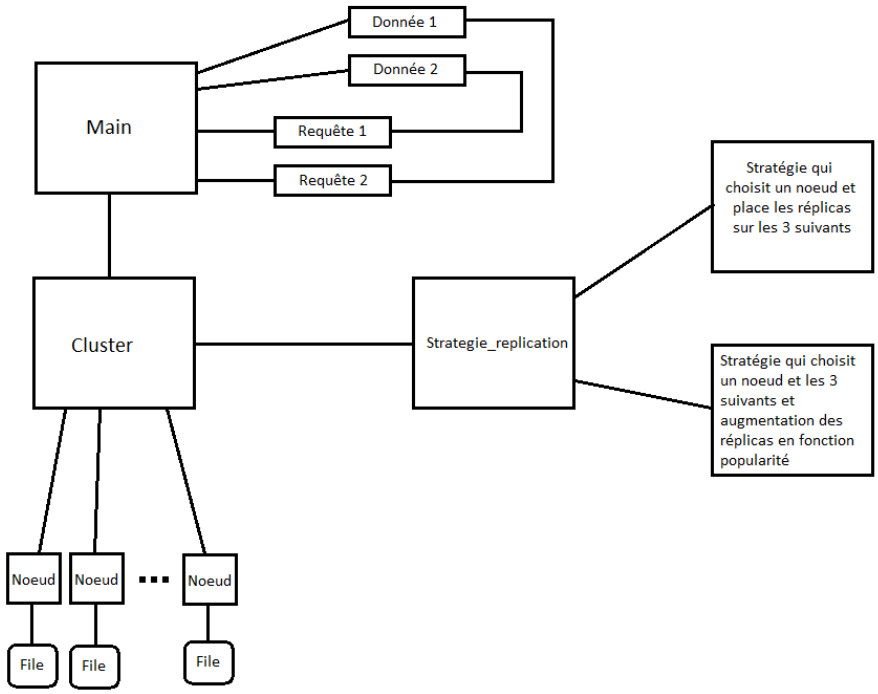


FIGURE 12 – Architecture des objets du simulateur