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

Identifications des besoins

Kevin Barreau Guillaume Marques Corentin Salingue

6 février 2015

Résumé

Dans une première partie, nous présenterons le projet, le contexte et les hypothèses. Ensuite, nous développerons les besoins fonctionnels et les besoins non-fonctionnels. De plus, nous dégageons une première version de la plannification du projet (GANTT). Enfin, nous présenterons les livrables.

Sommaire

1 Définition du projet

1.1 Contexte

Définition Un *environnement distribué* est constitué de plusieurs machines (ordinateurs généralement), appelées *noeuds*, sur lesquelles sont stockées des données et pouvant traiter des requêtes. Chaque noeud possède des informations locales propres à son fonctionnement (exemple : une liste des noeuds voisins).

Définition Une *base de données* est une entité permettant de stocker des données afin d'en faciliter l'exploitation (ajout, mise à jour, recherche de données).

Définition Une *donnée* ou un *objet* est un codage (une représentation sous forme binaire), propre au système de base de données, d'une information quelconque.

Définition Une *requête* est une interrogation d'une base de données afin de récupérer ou modifier des données.

Définition Une *charge* est associée à un noeud et désigne le nombre de requêtes restantes que le noeud doit traiter à l'instant T.

Définition La *réplication* d'une donnée est une action qui réalise des copies de cette donnée sur d'autres noeuds.

Définition Un *réseau* est un ensemble de noeuds qui sont reliés entre eux (par exemple par Internet) et qui communiquent ensemble.

L'expansion, au cours des deux dernières décennies, des réseaux et notamment d'Internet a engendré une importante création de données, massives par leur nombre et leur taille. Stocker ces informations sur un seul point de stockage (ordinateur par exemple) n'est bien sûr plus envisageable, que ce soit pour des raisons techniques ou pour des raisons de sûreté (pannes potentielles par exemple). Pour cela des systèmes de stockages dits distribués sont utilisés en pratique afin des les répartir sur différentes unités de stockages.

Pour répartir toutes ces données, notre client a développé de nouveaux algorithmes d'équilibrage de charge basés sur la réplication qu'il souhaite tester dans un environnement distribué.

1.2 Finalité

Nous devons développer une solution logicielle permettant de tester ces nouveaux algorithmes d'équilibrage de charge et de réplication proposés par le client dans un environnement distribué.

Définition Une *charge minimum* d'un environnement distribué est la plus petite charge trouvée sur un noeud parmi l'ensemble des noeuds. La *charge moyenne*, est la moyenne des charges de l'ensemble des noeuds.

Il s'agira d'implémenter les algorithmes développés par le client. On distingue les algorithmes d'affectation de requête :
— **SLVO** Si la charge du noeud est inférieure ou égale à la charge minimum, il s'affecte toutes les requêtes en attente et en avertit les autres noeuds.
— **AverageDegree** Si la charge du noeud est inférieure ou égale à la charge moyenne, il s'affecte toutes les requêtes en attente et en avertit les autres noeuds.

Définition La *popularité* d'un objet est le nombre de requêtes que va recevoir un objet durant un intervalle de temps T défini par l'utilisateur.

Ainsi que l'algorithme de gestion de copie, permettant d'établir le nombre de réplicas d'un objet en fonction de sa popularité.

Pour comparer l'efficacité de ces algorithmes, on peut visualiser l'état du réseau *en temps réel*.

1.3 Hypothèses

Nous évoluerons dans un environnement distribué constitué de n noeuds de stockage dans lequel on souhaite stocker m objets. C'est un réseau statique : on ne peut pas ajouter ou supprimer de noeuds après création du réseau.

Définition Un *message* est un envoi d'information d'un noeud vers un autre noeud pour mettre à jour ses données locales ou effectuer des actions particulières (autre que des requêtes, comme par exemple, "donne à tel noeud ta charge actuelle").

Données locales d'un noeud Un noeud contient les données locales suivantes :
— la charge de tous les noeuds du réseau
— la popularité de chaque objet stocké sur ce noeud
— une file d'attente de message à traiter
— la requête en cours de traitement

Requêtes Nous supposerons que les requêtes seront effectuées en un temps fixe.

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 et de leur criticité.

2.1 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	Description
1	Priorité haute	A implémenter en premier.
2	Priorité moyenne	A implémenter.
3	Priorité faible	A implémenter (en fonction du temps restant).

2.2 Criticité

Le niveau de criticité d'un besoin est un indicateur de l'impact qu'aura la non-implémentation de ce besoin sur le bon fonctionnement de l'application.

Valeur	Signification	Description
1	Criticité extrême	L'application ne sera pas utilisable par le client.
2	Criticité haute	L'application est utilisable par le client. En revanche, certaines fonctionnalités de l'application ne seront pas utilisables.
3	Criticité moyenne	L'application est utilisable par le client. En revanche, certaines fonctionnalités de l'application n'améneront pas au résultat attendu.
4	Criticité faible	L'application peut fonctionner sans l'ajout de ces fonctionnalités

3 Besoins fonctionnels

3.1 Gestion d'un réseau

Un réseau est un ensemble de noeuds qui sont reliés entre eux (par exemple par Internet) et qui communiquent ensemble afin de traiter toutes les requêtes reçues.

3.1.1 Gestion des noeuds

Création d'un noeud (*Priorité* : 1, *criticité* : 1) Il est possible de séparer ce besoin en plusieurs sous-besoins :
— créer un noeud dans l'environnement
— initialiser les données locales d'un noeud

Mise à jour des données locales (*Priorité* : 1, *criticité* : 1) Afin de connaître l'état du réseau de manière précise, les données locales doivent être mise à jour à chaque action, c'est à dire lors du traitement d'un message dans la file d'attente.

Communication des données locales (*Priorité* : 1, *criticité* : 1) Un noeud doit être capable de communiquer ses données locales à d'autres noeuds du réseau.

Récupération de l'état du réseau (*Priorité* : 1, *criticité* : 1) L'application doit permettre la description de l'état du réseau. On souhaite connaître :
— le nombre de requêtes en attente
— la popularité des objets

3.1.2 Réplication d'un objet

Définition Une *fonction de hachage* est une fonction mathématique déterministe (c'est à dire, si on lui donne la même entrée, elle renvoie la même sortie). Nous définissons ses entrées et sorties dans le paragraphe suivant.

Définition Un *token* permet comme une étiquette sur un produit, de désigner une donnée.

Il s'agit de copier un objet sur un autre noeud. Il est possible de définir le nombre de copies d'un objet au sein d'un ensemble de noeuds, appelé *data center*. Pour savoir quel noeud stocke l'objet, on utilise une fonction de hachage dans laquelle on fait passer la clé de l'objet (la clé de l'objet est une donnée permettant d'identifier un objet de manière unique). On obtient ainsi un *token*.

Tous les noeuds possèdent un intervalle (ou ensemble) de tokens dont ils sont responsables. On regarde le token de l'objet pour savoir quel noeud va le prendre en charge (voir la figure 2).

Une stratégie de réplication est la méthode qui permet de placer les copies d'un objet dans un data center. La stratégie consiste à utiliser une fonction de hachage différente pour chaque copie (voir la figure 3). Le numéro de la copie définit la fonction à utiliser. Ainsi, sur le schéma, la deuxième copie de tous les objets utilisera la fonction de hachage `Hash2` pour obtenir un token et placer la copie.

Définition des fonctions de hachage (*Priorité* : 1, *criticité* : 3)

Mise en place de la stratégie de réplication (*Priorité* : 1, *criticité* : 3)

3.1.3 Popularité d'un objet

Les algorithmes à implémenter nécessitent de connaître la popularité d'un objet dans le réseau. La popularité d'un objet est défini par le nombre de requêtes sur cet objet. Plus le nombre de requêtes est grand, plus l'objet est populaire.

Calcul de la popularité (*Priorité* : 1, *criticité* : 1)

Stockage de la popularité (*Priorité* : 1, *criticité* : 1) Chaque noeud stocke la popularité des objets qu'il contient.

Communication de la popularité (*Priorité* : 1, *criticité* : 1) Un noeud stockant des copies d'un objet doit communiquer la popularité de ces derniers au noeud possédant l'objet original.

3.2 Protocoles d'affectation

Une *affectation* consiste, pour un noeud, à effectuer le traitement d'une requête. Les requêtes peuvent arriver sur n'importe quel noeud. On dit alors que ce noeud devient le noeud *coordinateur* pour cette requête. Il transmet la requête aux noeuds possédant l'objet de la requête (voir la figure 4).

Les noeuds possédant l'objet mettent la requête dans leur file d'attente. Dès qu'un noeud aura à traiter cette requête, il communique aux autres noeuds possédant l'objet qu'il se charge de la traiter. Les noeuds suppriment la requête de leur file d'attente. Le noeud qui prend en charge la requête la traite et donne le résultat de la requête au noeud coordinateur, qui peut ainsi renvoyer le résultat.

Communication d'un message d'ajout d'une requête dans la file d'attente (*Priorité* : 1, *criticité* : 1)

Communication d'un message de suppression d'une requête de la file d'attente (*Priorité* : 1, *criticité* : 1)

Ajout d'une requête dans la file d'attente (*Priorité* : 1, *criticité* : 1)

Suppression d'une requête de la file d'attente (*Priorité* : 1, *criticité* : 1)

Des protocoles (équivalent algorithmes) d'affectation plus spécifiques peuvent être implémentés. Ils utilisent les données locales du noeud pour décider de l'affectation des requêtes.

Implémentation de l'algorithme SLVO (*Priorité* : 1, *criticité* : 1)

Implémentation de l'algorithme AverageDegree (*Priorité* : 1, *criticité* : 1)

3.3 Requêtes

3.3.1 Générations 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.

3.3.2 Importation d'un jeu de requête

Définition La notion d'efficacité est laissée à l'appréciation du client. Une brève approche serait de comparer les temps d'exécution.

Pour comparer l'efficacité des algorithmes, il doit être possible d'envoyer sur le réseau une même suite de requêtes : un jeu de requêtes.

Importation (*Priorité* : 1, *criticité* : 2) L'application doit pouvoir lire un fichier contenant une suite de requêtes et envoyer ces requêtes sur le réseau.

3.4 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.

3.4.1 Enregistrement des données

Ecriture dans un fichier (*Priorité* : 1, *criticité* : 2) 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;`

3.4.2 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 (*Priorité* : 3, *criticité* : 2) L'application doit permettre la représentation de chaque noeud par un sommet.

Analyse syntaxique (*Priorité* : 2, *criticité* : 1) 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 (*Priorité* : 3, *criticité* : 3) 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 (*Priorité* : 3, *criticité* : 3) 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 Gestion d'un réseau

4.3.1 Communication entre noeuds

Algorithme Le calcul de la popularité nécessite l'implémentation de l'algorithme d'approximation Space-Saving Algorithm [ADA05].

Pour connaître l'état du réseau, il faut regrouper les données locales des noeuds. Nous cherchons donc à récupérer ces données en un temps raisonnable ($O(log(n))$ pour n noeuds).

Pour cela, nous nous appuierons sur le protocole `Gossip` [Fou14a]. Périodiquement, chaque noeud choisi n noeuds aléatoirement dont un noeud *seed*[Fou14b], noeud en mesure d'avoir une connaissance globale du système, et il communique à ces noeuds ses statistiques (valeur de sa charge, objets les plus populaires...).

Ainsi, la connaissance globale du système se fait, dans la théorie, en $O(log(n))$.

4.3.2 Taille des données

La taille de chaque donnée est laissée à l'appréciation de l'équipe. Néanmoins, celle-ci doit être suffisamment importante, afin de permettre de créer des requêtes qui "stressent" le système pour avoir des résultats cohérents (sur la base de l'hypothèse : chaque requête prend un même temps à être traitée).

4.4 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.

4.5 Visualisation des données

4.5.1 Etat du réseau

La vue permet de montrer l'état du réseau.

Le réseau est représenté par un graphe, les machines par des noeuds. Pour chaque machine, les données affichées sont la charge ainsi que le contenu de la file d'attente.

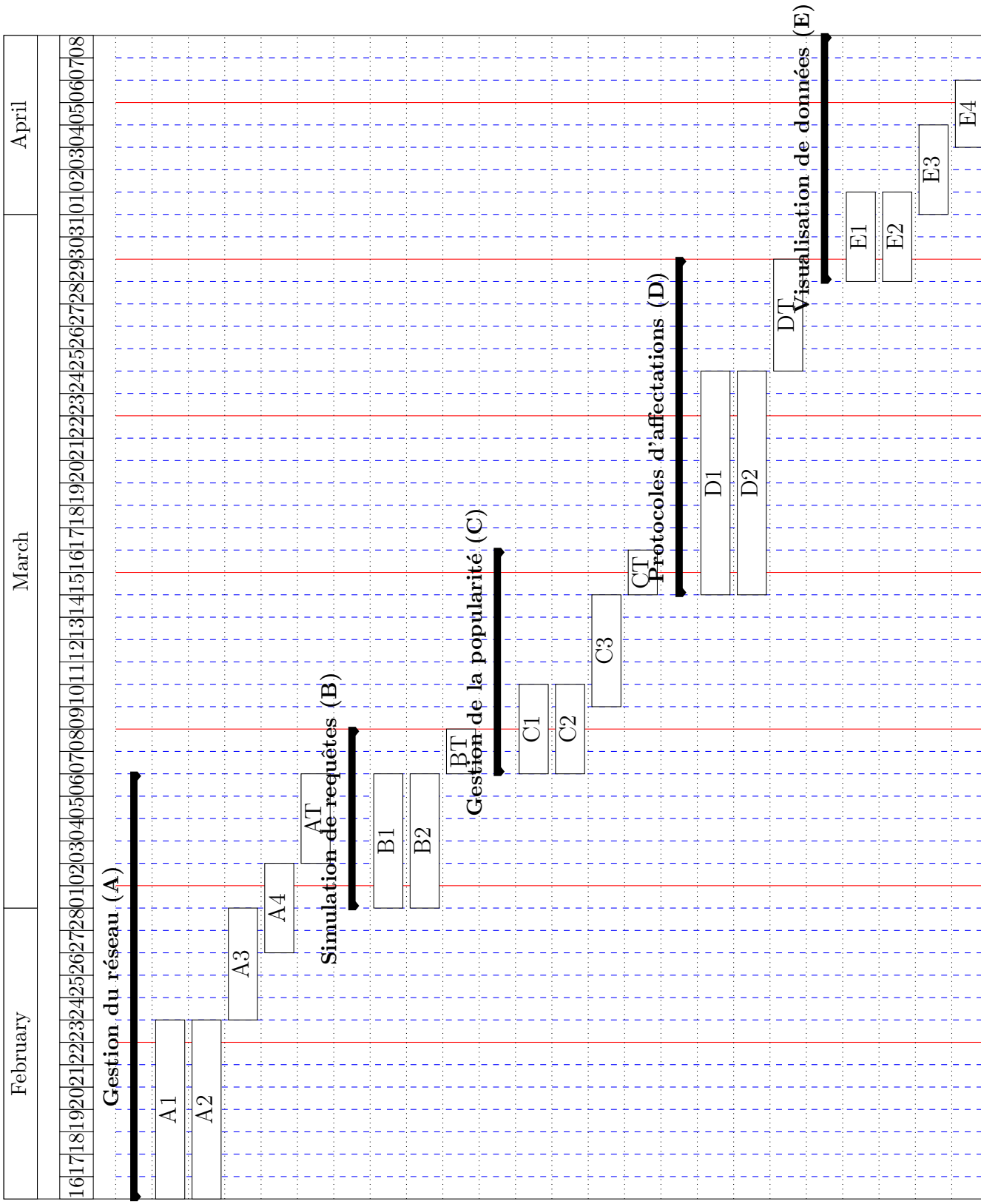
4.5.2 Actualisation de la vue

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 Répartitions des tâches

5.1 Diagramme de Gantt



5.2 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

Remarque Il s'agit d'une première version de notre GANTT. Nous n'avons pas encore défini l'affectation des tâches aux développeurs.

6 Livrables

6.1 Livrables intermédiaires

Un livrable intermédiaire est une ébauche de l'application. C'est à dire que seulement quelques fonctionnalités sont implémentées.

Il n'a pas encore été décidé de remettre un ou plusieurs livrables intermédiaires au client.

6.2 Livrable final

Il devra être remis le 8 Avril 2015. Il comportera les besoins de priorité 1 et 2.

Références

[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].

[Fou14a] The Apache Software Foundation. Architecturegossip - cassandra wiki. <`http://wiki.apache.org/cassandra/ArchitectureGossip`>, 2014. [Accessed 21 January 2015].

[Fou14b] The Apache Software Foundation. Faq cassandra wiki. <`http://wiki.apache.org/cassandra/FAQ#seed`>, 2014. [Accessed 21 January 2015].

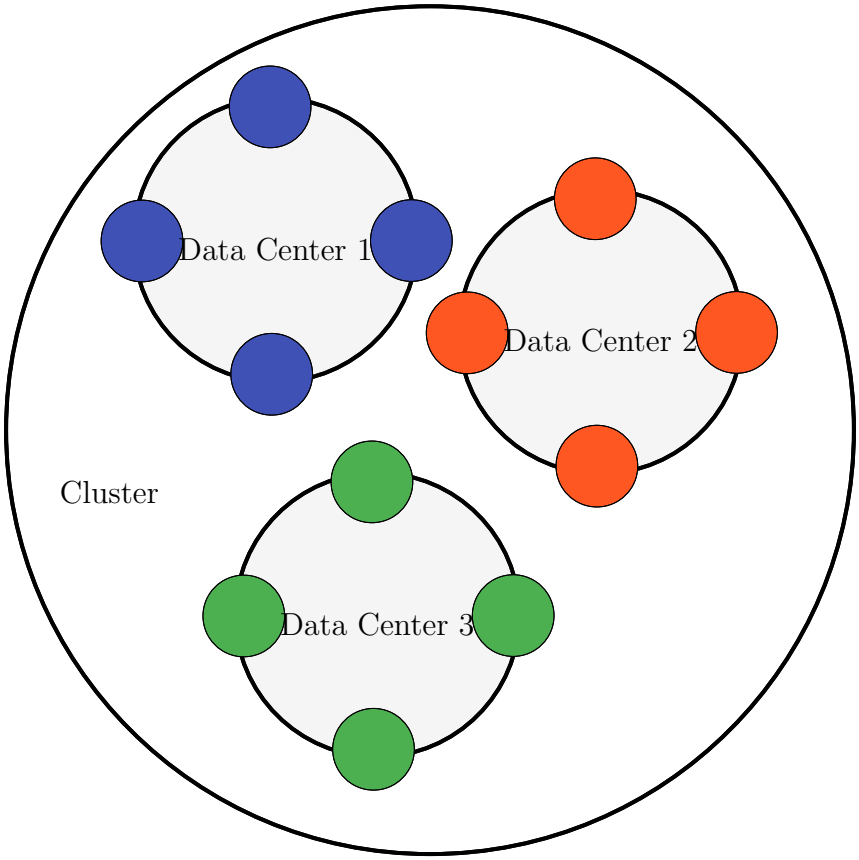


FIGURE 1 – Visualisation d'une base de données distribuée sous forme de clusters possédant quatre data center

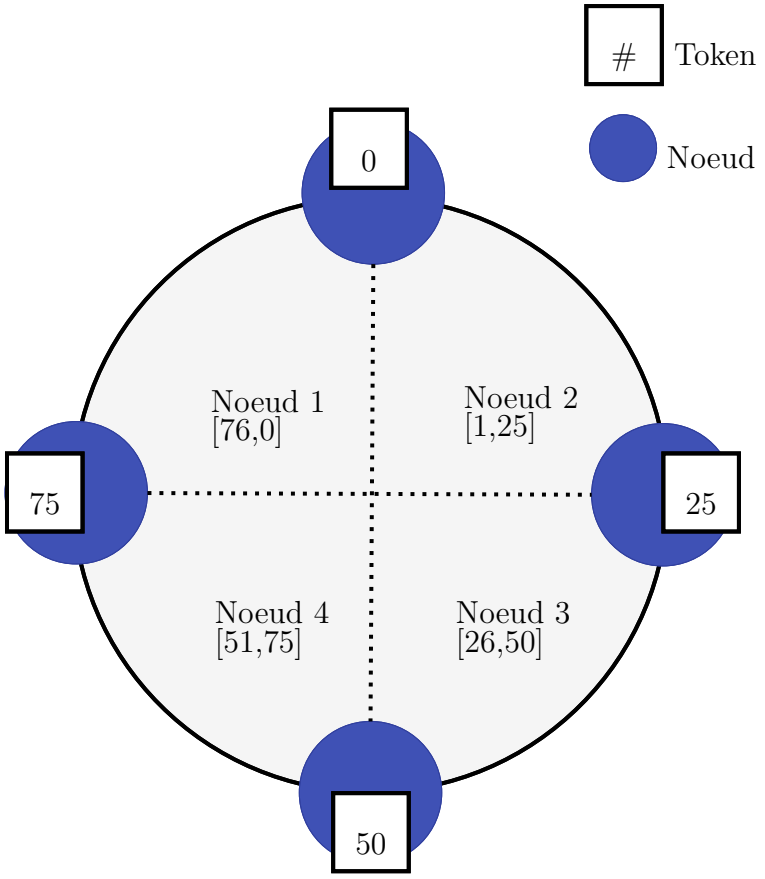


FIGURE 2 – Exemple de partitionnement des données dans une base de données distribuée

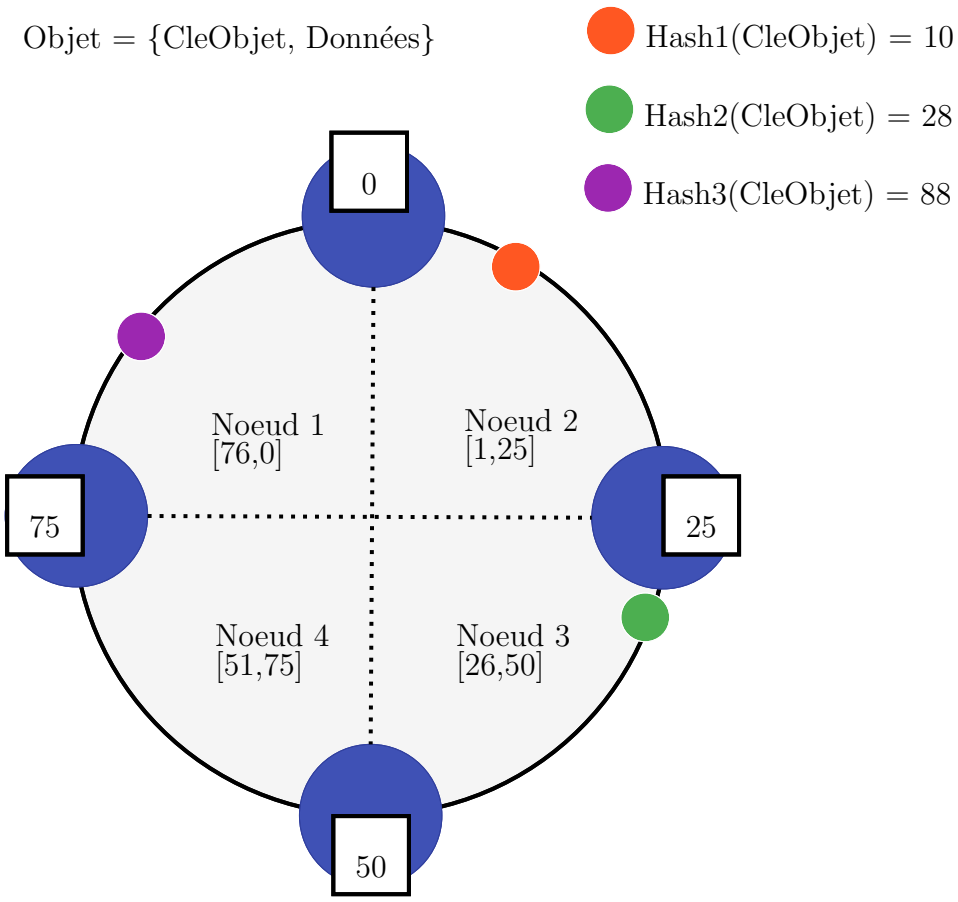


FIGURE 3 – Partitionnement des réplicas d'un objet avec une fonction de hachage pour chaque réplica

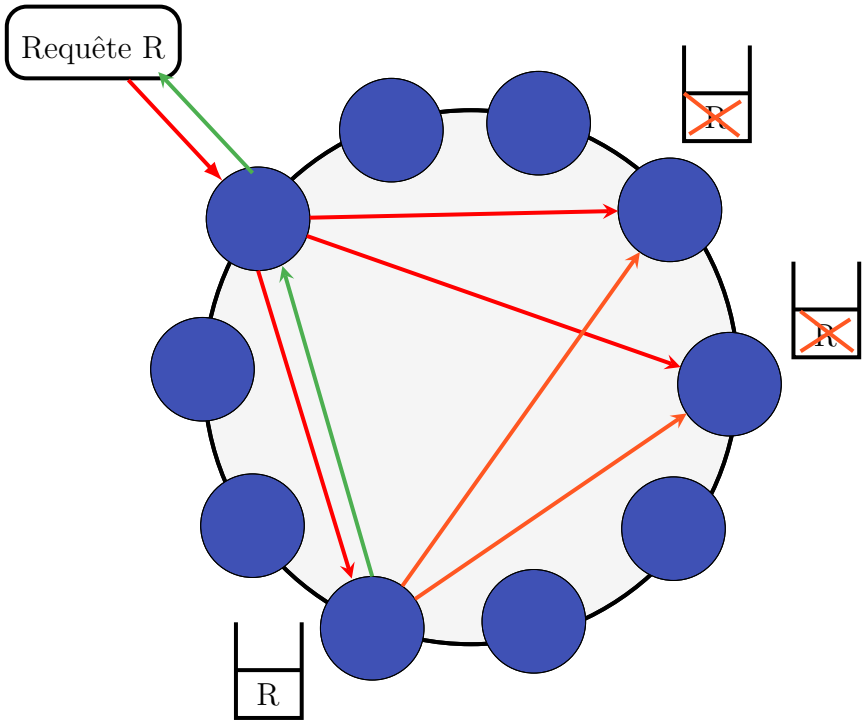


FIGURE 4 – Cheminement d'une requête dans une base de données distribuée avec la prise en charge de l'affectation (un seul noeud traite la requête)