Développement d'une méthode de distribution de métadonnée équilibrée pour un flux de requêtes exascale

Eloïse Billa

Thèse de doctorat

NNT : 2020UPASG044

université PARIS-SACLAY

Développement d'une méthode de distribution de métadonnées équilibrée pour un flux de requêtes exascale

Development of a well-balanced metadata distribution method for an exascale requests flow

Thèse de doctorat de l'université Paris-Saclay

École doctorale n° 580, Sciences et technologies de l'information et de la communication (STIC)
Spécialité de doctorat : Informatique
Unité de recherche : Université Paris-Saclay, UVSQ, LI PARAD, 78180, Saint-Quentin en Yvelines, France.)
Référent : Université de Versailles-Saint-Quentin-en-Yvelines.

Thèse présentée et soutenue à Paris-Saclay, le 11 décembre 2020 par

Éloïse BILLA

Composition du jury :

William Jalby Professeur des universités, Université de Versailles	Président
Hamamache Kheddouci Professeur des universités, Université de Lyon	Rapporteur et examinateur
Michael Krajecki Professeur des universités, Université de Reims	Rapporteur
Sebastien Gougeaud Ingenieur de recherche, CEA-DIF	Examinateur
Zineb Habbas Professeure des universités, Université de Lorraine	Examinatrice
Tarek Menouer Ingenieur de recherche, Umanis	Examinateur

Direction de thèse :

Soraya Zertal MCFHC, Université de Versailles	Directrice de thèse
Philippe Deniel Ingenieur de recherche, CEA-DIF	Co-encadrant

Remerciements

Cette thèse n'a pas été un long fleuve tranquille, mais elle a pu atteindre l'océan et c'est le plus important. Je tiens à remercier ceux qui ont permis à ce fleuve de s'écouler.

Tout d'abord, je remercie les membres du jury, qui ont pris le temps de s'intéresser à mon travail, l'UVSQ avec Soraya Zertal ainsi que le CEA, avec Philippe Deniel et Thomas Leibovici pour leur encadrement.

Plus personnellement, je remercie de tout cœur Franck Ledoux, qui m'a aidé par pure gentillesse pendant cette dernière année qui fut la plus difficile pour moi, donnant toujours de son temps même lorsqu'il n'en avait pas. Ses conseils furent de nombreuses fois une bouée m'empêchant de chavirer.

J'adresse aussi mes remerciements à l'école doctorale STIC représentée par Alain Denise, Leila Kloul et tout particulièrement Hanna Klaudel pour leur soutien et leur bienveillance vis-à-vis de ma thèse.

Viennent ensuite les remerciements pour mes collègues au CEA pour leurs conseils et leur gentillesse, en particulier Patrice Lucas, Quentin Bouget et Alexandre Pagnat, mais aussi pour l'université d'Évry et ses professeurs qui m'ont donné le goût de la recherche et toute l'équipe d'AS+ pour leur expertise technique et leurs barbecues.

Je n'oublie, bien évidemment pas les terateckiens qui n'ont jamais manqué d'ensoleiller mes journées de travail : les anciens tels que Gautier, Rémi, Sébastien, Hugo (les deux), Arthur et Hoby, mais aussi les actuels tels que Nestor, Anton, Alexiane, Florian et Simon (un magnifique couple je l'avoue ;p) ainsi que Brigitte et sa bonne humeur légendaire. Un remerciement particulier doit être fait à Théo qui s'est retrouvé dans la même galère que moi : chacun à évaluer la gravité de ses problèmes par rapport à ceux de l'autre. Je suis heureuse d'avoir pu avancer avec lui. Un autre remerciement tout particulier revient à Christina, toujours bienveillante et fournisseuse officielle de bonbons et gâteaux en tous genres : une amie que j'espère garder longtemps.

Pour finir, je voudrais de tout mon cœur remercie ma famille : parents, frères, sœur et leurs conjoints et enfants, ainsi que les amis pour simplement avoir été là pour moi. Mes derniers remerciements vont à Adrien : en plus de m'avoir supporté pendant ces 4 ans, il a su m'aider professionnellement et psychologiquement. Sans lui, tout cela aurait eu une fin bien plus morose. Même si aucun remerciement n'est à la hauteur de ma gratitude et ma reconnaissance : merci.

Table des matières

Introduction

Le calcul haute performance (*High Performance Computing* ou HPC) est un domaine qui cherche à proposer des supercalculateurs de plus en plus puissants, avec de meilleures puissances de calcul et/ou des solutions de stockages plus rapides et de plus grande capacités. Les calculs lancés dans le domaine du HPC correspondent à des domaines d'application variés, comme le traitement d'images, les études météorologiques ou encore l'intelligence artificielle. Les logiciels Abinit [GAA+20], cp2k [HISV14], gromacs [AMS+15] ou tensorflow [AAB+15] sont des exemples de codes de simulations classiques exécutés sur des supercalculateurs. Pour obtenir le plus haut niveau de précision possible, ces simulations génèrent de plus en plus de demandes d'accès à un nombre de données lui aussi grandissant. Il est donc nécessaire de concevoir des systèmes de stockage de grande capacité, tout en égalant ou améliorant les temps d'accès à ces données.

Une des caractéristiques importantes d'un système de stockage est sa capacité d'extension, qui correspond à sa faculté à ajouter des serveurs, ou encore son comportement en cas d'accès concurrents. La caractéristique *stateless* ou *stateful* d'un système indique si l'exécution d'une action est dépendante de l'état du système : un système *stateful*, comme un système de fichiers *Lustre* [Bra19], garde pour chaque donnée un état qui rend déterministe le prochain accès. Cet état est mis à jour à chaque accès et est nécessaire pour le suivant, limitant ainsi les opérations concurrentes. À l'inverse, dans un système *stateless*, comme un serveur *HTTP* [FGM+99], chaque accès est indépendant de l'état du système, ce qui facilite les accès concurrents.

Un système de stockage fournit aussi certaines garanties, les trois principales sont les suivantes :

- le niveau de *cohérence* des données, indiquant la visibilité de l'effet d'une opération. Il y a deux types de cohérence : la cohérence forte qui indique que l'effet d'une action est visible lors de l'action suivante et la cohérence faible qui autorise un retard de visibilité de l'effet d'une action.

- la *disponibilité* d'une donnée, définissant les moments où il est possible d'accéder à une donnée. Afin d'assurer une meilleure disponibilité des données, il est possible de répliquer les données et de stocker chaque réplica sur un serveur différent. Il est alors nécessaire d'intégrer un système de gestion de la cohérence afin de maintenir tous les réplicas à jour.
- la *tolérance* aux partitionnements, autrement dit la capacité du système à maintenir son fonctionnement lorsque le réseau est divisé et isole certains serveurs.

La dernière caractéristique importante d'un système de stockage évoquée ici correspond au compromis entre les trois principales garanties d'un système, comme l'indique le théorème *Consistency Availability Partition tolerance* (CAP) [FLP85, Bre12], présenté par la figure 1. Ce théorème annonce l'impossibilité de disposer de ces trois caractéristiques au même moment, il est donc nécessaire pour chaque système de réduire, quand cela est nécessaire, le niveau d'au moins une des trois caractéristiques. Un système toujours disponible et ayant une cohérence forte ne pourra fonctionner de cette manière en cas de partitionnement du réseau ; un système toujours disponible et résistant aux partitionnements réseau ne pourra garantir une cohérence forte pour ses données ; et finalement, un système ayant une cohérence forte et résistant aux partitionnements réseau ne permettra pas d'avoir des données toujours disponibles.

Afin d'établir un état des lieux objectif des ordres de grandeurs des systèmes HPC actuels, nous nous sommes basés d'une part sur les machines de calcul disponibles au CEA [CEA15, CEA16, CEA19], et d'autre part, sur les systèmes HPC les plus performants (en terme de puissance de calcul et en terme de puissance de stockage) en 2018 [KMBL18, DMS+18] comme le supercalculateur *Summit* [VdSB+18, TL18] mis en production au Laboratoire national d'Oak Ridge en 2018. Ces supercalculateurs nous ont permis de constater et déduire les chiffres suivants :

Les systèmes de stockage actuels sont des systèmes dits pétaflopiques, c'est-à-dire délivrant des dizaines voire des centaines de pétaFLOPS [1] (10^{16-17}), et cette puissance de calcul induit des systèmes de stockage correspondant aux ordres de grandeurs suivants :

- Plusieurs dizaines de milliers de clients (10^4) peuvent se connecter en concurrence afin d'accéder à leurs données. Cela résulte en un débit agrégé d'un Teraoctet (10^{12}) par seconde pour les données.

1. floating-point operations per second.

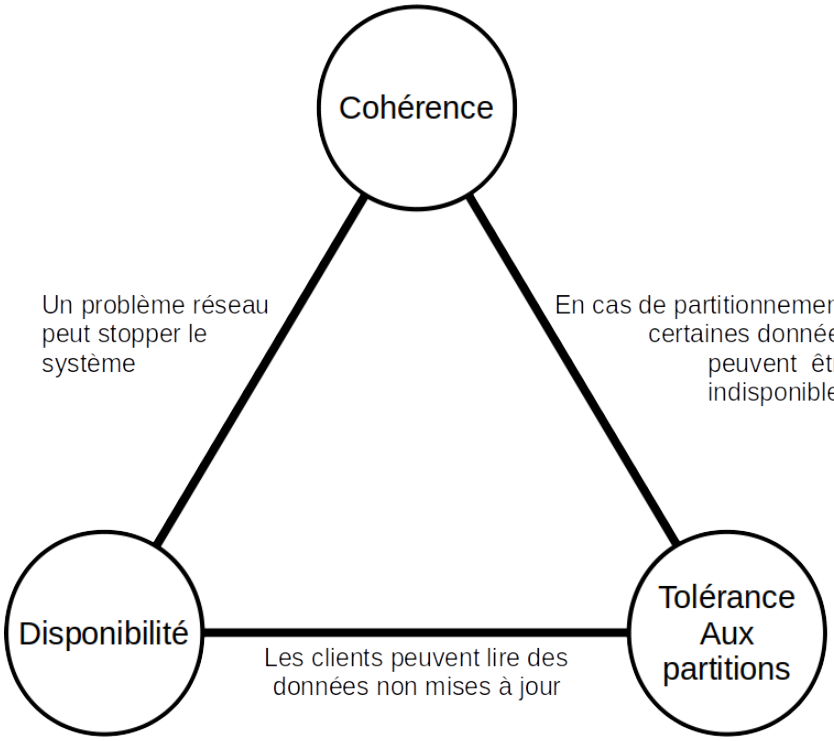


FIGURE 1: Illustration du théorème CAP.

- Le volume de données produites dans ces systèmes HPC dépasse la centaine de pétaoctets (10^{17}) et génère plusieurs centaines de millions de métadonnées associées, voire des milliards (10^{8-9}).
- Afin de stocker ce volume de données, plusieurs dizaines de serveurs de données sont disponibles. Pour les métadonnées, généralement un serveur de métadonnées pour les données froides (peu accédées) et quelques-uns ($2-4$) pour les données chaudes (fréquemment accédées), sont mis à disposition. Cela permet un flux de données de l'ordre du mégaIOPS [2] (10^6) et un flux de métadonnées de l'ordre de la dizaine de kiloIOPS (10^4).

Prochainement, il sera possible d'avoir des systèmes atteignant l'exaFLOPS (10^{18}). Plusieurs grands projets émergent partout dans le monde pour l'obtention de ces machines exaflopiques : le projet américain *Exascale Computing Project* [Mes17] prévoit leurs premières machines exaflopiques pour 2021, tout comme la Chine et sa machine exaflopique *Tianhe-3* [WAN19], tandis que le projet européen EUROHPC [Sko19] vise 2023.

2. input/output operations per second.

Cependant, la puissance de calcul n'est pas le seul paramètre à prendre en compte dans l'élaboration d'un supercalculateur exaflopique. Afin d'utiliser une puissance de calcul exaflopique sans être limité par les accès aux données, le système de stockage associé doit permettre les ordres de grandeur suivants :

- Plusieurs centaines de milliers voire des millions de clients (10^{5-6}) pourront se connecter en concurrence afin d'accéder à leurs données. Cela résulterait en un débit agrégé de plusieurs dizaines de Teraoctets (10^{13}) par seconde pour les données.
- Le volume de données produites pour ces systèmes HPC serait de l'ordre de la dizaine d'exaoctets [3] (10^{19}) et génèrerait des billions de métadonnées associées (10^{12}).
- Afin de stocker ce volume de données, plusieurs centaines de serveurs de données seraient nécessaires, ainsi que plusieurs dizaines de serveurs pour les métadonnées. Cela permettrait un flux de données de l'ordre de la centaine de mégaIOPS (10^8) et un flux de métadonnées de l'ordre de la dizaine de mégaIOPS (10^7).

Avec ces nouveaux systèmes de stockage, on constate un accroissement fort du nombre de clients et de données, ce qui génère un flux de requêtes très soutenu pour les métadonnées. La gestion de la cohérence des métadonnées est une problématique essentielle de ce passage à l'échelle : d'un côté, une cohérence forte est demandée, limitant la parallélisation des accès aux métadonnées, et de l'autre côté, beaucoup de clients accèdent à ces métadonnées et les temps d'accès doivent être les plus courts possibles, ce qui demande de paralléliser au maximum ces mêmes accès.

À la vue de ces ordres de grandeurs, les systèmes de fichiers actuels ne semblent pas en capacité de supporter les flux de requêtes futurs [GVM00, RFB11]. En effet, ceux-ci sont soumis à la norme POSIX qui impose une sémantique très stricte ainsi qu'une gestion des données en arborescence. Toutes ces contraintes rendent le passage des systèmes de type POSIX [IEE88] à l'échelle exaflopique complexe voire impossible. Ainsi, de nombreuses solutions [WBM+06, DHJ+15, MCA+16] s'orientent vers l'utilisation du stockage objet, qui permet, entre autres, un adressage plat des données et une sémantique d'accès simple. Le stockage objet offre de nouvelles possibilités telles que la séparation des flux de données et de métadonnées, ou encore un service de métadonnées distribué. Cependant, chaque innovation apporte aussi de nouvelles problématiques : utiliser un cluster de serveurs dédiés

3. ce qui est l'équivalent du trafic global d'internet sur *smartphone* pendant un mois en 2017 [Cis19].

aux métadonnées pose la question de la distribution des requêtes parmi tous ces serveurs. Un serveur recevant une proportion des requêtes trop importante peut se retrouver surchargé et ne sera pas en mesure de répondre rapidement à toutes ses requêtes. Nous basons notre étude sur les services de métadonnées, ainsi, lorsque nous parlons de serveurs, nous omettons de préciser qu'il est dédié aux métadonnées.

Afin de gérer la répartition des requêtes sur les différents serveurs, on utilise des méthodes de distribution de l'espace de nommage, associant à chaque métadonnées possible un serveur responsable de tous les accès et modifications pour cette métadonnée. Ces méthodes définissent le schéma de répartition des requêtes sur les différents serveurs mais aussi la méthode d'accès d'un client à la métadonnée demandée. De nombreuses méthodes de distribution d'un espace de nommage ont déjà été proposées. Bien que certaines soient adaptées aux systèmes de type fichier, ou d'autres propres à la distribution de données, elles restent applicables aux systèmes objets sans en exploiter au maximum les capacités.

Il existe toutefois des méthodes de distribution de métadonnées propres aux systèmes de stockage objet, mais celles-ci ne répondent pas forcément aux contraintes d'un important flux de requêtes comme nous l'avons décrit. En effet, les flux de requêtes associés à ce genre de systèmes ont tendance à être inconstants de par l'utilisation du supercalculateur par un grand nombre d'utilisateurs différents, et peuvent générer des moments de forts déséquilibres de charge entre les serveurs. Ces flux seront gérés différemment selon la méthode de distribution choisie. En effet, une méthode de distribution peut être efficace avec un flux de requêtes caractéristique d'une utilisation et être mise à défaut si on la soumet à un autre type de flux de requêtes. Une mauvaise distribution de l'espace de nommage amène alors des points de contention avec des serveurs de métadonnées surchargés et provoque des ralentissements du système dans sa globalité.

De plus, la comparaison entre ces différentes méthodes est complexe. En effet, chaque méthode a été conçue pour une architecture particulière et un flux de requêtes précis. A notre connaissance, aucun standard de description des méthodes de redistribution n'est défini et les implémentations de ces méthodes ne sont pas disponibles à la communauté scientifique du domaine. Il est donc relativent difficile de reproduire et comparer ces méthodes. Par ailleurs, il n'existe pas d'outil à code source ouvert permettant d'implémenter et de comparer ces méthodes dans un contexte homogène.

L'objectif de ce travail a été de développer une méthode de distribution de l'espace de nommage pour les métadonnées capable d'assurer une répartition équilibrée des requêtes en étant soumis à un flux de requêtes *HPC exascale*. Pour cela,

nous nous intéressons aux métadonnées et à leur serveurs dédiés.

Le travail réalisé s'est articulée suivant deux axes :

- Le développement d'une méthode de distribution nommée *Distribution adaptative à la charge* ou *LAD* pour *Load-adaptive Distribution*, ainsi que sa variante avec fenêtre temporelle *LAD-TW* pour *Load-adaptive Distribution with Temporal Window*, toutes deux capables d'assurer une répartition équilibrée des requêtes,
- La conception d'un outil nommé *MeDiE*, pour *Metadata Distribution Evaluator*, permettant de comparer différentes méthodes de distribution équitablement dans un même contexte d'évaluation.

La méthode *LAD* permet de redistribuer dynamiquement quand cela est nécessaire : nous avons élaboré une politique de déclenchement de distribution lorsque la charge de travail entre les serveurs est trop déséquilibrée. Cela permet de limiter les redistributions inutiles tout en gardant un flux de requêtes équilibré. De plus, notre algorithme de redistribution permet un rééquilibrage efficace même en cas de fort déséquilibre. Ainsi, même si un serveur est majoritairement surchargé, notre algorithme répartira cette surcharge sur l'ensemble des autres serveurs.

Notre outil *MeDiE* est un logiciel à code source ouvert qui permet une évaluation de différentes méthodes de distribution dans un même contexte et sur le même flux de requêtes. Sa structure autorise facilement le changement de méthodes de distribution, sans modifier le processus d'évaluation, garantissant une évaluation identique pour chaque méthode. Nous avons conçu notre outil afin qu'il soit aisé d'ajouter une nouvelle méthode de distribution, pour que chacun puisse évaluer la méthode de son choix. De plus, *MeDiE* fournit un module de génération de traces répondant aux caractérisations d'un flux, facilitant ainsi l'évaluation des méthodes sur différents flux de requêtes caractéristiques.

Ce manuscrit de thèse se compose de 4 chapitres.

Le premier chapitre propose une mise en contexte de notre étude : Tout d'abord, nous décrivons les motivations et les avantages d'un système de stockage objet, ainsi que les nouvelles problématiques découlant d'un tel stockage telles que la répartition de la charge des serveurs de métadonnées. Ensuite, nous présentons une définition des méthodes de distribution permettant de répondre à cette problématique, pour finalement exposer l'état de l'art associé.

Le second chapitre détaille notre approche pour obtenir la méthode *LAD* et sa variante *LAD-TW* qui exploitent les limites des méthodes existantes.

Le troisième chapitre présente *MeDiE*, notre outil permettant d'évaluer différentes méthodes de distribution dans un même contexte. Après l'expression des besoins, nous explicitons les étapes de conception et de développement de chacun

des modules de *MeDiE*, à savoir un simulateur de service de métadonnées, un processus de simulation et un générateur de traces.

Le chapitre quatre concerne l'évaluation des méthodes de distribution à l'aide de notre outil. Nous présentons d'abord l'environnement d'évaluation utilisé : l'environnement matériel de test, la méthodologie, ainsi que le choix des flux de requêtes pertinents lors de l'évaluation. Nous regroupons ensuite l'ensemble des résultats observés et proposons une analyse de ces méthodes.

Finalement, une conclusion permet un bilan de ces contributions et apporte de nouvelles perspectives sur les travaux futurs.

Un glossaire est disponible en fin de manuscrit pour un accès rapide aux définitions nécessaires à la compréhension de ce manuscrit.

Chapitre 1

Contexte et État de l'art

Dans ce chapitre, nous rappelons les termes et définitions nécessaires à nos travaux. Après avoir défini ce que sont les systèmes de stockage objet, nous considérons plus précisément les méthodes de distribution de données.

1.1 Le stockage Objet

Le stockage objet est un type de système de stockage émergeant qui répond à des limitations induites par la norme POSIX. Afin de mieux comprendre les bénéfices que fournit le stockage objet, nous exhibons tout d'abord les motivations de l'émergence de ce type de stockage, puis ses caractéristiques. La présentation de quelques systèmes de stockage objet réels est ensuite détaillée.

1.1.1 Limitations de la norme POSIX

Les systèmes actuels de type fichier (*File Systems*) suivent les normes POSIX [IEE88] qui imposent une sémantique forte et stricte, potentiellement limitante pour le stockage d'Exaoctets [GVM00, RFB11].

Historiquement, la conception de ces systèmes a été pensée de manière centralisée : un serveur répond à toutes les requêtes et chaque accès à une donnée nécessite une requête de métadonnées au préalable. Il est aussi possible d'effectuer des requêtes sur les métadonnées sans accès aux données ensuite. Ainsi, au moins 50% des requêtes totales d'un système correspondent à des accès aux métadonnées [RLA+00, MBO+11]. C'est pourquoi l'augmentation du nombre de clients influe directement sur la gestion des métadonnées qui devient un facteur limitant de l'extensibilité du système : l'augmentation du nombre de requêtes crée un point de contention et surcharge le serveur. De plus, un unique serveur pour répondre

aux requêtes de métadonnées présente un point unique de défaillance et bloque toute requête en cas de panne.

La norme POSIX force à gérer les fichiers sous la forme d'un arbre : les fichiers sont les feuilles, chaque dossier est un nœud et la racine de l'arbre est nommée "/". Accéder à un fichier nécessite de parcourir toute l'arborescence à partir de la racine, verrouillant chaque dossier parent pour garder la cohérence des données. Quand une autre requête a besoin d'accéder à un nœud de la même sous-arborescence, elle doit attendre la fin de la première requête afin de déverrouiller les nœuds déjà accédés. Cette procédure empêche une forte concurrence dans les accès aux éléments d'un arbre, ce qui limite l'extensibilité du système.

Cette structure en arborescence génère des machines à état complexe, de par la caractéristique *stateful* des accès dans POSIX (POSIX I/O). Plus l'arborescence grandit, plus la machine à états correspondante sera complexe. Le volume de données prédit pour l'*exascale* impliquant des arbres contenant des billions de nœuds, il devient trop ardu voire impossible d'implémenter des machines à états supportant une telle arborescence.

La gestion des métadonnées est elle aussi limitée par la norme POSIX. À chaque donnée est associé un ensemble spécifique de métadonnées comme le propriétaire, les permissions du fichier ou encore la date du dernier accès. La définition et la gestion de ces métadonnées sont contraints par des règles strictes complexifiant les accès :

- Certaines métadonnées sont identiques pour tous les fichiers d'un même dossier et pourraient être factorisées, cependant, la norme POSIX l'interdit. En effet, Factoriser les métadonnées signifie de nouveaux états à maintenir, et la machine à états deviendrait alors trop complexe.
- Certaines métadonnées sont modifiables en effectuant une action de métadonnées, par exemple le propriétaire du fichier, et d'autres nécessite une modification de la donnnée, par exemple la taille de la donnée. La norme POSIX mélange dans une même structure ces deux types de métadonnées, complexifiant la gestion de cette structure.
- Les attributs définis par la norme POSIX ne sont généralement pas suffisants pour décrire une donnée et des informations additionnelles doivent être stockées dans un second fichier de métadonnées (attributs étendus).

La rigidité de la gestion des métadonnées soumise à la norme POSIX amène un ralentissement des accès et limite encore une fois l'extensibilité.

1.1.2 Définition globale du stockage Objet

Le stockage objet est un type de stockage qui se détache de la gestion de fichiers en adoptant une sémantique plus simple appelée sémantique *Create Read Update Delete*) (CRUD) [BB08] et est par conséquent différent des normes POSIX. Comme l'explique G. Gibson [GVM00], les systèmes de stockage objet essaient de satisfaire des besoins existants tels que la consolidation de ressources, un déploiement rapide, une gestion centrale du système de stockage, une sauvegarde facile en cas de panne, une disponibilité élevée ou le partage des données. Mais ils tentent aussi de répondre aux besoins émergeants comme la séparation géographique des composants du système, l'augmentation de la sécurité des accès ou l'extensibilité du système de stockage et les performances que cela implique. Le stockage objet a émergé dans les années 2000, avec certaines contraintes : par exemple, les premiers systèmes objet ne permettaient pas de modifications de données, faisant de chaque objet une donnée immuable [GGL03, Cen02]. Les évolutions récentes ont permis de lever ces restrictions [CPK+18, DAI17] et continuent d'apporter des améliorations aux systèmes de stockage objet.

L'idée principale du stockage objet consiste en un système hautement distribué et détaché des contraintes imposées par le stockage en fichiers grâce, entre autres, à une sémantique simple. Dans la plupart des définitions [MGR03, Pan07], il différencie le flux de données et le flux de métadonnées en utilisant des serveurs dédiés aux métadonnées appellés *MetaData servers* (MDS) et des serveurs pour les données, appelés *Object Storage Device* (OSD). Les données sont stockées sous forme d'objets dans un espace de nommage plat sans arborescence appelé aussi adressage associatif. La majorité des systèmes objet sont de type *Key-Value Store* (KVS), c'est-à-dire que les accès aux données (*value*) se font via un identifiant unique (*key*).

Composition d'un système de stockage objet :

Comme illustré sur la figure 1.1, un système de stockage objet se compose de 5 éléments :

- les objets,
- les *Object Storage Devices*,
- l'interface client,
- le service de métadonnées,
- le réseau.

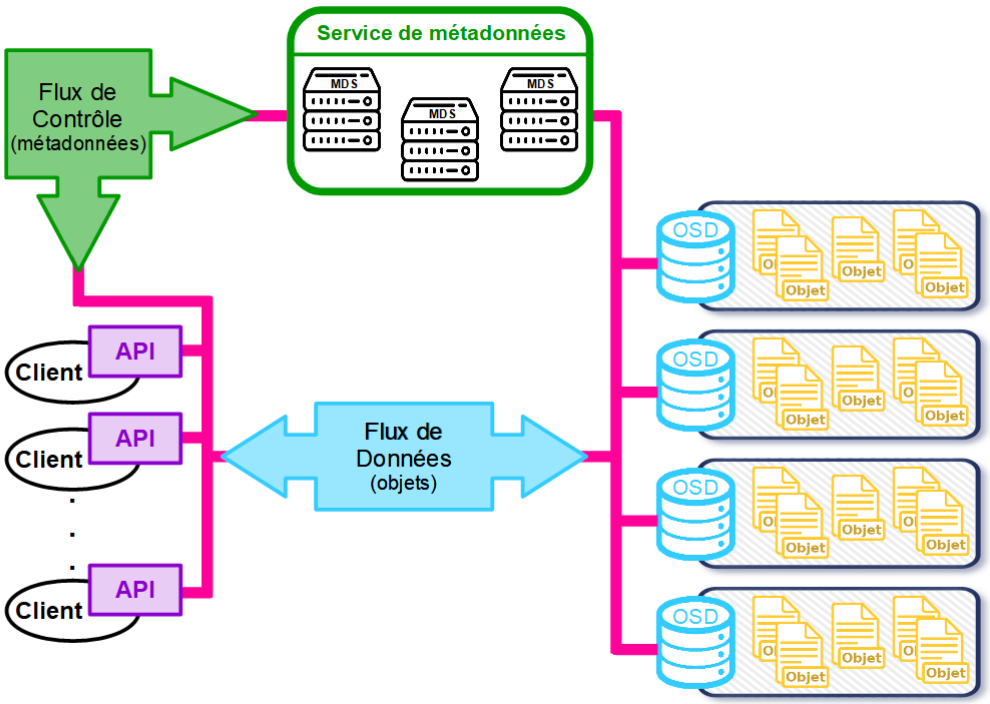


FIGURE 1.1: Architecture d'un système de stockage objet.

1. Un objet (en jaune sur la figure) est l'unité de stockage d'une donnée. Il est composé de la donnée et de ses attributs, ou métadonnées. L'ensemble des métadonnées associées est personnalisable et peut être réduit par rapport à ce qu'impose la norme POSIX, ou bien enrichi afin de compléter la description d'un objet. Il y a deux types de métadonnées : les attributs systèmes, comparables aux informations inode [1] d'un système de fichiers, et les attributs utilisateurs qui sont des informations de plus haut niveau telles que le propriétaire ou la date de la dernière modification. Chaque objet est unique et possède son propre identifiant.

2. Un *Object Storage Device*, ou OSD (en bleu sur la figure) est un média de stockage intelligent composé d'un disque, d'un processeur et de mémoire RAM. Une interface permet à l'OSD de gérer par lui-même l'espace de stockage

1. Les inodes contiennent notamment les métadonnées des fichiers, et en particulier celles concernant les droits d'accès

et toutes les opérations d'accès. Il est responsable de la disponibilité des données stockées et doit permettre l'accès à une donnée via un identifiant. Il autorise les accès en vérifiant les droits des utilisateurs et en respectant l'ordre d'exécution des opérations afin de maintenir la cohérence.

3. L'interface client (en violet sur la figure) fournit au client une interface de programmation (API) de stockage et traduit les requêtes clients en opérations compréhensibles par le système de stockage. Communément, cette interface obéit à la sémantique CRUD qui utilise des commandes simples comme le retrait (`get`), le dépôt (`put`) ou la suppression (`delete`). Cette sémantique est dite *stateless*, ce qui signifie que les exécutions des opérations n'influent pas sur l'état du système.

4. Le service de métadonnées (en vert sur la figure) est composé d'un ou de plusieurs serveurs de métadonnées (MDS) qui contrôlent les interactions entre les clients et les OSDs. Il donne aux clients les droits d'accès aux données et la localisation de l'OSD associé. Ces droits sont valables pendant un certain temps, ce qui autorise le client à échanger directement avec l'OSD lors de futures requêtes. Le service de métadonnées doit distribuer les objets parmi tous les OSDs afin d'équilibrer les charges de travail. Il est aussi responsable de l'initialisation d'un nouvel OSD et vérifie régulièrement la disponibilité des OSDs.

5. Le réseau (en rose sur la figure) fait le lien entre tous les autres composants. Il permet aux clients d'accéder à leurs données et au service de métadonnées de communiquer avec les OSDs. Le réseau supporte différents protocoles de communication comme les RPCs (*Remote Procedure Call*) pour interroger un MDS ou le protocole iSCSI (*Internet Small Computer System Interface*) pour échanger avec les OSDs.

Extensibilité d'un système de stockage objet :

Le stockage objet permet une forte extensibilité et cela grâce à trois principales caractéristiques :

- La séparation du flux de données et de métadonnées, comme il est possible de le voir dans la figure 1.1 afin de limiter les points de contention.
- La vérification des droits d'accès est faite sur un serveur différent que celui qui les octroie. Les serveurs de métadonnées génèrent des droits d'accès et les OSDs doivent seulement vérifier la validité de la permission que le client

leur présente. Le nombre de clients accédant aux OSDs peut donc augmenter indépendamment de l'espace de stockage et du nombre de médias.

- L'utilisation d'un espace de nommage plat : chaque objet est stocké au même niveau que tous les autres, évitant les problèmes de contention que l'on peut trouver avec une structure en arborescence. Cela permet d'ajouter des médias de stockage sans contraintes ou conséquences sur le système.

Ces points permettent une plus grande concurrence entre les accès des données, puisque les tâches sont distribuées plutôt que d'avoir une gestion centralisée. Ce gain de concurrence améliore grandement l'extensibilité du système et par conséquent les performances globales du système. De plus, les systèmes objet facilitent les accès des utilisateurs en permettant de personnaliser les métadonnées et d'accéder à leurs données avec un identifiant unique. L'utilisation d'une sémantique simple tel que la sémantique CRUD et la caractéristique *stateless* des systèmes de stockage objet facilitent aussi la gestion des accès.

Cependant, la distribution des tâches sur les différents composants rend les protocoles internes plus complexes et dépendants du réseau. Chaque composant doit informer les autres des changements qu'il a effectué, augmentant ainsi le nombre de communications. De plus, les systèmes objet sont totalement dissociés de la norme POSIX et de la structure en arborescence, et ne sont donc pas directement utilisables dans un contexte système de fichier classique, même si une adaptation peut être implémentée, comme avec *CephFS* [WBM+06].

1.1.3 Quelques systèmes objet connus :

L'un des systèmes objet les plus connus est Ceph [WBM+06], projet open-source garantissant une cohérence forte. Créé en 2006, ce projet s'est développé et a été racheté par *Red Hat* [SW14] en 2014 qui participe à son évolution depuis. Il est constitué d'un ensemble d'OSD appelé *Reliable Autonomic Object Storage* (RADOS) pour les données et d'un service de métadonnées. Les serveurs de métadonnées se partagent la responsabilité des données en utilisant le partitionnement en sous-arborescence dynamique [WPBM04]. Pour accéder à une donnée, le client contacte le service de métadonnées afin d'avoir un droit d'accès. Il utilise ensuite un algorithme de hachage appelé *Controlled Replication Under Scalable Hashing* (CRUSH) afin d'avoir la localisation de la donnée et peut contacter l'OSD correspondant. S'il y a une modification, le client recontacte enfin le service de métadonnée afin de l'informer des changements effectués. Pour garder une forte disponibilité des données, celles-ci sont répliquées et la cohérence est gérée grâce à l'algorithme de quorum PAXOS [Lam98] qui cherche à établir un consensus entre les serveurs.

Un deuxième système de stockage objet connu est Dynamo [DHJ+07, DHJ+15], le KVS d'Amazon, faisant partie de l'infrastructure de *Amazon Web Services*. Il fournit une forte disponibilité des données mais garantit une cohérence faible. C'est un système hautement distribué où chacun des nœud (ou serveur) du système a le même rôle et possède un statut de coordinateur, leur donnant la responsabilité d'un ensemble de données c'est-à-dire de garder la disponibilité et la cohérence des réplicas. Chaque nœud reçoit des requêtes et les transfère ensuite à un coordinateur pour ces données. Les données sont assignées aux coordinateurs grâce à une fonction de hachage et les réplicas sont distribués sur les différents serveurs avec une technique de hachage cohérent [KLL+97]. Pour accéder à une donnée, un client envoie une requête à un gestionnaire de charge qui transmet sa requête à un serveur libre. Si le serveur est un coordinateur pour cette donnée, il déclenche une procédure de quorum partiel (apellé *sloppy quorum*) afin que tous les coordinateurs valident le changement, il envoie ensuite au client le résultat de l'opération.

Il existe aussi d'autres systèmes objet comme Týr [MCA+16], un système de stockage objet, garantissant une cohérence forte. Il est composé de routeurs de requêtes qui redirigent les requêtes, d'un gestionnaire de cluster qui vérifie l'état du système, de processeurs de requêtes qui traitent les requêtes et d'agents de stockage qui stockent données et métadonnées. Pour chaque accès, le processeur de requêtes doit se coordonner avec le gestionnaire de cluster pour accéder à la donnée. Týr divise les données volumineuses en morceaux qui sont répliqués et répartis dans l'espace de stockage en utilisant le hachage cohérent. Les métadonnées sont répliquées et distribuées de la même manière. Pour garder la cohérence entre les parties, le système désigne le serveur responsable du premier morceau comme nœud de metadonnées qui doit alors maintenir un graphe de dépendance entre toutes les parties et leurs différentes versions.

1.2 Méthode de distribution et espace de nommage

Dans un système de stockage objet, on sépare le flux de données de celui de métadonnées avec un service dédié pour la gestion des métadonnées. Ce service regroupe les métadonnées de chaque objet et renvoie ces informations aux clients qui les demandent. Il contrôle aussi les accès concurrents des clients afin de limiter les incohérences. Un service de métadonnées peut se reposer sur un seul serveur ou bien un regroupement de MDS que l'on appelle cluster.

Le service de métadonnées est un composant majeur dans l'extensibilité d'un système de stockage : plus de 50% des accès du système sont destinés au service de métadonnées [RLA+00, MBO+11]. Si celui-ci ne permet pas un bon passage à l'échelle alors les requêtes de métadonnées vont créer des points de contention et le système entier sera bloqué. Il faut donc une répartition équilibrée des accès sur tous les serveurs. De plus, au vu de la taille croissante des données, il devient aussi important de se soucier de la taille des métadonnées et d'optimiser au maximum l'espace nécessaire à ce service.

Service de métadonnées centralisé :

L'idée la plus simple pour concevoir un service de métadonnées consiste en une solution centralisée, c'est-à-dire un gestionnaire unique qui répond à toutes les demandes des clients et des serveurs en charge de la réplication des métadonnées afin que le service résiste aux pannes. Le MDS traite seul les accès, il sait donc si une métadonnée est en train d'être modifiée. Il peut, en conséquence, refuser l'accès à cette métadonnée aux autres clients, pendant que la modification est en cours. Cependant, en suivant cette idée, on se heurte très vite à un souci de passage à l'échelle : si le nombre de clients augmente, un serveur unique ne suffit plus pour traiter toutes les demandes. Le nombre de requêtes arrivant pour le MDS augmente plus rapidement que le nombre de requêtes traitées par le serveur. On obtient alors un goulot d'étranglement qui limite les actions du serveurs, ralentissant ainsi le système dans son intégralité.

Service de métadonnées distribué :

Pour résoudre ce problème, il est possible de partitionner les métadonnées et distribuer les partitions entre plusieurs serveurs. Ainsi, chaque serveur est responsable d'une partie des métadonnées et ne traite qu'une partie des requêtes. Pour ce faire, on utilise l'espace de nommage (*namespace*) des objets stockés comme élément distinctif. L'espace de nommage des objets correspond à l'ensemble des noms possibles pour un objet. Suivant le nom de l'objet, les métadonnées associées peuvent être gérées par des serveurs différents mais toutes les métadonnées d'un même objet sont gérées par le même serveur.

Différentes méthodes de distribution vont produire différentes répartitions comme on peut le voir sur la figure 1.2 où chaque couleur correspond aux noms associé à chaque serveur. La distribution de l'espace de nommage répartit les métadonnées afin que chaque serveur reçoive un nombre équivalent de requêtes de métadonnées, ce qui en fait une difficulté essentielle des services de métadonnées distribués. En effet, une simple répartition uniforme n'est pas forcément adaptée : certaines mé-

tadonnées ou groupe de métadonnées peuvent être très demandées chargeant le serveur responsable de plus de requêtes (*hot spots*).

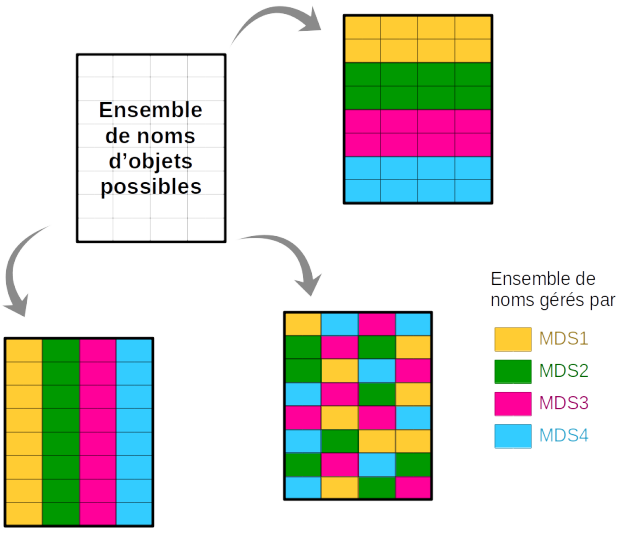


FIGURE 1.2: Exemples de répartition des métadonnées d'un espace de nommage.

L'unité de partitionnement des métadonnées peut être de l'ordre de l'objet ou plus large comme les dossiers pour des systèmes de stockage de type fichiers. Utiliser une unité de l'ordre de l'objet permet cependant une granularité plus ajustable de la distribution en fonction des besoins : les métadonnées de deux objets très demandés peuvent être gérées par des serveurs différents. Afin que le service de métadonnées identifie le serveur responsable d'une partie de l'espace de nommage, plusieurs méthodes sont utilisées et chacune peut se baser sur une structure différente : fonction de hachage, table d'index, *b-tree*...

Il existe de nombreuses méthodes de distribution qui possèdent chacune leurs avantages et leurs inconvénients [LWV07, WFWL09, XHL+10, XAYM14, YWL+14, TBD+17]. Suivant la répartition des métadonnées sur les serveurs, les localisations des *hot spots* vont varier et provoquer des goulots d'étranglement à différents niveaux du système. Chaque méthode peut aussi utiliser des structures différentes pour garder en mémoire la répartition ou bien s'en passer ou encore fournir une notion de localité spatiale différente.

Le principe de localité s'exprime selon deux axes :

- La *localité temporelle* : on accède à des données récemment accédées. Cette localité ne change pas suivant les distributions de l'espace de nommage donc nous n'en tiendrons pas compte ;

- La *localité spatiale* : on accède à des données stockées dans un espace mémoire proche de données déjà accédées. Cette localité change en fonction de la distribution de l'espace de nommage.

Dans un système de stockage de type fichier, la notion de localité s'applique pour les répertoires [2], c'est-à-dire que des fichiers dans le même dossier ont plus de chances d'être accédés dans une même exécution. Pour le stockage objet, la notion de localité n'est pas définie de manière fixe : dans certains systèmes, elle s'applique aux données dans un même OSD, mais on peut aussi envisager d'autres critères, par exemple, une localité par utilisateur.

Une méthode de distribution peut aussi agir différemment dans le temps, devenant ainsi une méthode *statique* ou *dynamique*. Les méthodes de distribution statiques assignent un MDS à une métadonnée qui vient d'être créée. Cette assignation ne changera qu'en cas de panne du serveur responsable de la métadonnée ou suppression de la donnée associée. Ce sont des algorithmes simples à mettre en place qui ne changent pas la distribution au cours du temps et qui permettent un équilibrage des requêtes à court terme, permettant des performances variables. Cependant, un système change au cours du temps : des données sont créées ou supprimées, certaines deviennent très populaires ou au contraire ne sont plus ou peu accédées. La structure du service de métadonnées évolue aussi : le système peut s'agrandir avec l'ajout de nouveaux MDS ou bien être restreint avec la suppression de certains. Ces changements ne sont pas prévisibles au moment de l'initialisation d'une distribution statique des données, c'est pourquoi certains services de métadonnées sont capables de s'adapter à ces changements au fur et à mesure qu'ils se produisent. Pour ces systèmes, la distribution de l'espace de nommage aux MDS doit donc se faire de manière dynamique et doit pouvoir évoluer au cours du temps afin de réguler la charge de travail de chaque serveur.

1.3 Les différentes catégories de méthodes de distribution

La taille croissante des systèmes de stockage encourage à disposer de plusieurs serveurs de métadonnées. Mais il existe encore des systèmes n'utilisant qu'un unique serveur de métadonnée comme GFS [GGL03].

L'état de l'art contient un certain nombre de méthodes de distribution de métadonnées proposées pour un système de stockage de type fichier [GGL03, WPBM04,

2. il existe aussi une localité intrafichier mais nous n'en tiendrons pas compte ici.

XZ12, STSM12]. Même si ces méthodes ne sont pas les plus adaptées au stockage objet, elles sont tout de même utilisables dans ce cadre.

De la même manière, il existe des méthodes de distribution liées aux données permettant de répartir des données sur les différents serveurs de données (de type objet ou bien fichier). Bien qu'elles ne soient pas optimisées pour, ces méthodes sont elles aussi applicables à notre contexte en considérant les métadonnées comme des données de très petite taille.
Dans cette partie, nous considérons toutes les méthodes de distribution, qu'elles soient conçues pour du stockage de données ou de métadonnées, ou qu'elles soient originellement prévues pour un système de stockage objet ou fichier. Nous avons classé les différentes méthodes en nous basant sur la structure utilisée, comme l'ont aussi fait Singh et. al. [SB18].

1.3.1 Méthodes basées sur une structure arborescente

Les méthodes basées sur une structure arborescente sont les méthodes les plus intuitives et les plus proches du stockage de type fichier. On considère l'espace de nommage sous forme d'arbre que l'on découpe en sous-arbres. La méthode du Directory Subtree Partitionning (DSP) en est l'exemple le plus simple : chaque sous-arbre est régi par le même gestionnaire, ce qui signifie que toutes les métadonnées liées à un élément du sous-arbre sont regroupées sur un même serveur. Cette méthode est utilisée dans beaucoup de systèmes comme *CODA* [SKK+90] ou *NFS* [PJS+94]. Un exemple de distribution de métadonnées est donné à la figure 1.3.

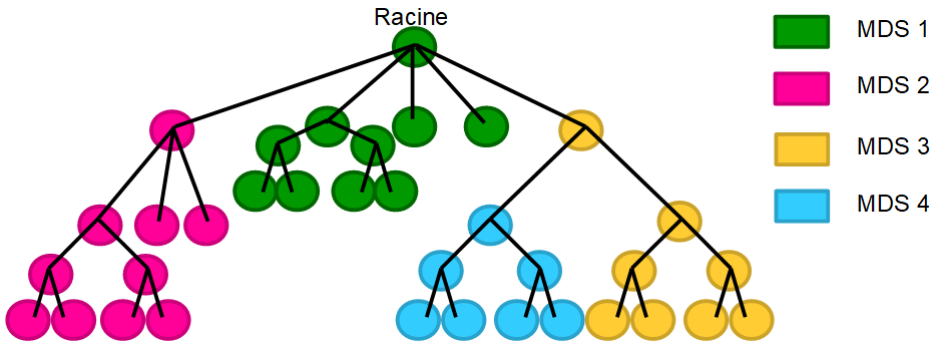


FIGURE 1.3: Répartition possible des métadonnées avec la méthode de partitionnement en sous-arbres.

La vision sous forme d'arbre est propre au stockage en mode fichier mais peut être simulée dans des systèmes de stockage objet compatibles avec la norme

POSIX, par exemple en prenant comme nom de l'objet le chemin dans un arbre. Le partitionnement en sous-arbres découpe cette arborescence en prenant comme plus petit élément possible le répertoire. Il permet ainsi de garder le principe de localité spatiale comme il est défini dans les systèmes de fichiers. La distribution en sous-arbres permet aux systèmes soumis aux normes POSIX de maintenir leurs protocoles d'accès : il est possible de verrouiller l'accès à des répertoires parents lors de l'accès à une localisation. De plus, aucune surcharge n'est signalée lors de la modification de l'arbre pour, par exemple, un renommage d'un sous-arbre. Il en va de même pour les permissions associées puisqu'un système hiérarchique est implicite dans la structure de l'arbre.

La méthode de partitionnement par sous-arbres (DSP) ne nécessite pas de table d'index. Lorsqu'un client veut accéder à une métadonnée et qu'il n'en connaît pas le serveur responsable, il va adresser une requête au premier MDS qu'il connait et celui-ci va lui indiquer quel est le gestionnaire en charge de la racine de l'arbre (*root*). Ce dernier va indiquer au client quel est le prochain serveur à questionner : le MDS immédiatement en dessous de lui dans l'arborescence qui est en charge du répertoire préfixant le chemin de la localisation demandée. Le client va donc parcourir toute l'arborescence de cette façon jusqu'à atteindre le serveur en charge du répertoire contenant sa métadonnée, comme le montre la figure 1.4. Cette méthode s'avère donc coûteuse en communications de par ses nombreuses indirections. Cependant, il est possible de réduire ce nombre de redirections pour les systèmes soumis aux normes POSIX, puisque la méthode DSP permet d'exploiter au maximum les notions de caches proposées dans les systèmes de fichiers.

On voit rapidement les limites de cette méthode : si un répertoire ou un objet devient populaire (*hot spots*), un goulot d'étranglement va se former, et les clients attendront pour questionner les mêmes serveurs. En effet, si plusieurs clients tentent d'accéder à des métadonnées situées dans un même répertoire, ils vont tous passer par le même chemin et donc la même suite de MDS pour atteindre le serveur du répertoire.

Le partitionnement en sous-arbres associe le stockage physique d'une métadonnée à un MDS. Ainsi, lorsque la distribution de l'espace de nommage change (par exemple, lorsqu'il faut modifier l'arborescence), les métadonnées doivent être réécrites sur le disque du nouveau serveur. Lorsqu'un gestionnaire est retiré ou ajouté, un important mouvement de métadonnées est nécessaire : l'arbre étant en totalité distribué parmi les MDS, l'ajout ou la suppression d'un serveur provoque la redistribution d'une partie des métadonnées.

Le partitionnement en sous-arbres permet un bon passage à l'échelle si l'arborescence grandit horizontalement c'est-à-dire en ajoutant de nouveaux sous-arbres

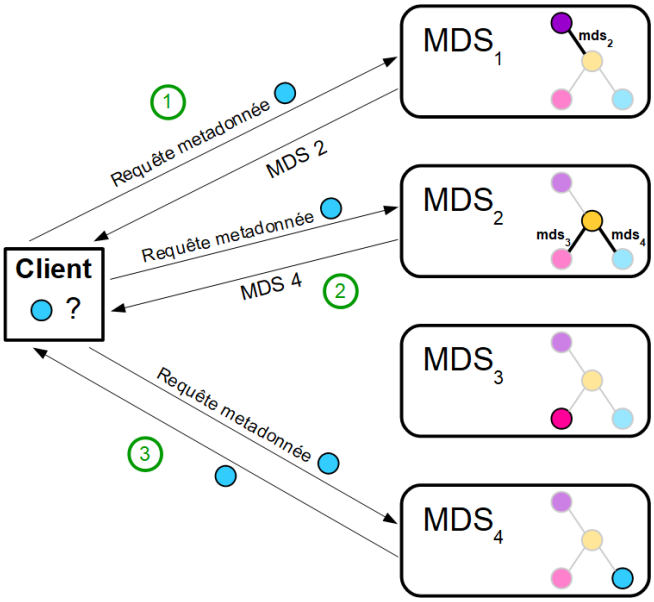


FIGURE 1.4: Accès à une métadonnée avec le partitionnement par sous-arbres.

à la racine, puisque chaque nouveau répertoire peut être attribué à n'importe quel serveur existant. Cependant, il ne permet pas une bonne extensibilité en profondeur, c'est-à-dire en imbriquant des sous-arbres entre eux et plus particulièrement si cette expansion n'est pas régulière et ne concerne qu'une portion d'un sous-arbre, puisque chaque nouveau répertoire doit être assigné au MDS en charge du répertoire parent et cela peut provoquer une redistribution partielle.

Pour pallier le goulot d'étranglement dû aux accès concurrents à un même répertoire, J. Xing et al. [XXSM09] ont décidé dans leur méthode de découper les répertoires contenant un grand nombre d'entrées et de les distribuer à d'autres MDS tout en gardant la hiérarchisation. Cette amélioration du partitionnement en sous-arbres est conçue pour diviser des répertoires de l'ordre de 10^{12} entrées en deux niveaux : un répertoire est découpé en partitions, elles-même composées de morceaux (*chunk*). On distribue tout d'abord les répertoires parents avec le partitionnement en sous-arbres, puis pour les répertoires volumineux, on associe les partitions à un serveur grâce à la technique de hachage cohérent (voir sous-section 1.3.2) et les clients peuvent ensuite accéder de façon concurrente à plusieurs morceaux dans une même partition. Cependant, cette couche d'indirection implique l'utilisation d'une table d'index.

Le nombre de partitions et de morceaux augmente de manière proportionnelle, l'un puis l'autre en gardant un bon équilibre entre les deux. On augmente le nombre

de partitions pour un meilleur passage à l'échelle ou on augmente le nombre de morceaux par partition pour diminuer le nombre d'accès concurrents à une même information et donc avoir de meilleures performances.

Cette méthode, utilisée dans le système *Dawning nebulae* [SXH+11], permet de réduire les goulots d'étranglements dûs aux répertoires populaires. Cependant, elle ne gère pas les accès fréquents à une même métadonnée et le découpage en partitions implique un surcoût dû au maintien d'une table d'index pour connaitre l'emplacement du MDS en charge d'une partition.

La méthode du partitionnement par sous-arbres a aussi servi de base pour la méthode du *Dynamic Directory Subtree Partitionning* par S. Weil et al. [WPBM04] afin qu'elle répartisse la charge dynamiquement sans perdre le principe de localité des systèmes de fichiers, pour les systèmes compatibles POSIX. La méthode de partitionnement par sous-arbres dynamique (*Dynamic subtree partitionning*) utilisée dans *Ceph* [WBM+06] se base sur le calcul régulier de la charge de travail de chaque MDS. Si un serveur est considéré comme surchargé, le système lui enlève une partie de son sous-arbre pour l'attribuer à un gestionnaire ayant une charge de travail plus faible. Si la charge de travail est due à un seul répertoire, celui-ci est répliqué et distribué parmi d'autres MDSs grâce à une fonction de hachage. Le serveur qui était en premier responsable de ce répertoire doit alors maintenir à jour les autres réplicas. Cependant, une fois qu'un répertoire est répliqué, il le reste même lorsque le nombre d'accès à ce répertoire diminue, ce qui entraine un surcoût en communication et en espace mémoire.

Lorsqu'un client veut accéder à une métadonnée, il envoie une requête à un MDS au hasard. Ensuite, celui-ci le redirige vers le serveur responsable du sous-arbre s'il le connaît ou vers un MDS dit "d'autorité" sinon. Le MDS d'autorité fait de même jusqu'à ce que le client arrive vers le bon serveur. Si la métadonnée demandée est répliquée et gérée par plusieurs serveurs (car dans un répertoire populaire), alors on donne au client la liste de tous les MDSs responsables d'un réplica.

La méthode du Dynamic subtree Partitionning a aussi été reprise par W. Xue et M. Zhu dans leur système de stockage *LandFile System* [XZ12] en modifiant entre autres les conditions de redistribution : pour chaque répertoire, une *heat value*, représentant le temps d'accès pour le répertoire est calculée en fonction des coûts en mémoire, des coûts CPU et des coûts en bande passante induits par le répertoire. La charge d'un serveur est calculée en sommant toutes les *heat values* des répertoires assignés à ce serveur. Le gestionnaire possédant la charge la plus importante est considéré comme surchargé et effectue un calcul de redistribution.

1.3.2 Méthodes basées sur une fonction de hachage

Ces méthodes de distribution, que l'on retrouve dans de nombreux systèmes comme *Lustre* [Bra19] ou *OpenStack Swift* [BDH+10], utilisent une fonction de hachage pour attribuer les métadonnées d'un objet à un serveur. De nombreux paramètres d'entrée pour la fonction de hachage sont possibles : nom du répertoire si le système est compatible POSIX, nom de l'objet ou encore combinaison de différents attributs... Le plus courant reste, ici encore, une approche en mode fichier avec le chemin absolu vers la donnée. Une fois cette fonction calculée pour tous les objets, on répartit l'ensemble d'arrivée de notre hachage, appelé aussi ensemble de clefs, parmi tous les serveurs de manière ordonnée. Le premier MDS gère le premier intervalle de clefs et ainsi de suite.

Toutes les fonctions de hachage peuvent posséder certaines propriétés. Elles garantissent, par exemple, une probabilité de collision, c'est-à-dire, la probabilité que deux métadonnées différentes génèrent un même résultat. Elles définissent aussi l'aspect aléatoire de la distribution, ce qui correspond à la capacité à prédire le résultat de la fonction de hachage pour une métadonnée grâce à des motifs dans la distribution. Cette dernière propriété est un facteur important permettant aux serveurs une charge de travail (*workload*) plus équilibrée : Si deux clés proches génèrent des résultats pointant vers des serveurs différents, la charge de chaque serveur sera plus équilibrée. À l'inverse, si deux clés proches sont dirigées vers le même serveur, l'équilibrage des requêtes pourrait être affecté. C'est la raison pour laquelle on recherche à optimiser la caractéristique aléatoire de la distribution pour une fonction de hachage destinée aux distributions de métadonnées. Il est important de noter que cette caractéristique n'influe pas sur la reproductibilité de la méthode mais sur la répartition des résultats de la fonction de hachage. Pour chaque élément de l'espace de nommage, il existe une clé unique qui ne change pas si on réitère le calcul.

Ainsi, la charge de travail est plus équilibrée pour chaque serveur et les goulots d'étranglement pour les répertoires trop populaires sont évités. En effet, les métadonnées associées à des données dans un même répertoire ne sont pas toutes attribuées au même gestionnaire, comme on peut le voir à la figure 1.5. Cependant, cette méthode n'apporte pas de solution pour les objets qui sont fréquemment accédés individuellement.

Cette méthode est aussi plus coûteuse pour les systèmes compatibles POSIX car un système de hiérarchie doit tout de même être implémenté pour gérer les permissions des données et des localisations. Ainsi, il est aussi plus compliqué de garder les avantages d'une arborescence de données, comme les optimisations utilisant les principes de localités et les systèmes de gestion de caches. Il est aussi

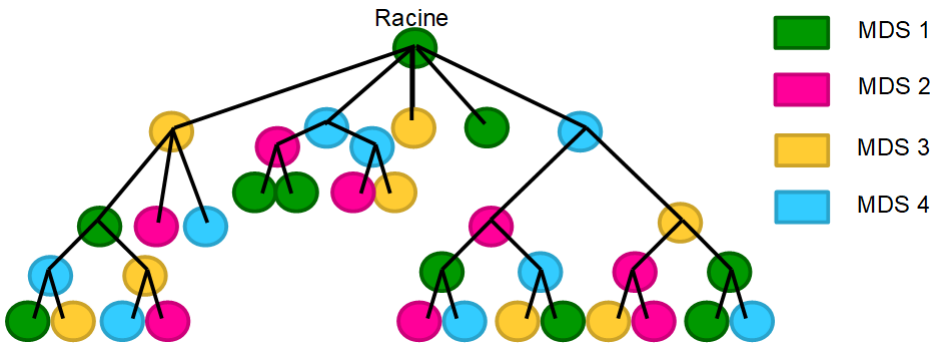


FIGURE 1.5: Répartition possible des métadonnées avec la méthode de hachage.

important de souligner que lors de la modification des localisations des données, les clefs concernées doivent être recalculées avec la nouvelle localisation comme paramètre.

Lorsqu'un client veut accéder à une métadonnée, il calcule la clef correspondante grâce à la fonction de hachage et utilise ensuite généralement une opération de modulo pour connaître le serveur responsable de cette clef. Le client a donc un accès rapide et n'effectue qu'une requête au service de métadonnées.

L'ajout ou la suppression d'un serveur est potentiellement coûteux : une modification de la fonction de hachage est nécessaire avec en paramètre de la fonction le nombre de MDS, impliquant le recalcul et la possible redistribution totale des métadonnées. Cette redistribution est d'autant plus coûteuse que l'on associe aux différents serveurs le stockage physique de la métadonnée et qu'il est donc nécessaire de la réécrire en cas de changement de MDS.

Nous pouvons retrouver ce coût de redistribution dans la méthode de B. Cai et al. [CXZ07] qui utilise un algorithme appelé *Self-Balancing Uniform* (SBU) pour choisir le serveur responsable de la prochaine métadonnée. Cet algorithme consiste à utiliser deux fonctions de hachage afin d'avoir un MDS principal et un autre alternatif pour chaque métadonnée que l'on souhaite accéder. Puis, suivant la charge de chaque serveur, la métadonnée est assignée soit au serveur principal, soit au serveur alternatif. La charge d'un serveur est calculée périodiquement en prenant comme métrique la latence des accès. Une fois qu'une métadonnée est assignée à un gestionnaire (principal ou alternatif), elle reste toujours associée à ce MDS. L'algorithme SBU est donc un algorithme statique où il n'y a pas de redistribution, seulement un placement intelligent au cours du temps.

Pour accéder à une métadonnée, un client doit calculer les deux serveurs possibles grâce aux deux fonctions de hachage. Il envoie d'abord une requête au serveur principal et si celui-ci n'est pas responsable de la métadonnée, le client contacte alors le MDS alternatif. Dans cette méthode, le coût de redistribution est doublé par rapport à une méthode de distribution par hachage simple, puisqu'il y a deux fonctions de hachage qui nécessitent d'être recalculées.

Pour réduire le coût de réorganisation, certains systèmes comme *Dynamo* [DHJ+07] ou *TYR* [MCA+16] utilisent une variante appelée hachage cohérent (*consistent hashing*). Cette méthode reprend la technique de hachage en ne prenant pas en compte les MDS dans la fonction de hachage. En plus d'attribuer une clef à chaque donnée, on utilise une fonction de hachage pour donner à chaque serveur un identifiant.

L'association d'une clef à un serveur est illustrée par la figure 1.6 et se fait de la façon suivante :

1. On visualise notre espace de clefs sous forme de cercle, ainsi la dernière clef de l'espace précède la première ;
2. On projette les identifiants des serveurs pseudo-aléatoirement sur ce cercle, en associant à chaque identifiant plusieurs points sur le cercle pour mieux équilibrer ;
3. On parcourt ensuite le cercle en associant à chaque fois la clef au MDS dont l'identifiant est le plus proche au-dessus d'elle.

Cette méthode permet de limiter la redistribution due à l'ajout ou la suppression d'un MDS. En effet, si un nouveau serveur arrive dans le système, on lui attribue un identifiant et des places sur le cercle. Il réclame ensuite au MDS suivant les clefs se trouvant entre lui et le gestionnaire précédent. De la même manière, si un serveur est supprimé, le MDS dont l'identifiant est le plus proche au-dessus de lui récupère la responsabilité des clefs dont il avait la charge.

D'autres systèmes réduisent leur coût de réorganisation en utilisant une famille d'algorithmes appelée *RUSH* (*Replication Under Scalable Hashing*) [HM04]. Ces méthodes ont été d'abord conçues pour distribuer des données et leurs réplicas sur les différents disques du système (appelées CRUSH [WBMM06]), mais peuvent aussi être appliquées pour la distribution de métadonnées. Cette famille d'algorithmes de hachage attribue à chaque MDS un poids indiquant sa capacité de gestion (en fonction des caractéristiques des disques de stockage) et propose trois variantes adaptées pour différents systèmes :

- $RUSH_p$ est une heuristique basée sur l'utilisation de nombres premiers et est plus adaptée aux petits systèmes avec des ajouts de MDS en petite quantité.

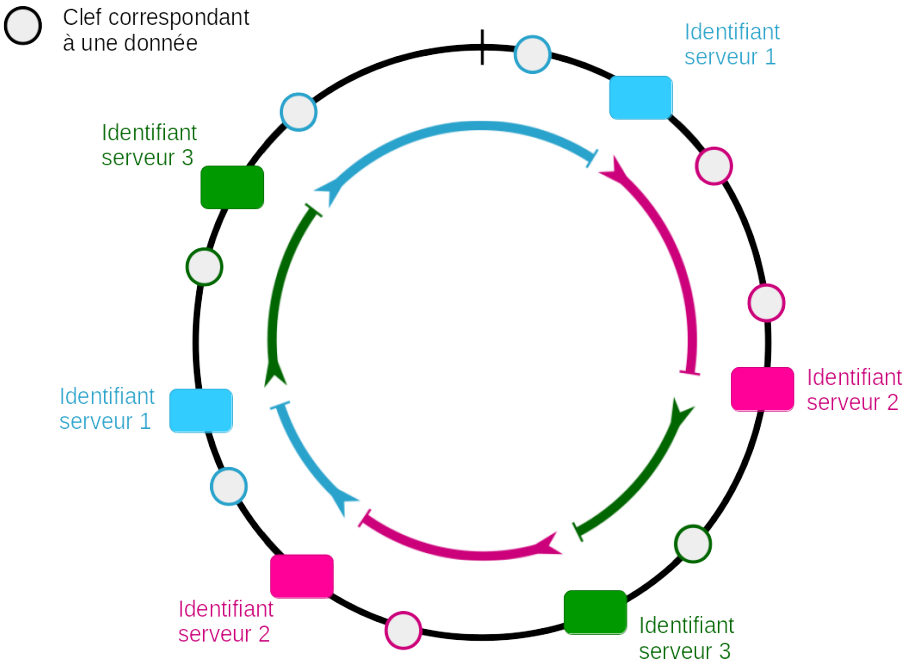


FIGURE 1.6: Attribution des données aux MDS par hachage cohérent.

Elle permet une bonne réorganisation sauf pour la suppression de groupes de serveurs ;

- $RUSH_r$ est basée sur les m$RUSH_p$ et permet une suppression des sous-groupes de serveurs sans réorganiser tout le système ;
- $RUSH_t$ prend comme structure de données des arbres et est plus adaptée aux systèmes de grande taille qui ajoutent de nombreux MDS. Elle permet ainsi un meilleur passage à l'échelle, mais s'avère moins performante que les deux précédentes.

Le système *Dynamic Ring Online Partitionning* (DROP) [XAYM14] tente de réduire les coûts de réorganisation en modélisant son service de métadonnées comme un anneau sur lequel sont placés différents serveurs virtuels. Chaque serveur virtuel est responsable d'une partie de l'espace de nommage et est exécuté sur un serveur physique. Plusieurs MDS virtuels peuvent être exécutés sur une même machine physique. Les métadonnées sont distribuées sur tous les gestionnaires virtuels de l'anneau grâce à une fonction de hachage nommée *Locality preserving Hashing* qui préserve le principe de localité des répertoires. Afin d'accéder à une métadonnée, un client calcule à l'aide de la fonction de hachage le serveur virtuel responsable de sa métadonnée. DROP crée ces MDS virtuels afin de réduire les

différents coûts de réorganisation (redistribution ou gestions des pannes) : si un serveur physique est trop chargé, il peut déléguer la gestion de MDS virtuels à d'autres machines physiques moins chargées. Pour calculer la charge d'un serveur physique, une approche basée sur des histogrammes de densité d'intervalles est utilisée : chaque serveur calcule son histogramme et échange périodiquement ses valeurs avec ses voisins. Si sa charge est deux fois supérieure à celle d'un de ses voisins, le dit voisin prend alors en charge un des MDS virtuels afin de rééquilibrer la charge de toutes les machines physiques.

1.3.3 Méthodes basées sur une structure centralisée

Ces méthodes de distribution se servent d'une structure de donnée centralisée afin de répertorier les associations métadonnées-serveurs. Cette structure est généralement une table d'index mais peut se trouver sous d'autres formes, tant qu'elle est centralisée, c'est-à-dire tant que toutes les modifications de distribution s'exécutent via cette structure. Lorsqu'un client veut accéder à une métadonnée, il calcule la clef correspondante et doit ensuite passer par une structure de données centralisée pour connaître le gestionnaire responsable de cette clef. Cette structure est mise à jour par les différents MDS mais est copiée chez le client pour certains systèmes. Le client a donc un accès rapide et n'effectue qu'une requête au service de métadonnées. Cependant, maintenir la structure de données à jour a un coût non négligeable qui augmente proportionnellement avec le nombre de clefs à gérer. C'est aussi un point unique de défaillance : si la structure n'est plus accessible, les modifications de distribution sont impossibles.

Le travail de S. Brandt et al. [BMLX03] associe les avantages des techniques de partitionnement en sous-arbres et de hachage pour concevoir une méthode pour les systèmes compatibles POSIX. Le fonctionnement hybride relâché utilise le hachage pour attribuer à chaque métadonnée une clef, puis cet ensemble de clefs est divisé en autant d'intervalles qu'il y a de MDS. Chaque serveur est alors associé à un intervalle de clefs via une table d'index. Cela permet d'équilibrer la charge de requêtes et ainsi éviter les goulots d'étranglements. Cependant, il garde la notion de hiérarchie des arbres du partitionnement par sous-arbres en stockant avec chaque métadonnée les droits d'accès. Cela permet d'alléger le nombre de requêtes pour un accès et d'éviter une surcharge élevée lors de modifications des localisations ou des permissions. De plus, ces modifications sont effectuées de façon relâchée, ce qui signifie de manière asynchrone quand un accès est demandé ou quand une faible charge des serveurs est constatée. Cela permet de répartir dans le temps la surcharge occasionnée.

L'accès aux métadonnées avec le fonctionnement hybride relâché est illustré par la figure 1.7. Pour accéder à une métadonnée, un client va calculer la clef correspondant à sa donnée, consulter la table d'index pour déterminer le MDS responsable de cette clef, puis lui adresser directement une requête. Cette table d'index permet d'accéder en une requête au bon gestionnaire, mais c'est aussi un goulot d'étranglement potentiel puisque la table est globale et accessible par tous les serveurs et clients sans être distribuée. Garder cette table d'index cohérente induit aussi un surcoût en synchronisation pour le système. De plus, même si cette méthode gère les accès fréquents à un répertoire (grâce au hachage), les goulots d'étranglement dûs aux accès de métadonnées d'un objet très populaire restent présents.

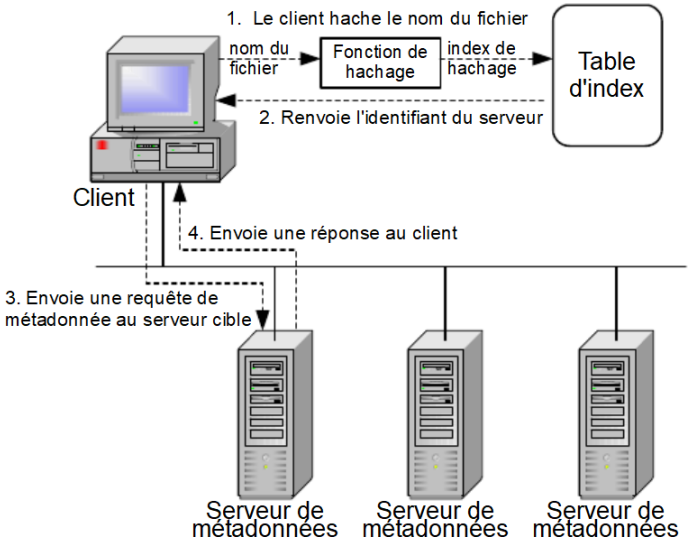


FIGURE 1.7: Accès à une métadonnée avec le fonctionnement hybride relâché d'après [BMLX03].

L'ajout d'une table d'index nécessite plus de communications liées à la synchronisation pour garder la table à jour. Toutefois, elle permet aussi de réduire de beaucoup le coût en réorganisation du système. En effet, lors de l'ajout ou la suppression d'un MDS, il n'y a pas besoin de déplacer des métadonnées, il suffit de faire le changement sur la table d'index en indiquant le nouveau serveur responsable de l'intervalle concerné.

Dans la méthode présentée par J. Wang et al. [WFWL09], une table d'index est utilisée pour répartir les métadonnées sur les différents serveurs tout en gardant

les avantages du partitionnement en sous-arbres. La méthode *Mimic the Hierarchical directory Structure* (MHS) utilise une fonction de hachage pour assigner à chaque répertoire un identifiant unique. Elle maintient une table d'index désignant le gestionnaire responsable de chaque identifiant et par extension de chaque métadonnée. Ensuite, elle simule une arborescence en créant une deuxième table d'index indiquant le MDS en charge d'un répertoire. Pour accéder à une métadonnée, un client doit se référer à la deuxième table d'index pour connaitre à partir du nom du répertoire le serveur qui en est responsable. Le dit gestionnaire calcule l'identifiant du répertoire grâce à la fonction de hachage et utilise ensuite la première table d'index pour connaitre le serveur responsable de la métadonnée. La demande du client est transférée au serveur responsable qui lui répond directement.

Le système de stockage proposé par J. Chen et al. [CDM+12] unifie plusieurs types de stockage fichier et objet. L'unification de plusieurs types de stockage dans un même système permet d'exploiter les ressources non utilisées par un service de stockage pour un autre système, mais rend la distribution plus difficile à gérer. Afin de garder une trace de cette distribution un registre global est maintenu grâce à *ZooKeeper* [HKJR10]. La distribution initiale s'effectue grâce à une fonction de hachage et un gestionnaire de partition manager est responsable de la gestion de la charge de chaque serveur. Afin d'accéder à sa métadonnée, un client doit consulter le registre qui lui indique le serveur à contacter.

La méthode conçue par W. Lin et al. [LWV07] sépare aussi l'emplacement physique des métadonnées et l'attribution de celles-ci aux MDS en utilisant une structure globale pour répartir les métadonnées sur les différents serveurs. La méthode *WPAR* (*Weighted Partitionning and Adaptive Replication*) distribue d'abord les métadonnées parmi plusieurs regroupements logiques appelés LUN (*Logical Unit Numbers*) en utilisant la technique du hachage universel (*universal hashing*). Cette méthode utilise une famille de fonctions de hachage très efficaces, causant très peu de collisions. Un conteneur global est partagé par tous les MDS et stocke l'ensemble des LUNs. Tous les MDS partagent le conteneur et accèdent uniquement et de manière exclusive aux regroupements dont ils sont responsables. Cela permet de réduire au maximum le coût de réorganisation en cas d'ajout ou de suppression de serveurs.

Une fois que les clefs sont associées à un regroupement, il faut distribuer la gestion de ces regroupements à chaque MDS et pour cela on utilise une variante du *B+tree* [CLRS09]. Les *B+trees* sont des arbres équilibrés et triés où seules les feuilles sont associées à des clefs. Les nœuds internes contiennent les intervalles permettant de retrouver les clefs. La particularité de cette variante est que l'on associe à chaque MDS un poids calculé en fonction de ses performances (puissance

du CPU, débit, mémoire disponible) et ce poids est pris en compte dans le calcul de la fonction de hachage pendant la construction du *B+tree*. Cette variante permet donc à des MDS avec un poids plus grand de gérer plus de métadonnées et ainsi équilibrer les requêtes par rapport à ce que chaque serveur peut supporter.

Le regroupement des métadonnées sous forme de LUNs, eux-mêmes stockés dans un conteneur partagé entre tous les MDS permet de ne pas réécrire les métadonnées en cas d'ajout ou de suppression d'un gestionnaire : une modification de l'attribution dans l'arbre *B+tree* est suffisante. En effet, si un MDS est ajouté, on va lui assigner un regroupement de métadonnées et son prédécesseur se verra retirer la gestion de ces LUNs. De la même manière, si un serveur est supprimé, d'autres MDS vont prendre en charge les regroupements logiques laissés, sans réécriture puisque le conteneur global est partagé entre tous les serveurs. Cependant, cette optimisation a un coût : savoir dans quel regroupement logique est la métadonnée recherchée, puis chercher le MDS responsable de ce regroupement implique un niveau d'indirection supplémentaire. Cela génère donc, pour chaque requête, un temps de calcul plus long qu'avec du hachage pur qui ne possède pas de niveau d'indirection.

La méthode WPAR permet aussi un comportement semi-dynamique afin de gérer les métadonnées très populaires. un système de réplication adaptatif prend en compte les performances des MDS sous forme de poids et la popularité de certaines métadonnées calculée à partir du nombre de requêtes pour celles-ci. Le poids d'un serveur est calculé à partir de la mémoire disponible, du débit réseau dont il dispose ainsi que de sa puissance CPU possible. Plus le poids d'un MDS est élevé, plus il peut gérer de métadonnées. Si un MDS gère une métadonnée qui devient populaire, c'est-à-dire que son nombre de requêtes dépasse un seuil défini, il demande alors à d'autres serveurs (avec les poids les plus importants) de répliquer cette métadonnée et les réplicas sont alors ajoutés dans la table d'index. Si l'accès à cette métadonnée redescend en dessous d'un certain seuil, les réplicas superflus sont supprimés et retirés de la table d'index. La définition de ces seuils de réplication est une contrainte importante au bon fonctionnement de la méthode.

La proposition de W. Li et al. [LXSZ06] est une méthode totalement dynamique qui redistribue l'espace de nommage en fonction des accès mesurés. Le hachage dynamique (*Dynamic Hashing*) utilise une table d'index afin de limiter les coûts de réorganisations qui deviennent conséquents avec une distribution dynamique : on modifie la table d'index plutôt que de déplacer les métadonnées. Pour équilibrer les charges des MDS, le hachage dynamique utilise une stratégie appelée RELAB (*RElative LoAd Balancing*). Celle-ci consiste à calculer la charge de travail de

chaque serveur en prenant en compte le nombre d'accès aux métadonnées et l'évolution de ce nombre dans le temps, ainsi que les capacités des MDS en terme de performances. Afin d'égaliser la charge des gestionnaires, RELAB distribue des intervalles de clefs gérées par des MDS en surcharge à des MDS moins occupés.

Le hachage dynamique prend aussi en compte la gestion des métadonnées populaires. Il détecte d'abord les métadonnées liées aux accès les plus fréquents grâce à un compteur associé à chaque métadonnée. Quand ce compteur atteint un seuil prédéfini, la métadonnée est déclarée comme populaire. Le MDS qui en est responsable demande à d'autres MDS de la répliquer afin que les clients puissent y accéder de façon concurrente. Le serveur reste responsable de la métadonnée et est chargé de transmettre les mises à jour aux réplicas. Une fois que le compteur d'accès diminue jusqu'à un seuil donné les réplicas sont supprimés et les accès clients sont réorientés vers le MDS principal.

1.3.4 Méthodes basées sur une structure distribuée

Ces méthodes utilisent pour garder en mémoire la distribution des métadonnées des tables d'index distribuées (*Distributed Hash-Table* ou DHT) dont la principale caractéristique est de distribuer leur structure, permettant à n'importe quel serveur de la modifier. Cela apporte une plus grande parallélisation des requêtes, mais pose de nouveaux problèmes sur la cohérence de cette structure.

Le système *Someta* de H. Tang et al. [TBD+17] utilise une DHT pour connaitre l'emplacement d'une métadonnée. Les entrées de cette table correspondent au hachage du nom de la métadonnée, indiquant alors l'emplacement des différentes métadonnées possédant le même nom. La particularité de *Someta* consiste à gérer rapidement des métadonnées avec le même nom : des attributs déclarés par l'utilisateur sont ajoutés à chaque métadonnée. Une fonction de hachage est ensuite utilisée avec en paramètre 4 attributs de la métadonnée afin d'obtenir un identifiant unique. La répartition des métadonnées sur les serveurs se fait à partir de l'identifiant unique de la métadonnée et une opération modulo, ce qui rend cette méthode statique, c'est-à-dire qu'elle ne change pas au cours du temps, puisqu'elle ne réalise pas de redistribution de la charge. L'accès à une métadonnée se fait via la table d'index en calculant le hachage du nom de la métadonnée. Il est aussi possible d'effectuer une recherche de métadonnées à partir d'un ou plusieurs attributs.

La méthode de D. Yang et al. [YWL+14] utilise une table d'index afin de stocker la distribution des métadonnées sur les différents serveurs. Celle-ci est globale et modifiée uniquement par le serveur d'index (*Metadata Index Server*) ou MIS : à chaque création de métadonnée, le serveur en charge envoie au MIS le répertoire parent dont il est responsable et le serveur d'index met alors à jour la table d'index.

La méthode *Peer-to-Peer Metadata Service* (PPMS) utilise le protocole pair-à-pair afin de réduire au maximum les consultations de la table d'index. Pour cela, chaque client va posséder un gestionnaire local responsable des métadonnées créées par le client et une liste de MDS voisins. Initialement cette liste est vide et à chaque accès à une métadonnée inconnue du serveur local, celui-ci va questionner le MIS. Le serveur d'index lui indique le serveur responsable du répertoire demandé et le serveur local ajoute alors le serveur dans sa liste de voisins avant de transmettre la requête. Ainsi, la liste des voisins de chaque MDS local se développe en fonction des accès demandés par le client. Cela permet à chaque gestionnaire de posséder sa propre table d'index contenant les voisins et de ne stocker que les informations dont son client a besoin.

Cassandra [LM10], utilisé par Facebook, adopte un comportement caractéristique des systèmes pair-à-pair, où tous les serveurs sont égaux et jouent un rôle de coordinateur pour une requête. Pour la gestion des métadonnées, Cassandra s'est inspiré du protocole CHORD [SMK+01]. L'espace de nommage est visualisé sous forme d'anneau et distribué sur les MDS grâce au hachage cohérent. Ensuite, tous les serveurs construisent une liste de serveurs connus qui leur est propre appelée *finger table*. Les gestionnaires dans cette liste sont les serveurs responsables de clés particulières calculées mathématiquement [3]. Pour accéder à une métadonnée, un client va contacter le serveur le plus proche. S'il est responsable de la métadonnée, il répond au client directement, sinon il consulte sa *finger table* et transmet la demande au MDS en charge s'il le connaît ou au serveur le plus proche qu'il connaisse si le hash demandé n'est pas couvert par sa liste de paires. Ce mécanisme est répété jusqu'à ce que la requête arrive sur le serveur responsable de la métadonnée demandée. Afin de toujours garder à jour la *finger table*, un protocole épidémique (*gossip protocol*) est mis en place : toutes les secondes chaque gestionnaire échange ses informations avec les autres serveurs.

1.4 Gestion spécifique des accès aux métadonnées

Le choix de la méthode distribution influe principalement sur la répartition des requêtes sur les différents serveurs mais a aussi un impact sur les protocoles d'accès aux métadonnées en terme de temps d'accès aux métadonnées et de gestion de la cohérence de celles-ci. Certaines méthodes peuvent être plus adaptées à différents niveaux de cohérence, améliorant les temps d'accès avec des requêtes moins contraignantes. L'utilisation de certaines structures de données ou bien le choix entre la réplication et la mise en cache de ces structures vont permettre

3. La liste des serveurs est calculée suivant la formule $n + 2^i mod(2^m)$ avec n la valeur de référence du serveur, i l'index de la *finger table* et 2^m l'espace de nommage.

d'accélérer les accès ou au contraire les rendront plus longs. Ce dernier point ne sera pas détaillé dans ce manuscrit, mais une étude comparative à ce sujet a été menée [XRZG15].

1.4.1 Filtres de Bloom

Plus le nombre de métadonnées gérées par un serveur augmente, plus le traitement des requêtes peut être long. Par exemple pour vérifier qu'une métadonnée existe, le serveur doit lister tous les éléments dont il est responsable et vérifier un par un s'il correspond à la métadonnée souhaitée. Pour palier cette limite, certaines approches exploitent des structures de données probabilistes appelées filtres de Bloom[Blo70]. Ces structures se basent sur des fonctions de hachage et permettent de savoir avec certitude qu'un élément est absent d'un ensemble et avec une certaine probabilité qu'il y est peut-être. Cela permet d'éviter en grande partie le coût de recherche lorsque la métadonnée demandée n'existe pas :

- Si le filtre de Bloom indique que la métadonnée n'existe pas, le serveur répond directement, évitant ainsi le coût lié à la vérification des éléments un par un ;
- Si le filtre de Bloom affirme que la métadonnée existe, le serveur va alors effectuer une recherche élément par élément. Il est possible que, finalement, le serveur ne trouve pas la métadonnée ; ce cas est appelé faux positif.

Une autre propriété du filtre de Bloom est sa taille constante qui ne dépend pas du nombre d'éléments dans l'ensemble. Cependant, le nombre de faux positifs augmente avec le nombre d'éléments, comme pour toute fonction de hachage. Le filtre de Bloom ne croît pas avec le nombre de métadonnées à stocker sur un serveur, ce qui permet de ne pas être une limite à l'extensibilité du système.

Plusieurs adaptations des filtres de Bloom ont été proposées afin de diminuer la probabilité de faux positifs lorsque le nombre de métadonnées pour un serveur augmente, comme dans les *Scalable Bloom Filters* [ABPH07]. L'idée principale consiste à ajouter un nouveau filtre de Bloom à chaque fois que le précédent est rempli, c'est-à-dire quand la probabilité de faux positif atteint un certain seuil. Le coût de vérification pour une métadonnée sera évité avec une probabilité donnée en échange du coût mémoire que représente l'ajout de plusieurs filtres de Bloom. Cependant, la taille des filtres ajoutés augmente par rapport aux précédents, ce qui permet de réduire le nombre de structures utilisées.

L'étude de Y. Zhu et al. [ZJWX08] s'inspire de la gestion hiérarchique du stockage (HSM) des systèmes de fichiers et hiérarchise deux filtres de Bloom ou plus : le premier filtre est utilisé pour les métadonnées fréquemment accédées et est souvent mis à jour. Le second filtre est plus grand et représente l'ensemble des fichiers

du système, mais est moins précis car mis à jour moins souvent. Lors d'une requête sur une métadonnée, le premier filtre de Bloom indique ou non l'existence de la métadonnée dans son filtre. Si le premier filtre trouve une seule et unique métadonnée correspondante, il répond directement. Par contre, Si le premier filtre trouve zéro ou plusieurs métadonnées, la réponse est considérée fausse et le second filtre est alors questionné. Si le second filtre donne aussi une réponse fausse, une demande est envoyée à tous les serveurs. Les *Hierarchical Bloom Filter Arrays* permettent une réponse rapide pour des métadonnées souvent accédées, et une réponse plus lente si la métadonnée demandée est peu fréquemment accédée. Une réponse fausse du deuxième filtre induit cependant un fort coût en communication.

Une adaptation des filtres de Bloom hiérarchiques appelé *Grouped-Hierarchical Bloom Filter Arrays* est donnée par Y. Hua et al. [HZJ+08] et tente de réduire les coûts lors d'une réponse fausse. Les serveurs sont regroupés et si une réponse fausse arrive d'un filtre, la demande est envoyée à tous les serveurs du groupe. C'est seulement si tous les serveurs du groupe donnent une réponse fausse que la demande est envoyée à tous les serveurs du système. L'ajout des groupes de serveurs permet de réduire le nombre de demandes transmises à l'ensemble des serveurs, réduisant ainsi les coûts de communication lors d'une réponse fausse.

1.4.2 Modification et cohérence des métadonnées

Un autre point important à prendre en compte dans la conception d'un service de métadonnées est la gestion des modifications des métadonnées, et plus particulièrement celles liées à l'espace de nommage comme les changements de permissions, l'ajout, la suppression ou le renommage d'éléments dans cet espace de nommage (éléments seuls ou une partie de l'espace de nommage). Ces modifications sont des opérations sensibles car elles doivent respecter le niveau de cohérence du système à chaque instant. Il est possible de catégoriser les méthodes selon les deux niveaux de cohérence (cohérence forte ou faible).

Méthodes assurant une cohérence forte

Une cohérence forte requiert une synchronisation globale du système, afin que chaque MDS ait le même espace de nommage à tout instant.

Pour cela certaines méthodes empêchent une partie des accès concurrents. Le partitionnement en sous-arbres verrouille les accès aux métadonnées dans le MDS concerné pour éviter une quelconque modification concurrente. La méthode de hachage utilise le même système de verrous, cependant si le changement intervient sur une métadonnée nécessaire pour calculer la fonction de hachage, alors il faut

recalculer la fonction et réorganiser si besoin. Pour effectuer une modification, *ShardFS* [XRZG15] verrouille tous les gestionnaires puisqu'ils ont tous un réplica de l'espace de nommage. Cette synchronisation globale est coûteuse en communication mais elle permet de garantir une cohérence forte pour les métadonnées.

D'autres méthodes permettent une cohérence forte, comme *BatchFS* [ZRG14] qui opère différement selon le contexte. Pour les accès concurrentes et totalement indépendantes, une mise à jour des MDS concernés est effectuée. Pour des accès concurrents à une même métadonnée (ou un même regroupement), un fonctionnement relâché est adopté. Ainsi, chaque client en concurrence possède une copie locale de l'espace de nommage où il effectue ses changements. Une fois qu'il a terminé, il transmet à des serveurs dits «auxiliaires» les modifications ainsi qu'une exécution valide prouvant sa validité, afin de fusionner les modifications avec les MDS principaux. Ce fonctionnement avec des serveurs auxiliaires permet d'alléger la charge des gestionnaires principaux responsables de l'espace de nommage tout en gardant un niveau de cohérence élevé.

Certaines méthodes tentent d'adapter la distribution des métadonnées à des systèmes centralisés tout en gardant une cohérence forte. *HopsFS* [NIG+16], une adaptation pour *Hadoop Distributed File System* (HDFS) [Bor08] a été proposée afin de distribuer les métadonnées en gardant le même niveau de cohérence pour les métadonnées que le système initial. Cette méthode distribue ses métadonnées de manière statique et hiérarchique en prenant en compte les accès fréquents connus lors de la distribution dans une table de base de données *NewSQL* [Sto12].

Les modifications effectuées par un client sont regroupées en transactions afin de diminuer le nombre de requêtes : le client commence une transaction, met en cache l'espace de nommage, modifie ce qu'il souhaite et soumet ensuite sa transaction au service de métadonnées. Les gestionnaires responsables de ces métadonnées vont ensuite verrouiller les lignes de la table concernées dans un ordre total défini de la même manière pour tous les MDS, pour finalement effectuer les modifications. Si la transaction concerne un répertoire volumineux, *HopsFS* utilise un protocole de verrous distribués parmi les sous-arbres du répertoire appelé *Subtree Operation Protocol* afin de diviser la charge en plusieurs transactions tout en gardant l'atomicité globale de la transaction. Disposer d'un ordre total dans la prise des verrous permet d'éviter tous les états bloquants. Si le système est compatible POSIX et considère son espace de nommage en arbres, l'ordre de prise des verrous est alors hiérarchique.

Une autre adaptation de services de métadonnées a été conçue par D. Stamatakis et al. [STSM12] et est destinée aux systèmes de stockage centralisés qui ne

possèdent pas de service de métadonnées répliquées, comme *PVFS* (Parallel Virtual File System) [RT+00] et *HDFS*. Celle-ci permet de répliquer les métadonnées en gardant une haute disponibilité et sans réellement dégrader les performances du système. Les modifications de l'espace de nommage doivent passer par tous les MDS car les métadonnées sont répliquées chaque modification est effectuée via un système de résolution de consensus comme *Paxos* [Lam98]. Ce système est coûteux en synchronisations mais il permet une cohérence forte pour tous les réplicas.

Méthode garantissant un niveau de cohérence relâché

Certains systèmes décident de ne pas surcharger les MDS avec le maintien d'une cohérence forte pour les métadonnées. C'est le cas de *DeltaFS* [ZRG+15] qui propose une gestion des métadonnées par vues individuellement cohérentes, c'est-à-dire que chaque client voit un espace de nommage qui lui est propre et qu'il n'y a pas de cohérence globale pour les métadonnées. Cela permet de ne pas avoir de MDS dédié à une partie de l'espace de nommage afin que tous les gestionnaires soient utilisés au maximum pour les requêtes en cours. Pour que tous les gestionnaires puissent gérer n'importe quelle vue, l'ensemble des sauvegardes de chaque vue est partagé.

Pour garder une cohérence par vue lors d'accès concurrents à la même vue, deux cas sont possibles. Dans le premier, tous les accès sont indépendants et on sait par avance qu'il n'y aura pas de conflit. On peut alors autoriser les exécutions en parallèle et les regrouper en une seule sauvegarde. Dans le deuxième cas, il est connu que des conflits vont se produire car les accès en concurrence modifient les mêmes métadonnées. On fait alors appel à des serveurs auxiliaires pour avoir un accès synchrone aux métadonnées qui sont temporairement distribuées dynamiquement sur les serveurs auxiliaires. Différencier ces deux exécutions permet de paralléliser au maximum les accès, tout en gardant une cohérence forte par vue.

1.5 Conclusion

La distribution de l'espace de nommage des métadonnées est un facteur important de la performance d'un système de stockage. Une distribution non adaptée cause des déséquilibres de charges pour les serveurs et cela induit des goulots d'étranglements. Nous plaçons notre étude dans un contexte HPC utilisant des systèmes exaflopiques et cela implique un flux de requêtes de métadonnées soutenu. Il est donc nécessaire de garder les serveurs équilibrés afin de ne pas ralentir le traitement des requêtes.

Les méthodes statiques comme le partitionnement par sous-arbres se retrouvent vite en situation de déséquilibre si l'accès aux métadonnées n'est pas uniforme. Le hachage comme dans *Someta* [TBD+17] est plus efficace, cependant si un fort déséquilibre est constaté, il ne sera pas capable de s'adapter. La méthode de B. Cai et al. [CXZ07] propose un placement intelligent des métadonnées selon la charge des serveurs sans autoriser de redistribution des charges. La méthode WPAR [LWV07] tente de réguler la charge des serveurs en répliquant de plus en plus les métadonnées lorsqu'elles sont souvent accédées. Cela implique un maintien de la cohérence entre tous les réplicas coûteux car le nombre de réplicas augmente mais cela ne garantit pas une baisse de la charge des serveurs.

Les méthodes dynamiques semblent donc plus indiquées pour répondre aux attentes d'un flux de requêtes de type HPC. Les méthodes dynamiques se basant sur le partitionnement en sous-arbres, comme celle proposée par S. Weil et al. [WPBM04], sont plus adaptées à un système de stockage de type fichier et rendent les redistributions coûteuses. D'autres méthodes dynamiques utilisent des redistributions de charge périodiquement comme W. Li et al. [LXSZ06] ou Q. Xu et al. [XAYM14]. Cela entraine un coût de redistribution régulier qui n'est pas forcément nécessaire ou qui peut ne pas suivre l'évolution du flux de requêtes. D'autres méthodes proposent des redistributions déclenchées selon un seuil basé sur l'espace mémoire utilisé et le coût en CPU [XZ12] ou bien par rapport au nombre de métadonnées stockées [XAYM14]. Cependant, ces mesures ne sont pas forcément un indice de la charge réel d'un serveur, ce qui peut fausser l'équilibrage des charges de travail des différents serveurs.

Il n'apparaît pas, d'après cette étude, de méthode capable de garder un équilibrage de la charge de travail des différents serveurs lorsqu'ils sont soumis à un flux de requêtes associé à un système HPC exaflopique.

Chapitre 2

Vers une méthode de distribution adaptative à la charge

Notre but est de construire une méthode de distribution adaptant la répartition des requêtes selon l'évolution de la charge des serveurs. Cette méthode doit permettre de gérer des flux de requêtes de métadonnées soutenus et qui évoluent au cours du temps, que ce soit en nombre total de requêtes reçues ou bien en terme de répartition de charge sur les différents serveurs.

Pour cela, nous partons de deux méthodes de l'état de l'art comme références, une statique et une dynamique, et analysons leurs limites sur les flux de requêtes à considérer pour proposer deux nouvelles méthodes de distribution adaptatives à la charge :

1. La méthode *Distribution adaptative à la charge*, ou *LAD* pour *Load-adaptive Distribution* ;
2. La méthode *Distribution adaptative à la charge avec fenêtre temporelle*, ou *LAD-TW* pour *Load-Adaptive Distribution with Temporal Window* qui est une évolution de la méthode *LAD*.

Nous commençons ce chapitre par présenter brièvement les méthodes de références choisies.

2.1 Choix d'une méthode statique de référence

La première méthode retenue est une méthode de l'état de l'art largement utilisée pour évaluer les différentes méthodes. Notre choix se porte sur le *Static Hashing* qui fait partie des méthodes basées sur une fonction de hash [BMLX03, HM04, WPBM04, DHJ+07] et qui permet une meilleure base de comparaison de par sa simplicité

2.1.1 Caractérisation du *Static Hashing*

La méthode du *Static Hashing* utilise une fonction de hachage afin d'assigner à chaque métadonnée un serveur en fonction du nom de l'objet associé. La fonction est définie avec comme ensemble d'entrée les noms possibles et comme ensemble de sortie les différents serveurs disponibles. Dans cette étude nous avons choisi la fonction de hachage *MurMur3* pour l'aléatoire de sa distribution : il n'est pas possible de prédire à partir des motifs dans la distribution le résultat de la fonction de hachage pour une clé donnée. Ainsi, il y a une plus forte probabilité d'obtenir des charges équilibrés pour les serveurs. C'est aussi une fonction rapide avec très peu de collision. Pour connaitre le serveur assigné à une métadonnée, chaque client calcule le résultat de la fonction de hachage correspondant à la métadonnée demandée. Les serveurs acceptent toutes les requêtes qu'on leur envoie. Ils n'ont pas besoin de savoir quels sont les métadonnées dont ils sont responsables puisque le choix du serveur se fait du côté client et ne changera pas.

2.1.2 Étude des limites

Les limites théoriques de la méthode du *Static Hashing* ont déjà été discutées et validée (voir la section 1.3.2). Elle permet une distribution initiale généralement bien répartie, permettant d'équilibrer la charge de chaque serveur, hormis quelques cas de *hot spots* où la distribution initiale est mise en défaut. La méthode étant statique, dans les cas où un déséquilibre apparaît, aucun algorithme de rééquilibrage n'est effectué, laissant ainsi le déséquilibre s'intensifier, possiblement jusqu'à la saturation du système.

2.2 Choix d'une méthode dynamique de référence

Afin d'empêcher un ralentissement du traitement des requêtes en cas de déséquilibre, nous avons opté pour une méthode dynamique et nous avons sélectionné le *Dynamic Hashing* proposé par Li et. al [LXSZ06] pour plusieurs raisons. Premièrement, cette méthode se base sur une fonction de hachage, comme la méthode statique. Cela nous permet de faire une comparaison juste et ainsi isoler les gains apportés par la dynamicité. Ensuite, nous avons fait le choix d'une méthode basant sa redistribution sur le nombre de requêtes reçues au lieu du nombre d'objets stockés ou encore le taux d'utilisation CPU des serveurs, car la capacité à traiter des requêtes d'un serveur nous parait la meilleure mesure pour évaluer une charge de travail. Finalement, nous avons pris une méthode dont les détails d'implémentation étaient explicités et clairs afin de restituer fidèlement son fonctionnement.

2.2.1 Caractérisation du *Dynamic Hashing*

La méthode du *Dynamic Hashing*, proposé par Li et. al [LXSZ06] utilise une table d'index appelée *Metadata Lookup Table* (MLT), afin de déterminer le serveur responsable d'une métadonnée. Pour cela, la table d'index est découpée en N_{entry} entrées et chaque entrée est assignée à un serveur. La MLT possède un autre champ qui est le numéro de la version la plus récente de la répartition pour l'entrée. Initialement, la distribution se fait grâce à une fonction de hachage et chaque numéro de version est positionné à la valeur 0. Les tables d'index sont utilisées du côté serveur afin de garder une cohérence dans la distribution des métadonnées mais aussi du côté client, puisque chaque client construit sa propre MLT qui va lui permettre d'accéder au bon serveur.

C'est une méthode dynamique qui redistribue périodiquement certaines entrées de la MLT selon la charge des serveurs. Pour déterminer la charge d'un serveur, un compteur d'accès par entrée est tenu et une mesure de charge est calculée en fonction de ces accès. Cette charge est pondéré par un facteur d'oubli afin de prendre en compte les accès passés.

Pour mesurer la charge globale d'un serveur on utilise le nombre de requêtes traitées par le MDS pendant la période. Il est déterminé par la somme des charge de chaque entrée dont le serveur a la charge. Les valeurs de charge globale de chaque serveur permettent ainsi de vérifier l'équilibrage des charges. De plus, le *Dynamic Hashing* permet de prendre en compte des différences de performances entre les serveurs en attribuant à chaque serveur un poids d'ajustement. Ainsi un serveur avec une capacité de stockage plus grande sera en charge de plus de métadonnées.

À la fin d'une période, un algorithme de redistribution appelé *RElative LoAd Balancing* (RELAB), détaillé ci-dessous dans l'algorithme 1, est exécuté afin de réduire les différences de charges entre les serveurs.

La première étape consiste à classer les serveurs surchargés dans l'ensemble L (pour *Large load*) ou libres dans l'ensemble S (pour *Small load*) selon leur charge relative. La charge relative correspond à l'écart de la charge du serveur et la charge idéale. Il faut ensuite pour chaque serveur libre déterminer un ensemble C de serveurs surchargés qui vont lui céder leur surplus de charge. Il faut donc que la somme des charges des serveurs inclus dans l'ensemble de serveur C, appelé $SUM(C)$ se rapproche au maximum de la charge idéale du serveur libre. On se sert ensuite du tableau *target* pour indiquer à quel serveur libre, chaque serveur surchargé doit transmettre de la charge. La dernière étape consiste à déterminer les entrées qui seront transférées, mettre à jour la MLT en augmentant le numéro de version de celles-ci et finalement transfèrer les métadonnées à leur nouveau responsable.

Algorithme 1 Algorithme RELAB

▷Classification des serveurs
$L = \emptyset$ ▷Ensemble des serveurs surchargés qui doivent transférer de la charge
$S = \emptyset$ ▷Ensemble des serveurs libres qui doivent recevoir de la charge
pour tout serveur i **faire**
    Calcul des x_i tel que $\sum_{i=1}^{n} x_i = \sum_{i=1}^{n} charge_i$ et $\frac{x_1}{w_1} = \frac{x_2}{w_2} = \ldots = \frac{x_n}{w_n}$
    Calcul de la charge relative : $y_i = charge_i - x_i$
    si $y_i > 0$ **alors** ▷serveur chargé
        $L = L \cup i$
    sinon si $y_i < 0$ **alors** ▷serveur libre
        $S = S \cup i$
▷Redistribution des serveurs
pour tout serveur $A \in S$ **faire**
    Choix d'un ensemble C de serveurs dans L tel que
        $|y_A + SUM(C)| \leq |y_A + SUM(O)|$ pour $\forall\ O \subset L$
    pour tout $srv \in C$ **faire**
        $target[srv] = A$
    $L = L \backslash \{C\}$
    $S = S \backslash \{A\}$
    si $L = \emptyset$ || $S = \emptyset$ **alors**
        pour tout $srv \in L$ **faire**
            $target[srv] = srv$
        Break
▷Choix des entrées à transférer
pour $i = 0$ *to* N_{srv} **faire**
    si $y_i > 0$ && $target[i] \neq i$ **alors**
        Choix d'un ensemble set d'entrées géré par i tq y_{set} le plus proche de y_i
        pour tout $entry \in set$ **faire**
            $MLT[entry] = i$ ▷mise à jour de la MLT

Suivant la charge des serveurs, des redistributions de métadonnées sont effectuées au cours du temps. À chaque redistribution, la table d'index des serveurs est mise à jour et les numéros de version des entrées transférées sont incrémentés. Cependant, le client n'est pas averti de ces changements. Il le sera quand il demandera à un serveur qui n'est plus responsable d'une donnée associée à une entrée modifiée. Le serveur en question l'informera de l'identité du nouveau serveur en charge de cette entrée ainsi que du numéro de version le plus récent.

Ce système de mise à jour appelé *lazy update* permet de lisser les communications de mises à jour dans le temps.

Exemple :

On considère l'algorithme RELAB sur 6 serveurs qui ont reçu respectivement 24, 19, 29, 22, 10 et 16 requêtes lors de la dernière période. Nous calculons d'abord les charges relatives à chaque entrée. Par simplicité, nous choisissons de prendre en compte uniquement les requêtes reçues pour cette époque.

La première étape de l'algorithme de redistribution consiste à calculer la moyenne idéale que chaque serveur devrait recevoir : celle-ci est de 20 requêtes. Nous calculons ensuite les charges relatives pour chaque serveur : respectivement [+4 ; -1 ; +9 ; +2 ; -10 ; -4] comme illustré par la figure 2.1a. Nous catégorisons chaque serveur dans les ensembles L et S : $L = \{0, 2, 3\}$ et $S = \{1, 4, 5\}$.

L'étape suivante consiste à remplir les serveurs disponibles.

Nous cherchons pour le serveur 4, celui ayant le plus de disponibilité, un ensemble de serveurs dont la somme des charges relatives vaut 10 (l'espace libre du serveur 4).

→ Le serveur 2 possède une charge relative de 9, il doit donc transférer une partie de sa charge au serveur 4.

Nous recalculons les charges relatives des serveurs : [+4 ; -1 ; 0 ; +2 ; -1 ; -4] et passons au serveur suivant avec le plus de disponibilité, c'est-à-dire le serveur 5.

→ Le serveur 0 possède une charge relative de 4, elle est transférée au serveur 5.

Nous mettons à jour les charges relatives : [0 ; -1 ; 0 ; +2 ; -1 ; 0]. Puis, nous cherchons pour le serveur 1 un ensemble de serveurs dont la somme des charges relatives vaut 1 (espace libre du serveur 1).

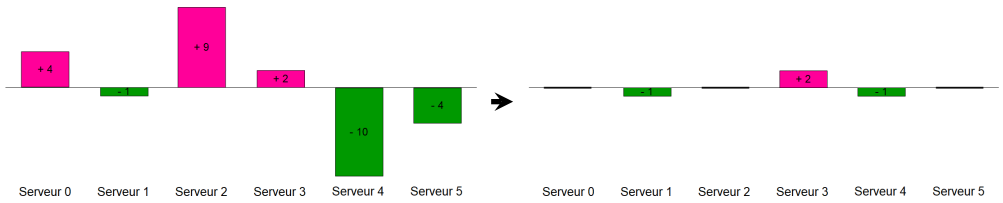
→ Il n'y a pas de serveur dont la charge est inférieure ou égale à 1.

Nous passons alors au serveur 4 car il reste des serveurs dans L et dans S.

→ Ici encore, il n'y a pas de serveur dont la charge est inférieure ou égale à 1.

Tous les serveurs de S ont été traités, l'algorithme s'arrête. Nous obtenons des charges relatives illustrées sur la figure 2.1b.

Une fois que les serveurs qui doivent transférer une partie de leur charge sont désignés, un ensemble d'entrées est selectionné pour être transféré pour chaque serveur surchargé.



(a) Avant redistribution. (b) Après redistribution.

FIGURE 2.1: Exemple : charges par serveur avant et après la redistribution.

2.2.2 Étude des limites

Le *Dynamic Hashing* est une méthode capable d'adapter la charge de travail au flux de requêtes, ce qui permet de gérer des flux qui mettaient en défaut la distribution initiale. Par rapport à la méthode du *Static Hashing*, cette méthode fait preuve d'une plus grande robustesse face à des flux mettant en difficulté la répartition initiale. Elle a par contre deux défauts : tout d'abord, la gestion de la table d'index induit un surcoût mémoire et elle doit être mise à jour régulièrement que ce soit côté client ou serveur ; ensuite, les redistributions se font à intervalle régulier sans analyser si elles sont nécessaires.

Ces redistributions peuvent aussi s'avérer inefficaces pour certains flux de requêtes, par exemple si les changements de flux sont plus fréquents que les rééquilibrages. Un cas problématique est celui où la majorité des requêtes sont dirigées vers un seul serveur alternativement, comme il est possible d'avoir si plusieurs utilisateurs chacun leur tour accèdent à leurs données. Sur l'exemple de la figure 2.2, lors de la redistribution, chacun des trois serveurs aura reçu le même nombre de requêtes et la redistribution ne changera rien à la répartition des accès, ce qui signifie que lorsque les utilisateurs reprendront leur même motif d'accès, chaque serveur sera de nouveau surchargé à tour de rôle.

Une autre limitation de cette méthode est l'algorithme RELAB car il ne permet pas de redistribution efficace dans le cas où un serveur est significativement plus chargé que les autres. Dans l'exemple développé précédemment, l'algorithme RELAB s'arrête sans pouvoir corriger les derniers déséquilibres. L'algorithme RELAB tente de trouver une charge serveur égale ou inférieure à sa disponibilité. Dans le cas où un serveur seul est surchargé, chacun des $N - 1$ serveurs libres cherchera un serveur dont la charge correspond à $1/N$ %. Comme il n'en existe pas, le calcul lié à la redistribution aura été inutile et ne permettra aucun transfert de métadonnées, laissant le serveur surchargé.

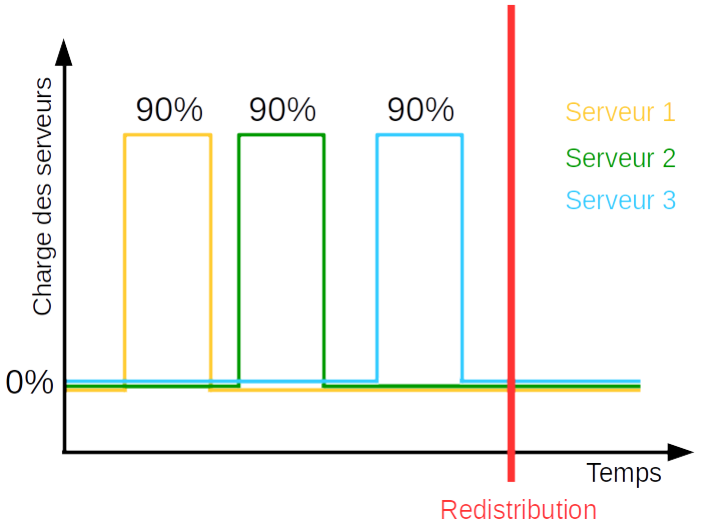


FIGURE 2.2: Exemple de flux problématique pour la méthode du *Dynamic Hashing*.

2.3 Méthode de distribution adaptative à la charge *LAD*

Afin de répondre aux problématiques exposées précédemment, nous proposons d'étendre la méthode du *Dynamic Hashing* [LXSZ06]. Nous avons décidé de prendre cette méthode comme point de départ pour concevoir notre propre méthode, appelée *Distribution adaptative à la charge* ou *LAD* pour *Load-Adaptive Distribution*, avec l'objectif qu'elle prenne en compte les cas la limitant. Nous avons gardé certaines structures et algorithmes du *Dynamic Hashing* et amélioré les points qui nous semblaient problématiques pour un flux très soutenu comme le flux HPC exascale, illustrés dans le chapitre 4.

Nous allons présenter ici de manière détaillée notre solution qui reprend la structure initiale de l'algorithme de *Dynamic Hashing* enrichie de mécanismes spécifiques à notre approche. Nous indiquerons de façon explicite ces ajouts dans la présentation de la méthode.

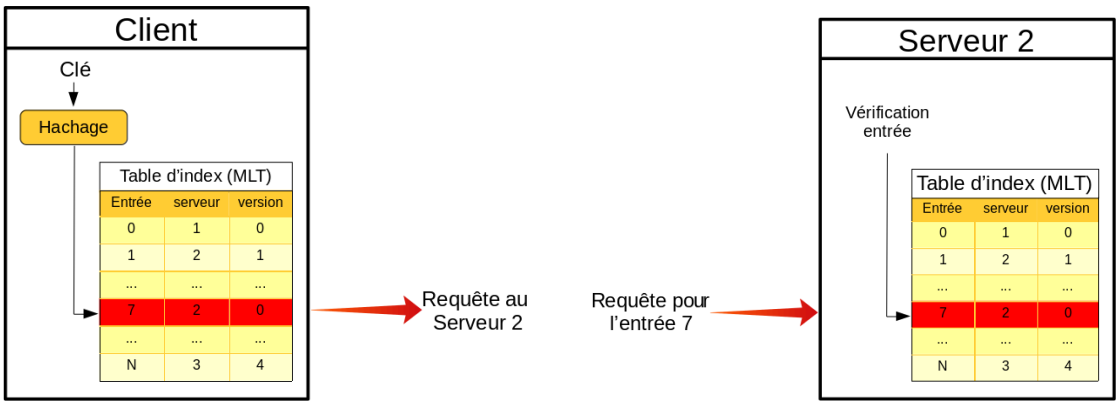
2.3.1 Structure fondamentale

Afin d'identifier le serveur responsable d'une métadonnée, deux étapes sont nécessaires. D'abord, une fonction de hachage est utilisée sur la métadonnée afin de définir un intervalle. Nous avons opté pour la fonction *MurMur3* [Sco11] pour le côté aléatoire de sa distribution. Cet intervalle est ensuite divisé en N_{entry} parties, chacune associée à une entrée dans la table d'index nommée *Metadata Lookup*

Table (MLT). La figure 2.3 illustre l'usage de la table d'index du coté client et du coté serveur :

- Pour le client, la table est utilisée afin de savoir quel serveur contacter, comme illustré par la figure 2.3a. Après avoir appliqué la fonction de hachage pour connaitre l'entrée accédée, le client se réfère à la table d'index afin de connaitre le serveur en charge.
- Pour le serveur, la table est utilisée afin de garder une cohérence dans la distribution des métadonnées et de vérifier que le serveur contacté est bien celui responsable de la métadonnée, comme illustré par la figure 2.3b.

La couche d'indirection que représente la table d'index, permet d'avoir une méthode dynamique avec des redistributions moins coûteuses : il n'est pas nécessaire de redistribuer la totalité de l'espace de noms mais seulement de modifier certaines entrées de la table d'index. De plus, chaque entrée de la MLT possède un numéro de version, permettant de mettre à jour seulement les entrées modifiées et non pas la table entière.



(a) côté client (b) côté serveur

FIGURE 2.3: Usage de la table d'index.

Nous avons choisi pour notre méthode de ne pas effectuer les rééquilibrages à fréquence régulière, mais seulement de les déclencher quand un serveur le demande. L'évaluation de la charge de chaque serveur reste cependant périodique et nous gardons le même procédé d'évaluation que dans le *Dynamic Hashing* : la charge d'un serveur est la somme des charges des entrées dont il est responsable et chacune de ces charges est calculée en fonction du nombre de requêtes que le serveur a reçues depuis la dernière évaluation. Chaque entrée de la MLT possède une structure de données appelée *Entry Access Counter List* (EACL) contenant un compteur d'accès (*access_count*) et une charge nommée *Synthetically Access*

Information (SAI). La charge SAI d'une entrée e est calculée à partir du compteur d'accès selon la formule 2.1.

$$SAI_t(e) = (1 - \alpha) * SAI_{t-1}(e) + \alpha * access\_count_t \qquad avec\ \alpha \in [0, 1]\ . \quad (2.1)$$

Le calcul d'une charge s'effectue suivant une moyenne mobile exponentielle : le calcul de la charge $SAI_t(e)$ au temps t prend en compte la valeur précédente $SAI_{t-1}(e)$ pour cette entrée et la pondère avec un facteur d'oubli α. Ce facteur rend les valeurs des charges passées plus ou moins importantes. Une valeur α proche de 1 donnera plus d'importance à la valeur du compteur d'accès que l'on vient d'obtenir et moins à la charge précédente. Cela permet de prendre plus en considération les variations immédiates de flux de requêtes. À l'inverse, une valeur α proche de 0 donne plus d'importance aux charges passées qu'aux accès présents. Cela permet de lisser la charge en étant peu influencé par les variations de flux de requêtes.

La méthode étant dynamique, la table d'index évolue donc au cours du temps. Contrairement aux serveurs qui possèdent une version toujours à jour de la MLT, un client peut avoir l'intégralité ou une partie de sa table d'index qui n'est plus à jour. Lorsqu'un client effectue une demande à un serveur qui n'est plus responsable de l'entrée de la table demandée, le serveur prévient le client que la demande ne lui est plus adressée et donne au client la nouvelle version de l'entrée demandée. Le serveur ne communique que la mise à jour de l'entrée demandée même si d'autres parties de la MLT du client ne sont plus à jour. Ce comportement appelé *Lazy Update* évite les mises à jour inutiles. Le client met à jour l'entrée expirée et envoie sa requête au nouveau serveur.

2.3.2 Redistribution à la demande

Selon notre méthode, une redistribution n'est exécutée que lorsqu'un ou plusieurs serveurs sont surchargés et le demandent. Cela permet de mieux gérer les périodes de rapide montée en charge, puisqu'il n'est pas nécessaire d'attendre le prochain rééquilibrage pour corriger la répartition de la charge de travail entre les serveurs. Pour permettre cette redistribution à la demande, il faut définir quand un serveur a besoin d'une redistribution de sa charge et comment la redistribution va se dérouler.

Nous avons choisi de mesurer la charge de chaque serveur périodiquement. Cela nous permet de contrôler la charge des serveurs de manière régulière afin de détecter au plus tôt une surcharge. Pour cela, nous introduisons la définition suivante :

DÉFINITION 2.3.1 (ÉPOQUE) : *Durée entre une évaluation et la suivante. Cette durée est configurable mais constante tout au long d'une exécution.* La charge d'un serveur correspond à la somme des charges SAI des entrées dont il est responsable et est basée sur le nombre de requêtes qu'il a reçues pendant la dernière époque.

Chaque serveur calcule sa charge pour l'époque actuelle et la compare à sa propre valeur d'un seuil R_{last} que le serveur ne doit pas dépasser pour rester équilibré et qui est donné par l'algorithme de redistribution lors du dernier rééquilibrage, comme illustré sur la figure 2.4. Le seuil R_{last} possède une valeur différente pour chaque serveur pour la même époque.

Lors d'une évaluation de charge pour l'époque i (en violet dans la figure 2.4), chaque serveur compare sa charge à sa valeur de R_{last}. Si une redistribution est demandée, un nouveau seuil est calculé ($R_{last} = Ri$) : R_i représente, pour chaque serveur, la charge idéale que le serveur aurait dû avoir pour l'époque i. Sa valeur peut différer d'un serveur à un autre car il est possible d'avoir des serveurs de capacités différentes. Ce nouveau seuil R_{last} est ensuite utilisé comme seuil pour la période $i + 1$ et garde la valeur R_i jusqu'à la prochaine redistribution.

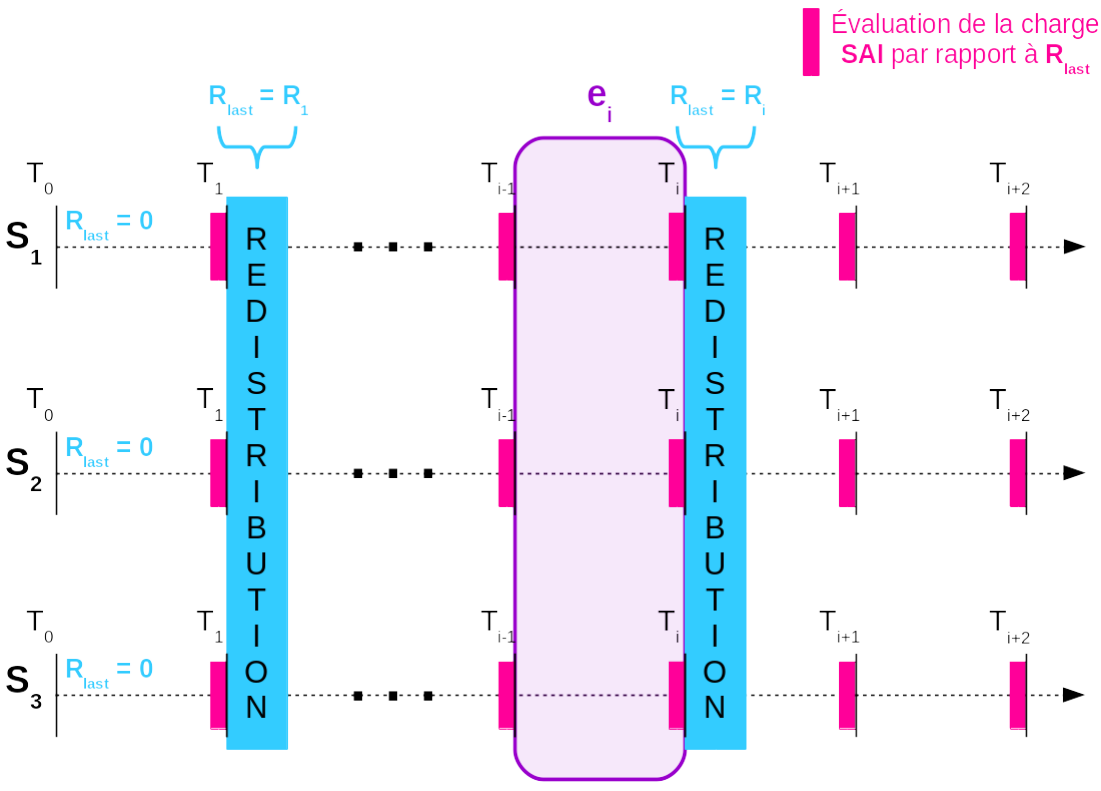


FIGURE 2.4: Procédé d'évaluation de la charge et de redistribution par époque.

Prendre comme seuil de redistribution la charge que ce serveur aurait dû avoir à l'époque précédente permet de suivre plus facilement le flux de requêtes. Tant que le seuil n'est pas atteint, nous pouvons considérer le flux de requêtes comme inchangé et il n'est pas nécessaire de modifier le seuil. À chaque fois que le seuil est dépassé, cela montre un fort changement dans le flux de requêtes. Un rééquilibrage permet alors de garder la distribution en accord avec le flux. Nous attribuons ensuite au seuil une nouvelle valeur afin de suivre le nouveau flux de requêtes.

Cependant, établir le seuil de surcharge à une valeur précise est une règle assez contraignante : obtenir pour chaque époque exactement le nombre de requêtes défini par le seuil pour chaque serveur provoquerait en pratique une redistribution à chaque évaluation. En effet, un niveau de finesse de l'ordre de la requête n'est pas nécessaire pour garder un flux équilibré, et il est possible de relâcher ce niveau de précision. Pour remédier à ce problème, nous avons ajouté à ce seuil une marge configurable, qui correspond à un pourcentage de R_{last} et qui définit un intervalle de tolérance ce charge $[min, max]$. Si le nombre de requêtes reçues lors d'une époque est en dehors de cet intervalle, alors une redistribution est demandée :

$$min = R_{last} - margin * R_{last}\,; \qquad max = R_{last} + margin * R_{last}\ .$$

À chaque époque, un ou plusieurs serveurs peuvent se considérer comme surchargés et faire une demande de redistribution. Nous appliquons alors l'algorithme de redistribution détaillé dans la section suivante et les serveurs concernés procèdent aux transferts des métadonnées qui en découlent. Si une entrée change de responsable, son numéro de version est incrémenté. Chaque serveur connaît aussi le nouveau seuil R_{last} pour la prochaine évaluation de charge.

2.3.3 Algorithme de redistribution

Lorsqu'une demande de redistribution est effectuée, l'algorithme 2 est appliqué afin de réduire au maximum les différences de charges entre les serveurs.

Nous commençons par la phase d'initialisation et créons tout d'abord une table nommée *target* avec autant d'entrées que la MLT en possède et initialisée à l'identité (1 à l'index 1, 2 à l'index 2 etc...). Cette table attestera des changements décidés par l'algorithme de redistribution. À la fin de l'algorithme, *target* correspond à la nouvelle table d'index et tous les serveurs doivent faire les transferts de métadonnées associés à ces changements. Nous créons aussi deux ensembles S (pour *Small*) et L (pour *Large*) qui servent pour la classification des serveurs selon leurs charges.

Algorithme 2 Nouvel algorithme de redistribution

▷Initialisation
pour tout entrée in MLT **faire**
    $target$[entrée] = serveur_actuel
$L = \emptyset$ ▷Ensemble des serveurs surchargés qui doivent réduire leur charge
$S = \emptyset$ ▷Ensemble des serveurs libres qui peuvent augmenter leur charge
▷Classification des serveurs
pour tout serveur i **faire**
    Calcul des x_i tel que $\sum_{i=1}^{n} x_i = \sum_{i=1}^{n} ALL_i$ et $\frac{x_1}{w_1} = \frac{x_2}{w_2} = \ldots = \frac{x_n}{w_n}$
    Calcul de la charge relative : $y_i = ALL_i - x_i$
    si $y_i > 0$ **alors** ▷serveur chargé qui doit se décharger
        $L = L \cup i$
    sinon si $y_i < 0$ **alors** ▷serveur libre qui peut être plus chargé
        $S = S \cup i$
▷Étapes de redistribution
Tri de L du serveur le plus chargé au moins chargé
pour tout serveur $A \in L$ **faire**
    Tri de S du serveur le plus libre au moins libre
    pour tout serveur $srv \in S$ **faire**
        ▷Choix d'un ensemble set d'entrées gérées par A tel que y_{set} le plus proche de $min\ (|y_{srv}|,\ y_A)$
        $set \leftarrow$ CHOIX_ENTREE$(0, entries_A, min\ (|y_{srv}|,\ y_A))$
        pour tout entrée $\in set$ **faire**
            $target$[entrée] = srv
        $y_{srv} = y_{srv} + y_{set}$
        si $y_{srv} \geq 0$ **alors**
            $S = S \backslash \{srv\}$
        $y_A = y_A - y_{set}$
        si $y_A \leq 0$ **alors**
            $L = L \backslash \{A\}$
            Break

Par la suite, une classification des serveurs est effectuée. Elle repose sur le calcul des charges de chaque serveur et nécessite les définitions suivantes.

DÉFINITION 2.3.2 (CHARGE ABSOLUE) : *Appelée aussi Absolute Load Level (ALL), elle correspond, pour chaque époque, à la somme des SAI des entrées dont le serveur est responsable.*

Si tous les ALL sont identiques ou proches, cela signifie que tous les serveurs ont la même charge et donc que la distribution est équilibrée.

DÉFINITION 2.3.3 (CHARGE IDÉALE) : *Elle correspond au seuil R_{last} du serveur, pondéré par le facteur de performance w_s.*

À partir de la somme de toutes les requêtes d'une époque i, nous calculons une moyenne qui représente la charge qu'un serveur aurait dû recevoir pour cette époque R_i. Cependant, tous les serveurs d'un même système peuvent avoir des performances différentes, nous avons alors un cluster hétérogène et la charge supportable pour chaque serveur peut varier. Cela est pris en considération par le facteur de performance w_s.

DÉFINITION 2.3.4 (CHARGE RELATIVE) : *Elle correspond à la distance entre la charge absolue d'un serveur et la charge qu'il aurait dû avoir pendant cette époque.*

Si tous les serveurs possèdent une charge relative proche de zéro, alors la distribution est équilibrée (tout en tenant compte des capacités de chaque serveur). Si la charge relative d'un serveur est positive, cela signifie que le serveur est surchargé d'autant et est donc placé dans l'ensemble L. À l'inverse, un serveur avec une charge relative négative est libre et est placé dans l'ensemble S.

Après l'initialisation et la classification des serveurs, l'étape de redistribution se charge de répartir la surcharge de chaque serveur de L sur les différents serveurs libres de S selon les tapes suivantes :

1. Tri de l'ensemble L du serveur le plus chargé au moins chargé.
2. Pour chaque serveur chargé A, recherche d'un surplus à donner au plus petit nombre de serveurs libres possible. Pour cela, Il faut trier l'ensemble S du plus disponible au moins disponible et considérer le premier serveur : srv.
3. Choix d'un ensemble *set* d'entrées dont A est responsable et dont la charge permet à un des deux serveurs (A ou srv) d'atteindre le seuil d'équilibre, c'est-à-dire que sa charge relative atteigne zéro. Le détail de l'algorithme permettant de choisir l'ensemble d'entrées est donné dans l'algorithme 3. C'est un algorithme glouton se basant sur le problème du rendu de monnaie [Cai09] et qui autorise de dépasser le seuil demandé pour un dernier élément si l'écart entre la valeur obtenue et le seuil est inférieur à l'écart obtenu sans ce dernier élèment. Les entrées choisies changent de responsable : A n'en est plus responsable au profit de srv.
4. Ajout des changements dans la table *target* et mise à jour des charges de A et de srv. Si un serveur atteint l'équilibre, il est retiré de son ensemble (l'ensemble L pour le serveur A et l'ensemble S pour le serveur srv).

5. Considération du serveur libre suivant, si le serveur A reste surchargé après avoir transféré une partie de sa charge à srv. Le nouveau serveur libre pourra recevoir un autre ensemble d'entrées du serveur A.
6. Tri de l'ensemble S à chaque changement de serveur surchargé A. Ainsi, il est possible d'avoir un minimum de transferts : le plus gros ensemble possible de A est déplacé dans le plus grand espace libre possible.

Algorithme 3 Algorithme glouton pour choisir un sous-ensemble d'entrées

Entrée:
1: $load_{current}$ - la charge de départ (zero),
2: $list$ - la liste de toutes les charges des entrées (celles du serveur surchargé),
3: $goal$ - la charge à atteindre (min ($|y_{srv}|$, y_A))
Sortie: $subset$ - l'ensemble d'entrées à transférer
4: **fonction:** CHOIX_ENTREE($load_{current}$, $list$, $goal$)
5: ▷Initialisation
6: $subset = \emptyset$
7: $Sum = load_{current}$ ▷charge évoluant au cours de l'algorithme
8: $last = -1$ ▷$last$ est le dernier élément que l'on n'a pas pu ajouter
9: ▷Étapes du choix d'entrées
10: Tri de $list$ par ordre décroissant
11: **pour tout** $element \in list$ **faire**
12: **si** $Sum + element \geq goal$ **alors**
13: **si** $element == 0$ **alors** ▷on ne transfere pas de charge nulle
14: Break
15: $Sum = Sum + element$
16: $subset = subset \cup \{element\}$
17: **sinon**
18: $last = index[element]$
19: ▷Si le dernier élément vu n'augmente pas l'écart de charge, on le rajoute
20: **si** $(last \neq -1)$ et $(absolue(Sum + list[last]) < absolue(Sum))$ **alors**
21: $subset = subset \cup list[last]$

En appliquant cet algorithme de redistribution, nous répartissons les surcharges sur un ensemble de serveurs libres. Cette vision prend en compte tous les cas de déséquilibres de charges possibles : une surcharge importante pourra, par exemple, être divisée et répartie sur plusieurs petits espaces libres ou être placée entièrement sur un seul espace libre de taille adéquate. De la même manière, plusieurs petites surcharges pourront être déplacées sur un seul large espace libre ou sur plusieurs petits espaces libres. La répartition commence par considérer les serveurs les plus

chargés, ce qui minimise le déséquilibre. Cet algorithme ne permet pas qu'un serveur de L reste significativement surchargé car il n'y a plus d'espace libre dû aux répartitions sur les serveurs précédents. Cela garantit aussi le plus petit nombre de transferts possibles, puisque nous remplissons d'abord les espaces libres les plus grands avec les charges les plus grandes. Cependant, cet algorithme ne garantit pas que l'ensemble de destinataires le plus réduit puisse être atteint pour un serveur chargé.

Exemple :

On applique cet algorithme de redistribution sur l'exemple de la section précédente avec 6 serveurs qui ont reçu respectivement 24, 19, 29, 22, 10 et 16 requêtes lors de l'époque précédente. Nous supposons, ici aussi, que l'espace de nom est divisé en 10, ce qui nous donne une MLT de 10 entrées avec une répartition des entrées sur les différents serveurs comme le montre la table 2.1. Nous calculons d'abord les SAI relatifs à chaque entrée. Par simplicité, nous choisissons la valeur de $\alpha = 1$, afin de prendre en compte uniquement les requêtes reçues pour cette époque. La structure EACL et la MLT associées sont sur la table 2.1.

Entrée	Serveur responsable	SAI = $acces\_count$	Serveur responsable après redistribution
1	0	20	0
2	0	4	5
3	1	19	1
4	2	20	2
5	2	9	4
6	3	1	1
7	3	1	4
8	3	20	3
9	4	10	4
10	5	16	5

TABLE 2.1: Exemple : EACL et MLT pour la méthode *LAD*.

La première étape de l'algorithme de redistribution consiste à calculer la moyenne idéale que chaque serveur devrait recevoir : celle-ci est de 20 requêtes. Nous calcu-

lons ensuite les charges relatives à chaque serveur : respectivement [+4 ; -1 ; +9 ; +2 ; -10 ; -4] comme illustré par la figure 2.5a. Nous plaçons chaque serveur dans les ensembles L et S : $L = \{0, 2, 3\}$ et $S = \{1, 4, 5\}$.

L'étape suivante consiste à répartir les surcharges. Nous cherchons pour le serveur 2, qui est le plus chargé, un ensemble d'entrées dont la somme des SAI vaut 9 (le surplus du serveur 2) ou 10 (l'espace libre du serveur le moins chargé, le serveur 4).

→ l'entrée 5 possède un SAI de 9, elle est donc transférée au serveur 4.

Nous recalculons les charges relatives des serveurs : [+4 ; -1 ; 0 ; +2 ; -1 ; -4] et passons au serveur suivant le plus chargé, c'est-à-dire le serveur 0.

→ L'entrée 2 possède un SAI de 4, elle est transférée au serveur 5.

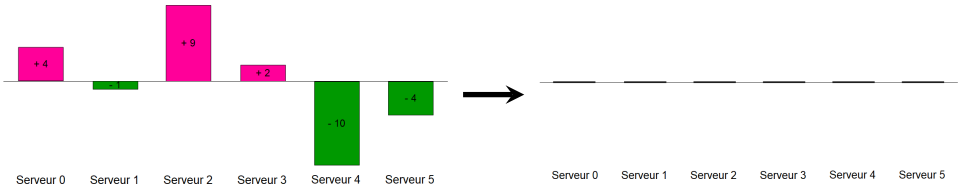
Nous mettons à jour les charges relatives : [0 ; -1 ; 0 ; +2 ; -1 ; 0]. Puis, nous cherchons pour le serveur 3 un ensemble d'entrées dont la somme des SAI vaut 2 (surplus du serveur 3) ou à 1 (espace libre du serveur 1).

→ L'entrée 6 possède un SAI de 1, elle est transférée au serveur 1.

Nous continuons avec le serveur 3 car sa charge relative est toujours positive.

→ L'entrée 7 possède aussi un SAI de 1, elle est alors transférée au serveur 4.

Nous mettons à jour les charges relatives : [0 ; 0 ; 0 ; 0 ; 0 ; 0]. Tous les serveurs de L ont été traités. Nous obtenons des charges relatives valant toutes 0 ce qui signifie que la charge de travail est équilibrée comme nous pouvons le voir sur la figure 2.5b. La table d'index est mise à jour avec les entrées transférées au cours de la redistribution et nous obtenons la répartition présentée dans la dernière colonne de la table 2.1.

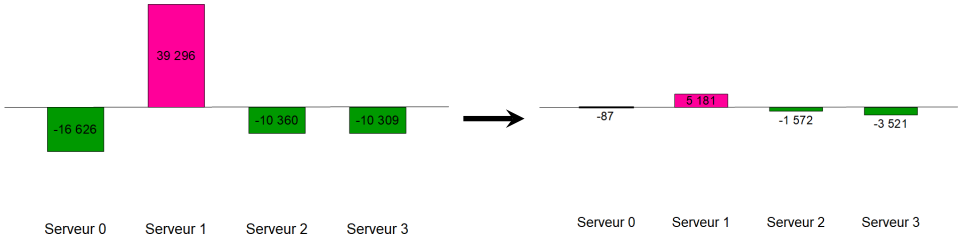


(a) Avant redistribution. (b) Après redistribution.

FIGURE 2.5: Exemple d'application de la méthode *LAD*.

Afin de donner une idée des valeurs des charges relatives lors d'une exécution dans un système réel, nous avons extrait un état possible d'une exécution de flux réel sur 4 serveurs avant et après l'algorithme de redistribution. Le calcul des charges relatives avant redistribution fournit les valeurs suivantes, montrant une surcharge du serveur 1 : respectivement −16626, +39296, −10360 et −10309. Après

application de l'algorithme de redistribution, nous pouvons observer la diminution des écarts de charges avec les valeurs suivantes : respectivement -87, $+5181$, -1572 et -3521. Ces charges sont illustrées par la figure 2.6 récapitulant l'impact de la redistribution.



(a) Avant redistribution. (b) Après redistribution.

FIGURE 2.6: Exemple d'application de la méthode *LAD* sur flux réel.

2.3.4 Étude des limites

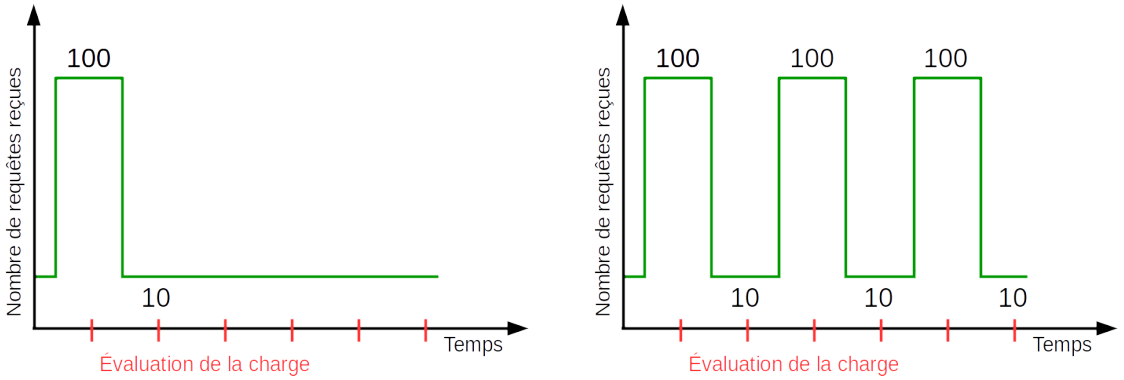
La méthode *LAD* diminue le nombre de redistributions inutiles en effectuant une redistribution quand un serveur en exprime le besoin. Cela permet aussi de gérer des flux variants fréquemment comme présenté sur la figure 2.2.

Toutefois, cette méthode possède des coûts notamment en espace mémoire qu'implique la table d'index du côté client et du côté serveur. Cependant, le coût en puissance de calcul est diminué par rapport à la méthode du *Dynamic Hashing*, puisque les redistributions ne sont pas effectuées à chaque époque. Il reste pour chaque serveur le coût de calcul de la charge. Le nombre de communications induites par l'algorithme de redistribution peut être supérieur à celui de la méthode précédente puisque la méthode *LAD* permet à un serveur de transférer sa charge vers plusieurs serveurs libres. Un surplus de communication pourrait être nécessaire afin de gérer les demandes de redistribution.

L'ajout d'un seuil pour chaque serveur peut aussi aboutir à des redistributions inappropriées. Tel est le cas lorsque le nombre de requêtes reçues par tous les serveurs augmente ou diminue significativement tout en gardant l'équilibre des charges, une redistribution sera demandée.

De plus, le système d'évaluation prend en compte tous les accès passés avec un facteur d'oubli. Cela signifie qu'une valeur de charge aura potentiellement un impact dans l'évaluation de toutes les charges futures, ce qui peut affecter le comportement de la méthode. En considérant le flux schématisé par la figure 2.7a, la méthode *LAD* exécutera 6 redistributions avant de se stabiliser (pour obtenir ce résultat nous avons arbitrairement défini les paramètres aux valeurs suivantes : $\alpha = 0.5$, $marge = 10\%$). Certains flux de requêtes très changeants, comme celui

schématisé par la figure 2.7b, pourraient donc mettre en défaut cette méthode : une redistribution serait demandée à chaque évaluation de la charge.



(a) Flux avec un pic de charges.

(b) Flux très changeant.

FIGURE 2.7: Exemple de flux mettant en difficulté la méthode *LAD*.

2.4 Variante avec fenêtre temporelle *LAD-TW*

Afin de gérer au mieux les flux identifiés précédemment, nous proposons une variante à la méthode *LAD*. Dans la version originale, l'évaluation de la charge de chaque serveur s'effectue en suivant la formule 2.1, reprise de la méthode initiale [LXSZ06]. Lors d'une évaluation, la totalité des accès passés sont pris en compte selon un facteur d'oubli, ce qui peut affecter le comportement de la méthode. Pour remédier à cela, nous proposons d'utiliser une fenêtre temporelle glissante qui permet de rendre visible et exploitable seulement une partie des accès.

2.4.1 Détection des limites de l'évaluation de la charge de la méthode *LAD*

L'évaluation précédente de la charge (formule 2.1) correspond à une moyenne mobile exponentielle et considère tous les accès passés en les pondérant avec un facteur d'oubli α. Ainsi, le nombre d'accès à un temps t sera pris en compte dans l'évaluation de toutes les charges futures, avec un poids de plus en plus faible.

Par exemple, considérons un flux léger avec un pic de charge à l'époque e_i, comme il a été schématisé sur la figure 2.7a. Les valeurs de la charge associées au moment de la rafale de requêtes auront un impact direct sur l'évaluation de la charge de l'époque e_i et une redistribution pourra être demandée. La charge de l'époque e_i sera ensuite comprise dans toutes les futures évaluations par inclusion

dans la charge suivante (voir formule 2.1), et d'autres redistributions pourront encore être demandées. Le nombre d'époques significativement impactées par la charge de l'époque e_i dépend de α. Au final, plusieurs redistributions auront été effectuées, sans que cela ne soit profitable pour le système : le flux redevenant léger après l'époque e_i, aucune des redistributions n'est nécessaire. Cependant la première permet de rééquilibrer le système dans l'hypothèse où le flux de requêtes resterait très important et ne peut donc pas être évitée. Les suivantes sont le résultat de la formule d'évaluation de la charge et le nombre de redistributions peut être réduit.

2.4.2 Nouvelle manière d'évaluer la charge

Nous proposons d'utiliser une fenêtre temporelle glissante précisant les accès pris en compte dans le calcul de charge. La taille de la fenêtre $size_w$ est configurable mais constante tout au long d'une exécution. Pour calculer la charge d'un instant t, nous effectuons la somme des accès de l'instant t et la moyenne des accès à des époques contenues dans la fenêtre glissante $[acces_{t-size_w} \dots acces_{t-1}]$. Nous conservons un facteur d'oubli α permettant de pondérer l'importance du passé par rapport aux requêtes de l'époque actuelle. La charge d'une entrée s'exprime donc avec la formule 2.2.

$$SAI_t(e) = \alpha * acces\_count_t + (1-\alpha) * (\frac{1}{size_w} * \sum_{i=1}^{size_w} acces\_count_{t-i}) \quad (2.2)$$

Une valeur α égal à 1 ne prendra en compte que les accès de l'époque actuelle, tandis que une valeur α proche de 0 prendra en compte uniquement la moyenne de la fenêtre temporelle. Pour avoir une moyenne uniforme entre toutes les valeurs des $acces\_count$, il faut fixer α à $1/(size_w + 1)$. De cette manière, l'évaluation de la charge peut se simplifier en la formule suivante :

$$SAI_t(e) = \frac{1}{size_w + 1} * \sum_{i=0}^{size_w} acces\_count_{t-i}.$$

Prendre en considération seulement une partie des accès passés permet de lisser plus vite les comportements atypiques et isolés et donc de provoquer moins de redistributions inutiles en s'adaptant au flux de requêtes, sans être influencé par un passé trop lointain. Plus la taille de la fenêtre temporelle est petite, plus la charge sera fidèle au flux actuel, mais plus elle sera sensible aux variations de flux. À l'inverse, une fenêtre de grande taille prendra en compte la tendance globale du flux de requêtes, mais aura un fort impact dans les évaluations futures pendant longtemps.

2.4.3 Étude des limites

La méthode *LAD-TW* permet une évaluation de la charge d'un serveur moins affectée par les actions lointaines et donc s'adapte au flux présent plus rapidement. Cela se traduit par un plus petit nombre de redistributions demandées suite à un bouleversement du flux de requête. Par exemple, pour le flux composé d'un court moment de forte charge suivi d'un flux faible constant schématisé par la figure 2.7a, la méthode *LAD-TW* permet de stabiliser le système au bout de 2 redistributions. Pour obtenir ce résultat nous avons arbitrairement défini les paramètres aux valeurs suivantes : $\alpha = 0.5$, $marge = 10\%$, $size_w = 4$ époques. De la même manière, il lui faudra 2 redistributions avant de s'adapter au flux très changeant schématisé par la figure 2.7b.

Cette méthode implique un coût en mémoire pour les serveurs, puisqu'ils doivent maintenir un tableau regroupant le nombre d'accès de chaque époque de la fenêtre, mais ne change pas significativement le coût de calcul de l'évaluation de la charge.

Il reste toutefois certains cas d'utilisation qui ne sont pas encore pris en compte par les méthode *LAD* et *LAD-TW*. La gestion des métadonnées par entrée de la table oblige le regroupement indissociable de certaines métadonnées. Ainsi, de forts accès à ce groupe de métadonnées génèrera une surcharge sur un serveur et un déséquilibre qui pourra être comblé. De la même manière, l'accès à une même métadonnée fortement demandée produira une forte surcharge ne pouvant être répartie sur différents serveurs.

2.5 Conclusion

Le *Static Hashing* et le *Dynamic Hashing* ont permis de construire pas à pas notre méthode adaptative à la charge. L'étude théorique des limites de ces deux méthodes ont permis de proposer notre méthode *LAD*. La méthode *LAD-TW* a ensuite été présentée afin de proposer une distribution adaptative à la charge se basant sur les accès enregistrés pendant la fenêtre temporelle glissante. Notre conception de cette méthode permet de gérer les cas de fort déséquilibre.

Chapitre 3

MeDiE : outil pour l'évaluation de méthodes de distribution

Ce chapitre présente *MeDiE* (***Me****tadata* ***Di****stribution* ***E****valuator*), un logiciel à code source ouvert développé au cours de cette thèse pour évaluer les différentes méthodes de distribution que nous avons proposées et les comparer à des méthodes de l'état de l'art. En effet, comparer différentes méthodes de distribution nécessite un simulateur permettant à la fois de modéliser aisément des flux d'entrée, de décrire et intégrer des méthodes de distribution et de configurer un service de métadonnées. Nous avons donc conçu les différents modules de cet outil :

- Un simulateur de requêtes entre les clients et les serveurs permettant de changer de méthode de distribution facilement ;
- Un protocole de simulation afin d'obtenir une gestion du temps modulable et d'adapter les traces à notre simulateur ;
- Ainsi qu'un générateur de traces, afin d'évaluer nos méthodes en considérant différents types de flux en entrée. Il a aussi été nécessaire de définir les différents flux de métadonnées à tester.

Dans ce chapitre, la section 3.1 détaille les motivations et le contexte dans lequel se place notre outil. Nous introduisons ensuite dans la section 3.2 le module permettant de simuler des interactions clients/serveurs et de changer la méthode de distribution. La section 3.3 présente le processus de simulation et la section 3.4 montre l'intégration dans *MeDiE* des méthodes de distribution présentées dans le chapitre 2. Finalement, la section 3.5 expose le module de génération de traces et présente les différents flux et scénarii pour évaluer des méthodes de distribution dans notre contexte.

3.1 Expression du besoin

La première étape consiste à établir les besoins et exprimer clairement dans quel contexte se place *MeDiE*, son rôle et les fonctionnalités qu'il doit garantir.

Nous souhaitons évaluer des méthodes de distribution équitablement, en les soumettant au même protocole d'évaluation. Pour cela, des outils d'évaluation comme *MDtest* [MLMK11] ou *PostMark* [Kat97] permettent de tester les performances des systèmes de type fichiers. Ils sont utilisés, par exemple par Xing et al. [XXSM09] ou encore par Yang et al. [YWL+14] si la méthode respecte les normes POSIX. Cependant, de nombreuses méthodes de distribution sont détachées de ces normes et ces outils ne sont alors pas appropriés. D'autres outils existent pour des systèmes de stockage objet, comme *COSBENCH* [ZCW+12] qui permet d'évaluer les performances globales d'un système. Comme ce n'est pas un outil spécialisé dans l'évaluation d'un service de métadonnées, il ne peut pas fournir d'analyses propres à la distribution des métadonnées, comme la charge de travail des différents serveurs. Le manque d'outils d'évaluation et de comparaison amène une implémentation des différentes méthodes dans des systèmes conçus spécifiquement à cet effet et l'évaluation de celles-ci suivant des protocoles de test dédiés, comme Landstore [XZ12] ou WPAR [LWV07]. Chaque méthode possède donc ses propres métriques. Dans la littérature, le choix porte très souvent sur la réimplémentation des algorithmes [XHL+10, WPBM04, WFWL09]. Toutefois, certaines méthodes ne correspondent pas forcément au système dans lequel elles sont réimplémentées et des adaptations sont nécessaires, ce qui peut générer des variations de performance par rapport à l'algorithme original et altérer la comparaison.

3.1.1 Contexte d'utilisation ciblé

L'outil dont nous avons besoin doit permettre la comparaison de méthodes de distribution facilement et sans adapter chaque méthode à chaque système. Pour cela, notre outil doit permettre de changer facilement de méthodes de distribution sans modifier l'environnement de test de l'outil. Ainsi, il est possible de comparer les performances des différentes méthodes dans un même environnement avec des résultats provenant d'exécutions dans le même contexte et soumis au même flux d'entrées.

Les méthodes évaluées sont des méthodes propres aux systèmes de stockage objet, ce qui implique des contraintes inhérentes à ceux-ci, comme l'utilisation de la sémantique CRUD. Le système dans lequel doivent s'intégrer les méthodes possède un cluster de MDS pour gérer les métadonnées. Il sépare le flux de requêtes

de métadonnées et celui des données et l'on considère une cohérence forte des métadonnées.
Nous avons choisi d'orienter notre étude vers les différentes méthodes de distribution des métadonnées plutôt que les mécanismes de réplication ou de tolérance aux pannes, déjà traités amplement [vRG10, ACZ03, Bre12, Tre05, ELSC13], même si une évolution de notre outil dans ce sens est envisageable à l'avenir.

Une fois le contexte établi, nous devons énumérer les caractéristiques dont doit disposer notre outil. Notre outil doit simuler un service de métadonnées où des clients peuvent lui adresser des requêtes. Les clients soumettent des requêtes avec la sémantique CRUD et désignent selon une procédure définie par la méthode de distribution de métadonnées le serveur à questionner. Les serveurs du service de métadonnées traitent les requêtes des clients en respectant les règles définies par la méthode de distribution.

3.1.2 Propriétés structurelles attendues

Le principal atout de cet outil doit être la généricité. Il doit permettre de changer de méthode de distribution facilement sans modifier la structure et l'implémentation de l'outil. Ainsi l'évaluation d'une méthode de distribution avec notre outil ne nécessite que l'implémentation de la méthode elle-même. Comme chaque méthode nécessite d'agir à des moments différents dans le traitement d'une requête, notre outil doit le permettre, afin de bien les supporter.

L'outil doit également permettre une généricité au niveau des médias de stockage : disques, bandes magnétiques, ou même un *cloud* avec un protocole propre à celui-ci. L'outil doit donc pouvoir gérer des requêtes pour des serveurs possédant des médias de stockage hétérogènes sans modifier la structure de l'outil.

Comme nous voulons évaluer les méthodes de distribution sur différents types de flux, notre outil doit être capable de prendre n'importe quel fichier de traces conformes au format minimal de requêtes :

$$\{timestamp,\ operation,\ key,\ jobid\}.$$

Une requête est définie par le temps auquel se passe l'opération (*timestamp*), le type d'opération (*operation*), la clé (*key*) sur laquelle l'opération agit et l'identifiant du job qui a lancé cet accès (*jobid*). Un fichier de traces est une liste ordonnée de requêtes effectuées par l'ensemble des clients pour le service de métadonnées dans son intégralité [1]. On qualifie de telles traces de *globales*.

1. Et non pas à un serveur en particulier.

3.1.3 Métriques d'évaluation nécessaires

Nous voulons évaluer l'impact d'une méthode de distribution sur les serveurs et sur les clients, il est donc nécessaire d'avoir une prise de mesure donnant des informations sur les deux côtés. Un des principaux prérequis d'un système de stockage est l'accès rapide aux métadonnées, et la méthode de distribution a un impact sur ce temps d'accès : une mauvaise distribution engendre des points de contention et un serveur surchargé ne pourra pas répondre rapidement à toutes les requêtes qu'il reçoit. Nous souhaitons donc mesurer le temps de réponse côté client, c'est-à-dire le temps entre l'envoi de la requête par le client et la réception de la réponse toujours par celui-ci. Afin de faire le lien entre le temps de réponse d'un client et la charge d'un serveur, nous avons choisi, côté serveur, une mesure de la charge en terme de requêtes reçues. Il est ensuite possible de calculer un pourcentage de charge pour chaque serveur afin de savoir si un serveur est trop chargé par rapport aux autres.

3.1.4 Choix de la structure de l'outil

Nous avons choisi de concevoir un outil nommé *MeDiE*, pour ***Me****tadata* ***Di****stribution* ***E****valuator*, illustré par la figure 3.1, contenant les modules suivants :

- Un simulateur de service de métadonnées (en bleu). Il permettra l'exécution des interactions entre les clients et les serveurs selon les différentes méthodes de distribution. Sa conception doit faciliter le changement de méthode de distribution.
- Un protocole de simulation (en vert). Il sera en charge de la gestion du temps et de la prise des mesures. Il est responsable du traitement des fichiers de traces globales.
- Un générateur de traces (en violet). Il facilitera la génération de fichiers de traces globales correspondant à un flux de requêtes selon différentes caractéristiques.

Nous avons choisi de faire de cet outil un projet à code source ouvert, disponible sur *Github* [Bil20], afin que chacun puisse évaluer sa propre méthode de distribution. Ainsi, chaque nouvelle évaluation permettrait d'enrichir l'ensemble des méthodes testées dans un même contexte et à terme peut-être d'uniformiser les protocoles d'évaluation des méthodes de distribution.

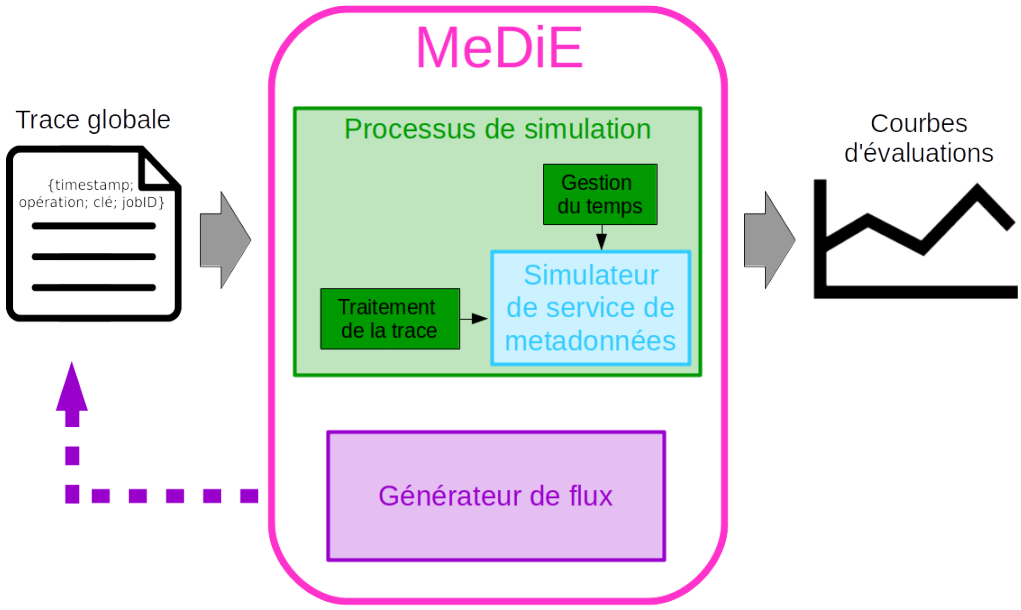


FIGURE 3.1: Structure de l'outil *MeDiE*

3.2 Simulateur de service de métadonnées

Le premier module de notre outil est le simulateur de service de métadonnées, permettant le traitement des requêtes de métadonnées. Il doit être capable de simuler des interactions entre les clients et les serveurs, en prenant en compte les règles définies par la méthode de distribution.

3.2.1 Conception logicielle du simulateur

En réponse à l'expression du besoin établi précédemment, nous avons décidé de cette structure globale pour le simulateur, représentée par le diagramme de classes de la figure 3.2.

Le schéma peut se décomposer en deux parties : le côté client et le côté serveur.

- Chacun de ces côtés possède une classe définissant le comportement de la méthode de distribution (*Distribution Client* et *Distribution Serveur*).
- Ils disposent aussi d'une classe permettant l'échange de requêtes et de réponses (*Client_API* et *Serveur*).
  Cette communication est la seule communication inter-machines prévue dans la structure du simulateur.

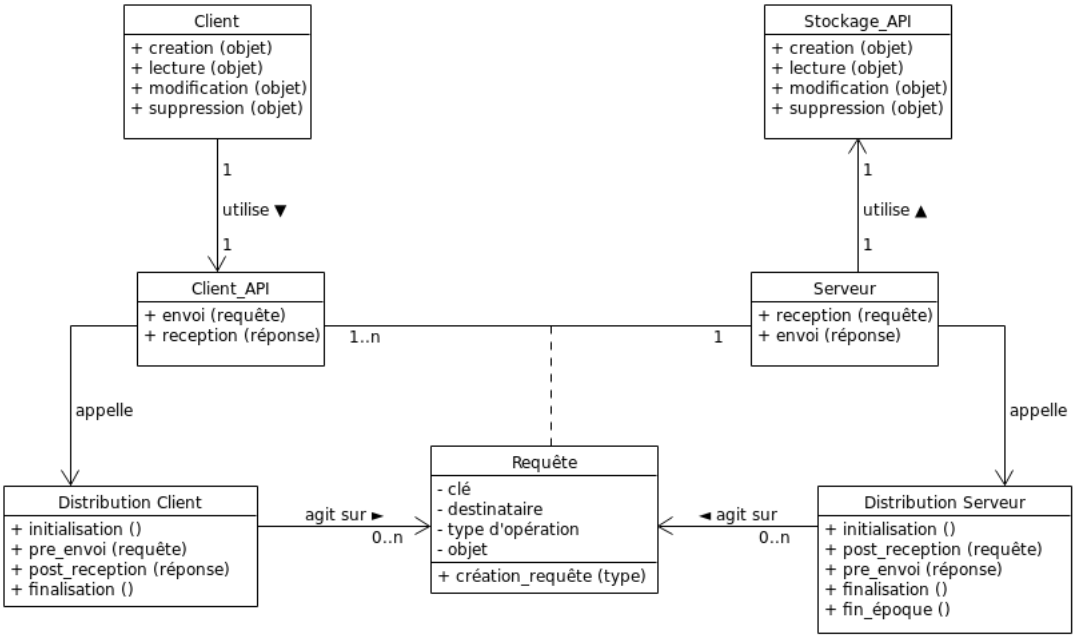


FIGURE 3.2: Diagramme de classes du simulateur de service de métadonnées

- Les requêtes sont définies dans la classe *Requête* : elles sont initialement composées d'une clé, d'un type d'opération CRUD et possiblement d'une métadonnée (*objet*) si la requête est une création ou une mise à jour. L'attribut *destinataire* correspond au serveur à interroger. Il est possible que d'autres champs propres à la méthode de distribution soient ajoutés. Un exemple de requête est donné à la figure 3.3a. Une réponse est en réalité une requête à laquelle nous ajoutons des attributs de réponse du serveur pour le traitement de la requête ou possiblement de la *Distribution Serveur*. Un exemple de réponse est illustrée par la figure 3.3b.
- La classe *Client* correspond au code utilisateur voulant accéder à une métadonnée.
- La classe *Stockage_ API* apporte au serveur les fonctions nécessaires pour traiter la requête.

Changement de méthode de distribution

Pour obtenir des comportements différents selon la méthode de distribution, les actions spécifiques aux distributions ont été rassemblées dans deux classes (*Distribution Client* et *Distribution Serveur*). Chacune possède deux fonctions

```
cle = "nom007"
destinataire = "serveur1"
type_operation = "suppression"
objet = ""
index_distribution = "42"
```

(a) Requête envoyée

```
cle = "nom007"
destinataire = "serveur1"
type_operation = "suppression"
objet = ""
index_distribution = "42"
repFlag = "fait"
```

(b) Réponse associée

FIGURE 3.3: Exemple d'échange client/serveur.

pre_envoi() et *post_reception()* décrivant le comportement de la méthode de distribution lors des communications pour le traitement d'une requête. Si aucune action n'est à exécuter lors d'un appel à une fonction de la distribution (selon la méthode choisie), alors la fonction correspondante sera vide.

Le processus complet de traitement d'une requête est le suivant :

1. La classe *Client* fournit à *Client_API* les informations en sémantique CRUD concernant sa requête : le nom de la métadonnée et l'opération à exécuter.
2. La classe *Client_API* crée un objet de type *Requête* et demande à la *Distribution Client* de remplir le champ *destinataire* (contenant le serveur à interroger) en appelant la fonction `pre_envoi`. La *Distribution Client* peut à cet instant effectuer d'autres actions propres à la méthode de distribution.
3. Le *Serveur* reçoit la requête et appelle la fonction `post_reception` de la *Distribution Serveur*.
4. Le *Serveur* effectue l'opération de la requête via la classe *Stockage_API*. Il complète ensuite la requête avec le retour de l'opération (si l'opération a réussi) et possiblement des métadonnées si la requête est une lecture. La requête devient alors une réponse.
5. Le *Distribution Serveur* effectue à nouveau des actions spécifiques à la méthode de distribution avec la fonction `pre_envoi` avant que le *Serveur* ne renvoie la réponse.
6. La classe *Client_API* reçoit la réponse du *Serveur* et appelle la fonction `post_reception` de la *Distribution Client*.
7. La classe *Client_API* renvoie au *Client* une validation de l'exécution de la requête.

Hétérogénéïté des médias de stockage

L'API de stockage (*Stockage_API*) fournit au serveur les différentes fonctions d'accès (en sémantique CRUD) afin de traiter les requêtes. Les modifications de

comportements des fonctions de stockage se font par l'intermédiaire de cette classe et n'impactent pas la structure du serveur. Pour que notre outil puisse s'adapter à n'importe quel système de stockage, cette API fournit des fonctions génériques permettant de redéfinir le comportement des opérations CRUD. En effet, la classe *Stockage_API* est réimplémentée à chaque nouveau média, afin que celui-ci puisse assigner les actions des opérations CRUD selon ses propres contraintes. De plus, chaque serveur possède sa propre API de stockage, ce qui permet de fédérer des serveurs utilisant des médias différents dans un même système.

Format des requêtes à l'entrée du simulateur de service de métadonnées

Le format des traces globales fournit des informations qui ne sont pas prises en compte par le simulateur de service de métadonnées. En effet, la gestion du temps ainsi que la répartition des requêtes parmi les clients selon leur identifiant de job est délégué au protocole décrit dans la section 3.3. Nous avons donc choisi de ne pas les intégrer dans le format de requêtes accepté par le simulateur. Le protocole est responsable de fournir au simulateur des fichiers de requêtes selon le format accepté.

Dans notre structure, une requête est composée d'une clé, d'un type d'opération et possiblement d'une métadonnée. La valeur de la métadonnée étant sans importance pour la simulation, nous avons décidé de l'ignorer dans nos fichiers de traces. Nous attribuons, à la place, la valeur par défaut `"a word"`, caractéristique de la petite taille d'une métadonnée, pour toute requête en nécessitant. Les fichiers de requêtes acceptés par notre simulateur sont donc de type *clé ;opération*. Ce format intègre l'essentiel d'une requête, à savoir la clé et le type d'opération à exécuter.

3.2.2 Implémentation et validation de l'outil

Nous détaillons ici l'implémentation de l'outil en discutant les choix de technologies et de langages utilisés ainsi que certaines fonctionnalités.

Choix du langage et des technologies

Nous avons choisi le langage C pour écrire notre outil car c'est un langage qui permet d'être proche de la machine. En général, les API utilisées dans les systèmes de stockage sont écrites en C et nous souhaitons reproduire au plus proche un système, avec des méthodes de distribution utilisables potentiellement dans des systèmes réels, ce qui implique des APIs codées dans le même langage de programmation que les systèmes réels.

En ce qui concerne le format des requêtes à envoyer, nous avons opté pour une encapsulation en JSON [Bra17] car c'est un standard ouvert simple et léger. L'ajout d'informations à la requête n'est pas coûteux et la désencapsulation se fait rapidement. Les requêtes circulant sur le réseau restent des messages de petite taille et ne surchargent pas la bande passante. Utiliser un même format défini côté client et côté serveur permet de détecter plus rapidement les potentiels messages corrompus et de faciliter l'écriture et la lecture des paquets. Pour construire nos requêtes en JSON, nous utilisons la bibliothèque *json-c* [Cla07].

En ce qui concerne la communication entre les clients et les serveurs, nous avons étudié différentes bibliothèques dédiées (Mercury [SKZ+13], Accelio [Mel16], ZeroMQ [Hin13]) et avons opté pour la bibliothèque ZeroMQ [Hin13] (ZMQ) qui se base sur un protocole de communication dérivant du protocole TCP [Ins81].

Nous avons choisi cette bibliothèque pour le nombre de motifs de communication qu'elle fournit, nous permettant de simuler facilement une grande variété de comportements. De plus, elle est *thread-safe* (compatible avec le multithreading), ce qui sécurise certains comportements possibles. Elle est aussi sans copie de données et autorise des connexions et déconnexions dynamiques, ce qui facilite grandement la gestion de l'interaction client-serveur. Les motifs de communications représentés sont :

- *request/reply* : l'expéditeur attend un retour du récepteur avant de pouvoir envoyer un nouveau message ;
- *publisher/subscriber* : plusieurs nœuds s'abonnent à un nœud publiant des messages (nœud publisher). Chaque message envoyé de ce nœud sera reçu par tous les nœuds abonnés (nœud subscriber) présents sur le réseau ;
- *pipeline ou push/pull* : un nœud de type *push* envoie un message à plusieurs nœuds de type *pull* et un nœud de type *pull* peut recevoir des messages de plusieurs nœuds de type *push*. C'est un schéma de distribution/collecte de données en parallèle qui peut se réaliser en plusieurs boucles et plusieurs étapes.

Choix d'implémentations

Après avoir choisi les technologies à utiliser, il est possible d'implémenter l'outil en prenant en considération les fonctionnalités qu'offre chacune d'elles.

Parallélisation du traitement des requêtes : Même si la bibliothèque ZMQ permet de paralléliser le traitement des requêtes (en permettant le multithreading), nous avons choisi de garder un comportement séquentiel pour l'interaction client-serveur car cela n'apporte rien à la comparaison des méthodes de distribution et complexifie la gestion de la cohérence. En effet, l'amélioration induite par la parallélisation des requêtes affecte de la même manière toutes les méthodes de distribution, ne changeant ainsi pas nos évaluations. Il est à noter que la décision de ne pas paralléliser les requêtes n'empêche en rien une méthode de distribution d'utiliser le multithreading afin d'exécuter des tâches propres à la méthode de distribution.

Découverte du réseau : À l'initialisation, chaque client et chaque serveur possède la liste et les adresses de tous les serveurs, afin de rendre possible les communications. De plus, la bibliothèque ZeroMQ gère les connexions et déconnexions, ce qui nous permet de ne pas nous soucier de l'ordre dans lequel chacun des acteurs se connecte.

Gestion de la généricité pour les méthodes de distribution : Afin de changer facilement de méthodes de distribution, nous avons séparé les codes globaux/génériques des codes spécifiques aux méthodes, comme illustré sur le diagramme de classes [2] de la figure 3.2 et chaque méthode de distribution est définie dans deux fichiers séparés : un fichier pour la distribution côté client et un autre pour la distribution côté serveur. Changer de méthodes de distribution revient à changer de *flag* lors de la compilation pour redéfinir les fonctions génériques.

Gestion de l'hétérogénéïté des médias de stockage : Nous avons utilisé le même principe de généricité pour permettre de gérer plusieurs types de média de stockage différents. Pour chaque média, une implémentation des fonctions d'accès génériques (*Création, Lecture, Modification, Suppression*) est définie. Contrairement à la gestion des méthodes de distribution, ici nous n'avons pas de *flag* à positionner : un seul fichier "generic_storage" est défini et chaque définition des fonctions remplace la précédente.

En résumé, le traitement d'une requête passe par 4 acteurs principaux aussi bien côté client que serveur, comme on peut le voir sur la figure 3.4.

Du côté client :

- L'utilisateur est l'entité effectuant une demande d'accès à une métadonnée ;
- L'API client (appelé *Client* dans la figure 3.4) effectue la requête : elle la crée,

2. Même si le language C n'est pas un langage objet, nous avons structuré notre code dans cet esprit.

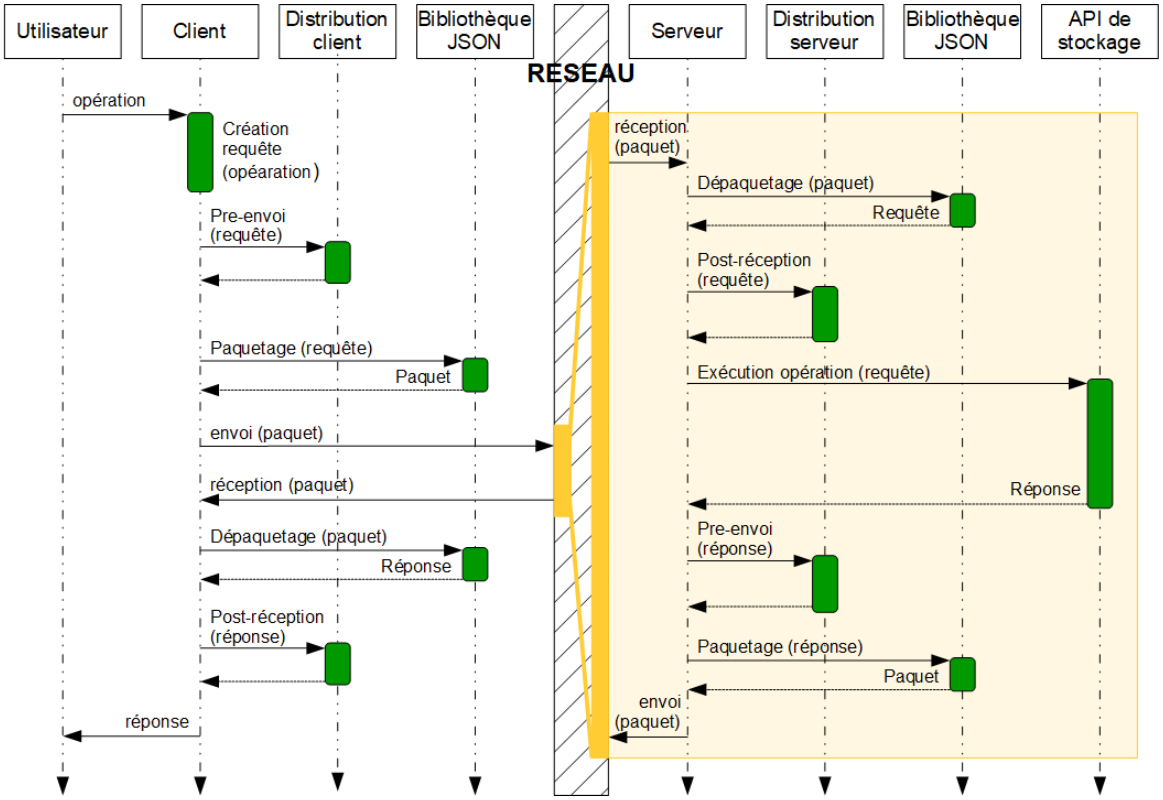


FIGURE 3.4: Diagramme de séquence du traitement d'une requête avec le simulateur de service de métadonnées.

l'envoie, réceptionne la réponse correspondante et renvoie à l'utilisateur le résultat de la demande ;

- La distribution client indique dans la requête quel serveur contacter et possiblement d'autres informations suivant la méthode choisie. Elle peut également effectuer des actions à la réception de la réponse ;
- La bibliothèque JSON traduit la requête avant l'envoi et extrait la réponse à la réception.

Du côté serveur :

- Le serveur reçoit la requête, la traite et renvoie la réponse ;
- L'API de stockage est utilisée afin d'exécuter l'opération ;
- La méthode de distribution est appelée à la réception de la requête et peut par exemple vérifier que la requête est bien adressée au serveur en charge de la métadonnée. Elle peut une nouvelle fois agir avant l'envoi de la réponse ;

- La bibliothèque JSON extrait les informations à la réception de la requête et traduit la réponse avant l'envoi.

3.3 Processus de simulation d'une méthode de distribution

Le simulateur de service de métadonnées permet d'exécuter un scénario client-serveur avec des méthodes de distribution différentes, mais il est nécessaire de définir comment à partir d'une trace globale, notre outil simule une exécution et fournit un post-traitement des mesures afin d'obtenir des courbes de performance comparables.

3.3.1 Conception du processus de simulation

Nous décrivons ici le processus de simulation pour l'exécution d'une trace globale ainsi que la collecte de mesures.

Gestion du temps

La première étape de ce processus de simulation consiste à choisir la prise en compte du temps. Les traces globales sont datées à l'arrivée des requêtes, ce qui définit la fréquence globale d'arrivée, que nous appellerons *profil temporel* de la trace (le nombre de requêtes par seconde).

Nous choisissons d'exécuter la trace globale par pas de temps plutôt que de simuler les requêtes en temps réel. Nous découpons donc la trace par pas de temps que nous appellerons **époques** et nous exécutons toutes les requêtes d'une même époque en parallèle. Toutes les requêtes possédant une date d'arrivée comprise entre T et $T + \textit{pas de temps}$ (avec T le début d'une époque) seront simulées en même temps, lors de la même époque. Le pas de temps est configurable afin de choisir le niveau de précision que l'on souhaite.

Exécuter les requêtes par époque nous permet d'accélérer le temps de simulations, puisque nous n'exécutons pas les requêtes en temps réel, mais quand l'époque précédente se termine. Ainsi, il n'est pas nécessaire d'attendre pendant les périodes de faible intensité. En plus de réduire le temps de simulation, cette approche permet aussi de réduire les risques de panne dans l'environnement de simulation : plus une simulation est longue, plus la probabilité qu'un incident arrive est grande, l'accélération de la simulation réduit donc cette probabilité. Si un incident arrive tout

de même, rejouer la simulation reste moins coûteux qu'avec une simulation en temps réel.

Plus le pas de temps est petit, plus le réalisme de l'exécution augmente, mais ralentit l'exécution de la trace. À l'inverse, plus le pas de temps est grand, plus la simulation est rapide, cependant, elle perd des informations sur le profil temporel de la trace et peut changer le comportement de la simulation. C'est pourquoi il est nécessaire d'avoir un pas de temps configurable afin d'avoir un compromis entre rapidité d'exécution et le réalisme de l'exécution par rapport à la trace globale.

Répartition des requêtes sur les différents clients

L'étape suivante consiste à répartir toutes les requêtes d'une époque parmi un certain nombre de clients afin de les exécuter en parallèle. Pour que la simulation soit la plus rapide possible et qu'il soit possible d'observer des rafales de requêtes, toutes les requêtes d'une époque sont réparties entre les clients de telle sorte qu'ils aient tous approximativement le même nombre de requêtes à effectuer durant cette époque. Pour éviter des problèmes de cohérence, nous avons choisi d'attribuer toutes les requêtes liées à une même clé à un même client. Nous n'avons pas pris en compte le *jobid* dans notre répartition car il n'est pas possible de savoir si un job va produire beaucoup de requêtes ou non, rendant plus difficile la répartition sur les clients équitable. Chaque client va donc jouer approximativement le même nombre de requêtes, et chaque client s'exécute en parallèle avec les autres.

Prises de mesures

La troisième et dernière étape consiste en la prise des différentes mesures. Nous avons choisi de l'effectuer à la fin de chaque époque plutôt qu'une prise à la volée qui pourrait d'une exécution à une autre attribuer le traitement d'une requête à une époque différente : suivant l'instant où la prise de mesures est faite, le traitement peut être pris en compte à l'époque courante ou à la suivante. Ainsi, les charges serveurs pourraient pour une même époque différer d'une exécution à une autre.

Nous avons donc opté pour une solution contrôlant l'état du système avant de le stopper pour une prise de mesures. La fin d'une époque est une indication de l'état du système pertinente pour une prise de mesures. Nous récupérons le nombre de requêtes reçues par chaque serveur pendant toute l'époque pour la charge côté serveur et pour le temps de réponse côté client, nous avons choisi de mesurer le temps que met chaque client pour effectuer l'ensemble de ses requêtes pour une époque. Prendre des mesures sous condition que l'époque entière ait été

simulée permet une meilleure reproductibilité des tests pour la mesure de charge des serveurs, puisque chaque requête ne peut être traitée que pendant l'époque associée.

Les métriques sont enregistrées à la fin de chaque époque et seront traitées à la fin de l'exécution afin de générer des courbes qui donnent le comportement de la méthode tout au long de la trace.

3.3.2 Déroulement du processus de simulation

La figure 3.5 montre une vue globale de notre protocole de simulation et du déroulement d'une exécution. Les flèches bleues correspondent aux scripts exécutés par le protocole de simulation. La trace globale est d'abord traitée (flèche bleue verticale) afin d'extraire les informations relatives à la gestion du temps et d'obtenir des listes d'instructions correspondant au format demandé par le simulateur. Le protocole de simulation est ensuite responsable du déroulement d'une exécution (flèche bleue horizontale). Après l'initialisation, les requêtes de chaque époque (en jaune) sont exécutées par notre simulateur et entre chacune d'entre elles un point de synchronisation (carré rose) est effectué par le protocole de simulation.

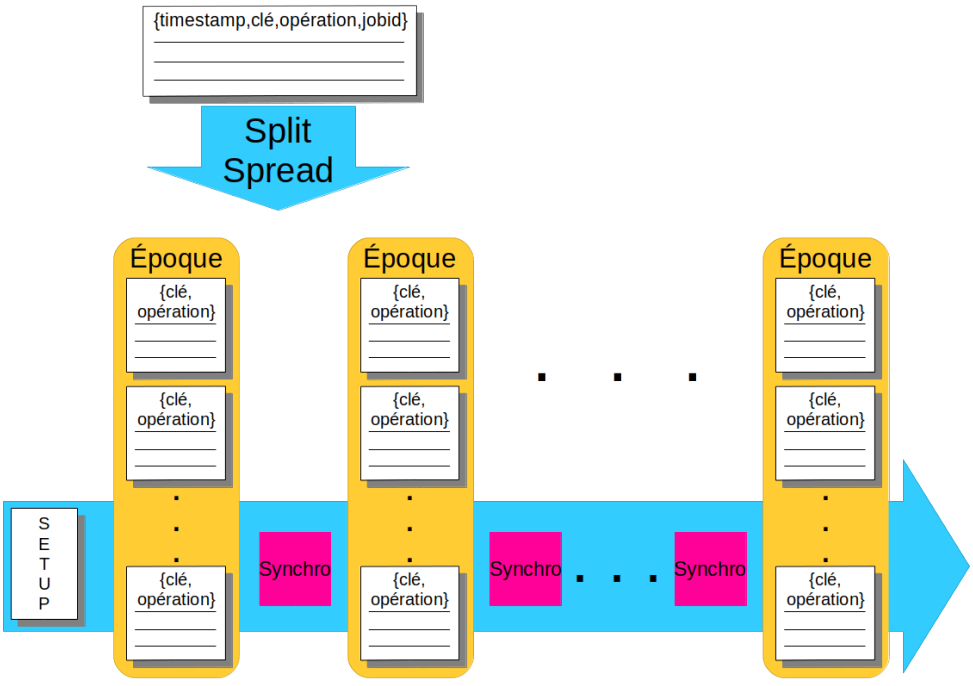


FIGURE 3.5: Déroulement du processus de simulation.

Traitement de la trace globale

Avant de lancer l'exécution d'une simulation, nous effectuons un prétraitement de la trace globale pour que chaque client puisse avoir une liste de requêtes à exécuter à chaque époque. Ce prétraitement se fait par des scripts *Python*.

Le script *split* prend une trace globale et un pas de temps et découpe la trace par pas de temps grâce aux datations des requêtes. Nous obtenons alors un certain nombre de traces locales contenant toutes les requêtes de chaque époque. Il est ensuite nécessaire de répartir ces requêtes sur tous les clients disponibles.

Le script *spread* prend en argument un chemin contenant les traces locales ainsi qu'un nombre de clients et répartit les requêtes équitablement en plusieurs listes d'instructions à donner aux clients. Ces listes sont au format *{key, operation}*, comme demandé par notre simulateur. Sur la figure 3.5, ces scripts sont rassemblés dans la première flèche bleue, montrant la transformation de la trace globale en plusieurs groupes d'instructions classés par époques.

Initialisation

Il peut aussi être demandé que le système de stockage contienne déjà un certain nombre de métadonnées avant d'exécuter une trace. Pour les traces réelles par exemple, beaucoup de clés sont mises à jour sans avoir été créées au préalable, puisque ces traces sont extraites d'un système en pleine exécution. Il nous faut donc créer ces clés avant de jouer la trace. Pour ce faire, nous avons permis l'exécution d'une trace *setup* afin de préparer le service de métadonnées. Nous avons aussi écrit un script du même nom (*setup*) permettant de parcourir une trace et de générer la trace *setup* correspondante.

Exécution

La gestion globale de l'exécution est prise en charge par un script en *bash* et est illustrée par la flèche bleue horizontale sur la figure 3.5. Le script initialise les serveurs du simulateur sur les premières machines virtuelles, exécute la trace *setup* pour préparer le système et commence l'exécution de la trace choisie. Pour cela, le simulateur génère des clients sur les machines suivantes qui exécutent toutes les requêtes d'une époque. Une fois que tous les clients ont fini, ils notent dans un journal de mesure le temps d'exécution de l'ensemble de leurs requêtes. Le script attend le retour de tous les clients avant de lancer la prise de mesures côté serveur.

Afin de signaler à chaque serveur que l'époque est finie et qu'il doit faire une prise de mesures, nous avons implémenté une gestion des signaux utilisateurs *SIGUSR1* et *SIGUSR2*.

- Le signal *SIGUSR1* correspond à un changement d'époque avec simple prise de mesures : À la réception de ce signal, chaque serveur note le nombre de requêtes reçues pour cette époque.
- Le signal *SIGUSR2* correspond à un changement d'époque avec action de la part de la méthode de distribution. La gestion du temps étant contrôlée par notre script, les méthodes de distribution ne peuvent pas faire usage d'une horloge interne. Si certaines méthodes agissent périodiquement, il faut leur permettre d'agir lors du passage à une autre époque qui représente le passage du temps. C'est dans ce but que la fonction `fin_époque` de la *Distribution Serveur* doit être utilisée. Le signal *SIGUSR2* déclenche la prise de mesures ainsi que des actions spécifiques à la distribution avec la fonction `fin_époque`.

Ainsi, la prise de mesures peut se faire de manière plus fréquente que la période de temps imposée par une méthode de distribution sans conséquence sur celle-ci, ce qui permet, par exemple, de visualiser la répartition de la charge des serveurs avant une action de la distribution.

Une fois toutes les actions liées aux signaux effectuées, chaque serveur va générer un fichier de validation indiquant la complétion du changement d'époque. Le script attend la génération de tous les fichiers de validation avant de passer à la prochaine époque et de lancer de nouveaux clients. Ces vérifications correspondent aux points de synchronisation entre les époques et sont représentées par les carrés nommés "Synchro" sur la figure 3.5.

Génération de courbes

À la fin de l'exécution, quand toutes les époques ont été simulées, il est possible de traiter les mesures prises. Pour cela, nous avons un script de post-traitement global qui va faire appel à un post-traitement pour les mesures du côté client et un autre pour le côté serveur et qui générera des courbes de synthèse en utilisant *Gnuplot* [WK86]. Séparer le post-traitement serveur et client permet de rajouter facilement d'autres traitements si nécessaire. Nous avons choisi de fournir :

- Des courbes représentant la charge en nombre de requêtes reçues par chaque serveur en fonction du temps, afin d'attester du profil temporel de la trace ;
- Des courbes représentant le pourcentage de charge pour chaque serveur en fonction du temps, afin d'évaluer les possibles déséquilibres ;
- Des courbes représentant le temps de réponse moyen pour un client en fonction du temps, afin de voir des possibles latences en cas de déséquilibre.

3.3.3 Validation du processus d'évaluation

Notons que tout au long du développement et plus spécifiquement aux étapes d'implémentation du simulateur et des scripts du protocole de simulation, nous avons effectué des tests de fonctionnement et de non-régression afin de garantir un outil répondant aux besoins exprimés. Cette dernière étape consiste à vérifier le bon fonctionnement du processus de simulation dans sa globalité :

- Vérifier la bonne transformation de la trace globale en listes d'instructions pour chaque client ;
- Confirmer qu'il n'y a pas de possibilité de blocage pendant l'exécution ;
- Constater l'exécution des opérations d'accès des métadonnées sur les médias de stockage ;
- Attester des prises de mesures effectuées à chaque fin d'époque ;
- S'assurer que les synchronisations se font en un temps raisonnable ;
- Valider que le post-traitement et la génération de courbes fournissent bien des résultats cohérents avec la trace.

Nous avons effectué toutes ces vérifications avec une trace synthétique dont le comportement est connu afin de faciliter la vérification.

3.4 Intégration des méthodes de distribution dans *MeDiE*

Afin d'analyser le comportement des méthodes de distributions choisies dans le chapitre 2 et de les comparer, nous les avons intégrées dans *MeDiE*.

Afin d'exploiter au mieux les capacités de chaque méthode, il est important de prendre en compte leurs caractéristiques lors de l'intégration dans notre outil. Pour cela, une première étape de spécification technique est nécessaire avant l'implémentation de la méthode et ses tests de bon fonctionnement. La spécification technique d'une méthode consiste à définir les comportements propres à la distribution client et propres à la distribution serveur, tel qu'ils sont indiqués dans le diagramme de classes présenté à la figure 3.2. Cela nécessite une classification des fonctionnalités et la distinction entre celles propres aux clients et celles propres aux serveurs. Il est aussi important de détailler le déroulement complet de l'accès à une métadonnée et d'établir les actions à effectuer à la fin d'une période.

3.4.1 Intégration du *Static Hashing*

La méthode du *Static Hashing* est une méthode simple et les comportements liés au côté client et ceux au côté serveur sont basiques : la fonction `pre_envoi` de la *Distribution Client* contient le calcul de la fonction de hachage tandis que la fonction `post_reception` du côté client est vide : le client n'a pas de comportement défini à la réception de la réponse. Les fonctions `init` et `finalise` sont aussi vides, puisqu'il n'y a pas de structure particulière à la méthode du *Static Hashing*. Du côté serveur, les fonctions `init` et `finalise` ainsi que les fonctions propres à la distribution sont toutes vides : le serveur reçoit des requêtes et les traite sans vérifier si elles lui sont bien adressées. Le diagramme de séquences sur la figure 3.6 illustre ces comportements avec le traitement d'une requête effectuée par l'utilisateur à la réception par celui-ci de la réponse. Il n'y a pas non plus de comportement particulier lors de la fin d'une époque.

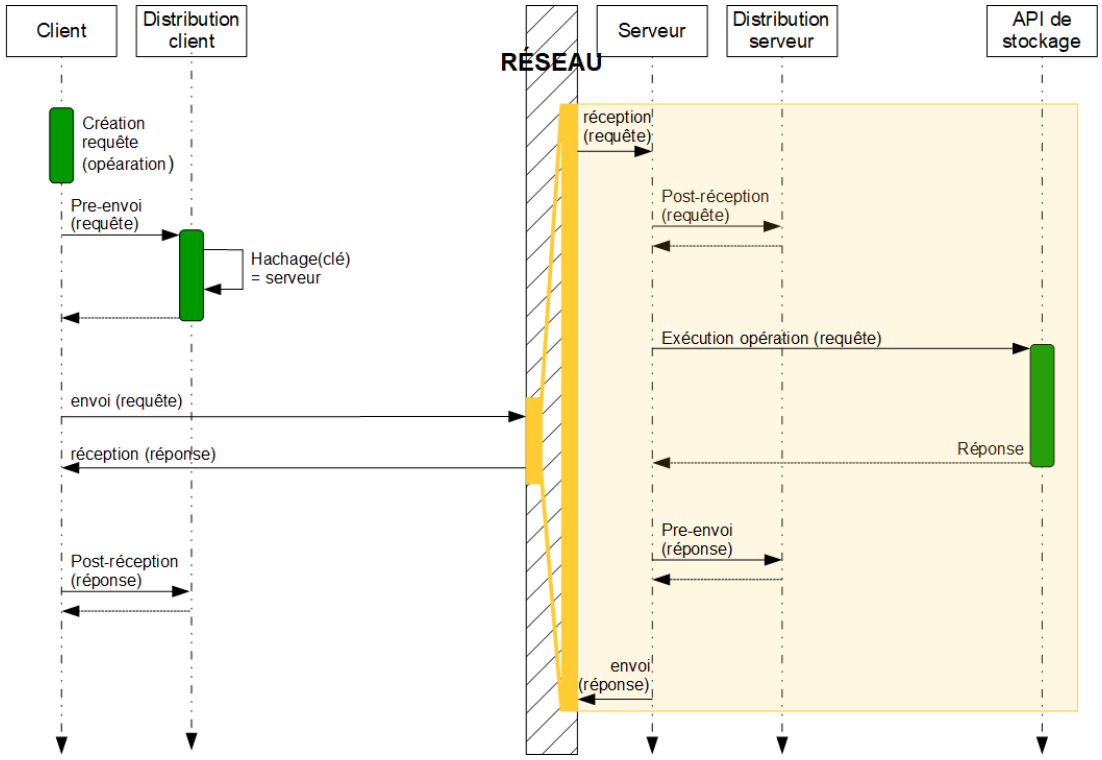


FIGURE 3.6: Déroulement d'une requête pour le *Static Hashing*.

Implémentation

Nous avons choisi de ne pas réimplémenter la fonction de hachage MurMur3 et d'utiliser une implémentation existante dans le langage C [Sco11]. Pour désigner le

serveur responsable d'une métadonnée, nous exécutons la fonction de hachage et appliquons un calcul de modulo en fonction du nombre de serveurs sur le résultat. L'utilisation du modulo évite aux clients et aux serveurs de stocker en mémoire des intervalles de valeurs (chacune correspondant à un intervalle géré par un serveur). Comme précisé plus haut, la méthode du *Static Hashing* ne possède pas de comportement particulier à la fin d'une époque, la fonction `fin_époque` de la *Distribution Serveur* est donc vide.

3.4.2 Intégration du *Dynamic Hashing*

Les différents comportements de la méthode du *Dynamic Hashing* sont explicités ci-après et sont illustrés sur la figure 3.7 qui montre le diagramme de séquences pour l'exécution d'une requête. La table d'index est représentée dans notre figure comme un acteur du déroulement afin de simplifier la compréhension, elle n'exécute, en réalité, aucune action.

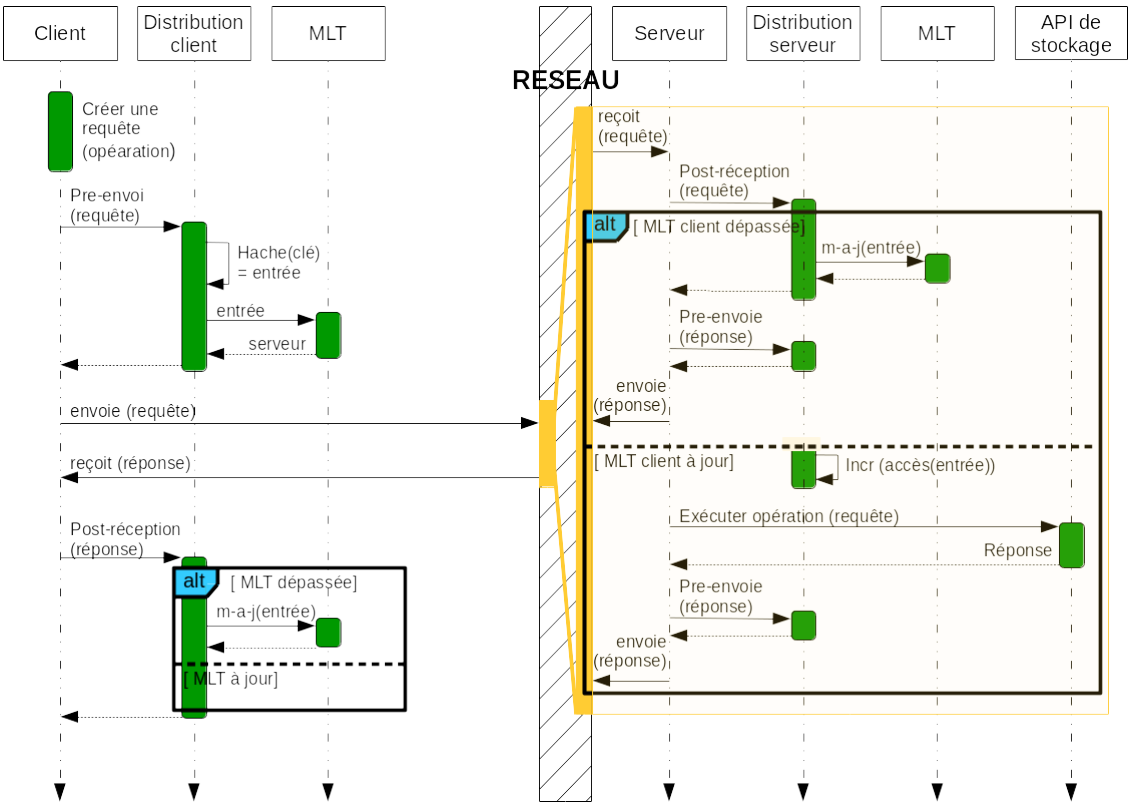


FIGURE 3.7: Diagramme de séquence du déroulement d'une requête pour le *Dynamic Hashing*.

Les comportements de la distribution côté client sont définis de la manière suivante :

- Les fonctions `init` et `finalise` sont en charge de la préparation et la révocation de la table d'index ;
- La fonction `pre_envoi` désigne le serveur à questionner : elle calcule la fonction de hachage et consulte la MLT afin d'obtenir le serveur responsable. Pour les mêmes raisons que la méthode précédente, nous avons choisi la fonction de hachage *MurMur3* ;
- La fonction `post_reception` vérifie la réponse de la *Distribution Serveur* au sujet de la table d'index : si sa MLT n'est plus à jour, elle la modifie et notifie le *Client* afin qu'il renouvelle sa requête. Si la table est à jour, la fonction ne fait rien ;

Pour le côté serveur, les fonctions propres à la distribution sont les suivantes :

- Les fonctions `init` et `finalise` sont en charge de la préparation et la révocation de la MLT et de la table d'EACL ;
- La fonction `post_reception` incrémente le compteur d'accès de l'entrée (EACL) et vérifie que la requête lui est bien adressée, c'est-à-dire, si le numéro de version de l'entrée demandée correspond à celui de sa propre MLT. Si la table n'est pas à jour, la fonction ajoute à la *Réponse* le serveur en charge de l'entrée demandée et le numéro de version associé. Elle notifie ensuite au serveur que la requête ne lui est pas destinée, afin de ne pas la traiter ;
- La fonction `pre_envoi` est vide car la méthode du *Dynamic Hashing* ne nécessite pas de comportement particulier précédant l'envoi de la réponse.

Pour intégrer la redistribution périodique à notre protocole qui contrôle le temps, nous avons unifié la notion de temps utilisée par la méthode de distribution et celle fournit par *MeDiE*, et fait un lien entre les périodes entre les redistributions et les époques : une période de redistribution est un multiple d'une époque de l'outil. Il est par exemple possible de prendre des mesures toutes les 5 minutes et de déclencher le changement de période de la redistribution toutes les heures. Nous avons décidé d'effectuer les traitements liés à la redistribution à la fin de l'époque, pendant la période de synchronisation, afin de ne pas les prendre en compte dans la mesure du temps de réponse côté client. Cela permet d'évaluer réellement le temps de réponse du côté client sans le bruits liés à d'autres calculs pendant l'époque. La fonction `fin_epoque` lance donc une redistribution en exécutant l'algorithme RELAB.

Nous avons fait le choix d'une entité distincte que nous nommerons *manager*, pour effectuer la redistribution. Cela permet de rester plus fidèle à la conception initiale en ne rajoutant pas d'hypothèses sur les politiques désignant le serveur

exécutant la redistribution. De plus, l'algorithme RELAB nécessite une connaissance globale des MLT de chaque serveur et il est plus pratique d'avoir une entité collectant une fois les informations de tous (EACL de chaque serveur). Un système d'échange d'informations entre tous les serveurs serait coûteux en nombre de communications si on suppose que chaque serveur peut effectuer l'algorithme de redistribution. À la fin du calcul RELAB, la nouvelle table d'index doit être mis à jour pour chaque serveur et une cohérence forte doit être maintenue. Nous considérons que le manager est la seule entité autorisée à modifier la MLT et que sa table est toujours à jour. Ainsi quand celui-ci envoie sa table à tous les serveurs, ils reçoivent forcément une table plus récente que la leur et doivent la mettre à jour.

Dans le calcul de l'algorithme RELAB (voir algorithme 1), le choix d'un ensemble de serveurs satisfaisant la condition "$|y_A + SUM(C)| \leq |y_A + SUM(O)|$ pour tout sous-ensemble O de L" peut se traduire par la recherche de l'ensemble C minimisant la somme de la charge du serveur A et celle de l'ensemble C. Nous avons donc simplifié cette formule en appliquant la condition suivante : "$|y_A + SUM(C)|$ le plus proche de 0". De plus, nous avons décidé d'utiliser l'algorithme glouton du rendu de monnaie [Cai09] pour résoudre le choix des serveurs satisfaisant cette condition.

Implémentation

Par simplicité, nous avons décidé d'héberger le manager, responsable du calcul RELAB, sur le serveur 0. Cependant, il est tout à fait possible de séparer le manager sur une machine dédiée, ou bien d'instaurer un système où chaque serveur est manager à tour de rôle, comme dans *PAXOS* [Lam98].

L'algorithme RELAB indique quelles sont les entrées à transférer entre les différents serveurs, mais ne connaît pas quelles sont les métadonnées associées à ces entrées. Il est donc nécessaire pour chaque serveur de maintenir une liste de toutes les métadonnées liées à chaque entrée. Pour cela, nous utilisons une structure, appelée *md_entry* contenant pour chaque entrée, le nom des métadonnées associées. Cette structure est mise à jour à chaque création ou suppression d'une métadonnée ainsi que pendant les transferts si le serveur est concerné.

À la réception du signal *SIGUSR2*, la fonction `fin_époque` pour chaque serveur envoie sa table EACL au manager en utilisant le motif de communication *push/-pull* de ZMQ. Le manager exécute l'algorithme RELAB et envoie aux serveurs la nouvelle MLT en utilisant le motif *publisher/subscriber*. Les serveurs mettent à jour leur propre table et procèdent aux transferts, via des communications ZMQ *request/reply*. Le fichier de validation de changement d'époque est généré une fois tous les transferts éffectués.

3.4.3 Intégration de la méthode *LAD* et de sa variante

L'intégration de cette méthode dans *MeDiE* est similaire à celle de la méthode du *Dynamic Hashing*, en ce qui concerne l'interaction entre les clients et les serveurs. Les fonctions `pre_envoi` et `post_reception` de la *Distribution Client* et de la *Distribution Serveur* ont les mêmes comportements, et le diagramme de séquences sur la figure 3.7 illustre également le traitement d'une requête avec cette méthode. Nous avons aussi choisi de garder la gestion des redistributions par une entité distincte, le *manager*, afin de faciliter le maintien de la cohérence de la table d'index et de limiter les communications inter-serveurs dues aux redistributions.

La différence réside dans la fonction `fin_époque`. En effet, ici, la fin d'une époque ne signifie pas forcément une redistribution, le comportement de cette fonction est donc le suivant : chaque serveur effectue une évaluation de sa charge et le compare à son seuil R_{last} ; si celui-ci est dépassé, le serveur demande une redistribution au manager. Comme les redistributions ne sont pas automatiques à chaque changement d'époques, il est possible d'augmenter la fréquence des époques agissant sur la distribution sans augmenter le coût de redistribution.

Les communications spécifiques à la demande de redistribution sont illustrées par un exemple sur la figure 3.8. Dans cet exemple, nous considérons 4 serveurs. À la fin de l'époque actuelle, tous les serveurs évaluent leur charge par rapport à leur valeur de R_{last} et nous obtenons l'état suivant : le serveur 1 est considéré comme libre et le serveur 3 comme surchargé, les serveurs 0 et 2 sont équilibrés. Le serveur 3, ayant atteint le seuil max, demande une redistribution auprès du manager (flèche nommée "Problème !"). Le manager collecte les EACLs de chacun des serveurs (flèches nommées "EACL ?" et "EACL !") pour ensuite exécuter l'algorithme de redistribution ("ÉQUILIBRAGE"). Il en résulte une nouvelle table d'index qui est envoyée à tous les serveurs (flèches "MLT !") afin que chaque serveur mette à jour sa propre MLT. Dans notre cas, la table indique au serveur 3 de transférer la gestion de certaines entrées au serveur 1. Après ces transferts, les serveurs sont considérés comme rééquilibrés.

Implémentation

Pour l'implémentation, nous avons repris certains choix d'intégration du *Dynamic Hashing* : le manager est hébergé par le serveur 0 arbitrairement et le tableau *md_entry* indique, pour chaque serveur, les métadonnées de chaque entrée dont il est responsable.

À la réception du signal *SIGUSR2*, la fonction `fin_époque` calcule sa charge et possiblement une redistribution s'exécute. Lorsque les serveurs envoient un mes-

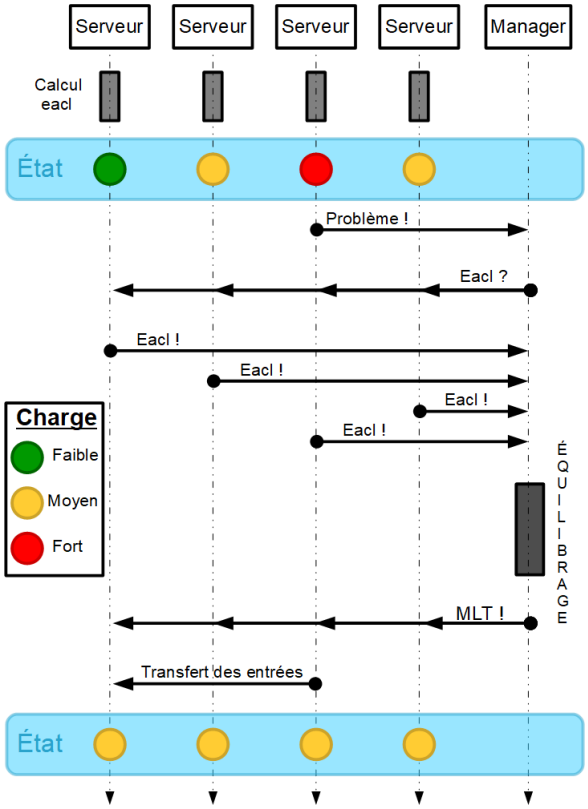


FIGURE 3.8: Déroulement d'une redistribution selon la méthode *LAD*.

sage au manager, ils utilisent le motif de communication *push/pull* de ZMQ. Le manager s'adresse aux serveurs via le motif *publisher/subscriber*. Les serveurs procèdent aux transferts nécessaires en ouvrant des canaux de communication ZMQ de type *request/reply*.

D'après l'algorithme de redistribution (voir algorithme 2.3.3), le seuil envoyé à chaque serveur (R_{last}) peut être différent en fonction du facteur de performance w_i. En pratique, nous utilisons un ensemble de serveurs gérés par l'outil PCOCC [DB16] et celui-ci n'autorise pas de cluster hétérogène. Nous avons décidé de simplifier l'implémentation en fixant un seuil unique à tous les serveurs au lieu d'un seuil dédié à chacun.

La redistribution n'étant pas systématique, la gestion de la synchronisation indiquant la fin d'une époque se complexifie. En effet, un unique type de fichiers de validation peut provoquer des synchronisations incomplètes : si parmi N serveurs, seul un demande une redistribution, les autres serveurs génèreront leur fichier de validation avant l'exécution de l'algorithme de redistribution. Quand le serveur ayant demandé une redistribution aura pris en compte la nouvelle table d'index,

il génèrera à son tour son fichier de validation et l'étape de synchronisation se terminera avant que les autres serveurs aient pris en compte la redistribution.

Afin d'éviter de potentiels blocages ou synchronisations partielles, nous avons prévu deux types de fichiers de validation qui sont générés dans les conditions suivantes :

- Le serveur n'a pas demandé de redistribution, il génère alors directement un fichier de validation de type 0 ;
- Le serveur a demandé une redistribution et il génère un fichier de validation de type 1 une fois les transferts nécessaires effectués.

Il est maintenant nécessaire de prendre en compte dans le script de synchronisation ces différents types de fichiers de validation : Le script termine la synchronisation si tous les serveurs génèrent un fichier de validation du même type.

Ainsi, si un serveur demande seulement une redistribution, nous obtenons les comportements suivants : les autres MDS génèrent des fichiers de validation de type 0, tandis qu'une redistribution est demandée. Une fois celle-ci achevée, le serveur ayant demandé la redistribution génère un fichier de type 1. Le script va donc attendre la génération des autres fichiers de validation de type 1 avant de terminer la synchronisation. Les serveurs n'ayant pas demandé de redistribution recevront la nouvelle MLT et après sa prise en compte pourront générer un fichier de validation de type 1.

3.5 Générateur de traces

Afin d'évaluer les méthodes de distribution dans différents contextes, il est nécessaire de disposer de plusieurs types de flux. Grâce à la collaboration avec le CEA, nous disposons d'une trace réelle, prise pendant 24h sur un supercalculateur industriel, dont les caractéristiques sont détaillées à la section 3.5.5. Cependant un seul type de flux ne suffit pas et les flux de requêtes des systèmes futurs ne sont par définition pas encore observables sur les supercalculateurs. C'est pourquoi il est nécessaire de générer synthétiquement ces types de flux ainsi que d'autres représentant des cas particuliers qui permettront d'évaluer les limites de nos méthodes. Pour cela, nous avons développé un outil de génération de flux permettant de générer des traces dans un format défini préalablement et suivant des motifs choisis.

Nous voulons générer des flux dans leur globalité, correspondant au fonctionnement de tout le service de métadonnées du système pendant plusieurs heures avec toutes les modifications possibles liées au flux caractérisé. Nous utilisons donc le

format de traces globales choisi à la section 3.1 : *{timestamp, operation, key, jobid}*. Le temps permet de savoir si le workload est chargé (s'il y a beaucoup de requêtes en même temps), le couple *(opération, clé)* représente la requête et le *jobid* est important pour garder la cohérence entre les requêtes : s'il y a plusieurs requêtes avec la même clé, sont-elles indépendantes et, dans ce cas, peuvent-elles s'exécuter en parallèle ou font-elles partie du même job et donc doivent-elles se faire de manière séquentielle ?

3.5.1 Expression du besoin

La première étape est d'établir la liste des paramètres importants à faire varier pour obtenir différents types de flux. Nous souhaitons pouvoir faire varier les caractéristiques de flux suivantes :

- **La durée totale de la trace** : il doit être possible de contrôler la durée d'observation du comportement d'une méthode sur un temps plus ou moins long. Certaines méthodes peuvent être meilleures à court terme et d'autres à long terme, comme par exemple, les méthodes dynamiques.
- **Le profil temporel** : il faut pouvoir faire varier le nombre de requêtes par seconde que reçoit le système. Ainsi, il est possible de configurer les périodes de faible intensité ou de rafales.
- **La répartition des requêtes sur les différents serveurs** : il est nécessaire de contrôler le nombre de requêtes devant arriver sur chaque serveur pour modéliser différents cas de répartition. Forcer la répartition initiale des requêtes permet de créer artificiellement des déséquilibres plus ou moins importants. Il est alors possible d'évaluer le comportement des méthodes dans certains cas de déséquilibre, mais aussi la gestion de ces déséquilibres dans le temps.
- **Le pourcentage de clés distinctes accédées** : il doit être possible de dimensionner l'espace de stockage en accédant à plus ou moins de clés. La génération d'un grand nombre de clés ou au contraire l'accès à un ensemble de clés très restreint peut mettre en difficulté certaines méthodes de distribution.

3.5.2 Conception du générateur

Nous avons divisé la génération de traces en trois étapes :

1. **La génération du profil temporel indiquant tout au long de la trace le nombre de requêtes par seconde** : la courbe du profil temporel indique à chaque seconde combien de requêtes le système doit recevoir. Cette courbe

nous permet de faire varier l'aspect temporel de la trace, à savoir la durée totale et le profil temporel.

2. **La définition de la répartition des requêtes sur les serveurs** : la définition de la répartition des requêtes sur les serveurs se fait aussi via une fonction donnant à chaque seconde un tuple. Ce tuple possède autant d'éléments que de serveurs et chacun représente un pourcentage des requêtes destiné à un serveur. La somme des éléments de ce tuple est donc égale à 100.
3. **L'écriture des requêtes dans un fichier de traces pour chaque seconde de la trace** : cette dernière étape consiste à générer la trace avec comme pas de temps la seconde. Nous nous appuyons sur les courbes précédentes afin de déterminer le nombre de requêtes pour chaque serveur et par seconde. Nous les écrivons ensuite dans un fichier global en prenant en compte le pourcentage de clés distinctes.

Nous avons choisi de prendre la seconde comme unité de temps car la précision qu'elle apporte est largement suffisante pour des traces représentant plusieurs heures.

3.5.3 Implémentation du générateur

Pour l'implémentation, nous avons choisi le langage Python, car c'est un langage adapté aux traitements mathématiques rapides avec des bibliothèques spécialisées permettant une implémentation simple. Par souci de clarté, nous avons regroupé au début de l'outil de génération de traces, tous les paramètres utilisés tout au long de la génération de traces : paramètres définissant la courbe du profil temporel, paramètres de répartition des requêtes sur les serveurs, taille de la trace, pourcentage de clés distinctes, paramètres liés à la distribution.

Pour générer la courbe du profil temporel nous avons choisi d'implémenter un modèle de fonction simple pouvant représenter une grande majorité des cas, dont la figure 3.9 en est un exemple. Cependant il est facile d'utiliser dans notre outil d'autres fonctions mathématiques plus complexes pour définir le profil temporel. Dans le modèle de fonction implémenté, nous considérons alternativement deux niveaux de flux de requêtes : un niveau haut et un niveau bas. Il est possible de paramétrer le nombre de requêtes reçues à chaque niveau, le temps écoulé dans chacun des niveaux ainsi que le temps nécessaire pour passer du niveau bas au niveau haut et du niveau haut au niveau bas.

Ce modèle simple et configurable permet de simuler la majorité des flux de requêtes en mélangeant différents motifs caractéristiques : alterner des moments

de faible intensité et des possibles rafales, choisir la pente pour une montée (ou descente) en charge plus ou moins rapide, ou encore garder un flux constant.

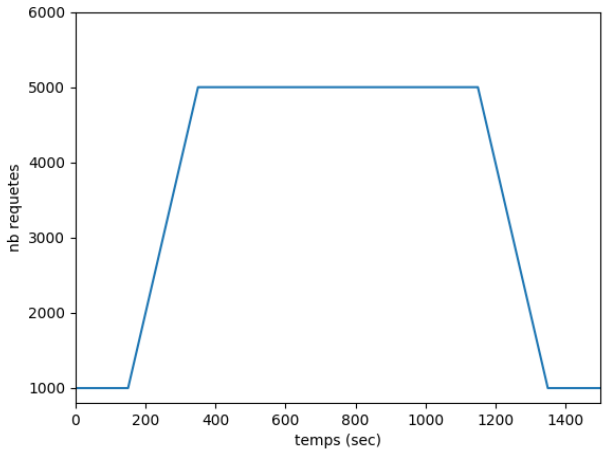


FIGURE 3.9: Exemple de courbe de profil temporel générée par notre outil.

Pour la génération de la fonction définissant la répartition des requêtes sur les différents serveurs, nous avons choisi d'implémenter deux politiques de répartition initiale : une politique aléatoire et une politique constante mais configurable. La fonction remplit à chaque seconde un tuple avec un élément par serveur représentant le pourcentage de charge associé à ce serveur. la somme de tous les éléments du tuple est égale à 100% des requêtes. Dans la politique aléatoire, chaque élément d'un tuple est choisi au hasard puis normalisé, tandis que dans la politique constante, une répartition valide est donnée en paramètre. Ici encore, nous avons implémenté des schémas simples modélisant une grande partie des cas d'utilisation, mais il est facile d'intégrer d'autres fonctions de répartition plus complexes.

L'implémentation de la dernière étape est la plus coûteuse, car c'est elle qui écrit réellement le fichier de traces. Une fois le nombre de requêtes prévu pour chaque serveur connu, il faut ensuite générer chaque ensemble de requêtes. Or, le serveur destinataire d'une requête est identifié grâce à la méthode de distribution. Notre outil de génération de traces n'ayant pas accès à ces méthodes, il n'est pas capable de générer une requête ciblant un serveur en particulier. Nous avons donc choisi d'appeler dans notre générateur (en *Python*) un sous-programme codé en C, nommé *gen_timestep*.

Ce sous-programme utilise l'API client du simulateur de service de métadonnées, détaillé à la section 3.2, afin de générer des requêtes destinées à un serveur

en particulier. Il génère des requêtes par brute force [3] et vérifie avec la méthode de distribution si pour chaque clé, le serveur en charge de cette clé correspond au serveur demandé dans la génération de requêtes. Lors de la génération de l'ensemble de requêtes, le sous-programme prend en compte le pourcentage de clés distinctes donné en argument. Il est aussi responsable d'assigner un *jobid* à chaque requête. Avoir des requêtes accédant aux mêmes clés en parallèle n'impacte pas l'évaluation des méthodes de distribution, c'est pourquoi nous avons choisi d'inscrire de manière arbitraire le *jobid* comme étant le nom de la clé accédée pour la requête.

Il est important de noter que les traces générées avec ce sous-programme garantissent une répartition des serveurs pour une distribution donnée : si l'on change la méthode de distribution, ou même certains paramètres d'une méthode, la distribution des requêtes sur les différents serveurs peut changer. De la même manière, la répartition est calculée à partir de l'état initial de la méthode, si une méthode change de répartition au cours du temps, ces changements ne seront pas pris en compte dans la génération de la trace. Pour ces deux raisons, il est nécessaire de fournir en paramètres au générateur de traces (en *Python*) certaines informations relatives à la méthode de distribution.

3.5.4 Vérification du générateur

Il est nécessaire de vérifier que les traces générées ont bien les caractéristiques attendues : profil temporel et répartition sur les différents serveurs ainsi que le pourcentage de clés distinctes. Pour vérifier cela, nous exécutons une trace générée avec notre simulateur de service de métadonnées. Celui-ci produit une courbe de charge des différents serveurs, en requêtes par seconde et en pourcentage du nombre de requêtes reçues, nous permettant de valider notre générateur de traces. La vérification du temps d'exécution de notre générateur se fait à vue d'œil : les traces n'ayant pas vocation à être régénérées à chaque test, un temps de génération plus long est acceptable (une dizaine de minutes pour des traces de plusieurs heures).

3.5.5 Choix des flux pertinents

En parcourant l'état de l'art [CXZ07, XZ12, TBD+17], nous pouvons constater qu'il n'existe pas de flux de requêtes standard et uniforme permettant d'évaluer des méthodes de distribution pour les systèmes objets : certains utilisent des bancs des test propres aux systèmes de fichier comme *MDtest* [MLMK11] ou *Postmark* [Kat97] ; d'autres se servent de COSBENCH [ZCW+12] qui est un bench-

3. Méthode utilisée en cryptanalyse pour trouver un mot de passe ou une clé. Il s'agit de tester, une à une, toutes les combinaisons possibles.

mark pour l'évaluation globale d'un système objet ; et d'autres encore se basent sur des articles dédiés à la caractérisation de flux, comme l'analyse de Mummert et. al. [MS96] ou celle de Wang et. al. [WXH+04] ; mais la plupart préfèrent générer leur propres traces afin d'évaluer leur méthode sur des flux choisis.

Il est cependant possible de trouver une tendance commune dans les flux d'entrées choisis : beaucoup de clients envoient un grand nombre de requêtes (de l'ordre des milliers de requêtes), produisant des moments de bursts. Ces requêtes peuvent être adressées à des serveurs différents (en aléatoire généralement) pour avoir une répartition plus réaliste, ou bien toutes adressées à un même serveur afin d'évaluer leur résistance à la charge. Les requêtes utilisées sont en général des opérations de création car ce sont les plus coûteuses.

Pour la génération de nos propres flux de requêtes, nous nous sommes inspirés de ces tendances et nous avons choisi de modéliser plusieurs profils temporels : le modèle *Montée en Charge (MC)*, le modèle *On/Off*, le modèle *Pic* et le modèle *Chaotique*. Ces modèles ont été nommés suivant les motifs du flux et ne suivent aucun standard existant.

Le modèle *Montée en Charge (MC)* : Ici, nous avons une faible densité de requêtes au début, puis le nombre de requêtes par secondes augmente jusqu'à obtenir un flux de requêtes intense. Ce profil temporel est représentatif d'une d'utilisation HPC : la plupart des simulations doivent être exécutées un grand nombre de fois (afin de faire varier des paramètres par exemple) et sont lancées en parallèle. Ces exécutions ne vont pas débuter en même temps et leur superposition en décalée va créer, au fur et à mesure de leur lancement, une forte demande en métadonnées, créant ainsi une montée en charge progressive. Un exemple de courbe représentant le profil temporel *Montée en charge* est présenté à la figure 3.10.

Pour générer des traces représentatives de ce modèle, nous avons fourni au générateur les paramètres temporels suivants : la trace totale dure 3h, elle reste initialement au niveau bas pendant 1h, monte jusqu'au niveau haut en 1h et enfin reste au niveau haut l'heure restante. Nous avons établi le niveau haut à 90% du nombre de requêtes que peut supporter le service de métadonnées et le niveau bas à 50%.

Le modèle *On/Off* : Ici, le flux de requêtes possède des périodes de rafales et des périodes creuses qui s'alternent. C'est un flux représentatif de l'usage d'un supercalculateur qui ne tourne pas toujours au maximum de ses capacités. Un exemple de courbe représentant le profil temporel *On/Off* est présenté à la figure 3.11.

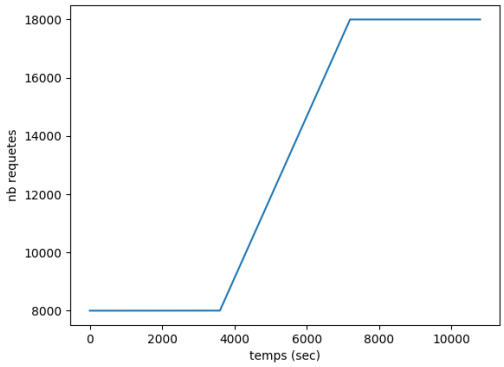


FIGURE 3.10: Profil temporel de type *MC* pour un système à 4 serveurs.

Pour générer des traces représentatives de ce modèle, nous avons fourni au générateur les paramètres temporels suivants : la trace totale dure 3h, elle reste initialement 20 minutes au niveau bas, monte en 5 minutes, reste au niveau haut pendant 30 minutes pour redescendre en 5 minutes aussi. Ce motif est répété 3 fois jusqu'à la fin de la trace. Ici encore, le niveau haut correspond à 90% du nombre de requêtes que peut supporter le service de métadonnées et le niveau bas à 50%.

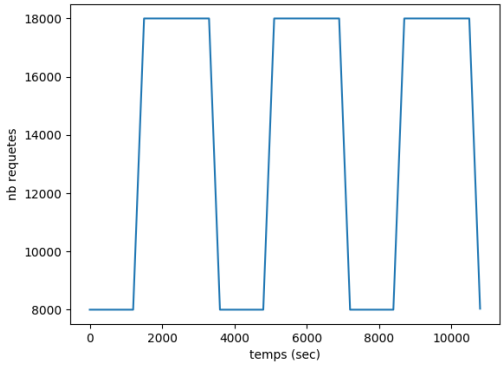


FIGURE 3.11: Profil temporel de type *On/Off* pour un système à 4 serveurs.

Le modèle *Pic* : Ici, le flux consiste en une faible densité de requêtes, qui va être perturbé par une courte mais intense arrivée de requêtes, pour finalement revenir au flux constant de faible densité. C'est un flux représentatif de l'usage d'un supercalculateur où serait lancé un calcul important nécessitant une grosse majorité des ressources du calculateur et générant énormément d'opérations d'accès

sur un court laps de temps. Un exemple de courbe représentant le profil temporel *Pic* est présenté à la figure 3.12.

Pour générer des traces représentatives de ce modèle, nous avons fourni au générateur les paramètres temporels suivants : la trace totale dure 3h, elle reste initialement 1h au niveau bas, monte en 10 minutes et redescend immédiatement en 10 minutes. Il reste ensuite au niveau bas jusqu'à la fin de la trace. Le niveau haut correspond à 95% du nombre de requêtes que peut supporter le service de métadonnées et le niveau bas à 20%.

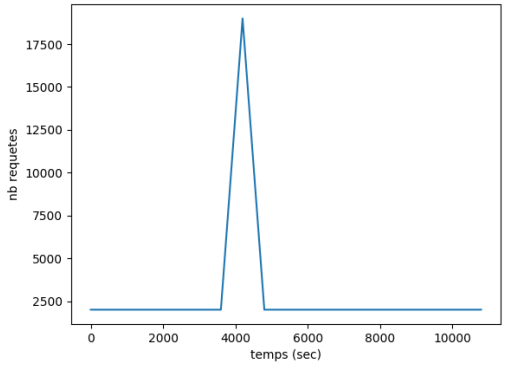


FIGURE 3.12: Profil temporel de type *Pic* pour un système à 4 serveurs.

Le modèle *Chaotique* : Ici, le flux de requêtes varie de manière aléatoire à chaque seconde. C'est un flux représentatif de l'usage d'un supercalculateur car une grande quantité de simulations aux comportements hétérogènes sont exécutées à tout instant sur le supercalculateur. L'aggrégation de tous ces accès produit un nombre aléatoire d'accès.

Pour générer des traces représentatives de ce modèle, nous avons fourni au générateur les paramètres temporels suivants : la trace totale dure 3h et pour chaque seconde, le nombre de requêtes générées prend une valeur aléatoire comprise entre le niveau haut et le niveau bas. Le niveau haut correspond à 90% du nombre de requêtes que peut supporter le service de métadonnées et le niveau bas à 25%.

Pour la répartition sur les différents serveurs, nous avons cherché des distributions pouvant mettre en difficulté les méthodes de distribution de métadonnées :

- En imposant un fort déséquilibre où un serveur est beaucoup plus chargé que les autres (au moins le double de la charge de chaque serveur) ;

- En imposant un faible déséquilibre où au contraire un serveur est un peu moins chargé que les autres (au plus le double de la charge de chaque serveur) ;
- En imposant une répartition aléatoire représentative des accès chaotiques des métadonnées dans des systèmes de production.

Ces répartitions permettent de couvrir la majorité des cas possibles, afin d'observer la majorité des différents comportements des méthodes.

Nous avons choisi pour l'ensemble des flux (sauf pour le type *Chaotique* où cela est précisé) d'appliquer un pourcentage de clés distinctes de 10%, ce qui signifie que 10% des requêtes ciblent une nouvelle clé. Cela permet de représenter un ensemble d'utilisateurs accédant chacun à leurs données et à des données partagées proposées par le système par exemple.

La répartition du flux de requêtes sur les différents serveurs dépend de la méthode de distribution prise en compte lors de la génération des traces. Afin de ne pas se perdre dans un trop grand nombre de flux lors de l'évaluation des différentes méthodes, nous avons fait le choix de générer toutes les traces en se basant sur la méthode de distribution du *Static Hashing* et d'utiliser ces mêmes traces pour évaluer toutes les méthodes. La répartition des requêtes sur les serveurs est donc garantie pour notre méthode de référence mais pourrait varier avec d'autres méthodes de distribution. Cela ne pose pas de problêmes dans notre cas, puisque l'ensemble des méthodes évaluées dans ce manuscrit sont basées sur la même répartition initiale.

Les choix précédents nous ont permis d'établir une liste de traces correspondant à des flux d'entrées utiles à l'évaluation des méthodes de distribution de métadonnées. Pour obtenir des courbes de comparaisons lisibles, nous avons choisi de limiter le nombre de serveurs à 4 et avons généré les traces en conséquence (pour le nombre de requêtes que doit supporter le système, ou encore la répartition entre les serveurs).

- Les traces avec un profil temporel de type *Montée en Charge* : la caractérisation du profil temporel est montrée sur la figure 3.10. Pour la répartition entre les serveurs nous avons choisi les pourcentages qui suivent : [85-5-5-5] pour avoir un fort déséquilibre, [10-30-30-30] pour avoir un faible déséquilibre.
- Les traces avec un profil temporel de type *On/Off* : la caractérisation du profil temporel est montrée sur la figure 3.11. Pour la répartition entre les serveurs nous avons choisi les mêmes pourcentages que précédemment : [85-5-5-5], [10-30-30-30].
- Les traces avec un profil temporel de type *Pic* : la caractérisation du profil temporel est montrée sur la figure 3.12. Pour la répartition entre les serveurs

nous avons choisi d'utiliser seulement le motif de fort déséquilibre [85-5-5-5] car c'est le seul cohérent avec le cas d'usage représentatif du profil temporel.

- Les traces avec un profil temporel de type *Chaotique* : la caractérisation du profil temporel est aléatoirement définie prenant à chaque seconde une valeur entre 25% et 90% du nombre de requêtes que peut supporter le service de métadonnées. Pour la répartition entre les serveurs nous avons choisi de ne caractériser que la répartition aléatoire [x_1-x_2-x_3-x_4] (avec $\sum x_i = 100$), afin de représenter un flux complètement aléatoire. De plus pour la génération de ce flux, nous avons appliqué un pourcentage de clés distinctes de 50%, ce qui accroît le caractère aléatoire de la distribution.
- La trace réelle extraite d'un système de production : la trace est extraite du supercalculateur industriel et académique Joliot Curie [CEA19]. Avec 1 656 nœuds Skylake (Intel Xeon 8168 bi-processeurs) et 828 nœuds Knight Landing (*many cores* Intel Xeon Phi 7250), ce supercalculateur possède une puissance de calcul de 9 Pflops. Le système de stockage dédié de 5 PB est un système Lustre gérant 2 MDSs et 42 OSTs. La trace reflète 24h d'usage quotidien d'un cluster de calcul et est représentative de l'usage d'un large ensemble d'applications HPC. En effet, l'utilisation de ce cluster réunit plusieurs champs de recherche comme l'apprentissage automatique, la chimie, la physique ou la climatologie avec des codes de simulation connus comme abinit [GAA+20], cp2k [HISV14], gromacs [AMS+15] ou tensorflow [AAB+15]. Initialement, les traces ont été enregistrées sur le système de fichiers Lustre, puis les opérations spécifiques à l'arborescence (*mkdir, rmdir, unlink...*) ont été adaptées à la sémantique CRUD.

3.6 Conclusion

Nous avons présenté *MeDiE*, notre outil d'évaluation de méthodes de distribution, l'utilisation et l'intégration dans cet outil de certaines méthodes de distribution, mais aussi la génération de traces globales correspondant à un flux caractérisé.

Le simulateur rend possible l'exécution des interactions entre les clients et les serveurs en prenant en compte les comportements particuliers induit par une méthode de distribution pour ces deux côtés. Ensuite, la définition du processus de simulation a permis de contrôler la gestion du temps de la simulation et de la prise de mesures, afin d'obtenir des évaluations comparables.

L'intégration de chaque méthode de distribution dans *MeDiE* a nécessité une association des fonctionnnalités des méthodes aux différents modules de l'outil avant l'implémentation de la méthode.

Pour finir, l'écriture d'un générateur de traces et la caractérisation de différents types de flux complète notre outil. Il offre la possibilité d'évaluer les méthodes de distribution sur des flux particuliers correspondant à des usages d'un supercalculateur HPC.

Chapitre 4

Évaluation

Ce chapitre est dédié à l'analyse des performances des méthodes de distribution présentées dans le chapitre 2. Nous commençons par présenter l'environnement dans lequel se dérouleront nos évaluations ainsi que le protocole suivi. Ensuite nous évaluons chaque méthode et la comparons à la précédente afin d'attester du gain qu'apporte la nouvelle méthode : nous débutons par la méthode du *Static Hashing*, puis la méthode du *Dynamic Hashing*, ensuite la distribution adaptative à la charge *LAD* et enfin la variante avec fenêtre temporelle *LAD-TW*.

4.1 Environnement et conduite de test

Tous les résultats présentés dans ce document proviennent de tests effectués sur les machines d'un cluster interne du CEA. Les tests ont été validés sur des nœuds de calcul d'architecture Haswell, composés de processeurs Intel(R) Xeon(R) CPU E5-2698 v3 fréquencés à 2.30GHz avec 32 cœurs par nœud (répartis sur 2 sockets). La mémoire pour un nœud est de 125G et l'interconnexion se fait par infiniband FDR avec un débit aggrégé de 60Gb/s.

Les processus clients et serveurs sont exécutés sur des machines virtuelles (VM) indépendantes coordonnées par un gestionnaire de machine virtuelle développé au CEA appelé PCOCC [DB16], permettant d'héberger des VMs sur des nœuds de calcul comme une allocation de job usuelle. Chaque machine virtuelle s'exécute sur 4 cœurs avec Centos 7.4 comme système d'exploitation. Chaque test représente un nouveau commencement du système et tous les espaces de stockage sont initialement vides.

Cet environnement de test est à prendre en compte lors de la génération des traces. Dans cette optique, nous avons choisi que la dernière requête associée à chaque clé serait une opération de suppression afin de pallier une surcharge du stockage des machines virtuelles [1].

Pour notre évaluation, nous avons choisi de suivre le même cheminement que celui du chapitre 2 en analysant chaque méthode par rapport à la précédente.

Nous commençons par évaluer la méthode de référence qu'est le *Static Hashing* sur tous les flux exprimés, et exposons les limites observées pour cette méthode. Afin de limiter les simulation en un nombre réalisables (et en particulier lors des étapes d'études de paramètres), nous avons choisi de ne pas garder l'intégralité des flux bien gérés par la méthode pour l'évaluation de la suivante. Par contre, tous les flux mettant en difficulté la méthode sont utilisés pour évaluer la suivante.

Nous procédons tout d'abord à une étude de paramètres pour chaque méthode afin de trouver la combinaison de paramètres permettant la meilleure distribution. Cependant, ces paramètres sont aussi dépendants du flux sur lequel ils sont évalués, et cela doit être pris en compte dans l'étude de paramètres. La démarche idéale est une étude exhaustive mais bien évidemment coûteuse. Par exemple, la méthode *LAD-TW* possède 132000 combinaisons de paramètres possibles à tester pour chaque flux.

Nous avons donc choisi de tester les combinaisons de paramètres en utilisant une approche itérative sauf pour la méthode du *Static Hashing* qui ne demande aucun paramètre :

- Les paramètres à considérer pour la méthode du *Dynamic Hashing* sont : α, N_{entry} et le temps entre chaque redistribution. Pour cette première méthode, il est nécessaire de tester toutes les combinaisons, et il sera ensuite possible de conserver les valeurs de certains paramètres.
- Pour la méthode *LAD*, il faut prendre en compte les paramètres suivants : la marge utilisée pour l'évaluation du seuil de redistribution et N_{entry} qui est directement impacté par le nouvel algorithme de redistribution et qui doit être réétudié. Le paramètre α n'est pas impacté par les modifications de la méthode *LAD*, il n'est pas nécessaire de l'évaluer à nouveau.
- La méthode *LAD-TW*, possède 4 paramètres : α, N_{entry}, la marge utilisée pour l'évaluation du seuil de redistribution et la taille de la fenêtre pour l'évaluation de la charge. Parmi ces paramètres, seuls α et la taille de la fenêtre doivent être étudiés car ils sont en lien direct avec les modifications au sujet de l'évaluation de la charge.

1. L'espace alloué pour une machine virtuelle ne permet pas de stocker plus de 16 GigaOctets, ce qui est problématique pour des flux de requêtes très déséquilibrés.

Chaque combinaison de paramètres est testée sur tous les flux utilisés pour l'évaluation de la méthode. Pour les traces réelles, l'étude de paramètres se fait sur un échantillon de 3h.

Afin d'évaluer équitablement chaque méthode, il est nécessaire d'établir une métrique permettant de juger objectivement une méthode. *MeDiE* prend une mesure de la charge des serveurs, ainsi que du temps de réponse des clients pour une requête. Pour l'évaluation côté client, la comparaison des temps de réponse pour une requête est une métrique suffisante que l'on veut minimiser.

Pour l'évaluation côté serveur, les métriques de charge ne suffisent pas pour juger de l'efficacité d'une méthode : les méthodes permettant des redistributions amélioreront la répartition des charges, mais chaque redistribution correspond aussi à un coût qui doit être pris en compte dans l'évaluation. Nous avons donc décidé de créer un score pour évaluer la rentabilité de l'équilibrage de charge d'une méthode de distribution.

Afin de mettre en place ce score, nous avons calculé une métrique que nous appellerons **écart de charge** et qui correspond pour chaque serveur à l'écart entre sa charge actuelle et la charge idéale qu'il aurait dû avoir. Nous prenons en compte l'*écart de charge moyen* et l'*écart de charge maximum*. Cette métrique nous permet de comparer l'équilibrage de la charge pour chaque méthode : plus l'écart de charge est faible, meilleur est l'équilibrage de la méthode. Cependant, elle ne prend pas en compte le coût qu'induit toutes les redistributions, et une évaluation avec cette métrique seule avantagerait les méthodes redistribuant trop régulièrement.

Il nous faut donc une métrique calculant le coût qu'impliquent ces redistributions. Nous avons choisi de prendre en compte le nombre de communications induites par les redistributions. Cette métrique dépend de l'implémentation de la méthode dans l'outil, mais permet d'établir un coût concret. Le nombre de communications d'une redistribution correspond à la récupération d'informations pour l'algorithme de redistribution ainsi que les transferts de métadonnées qui en découlent. Afin de ne pas minimiser ce cout, nous avons choisi de comptabiliser le nombre de communications dans le pire cas. D'après l'implémentation des méthodes dans notre outil (voir section 3.4), la méthode du *Dynamic Hashing* demande pour chaque redistribution $2N+N/2$ communications (avec N le nombre de serveurs), tandis que les méthodes *LAD* et *LAD-TW* en nécessitent pour chaque redistribution $4N + N/2$.

Nous possédons à présent une métrique permettant d'attester de l'efficacité de l'équilibrage de charge d'une méthode (l'écart de charge) que nous souhaitons minimiser, ainsi que d'une métrique attestant des coûts induits par chaque méthode (nombre de communications dues aux redistributions) à minimiser également. Il faut maintenant les associer pour créer un score objectif. Cependant, ces deux métriques sont indépendantes et ne sont pas exprimées dans la même unité, ce qui empêche de les combiner directement. Par conséquent, nous avons tout d'abord normalisé ces métriques, c'est-à-dire ramené leur valeur dans le même intervalle $[0, 1]$ avant de procéder à une linéarisation, généralement utilisée dans le domaine de l'optimisation multicritère afin d'obtenir un unique score. Notre score se traduit par la formule suivante que nous souhaitons minimiser :

$$score = w_1 * ecart\ de\ charge + w_2 * nombre\ de\ communications \tag{4.1}$$

où w_1 et w_2 sont deux réels positifs dont la somme est 1 et permettant d'accorder plus ou moins d'importance à une métrique. L'établissement de ce score nous permet de comparer équitablement différentes méthodes de distribution : la méthode possédant le score le plus petit est celle ayant la meilleure rentabilité, c'est-à-dire, permettant un meilleur équilibrage de la charge par rapport aux coûts de redistribution engendrés.

En complément du score de chaque méthode, nous avons choisi d'analyser à posteriori et de classer chaque redistribution effecuées selon son impact. Si la redistribution a été effectuée alors que l'écart de charge était supérieur à 5%, elle est considérée comme *nécessaire*. Si en plus, elle a permis lors de l'étape suivante, de réduire l'écart de charge d'au moins 5% ou bien d'obtenir un écart inférieur à 5%, la redistribution est considérée comme *utile*. Toute redistribution non utile est considérée *inutile*. Ainsi, lors de l'évaluation, les courbes permettent d'identifier si une redistribution a été utile ou non.

Tout au long de cette évaluation nous allons regrouper les différents flux en deux catégories : les flux d'utilisation normale qui ne provoque qu'un faible déséquilibre et les flux intensifs qui génèrent de fort déséquilibres entre les serveurs.

4.2 Évaluation du *Static Hashing*

La méthode du *Static Hashing* ne possède pas de système de redistribution, le coût induit est donc nul.

4.2.1 Étude des flux d'utilisation normale

La sous-figure 4.1a correspond aux flux avec un faible déséquilibre : les traces ayant été générées pour obtenir ce déséquilibre, il n'est pas intéressant de s'attarder dessus, mais elles sont toutefois nécessaires comme base de comparaison pour l'étude des méthodes dynamiques. Les courbes de certains serveurs peuvent sembler absentes de la figure, mais celles-ci sont simplement superposées à d'autres courbes. Le flux *Chaotique* est présenté à la sous-figure 4.1b et nous constatons que le flux est assez équilibré, ce qui confirme le caractère aléatoire de la fonction de hachage.

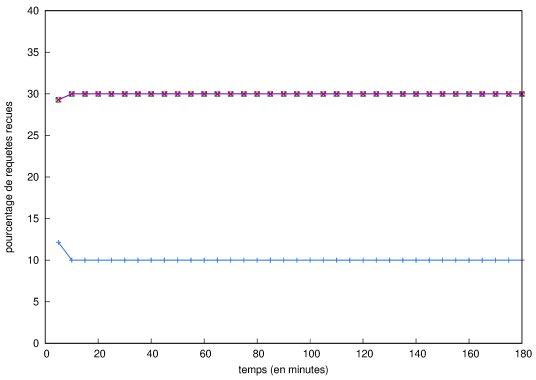
Nous pouvons aussi étudier le comportement du *Static Hashing* sur la trace réelle dont nous disposons : la sous-figure 4.1d indique le pourcentage de charge de chaque serveur. Nous pouvons constater que le flux de requêtes est bien équilibré, ce qui s'explique par la rigueur demandée pour l'utilisation du supercalculateur. En effet, si des utilisateurs provoquent des motifs d'accès aux métadonnées pouvant affecter la performance globale du système, ceux-ci sont contactés par les administrateurs du supercalculateur afin d'optimiser leurs méthodes d'accès.

L'ensemble des courbes affichant le nombre de requêtes reçues pour chaque serveur selon chaque flux d'entrée se trouve à la figure 4.2 (colonne de gauche pour les flux d'utilisation normale). Nous pouvons ainsi observer le comportement d'une méthode de hashing sans redistribution et repérer les moments de déséquilibres problématiques. Mis en correlation avec les courbes de répartition de charge, nous pouvons évaluer l'ampleur du déséquilibre constaté et en déduire si cela est problématique.

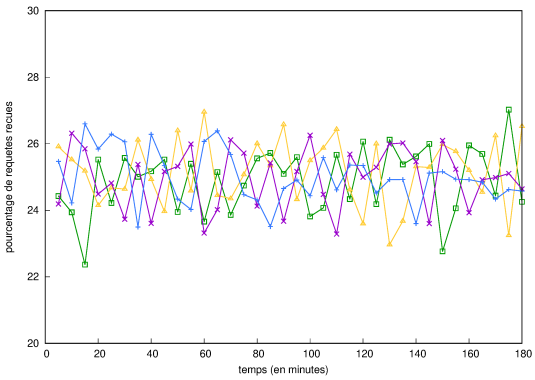
De la même manière qu'avec les charges moyenne des serveurs, la figure 4.3 regroupe le temps d'accès moyen pour une requête pour un client selon les différents flux. Chaque boite à moustache correspond à la prise de mesure d'une période pour tous les clients ayant envoyé des requêtes. L'étude des temps de réponse pour une requête est exposée ici afin de servir de base de comparaison et ne reflète pas la performance d'un système réel.

4.2.2 Étude des flux intensifs générés

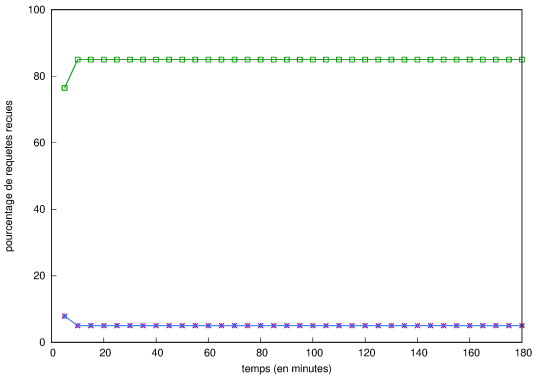
La sous-figure 4.1c montre le pourcentage de charge de chaque serveur pour les flux avec un fort déséquilibre et nous pouvons retrouver dans les figures 4.2 et 4.3 respectivement le nombre de requêtes reçues pour chaque serveur et le temps de réponse pour une requête pour un client selon tous ces flux. Ici encore, les traces ont été générées pour obtenir ce déséquilibre, et l'étude de ces courbes seules ne permet pas de ressortir beaucoup d'informations. Elles seront cependant utilisées comme base de comparaison pour l'étude des méthodes dynamiques.



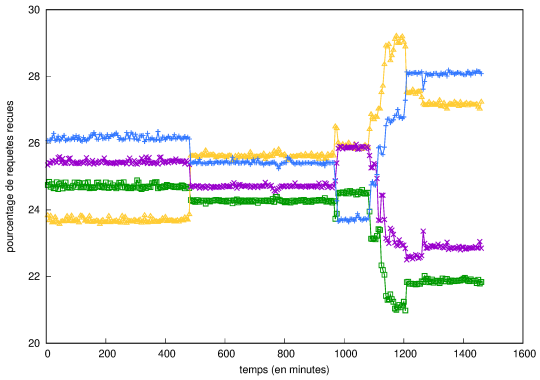
(a) Flux de type *MC* et *On/Off* avec un faible déséquilibre.



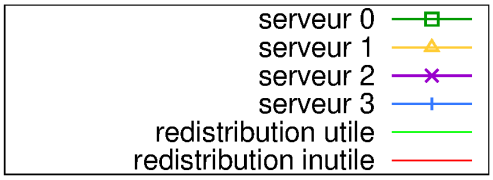
(b) Flux de type *Chaotique*.



(c) Flux de type *MC*, *On/Off* et *Pic* avec un fort déséquilibre.

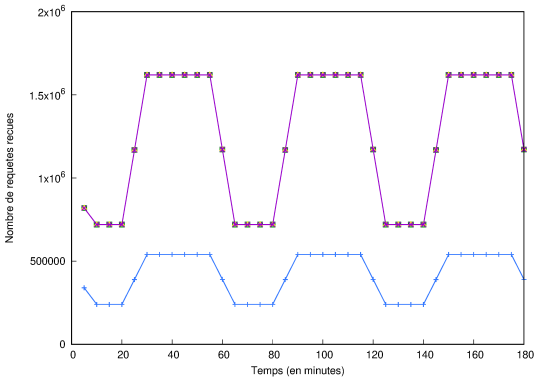


(d) Trace réelle.

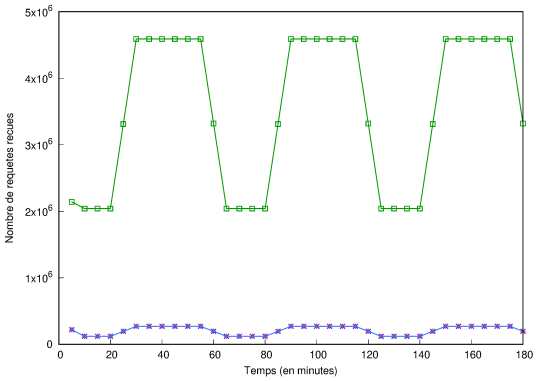


(e) Légende.

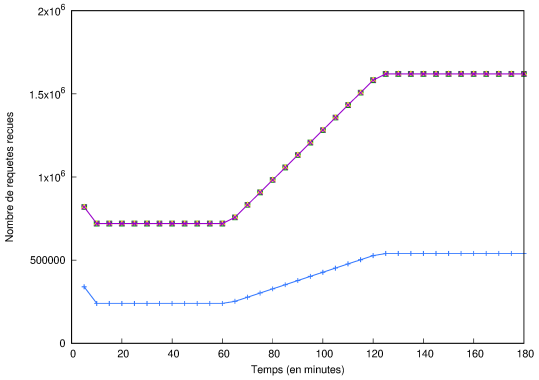
FIGURE 4.1: Répartition de la charge avec le *Static Hashing*.



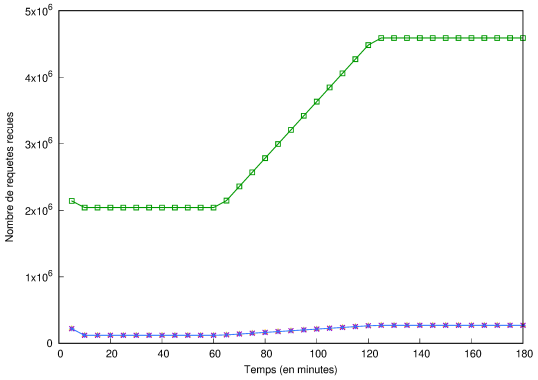
(a) Flux de type *On/Off* avec un faible déséquilibre.



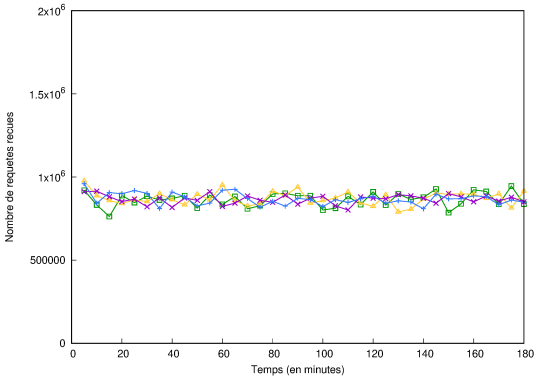
(b) Flux de type *On/Off* avec un fort déséquilibre.



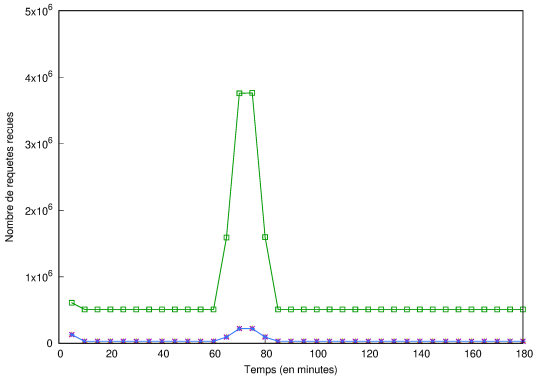
(c) Flux de type *MC* avec un faible déséquilibre.



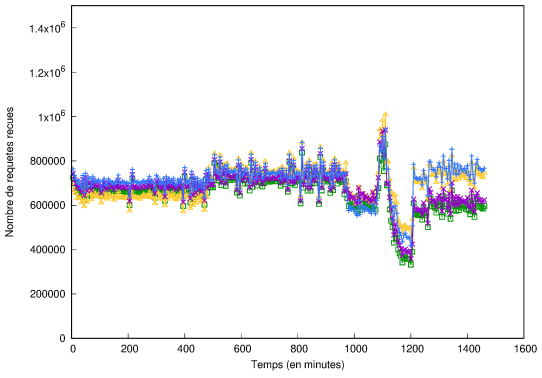
(d) Flux de type *MC* avec un fort déséquilibre.



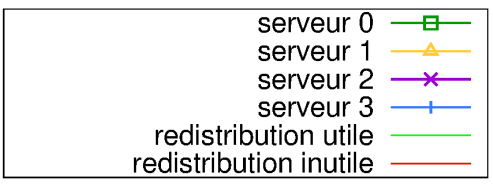
(e) Flux de type *Chaotique*.



(f) Flux de type *Pic* avec un fort déséquilibre.

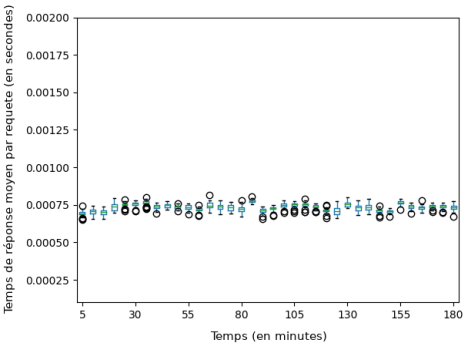


(g) Trace réelle.

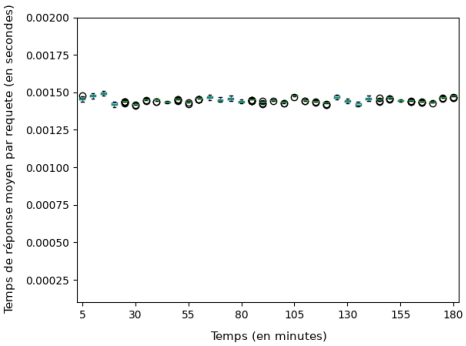


(h) Légende.

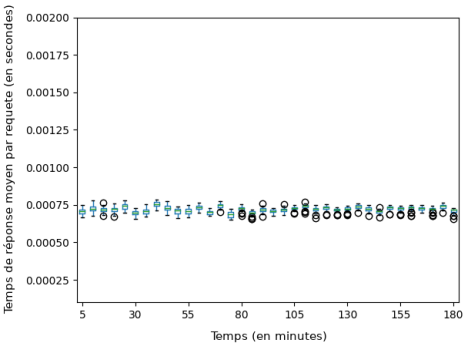
FIGURE 4.2: Charge moyenne avec le *Static Hashing*.



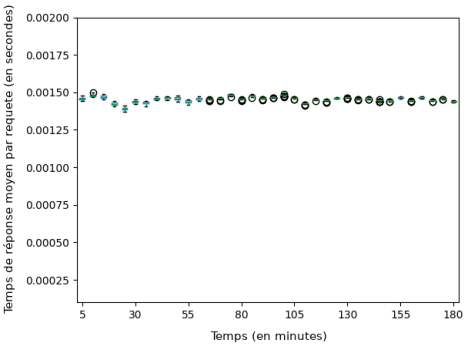
(a) Flux de type *On/Of* avec un faible déséquilibre.



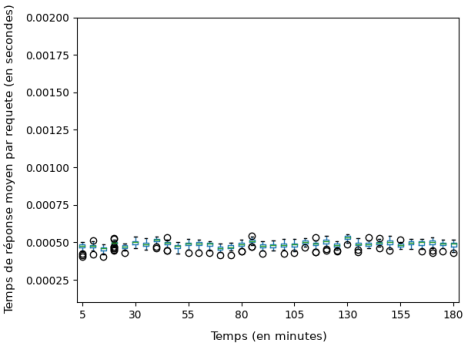
(b) Flux de type *On/Off* avec un fort déséquilibre.



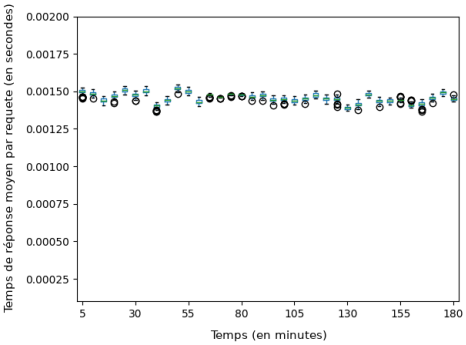
(c) Flux de type *MC* avec un faible déséquilibre.



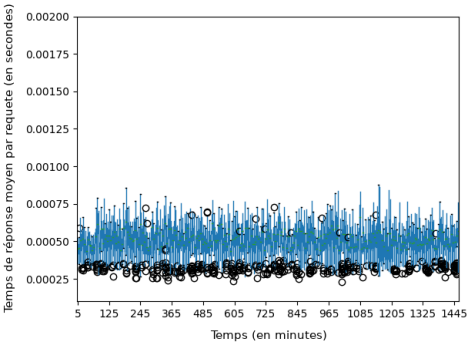
(d) Flux de type *MC* avec un fort déséquilibre.



(e) Flux de type *Chaotique*.



(f) Flux de type *Pic* avec un fort déséquilibre.



(g) Trace réelle.

FIGURE 4.3: Temps moyen par requête avec le *Static Hashing*.

4.2.3 En résumé

Pour l'ensemble des flux testés, nous avons calculé le score qu'obtient le *Static Hashing* et le tableau 4.1 en est un récapitulatif.

Flux	*On/Off* faible	*On/Off* fort	*MC* faible	*MC* fort	*Chaotique*	*Pic* fort	Réel
Score obtenu	0.0498	0.1992	0.0498	0.1992	0.0504	0.1977	0.0076

TABLE 4.1: Scores du *Static Hashing*.

Cette évaluation nous a permis de retrouver les limites théoriques exhibées à la section 2.1 : la méthode est initialement bien répartie, comme nous pouvons le voir sur les courbes d'exécution de la trace réelle. Toutefois, lorsque la distribution est mise en défaut et provoque des déséquilibres, le *Static Hashing* ne permet pas d'y remédier et le déséquilibre subsiste. C'est pourquoi l'étude d'une méthode dynamique est essentielle.

Afin d'évaluer intégralement la prochaine méthode, nous avons choisi de garder tous les flux testés sur le *Static Hashing*, ce qui correspond aux flux suivants :

- le flux réel,
- le flux de type *MC* avec faible déséquilibre,
- le flux de type *On/Off* avec faible déséquilibre,
- le flux de type *Chaotique*,
- le flux de type *MC* avec fort déséquilibre,
- le flux de type *On/Off* avec fort déséquilibre,
- le flux de type *Pic* avec fort déséquilibre.

4.3 Évaluation du *Dynamic Hashing*

Avant d'évaluer la méthode et la comparer au *Static Hashing* pour voir les effets qu'apporte la dynamicité, nous devons d'abord trouver les bons paramètres pour la configurer. En effet, ceux-ci peuvent avoir une forte influence sur les performances de la méthode.

4.3.1 Étude des paramètres

La méthode du *Dynamic Hashing* nécessite trois paramètres :

- Le paramètre d'oubli α dont les valeurs testées doivent être comprises entre 0 et 1 et varient par pas de 0.1.

- Le paramètre N_{entry} qui représente la taille de la table d'index et dont les valeurs testées sont présentées dans le tableau 4.2.
- L'intervalle de redistribution, qui correspond à la période de redistribution et dont les valeurs testées sont présentées dans le tableau 4.3.

N_{entry}	50	100	500	1000	5000	10000

TABLE 4.2: Valeurs de N_{entry} testées pour le *Dynamic Hashing*.

Intervalle de redistribution (min)	10	30	60	120

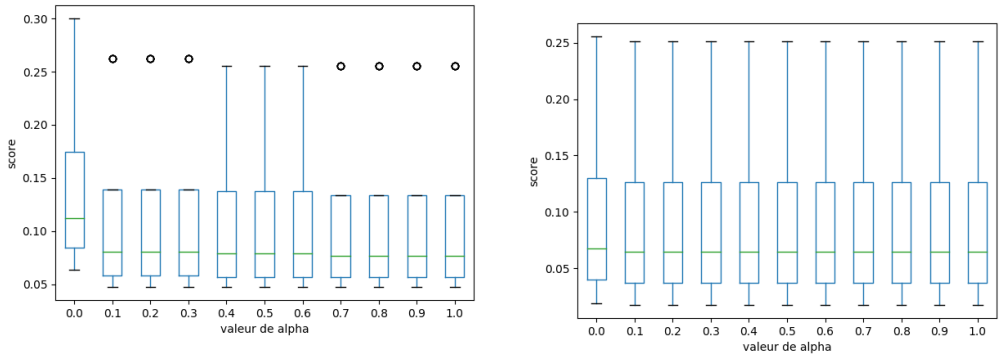
TABLE 4.3: Périodes de redistribution testées pour le *Dynamic Hashing*.

Afin d'étudier l'impact de chaque paramètre indépendamment, nous avons testé toutes les combinaisons de valeurs et nous avons exécuté l'ensemble de ces simulations sur tous les flux. Pour chaque simulation, nous calculons le score comme défini par la formule 4.1 qui nous indique l'impact de chaque paramètre.

Pour tous les flux avec un fort déséquilibre, faire varier les paramètres ne permet pas d'observer une quelconque différence de performance. Ceci est dû à l'algorithme de redistribution RELAB, et sera expliqué en détail par la suite. Pour cette raison, nous avons décidé d'écarter ponctuellement les flux comportant un fort déséquilibre. Pour les flux à faible déséquilibre, une analyse est possible. Nous constatons que les résultats pour tous les flux avec un faible déséquilibre sont similaires, c'est pourquoi nous ne présenterons ici qu'une seule de ces études. Nous avons également effectué cette étude en utilisant la trace réelle tronquée à 3h.

La figure 4.4 exprime l'impact du facteur d'oubli α avec un flux de type *MC* et avec la trace réelle.

Nous comparons l'impact des différents paramètres en fonction du score obtenu lors de chaque simulation qui est établi à partir de l'équilibrage des charges. L'étude de la trace réelle avec la méthode du *Static Hashing* nous a permis d'affirmer que le flux de requêtes était équilibré sur les 3 premières heures. Cela a donc un impact sur notre étude de paramètres : il n'y a pas de forts écarts de charges et les redistributions exécutées ont peu d'influence. Les scores obtenus sont donc très proches, ce qui rend plus complexe l'évaluation de ce paramètre. Les flux générés forcent un faible déséquilibre, ce qui nous permet tout de même d'observer des



(a) Flux de type *MC* avec un faible déséquilibre.

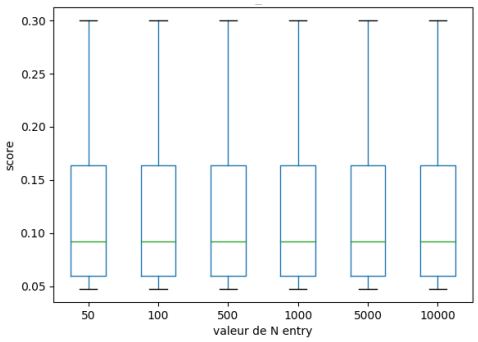
(b) Trace réelle.

FIGURE 4.4: Score en fonction du paramètre α.

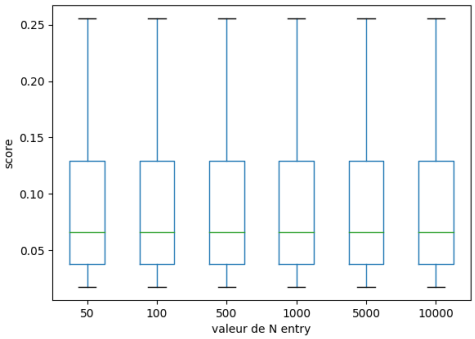
différences de score pour certaines valeurs de α.SUr la figure 4.4, la valeur 0 provoque un score significativement plus grand. Ceci peut s'expliquer intuitivement : $\alpha = 0$ signifie qu'on ne prend en compte que le passé afin d'évaluer une charge. Ainsi, peu importe les requêtes reçues à l'époque actuelle, l'évaluation de charge prendra toujours le nombre de requêtes initiales, c'est-à-dire 0 dans ce cas, rendant toute redistribution inutile. Nous pouvons aussi noter que les valeurs 0.4, 0.5 et 0.6 induisent un score avec une plus grande variabilité par rapport aux autres valeurs testées. Nous n'expliquons pas ces variations, ce qui nécessiterait une étude approfondie avec d'autres exemples. Globalement, nous pouvons constater que le paramètre α n'a pas un fort impact sur ces simulations, mise à part la valeur 0. Dans la suite de cette étude, nous avons choisi pour α la valeur de 0.7, car nous souhaitons favoriser les accès de l'époque actuelle par rapport à ceux des époques passées, sans pour autant les mettre de côté.

La figure 4.5 exprime l'effet du paramètre N_{entry} avec un flux de type *MC* et avec la trace réelle.

Nous pouvons observer que ce paramètre n'a pas d'impact sur le score obtenu. Ici encore, il est possible de l'expliquer par le faible déséquilibre imposé par les flux. En effet, le paramètre N_{entry} joue un rôle fort lors de la redistribution et permet une meilleure granularité dans les métadonnées transférées. Des flux relativement équilibrés aboutissent à peu de redistribution et ne permettent pas d'évaluer l'influence de cette granularité. Nous avons choisi pour N_{entry} la valeur 1000, car nous souhaitons garder une MLT de taille réaliste et ne surchargeant pas en mémoire les clients, y compris ceux possédant peu de ressources.



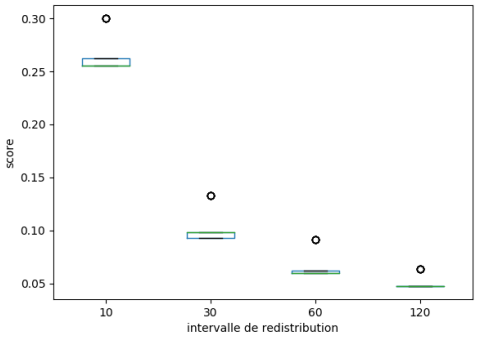
(a) Flux de type *MC* avec un faible déséquilibre.



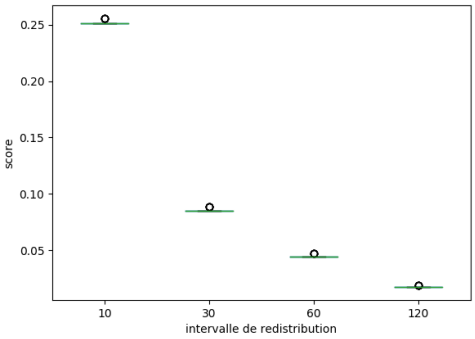
(b) Trace réelle.

FIGURE 4.5: Score en fonction du paramètre N_{entry}.

La figure 4.6 exprime l'impact de l'intervalle de redistribution avec un flux de type *MC* et avec la trace réelle.



(a) Flux de type *MC* avec un faible déséquilibre.



(b) Trace réelle.

FIGURE 4.6: Score en fonction de l'intervalle de redistribution.

Pour ce paramètre, nous pouvons observer de fortes différences, ce qui fait de l'intervalle de redistribution le paramètre dominant de cette étude. Nous constatons clairement que plus le nombre de redistributions augmente, plus le score croît. En effet, chaque redistribution possède un coût qui est pris en compte dans le calcul du score de chaque simulation. Les exécutions procédant à beaucoup de redistributions génèrent un plus fort coût de redistribution qui n'est pas compensé par le gain en équilibrage de charge. Ainsi, nous pouvons remarquer qu'il est préfé-

rable d'attendre 2h avec un faible déséquilibre, plutôt que de redistribuer toutes les heures par exemple. Nous avons choisi pour l'intervalle de redistribution la valeur de 120 (redistribution toutes les 2h) car c'est incontestablement la valeur nous donnant le meilleur score.

Cette étude de paramètres nous a permis d'établir que la combinaison de valeurs $\alpha = 0.7$, $N_{entry} = 1000$ avec un intervalle de redistribution de 120 minutes nous apporte le meilleur score possible et nous allons donc étudier le comportement du *Dynamic Hashing* avec ces valeurs.

4.3.2 Étude des flux d'utilisation normale

Pour l'étude des méthodes dynamiques nous avons introduit le principe de redistributions utiles et inutiles (voir section 4.1), où une redistribution utile permet de diminuer l'écart de charge entre un serveur et l'optimal d'au moins 5% ou bien d'obtenir un écart de charge inférieur à 5%. Pour la suite de ce chapitre, dans l'ensemble des figures, une ligne verticale verte signifie qu'une redistribution utile a été effectuée, tandis qu'une ligne verticale rouge signifie que la redistribution n'a pas été utile.

Pour le *Dynamic Hashing*, les redistributions s'effectuent suivant l'intervalle de redistribution dont la valeur a été définie à 120 minutes, il n'y a donc qu'une seule redistribution pour les exécutions des flux générés et 12 redistributions pour la trace réelle. Les courbes de répartition des charges sur les différents serveurs s'avèrent être similaires pour certains flux, nous avons donc choisi de montrer sur la figure 4.7 seulement les courbes différentes. La sous-figure 4.7a correspond aux flux de type *MC* et *On/Off* et la sous-figure 4.7b au flux *Chaotique*. Bien que certaines courbes de répartition des charges soient similaires, il est possible de voir le comportement de la méthode sur les différents flux avec le nombre de requêtes reçues, en regardant les courbes de la figure 4.8. Pour l'étude des flux provoquant un faible déséquilibre, nous pouvons faire l'observation suivante : une fois la redistribution exécutée, les courbes de chaque serveur convergent vers un nombre de requête équitable que chacun reçoit. L'équilibrage a donc été utile, et nous gardons cet équilibre jusqu'à la fin de l'exécution. Cela nous amène à penser que si la redistribution avait été faite plus tôt, le déséquilibre aurait été compensé plus tôt. Les redistributions du *Dynamic Hashing* étant périodiques, cela aurait induit un plus grand nombre de redistributions : la première aurait été utile tandis que celles d'après auraient été superflues.

Les courbes du temps de réponse moyen de la figure 4.9 soulignent l'impact de la redistribution : nous pouvons voir une réduction du temps de réponse à partir

de la 120^{e} minute, ce qui correspond à la redistribution effectuée. Ce gain de temps est très visible pour le flux *On/Off*, tandis qu'il est moins prononcé pour le flux de type *MC*.

La redistribution répartie les entrées plus accédées que d'autres, impactant aussi les futurs accès : les requêtes visant les entrées déplacées seront par la suite mieux réparties sur les différents serveurs. Les serveurs n'étant pas surchargé, les temps d'attente pour le traitement d'une requête sont réduits.

4.3.3 Étude des flux intensifs générés

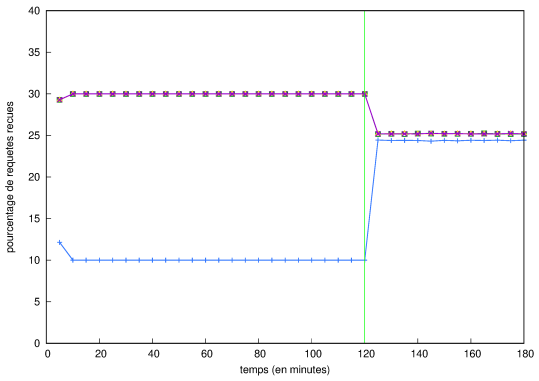
Le comportement du *Dynamic Hashing* s'avère être identique pour tous les flux intensifs, par soucis de lisibilité, nous ne présenterons donc pour la répartition des charges qu'une seule de ces courbes (sous-figure 4.7c). Nous pouvons voir que la redistribution n'a pas eu d'impact sur la répartition des charges. Les courbes de la figure 4.8 indiquant le nombre de requêtes reçues par serveur pour les flux provoquant un fort déséquilibre, et celles de la figure 4.9 affichant les temps de réponse associés viennent renforcer ce constat.

L'origine de ces redistributions inefficaces se trouve dans l'algorithme *RELAB*. Celui-ci cherche pour chaque serveur libre, une surcharge correspondant à sa disponibilité. Il faut donc que le serveur surchargé donne une charge plus faible ou égale à la disponibilité du serveur libre. Le problème a lieu lorsqu'un serveur est significativement plus chargé que tous les autres, comme cela se produit pour les flux intensifs. L'algorithme *RELAB* cherche donc un serveur surchargé dont la surcharge est inférieure ou égale à la disponibilité d'un des serveurs libres. Comme un seul serveur est surchargé, cela signifie que la compensation de cette surcharge est divisée sur les autres serveurs et qu'il n'existe pas de serveurs libres dont la disponibilité est égale à la surcharge donnée. L'algorithme n'étant pas conçu pour décomposer la surcharge d'un serveur, il n'est alors pas capable de trouver une nouvelle répartition équilibrée.

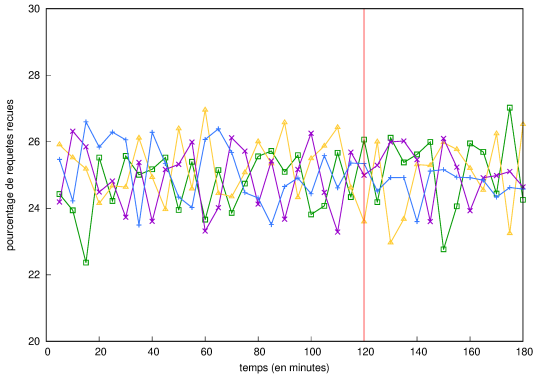
Pour ces types de flux, le *Dynamic Hashing* est donc moins efficace que le *Static Hashing* puisque la distribution des charges est identique mais que le *Dynamic Hashing* génère un surcoût dû aux redistributions inefficaces.

4.3.4 En résumé

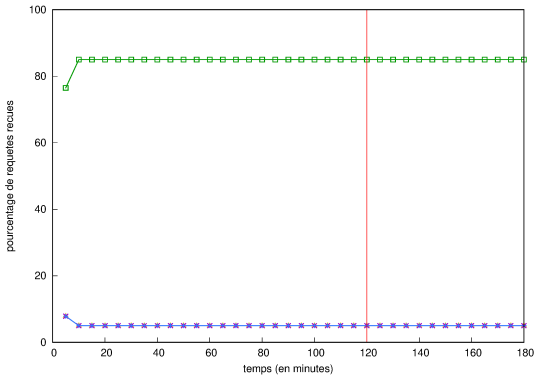
Le dynaminc hashing est une méthode permettant un rééquilibrage périodique. Celui-ci se révèle être utile pour des cas de faibles déséquilibres, où il n'y a pas de serveur plus chargé que la plus grande disponibilité des serveurs libres. Cela nous permet de confirmer le gain d'équilibrage de charge par rapport à une méthode statique. En revanche, les flux provoquant un fort déséquilibre rendent les redistributions inefficaces et la méthode plus coûteuse que sa version statique. Cela



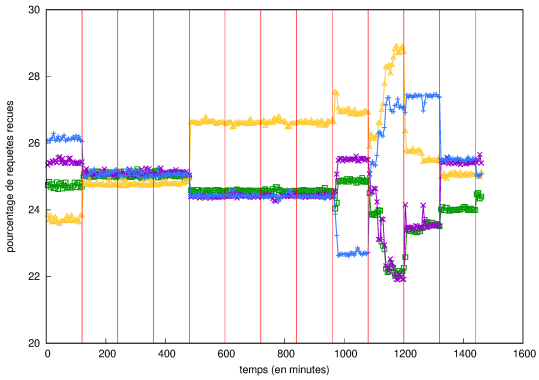
(a) Flux de type *MC* et *On/Off* avec un faible déséquilibre.



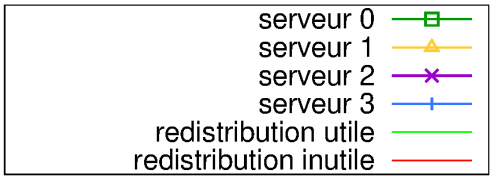
(b) Flux de type *Chaotique*.



(c) Flux de type *MC*, *On/Off* et *Pic* avec un fort déséquilibre.

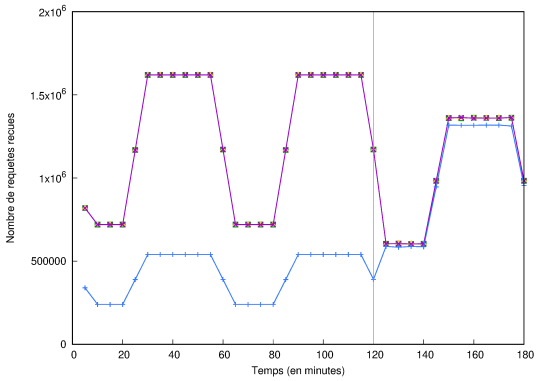


(d) Trace réelle.

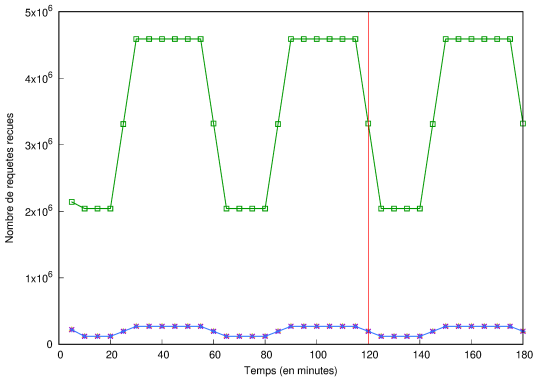


(e) Légende.

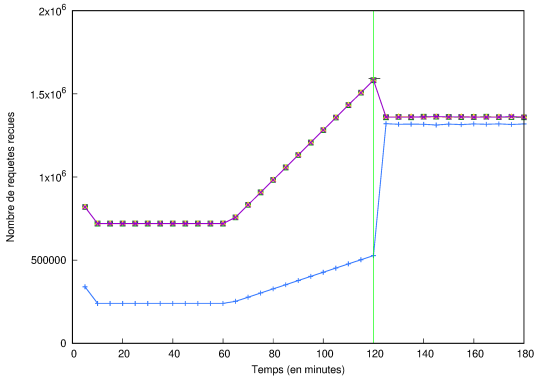
FIGURE 4.7: Répartition de la charge avec le *Dynamic Hashing*.



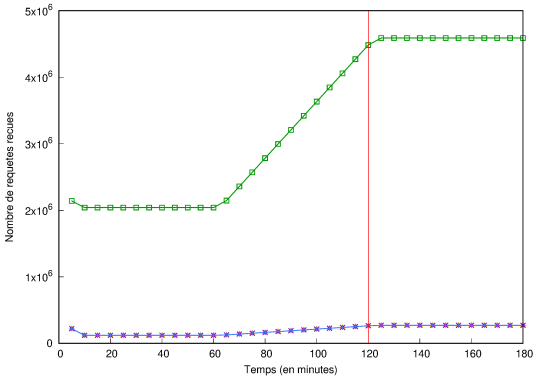
(a) Flux de type *On/Off* avec un faible déséquilibre.



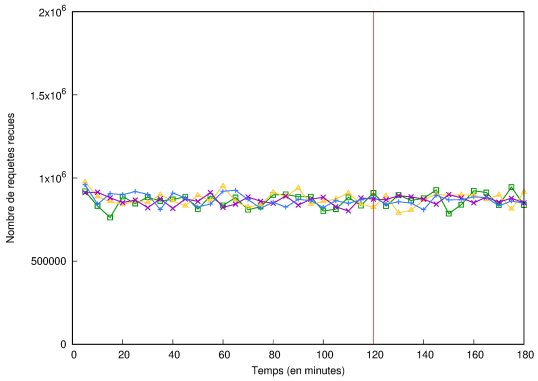
(b) Flux de type *On/Off* avec un fort déséquilibre.



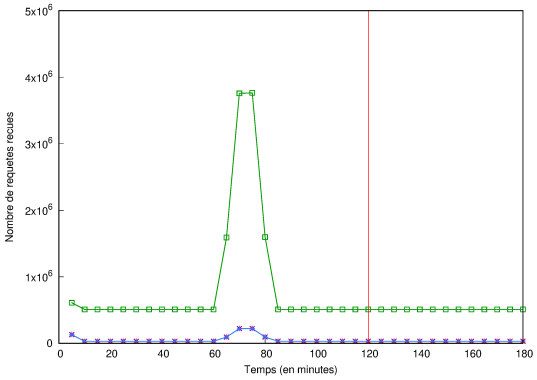
(c) Flux de type *MC* avec un faible déséquilibre.



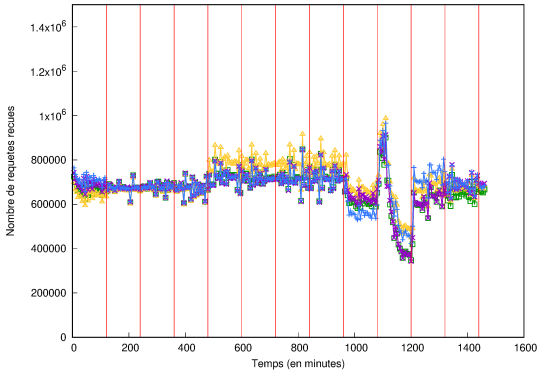
(d) Flux de type *MC* avec un fort déséquilibre.



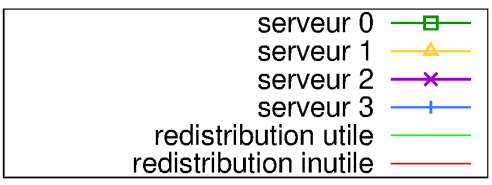
(e) Flux de type *Chaotique*.



(f) Flux de type *Pic* avec un fort déséquilibre.

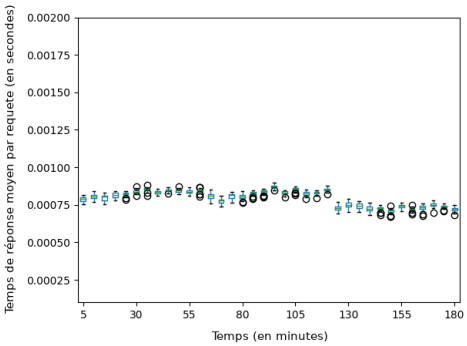


(g) Trace réelle.

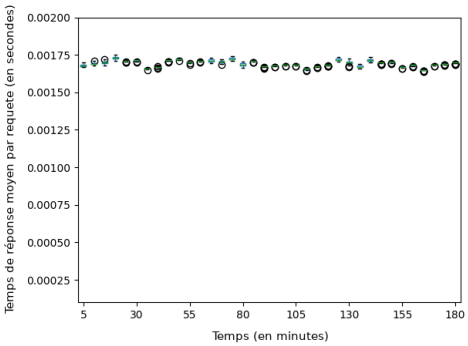


(h) Légende.

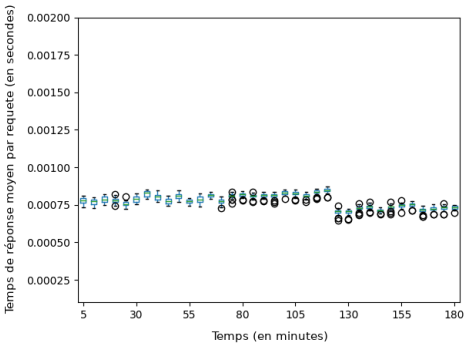
FIGURE 4.8: Charge moyenne avec le *Dynamic Hashing*.



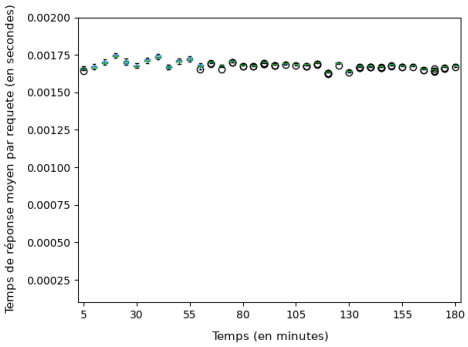
(a) Flux de type *On/Off* avec un faible déséquilibre.



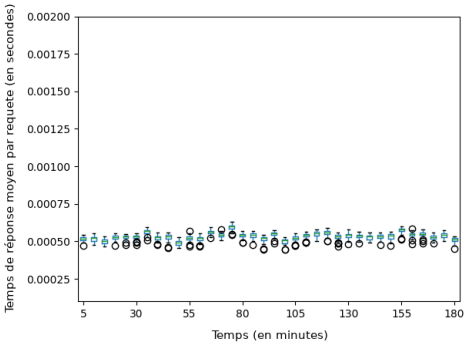
(b) Flux de type *On/Off* avec un fort déséquilibre.



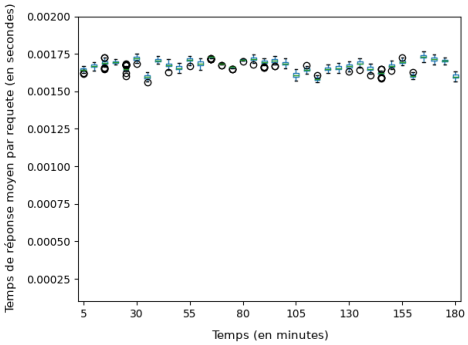
(c) Flux de type *MC* avec un faible déséquilibre.



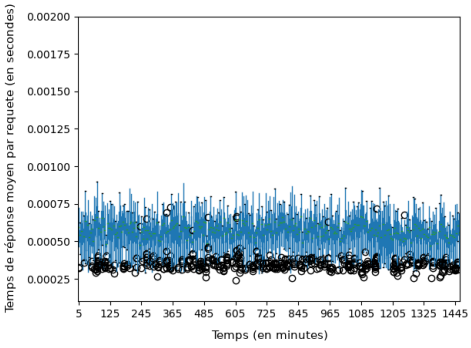
(d) Flux de type *MC* avec un fort déséquilibre.



(e) Flux de type *Chaotique*.



(f) Flux de type *Pic* avec un fort déséquilibre.



(g) Trace réelle.

FIGURE 4.9: Temps moyen par requête avec le *Dynamic Hashing*.

nous a ensuite permis de déceler un problème de conception dans l'algorithme de redistribution *RELAB*.

Nous exposons dans le tableau 4.4 le score qu'obtient le *Dynamic Hashing* pour l'ensemble des flux testés.

Flux	*On/Off* faible	*On/Off* fort	*MC* faible	*MC* fort	*Chaotique*	*Pic* fort	Réel
Score obtenu	0.0476	0.2131	0.0476	0.2131	0.0190	0.2116	0.0258

TABLE 4.4: Scores du *Dynamic Hashing*.

Afin d'alléger les évaluations pour la méthode *LAD*, nous avons choisi de garder un flux généré d'utilisation normale en addition de la trace réelle. Cela permet de confirmer les bénéfices déjà observés pour cette méthode, sans rallonger le processus d'évaluation. Les flux provoquant de forts déséquilibres n'ayant pas montré d'améliorations, nous les gardons tous pour l'évaluation de la méthode suivante. Les flux gardés pour évaluer la prochaine méthode sont donc les suivants :

- le flux réel,
- le flux de type *MC* avec faible déséquilibre,
- le flux de type *MC* avec fort déséquilibre,
- le flux de type *On/Off* avec fort déséquilibre,
- le flux de type *Pic* avec fort déséquilibre.

4.4 Évaluation de la Distribution adaptative à la charge *LAD*

Avant de pouvoir évaluer la méthode sur les différents flux, il est important d'étudier l'impact des différents paramètres afin de comparer la méthode dans son meilleur cas.

4.4.1 Étude des paramètres

La méthode *LAD* nécessite 3 paramètres :

- Le paramètre d'oubli α dont l'évaluation s'avère inutile car l'évaluation de la charge des serveurs n'a pas été modifiée par rapport à la méthode précédente.
- Le paramètre N_{entry}, qui représente la taille de la table d'index. L'algorithme de redistribution étant différent de celui utilisé dans la méthode du *Dynamic Hashing*, il faut l'évaluer de nouveau. Les valeurs testées pour ce paramètre restent les mêmes (voir tableau 4.2).

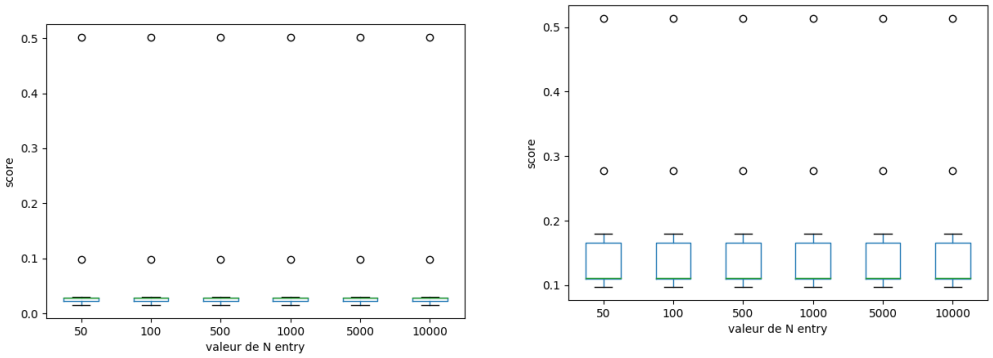
- La marge utilisée pour le seuil d'évaluation de charge déclenchant des redistributions. Les valeurs testées pour ce paramètre sont présentées dans le tableau 4.5.

Marge (%)	5	10	15	20	25	30	35	40	45	50

TABLE 4.5: Marges du seuil d'évaluation de charge testées pour la méthode *LAD*.

Afin d'étudier l'impact de chaque paramètre indépendamment, nous avons testé toutes les combinaisons de valeurs sur tous les flux. Pour chaque simulation, nous calculons le score selon la formule 4.1, et nous comparons l'impact de chaque paramètre en fonction de ce score que nous souhaitons minimiser. Pour plus de clarté, nous présentons ici l'étude de paramètres seulement sur deux flux, les autres flux présentant des résultats similaires.

La figure 4.10 exprime l'impact du paramètre N_{entry} sur la trace réelle et sur un flux de type *MC* avec un fort déséquilibre.



(a) Trace réelle.

(b) Flux de type *MC* avec un fort déséquilibre.

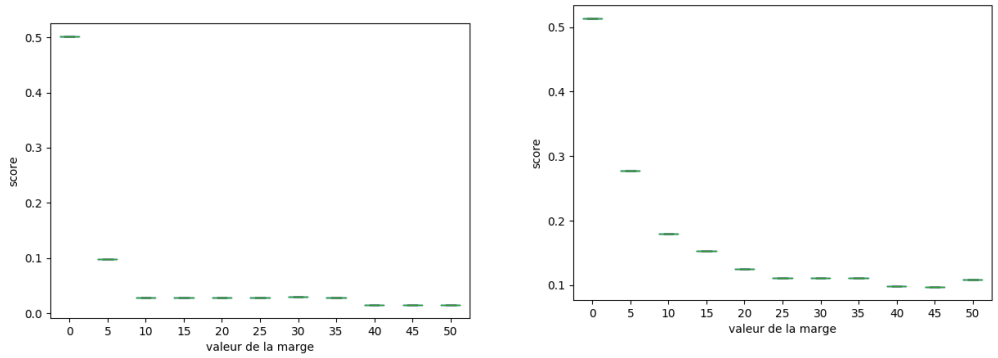
FIGURE 4.10: Score en fonction du paramètre N_{entry}.

Nous pouvons observer que ce paramètre impacte peu le score d'évaluation, même en l'étudiant sur un flux avec un fort déséquilibre. Il serait attendu que l'on puisse voir une forte différence de performances puisque la taille de la table d'index joue un rôle important dans la bonne répartition de l'algorithme de redistribution. Cependant, la normalisation utilisée afin de pouvoir évaluer les différentes méthodes entre elles impose à notre score des limites inférieures et supérieures les

plus grandes possible, cela a pour effet de diminuer la visibilité des impacts des paramètres.

Notre choix pour le paramètre N_{entry} s'est fait en consultant l'ensemble des flux sur lesquels nous étudions cette méthode. Nous avons choisi la valeur de 1000 afin de permettre une redistribution précise, sans surcharger en mémoire des clients ayant peu de ressources.

La figure 4.11 exprime l'impact de la marge utilisée pour l'évaluation du seuil de redistribution sur la trace réelle et sur un flux de type *MC* avec un fort déséquilibre.



(a) Trace réelle.

(b) Flux de type *MC* avec un fort déséquilibre.

FIGURE 4.11: Score en fonction de la marge.

Pour ce paramètre, nous pouvons observer une décroissance du score lorsque la marge augmente. Plus la marge est grande, plus la différence de charge entre l'époque actuelle et l'époque de la dernière redistribution doit être grande pour déclencher une redistribution, ce qui signifie que plus la marge est grande, moins il y a de chance de faire une redistribution inutile. Toutefois, nous constatons que le score commence à remonter si la marge dépasse les 45%. En effet, une trop grande marge induit forcément un plus grand écart de charge entre les serveurs et nous arrivons au point où le gain d'écart de charge est trop faible pour compenser le coût des redistributions. En consultant les résultats produits par l'ensemble des flux sur lesquels nous étudions cette méthode, nous avons choisi pour ce paramètre la valeur de 40%.

Cette étude de paramètres nous a permis d'établir que la combinaison de valeurs $N_{entry} = 1000$ avec une marge pour l'évaluation du seuil de redistribution de 40% nous apporte le meilleur score possible et nous allons donc étudier le comportement de la méthode *LAD* en utilisant ces valeurs.

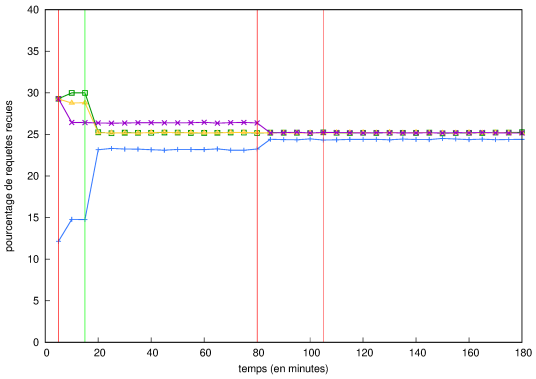
4.4.2 Étude des flux d'utilisation normale

La méthode précédente ayant donné de bons résultats pour l'étude des flux d'utilisation normale, nous évaluerons cette méthode sur seulement deux de ces flux : un flux généré de type *MC* avec un faible déséquilibre et la trace réelle, caractéristique de notre utilisation du supercalculateur.

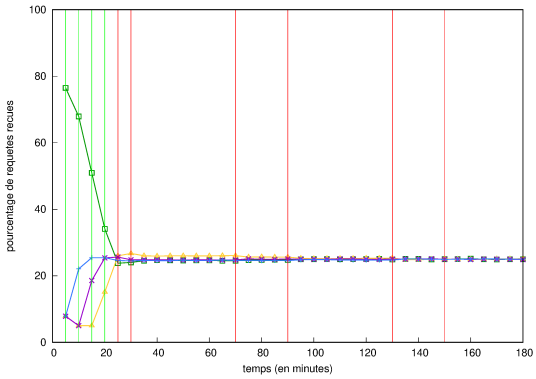
Les sous-figures 4.12a et 4.12e décrivent la répartition de la charge des serveurs pour le flux d'utilisation normale généré et la trace réelle. L'étude des sous-figures 4.13a et 4.13e, indiquant le nombre de requêtes reçues pour chaque serveurs et les sous-figures 4.14a et 4.14e, affichant les temps de réponse d'une requête, permettent de compléter l'analyse du comportement de la méthode sur ces flux. Nous pouvons observer des redistributions dès le début de l'exécution, ce qui signifie que la méthode détecte directement les déséquilibres. Une fois, ces déséquilibres comblés la méthode ne redemande plus de redistribution tant que le flux de requêtes ne change pas. Lorsque le nombre de requête augmente significativement, une redistribution est demandée. Elle n'est pas forcément effective, puisque le serveur se base sur le nombre de requêtes qu'il aurait dû recevoir lors de la dernière redistribution et n'a pas connaissance de la charge des autres serveurs : il voit sa charge augmenter et demande une redistribution. La redistribution lui permet de mettre à jour son seuil de redistribution et de s'adapter au nouveau flux de requêtes. L'étude des temps de réponse clients montre une amélioration après chaque redistribution ayant eu un effet sur l'équilibrage de charge.

4.4.3 Étude des flux intensifs générés

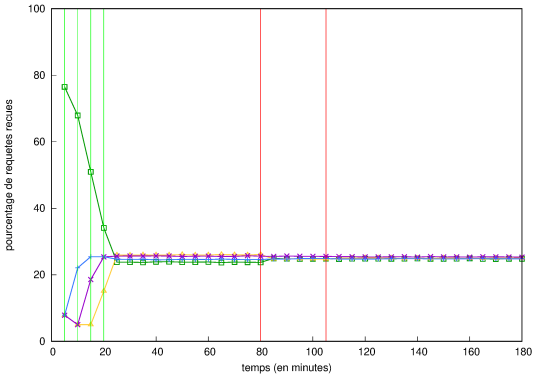
Les figures 4.12 et 4.13 affichent les courbes décrivant la répartition de la charge des serveurs ainsi que le nombre de requêtes reçues pour les flux intensifs. Pour tous ces flux nous pouvons constater que les 4 premières évaluations de la charge résultent en redistrubutions utiles. La charge semble ensuite se stabiliser à l'équilibre. À chaque changement de flux de requêtes (forte augmentation ou diminution du nombre de requêtes), des redistributions sont demandées mais inutiles, comme on peut le voir pour le flux de type *Pic*. En effet, lors des premières redistributions, un certains nombre d'entrées de la table d'index ont été transférées, ce qui équilibre aussi la charge future. L'augmentation de charge lors de la rafale de requêtes provoque des redistributions, cependant la répartition a déjà été équilibrée, rendant les redistributions inutiles. La figure 4.14 indique les temps de réponse d'une requête et permet de renforcer ces constats. Le temps de réponse diminue significativement après les 4 premières redistributions et reste constant par la suite.



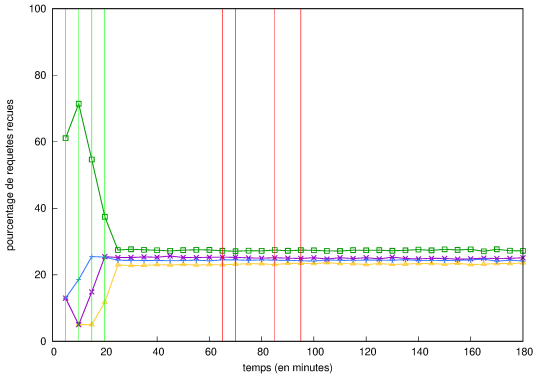
(a) Flux de type *MC* avec un faible déséquilibre.



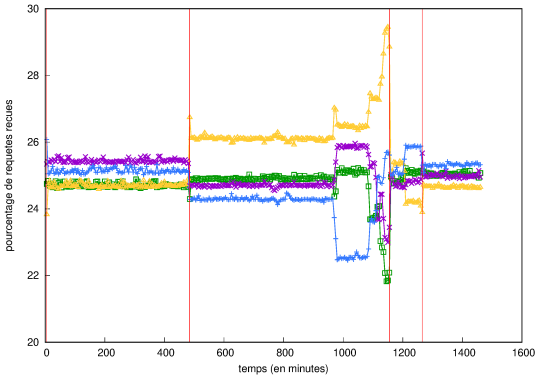
(b) Flux de type *On/Off* avec un fort déséquilibre.



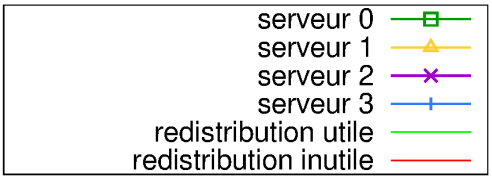
(c) Flux de type *MC* avec un fort déséquilibre.



(d) Flux de type *Pic* avec un fort déséquilibre.

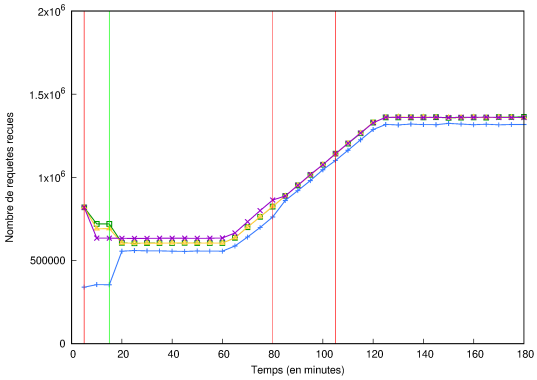


(e) Trace réelle.

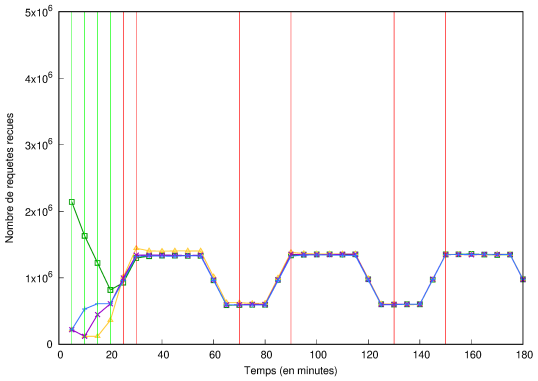


(f) Légende.

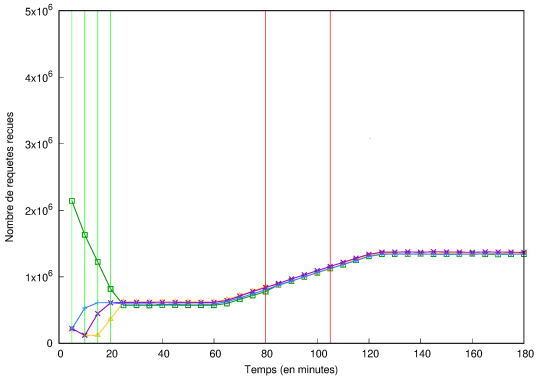
FIGURE 4.12: Répartition de la charge avec la méthode *LAD*.



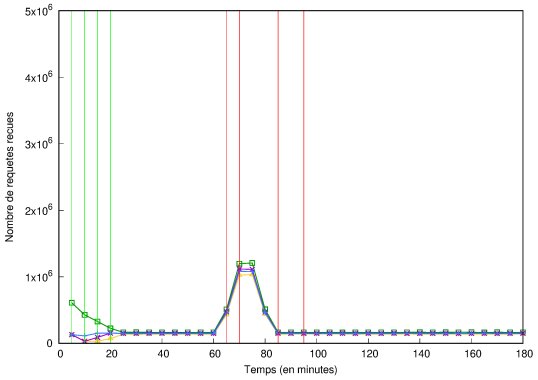
(a) Flux de type *MC* avec un faible déséquilibre.



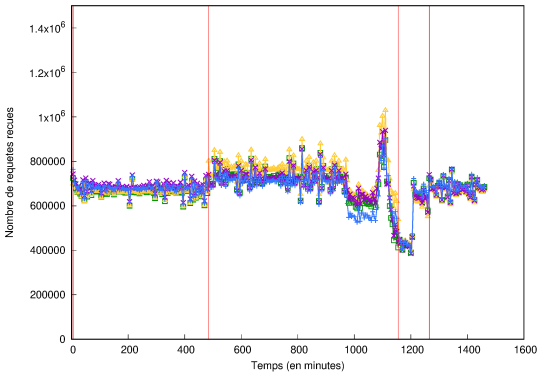
(b) Flux de type *On/Off* avec un fort déséquilibre.



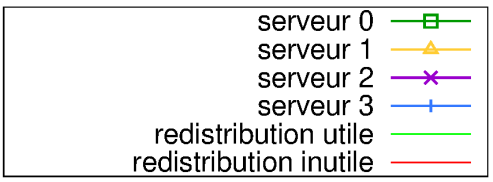
(c) Flux de type *MC* avec un fort déséquilibre.



(d) Flux de type *Pic* avec un fort déséquilibre.

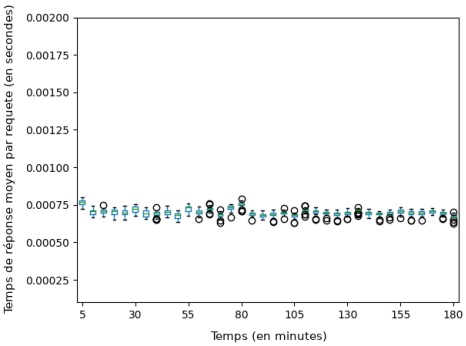


(e) Trace réelle.

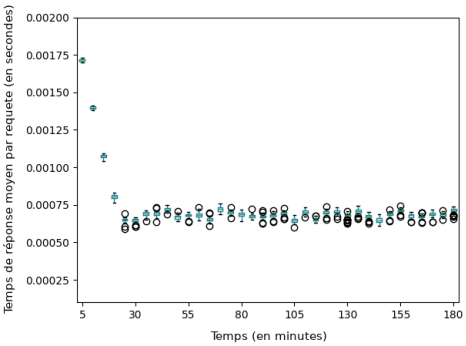


(f) Légende.

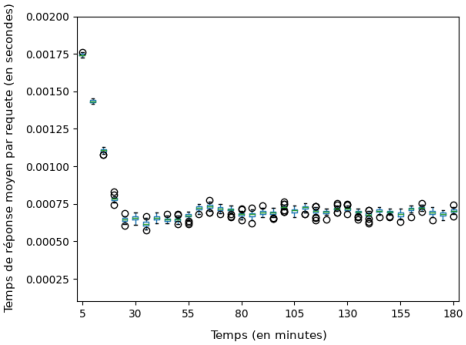
FIGURE 4.13: Charge moyenne avec la méthode *LAD*.



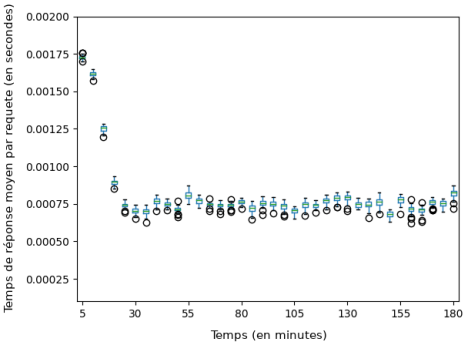
(a) Flux de type *MC* avec un faible déséquilibre.



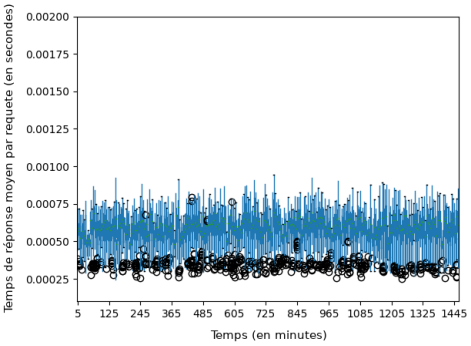
(b) Flux de type *On/Off* avec un fort déséquilibre.



(c) Flux de type *MC* avec un fort déséquilibre.



(d) Flux de type *Pic* avec un fort déséquilibre.



(e) Trace réelle.

FIGURE 4.14: Temps moyen par requête avec la méthode *LAD*.

4.4.4 En résumé

L'évaluation de la méthode *LAD* a montré sa capacité à redistribuer dès qu'un déséquilibre se présente, afin de prévenir des fortes surcharges. Le coût d'une redistribution est un peu plus élevé qu'avec le *Dynamic Hashing*, puisqu'un système de demande de redistribution avec récolte d'informations est nécessaire. Cependant cette méthode permet d'obtenir une répartition de la charge aussi équilibrée qu'avec une période de redistribution fréquente sans pour autant obtenir le coût de redistribution associé. De plus, cette méthode permet de gérer les cas de fort déséquilibre qui mettait le *Dynamic Hashing* en défaut.

Le tableau 4.6 répertorie les scores obtenus par la méthode *LAD* pour l'ensemble des flux testés.

Flux	*MC* faible	*On/Off* fort	*MC* fort	*Pic* fort	Réel
Score obtenu	0.0619	0.1525	0.098	0.1301	0.0102

TABLE 4.6: Scores de la méthode *LAD*.

Nous remarquons que les scores liés aux flux d'utilisation normale sont moins bons qu'avec les méthodes précédentes, cela s'explique par la complexification de la méthode afin de gérer plus de cas. En effet, une redistribution est plus coûteuse, et le déclenchement de redistributions vient de l'évaluation de la charge d'un serveur qui n'a aucune vision globale. Demander une redistribution se fait donc de manière locale et il arrive qu'un serveur initie une redistribution inutile. La méthode est donc un peu moins performante sur des flux déjà bien gérés par les méthodes précédentes comme le flux *MC* avec un faible déséquilibre, mais la méthode *LAD* permet une meilleure gestion des cas avec de plus forts déséquilibres.

Pour l'étude de la prochaine méthode, nous avons décidé de garder les mêmes flux d'utilisation normale pour vérifier qu'il n'y a pas de régression pour la bonne gestion de ces flux. Nous avons aussi choisi de garder l'ensemble des flux provoquant un fort déséquilibre, afin de comparer les bénéfices apportés. Les flux gardés pour évaluer la prochaine méthode sont donc les suivants :

- le flux réel,
- un flux de type *MC* avec faible déséquilibre,
- le flux de type *MC* avec fort déséquilibre,
- le flux de type *On/Off* avec fort déséquilibre,
- le flux de type *Pic* avec fort déséquilibre.

4.5 Évaluation de la variante avec fenêtre temporelle *LAD-TW*

Avant d'évaluer la méthode *LAD-TW* sur les différents flux, l'étude des nouveaux paramètres induits par cette variante est nécessaire.

4.5.1 Étude des paramètres

La méthode *LAD-TW* nécessite 4 paramètres :

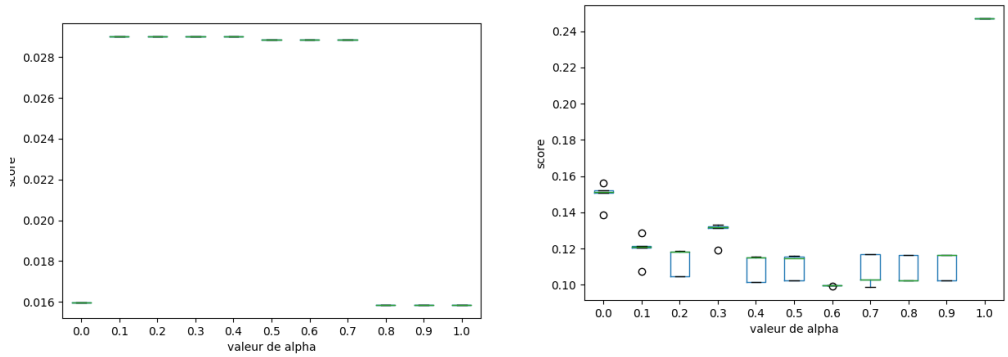
- Le paramètre d'oubli α qui doit être réévaluer puisque cette variante modifie l'évaluation de la charge et dont les valeurs testées restent les mêmes.
- Le paramètre N_{entry}, qui représente la taille de la table d'index et dont l'évaluation est inutile car l'algorithme de redistribution n'a pas été modifiée.
- La marge utilisée pour le seuil d'évaluation de charge déclenchant des redistributions dont l'évaluation s'avère inutile également.
- La taille de la fenêtre temporelle d'évaluation dont les valeurs testées sont présentées dans le tableau 4.7.

Taille de la fenêtre (min)	10	30	60	120

TABLE 4.7: Tailles de la fenêtre d'évaluation testées pour la méthode *LAD-TW*.

Les courbes obtenues pour l'étude de ces paramètres nous permettent un constat similaire, et par souci de lisibilité, nous ne proposerons que deux de ces courbes pour chaque paramètre testé. La figure 4.15 exprime l'impact du paramètre α sur la trace réelle et sur un flux de type *MC* avec un fort déséquilibre.

Comme pour l'évaluation de ce paramètre avec la méthode du *Dynamic Hashing*, nous pouvons observer une faible divergeance de scores pour des flux avec des faibles déséquilibres comme la trace réelle dont les valeurs du score restent très prochent de zéro (0.01 à 0.02). Les flux avec de forts déséquilibres apportent une plus grande variation des scores : la valeur 0 engendre un score majoritairement plus haut que les autres. En effet, la prise en compte unique du passé ne permet pas une évaluation de la charge pertinente, cependant, le score est meilleur qu'avec le *Dynamic Hashing* car le passé pris en compte ne reste pas identique et suit le flux de requêtes (avec du retard). La valeur 1 fournit aussi un score haut, traduisant une mauvaise évaluation de la charge. Cela signifie que la prise en compte du présent seule ne permet pas une évaluation pertinente.



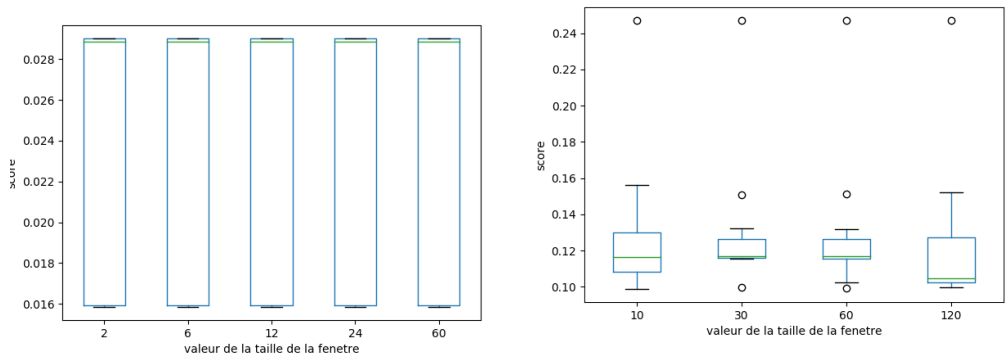
(a) Trace réelle.

(b) Flux de type *MC* avec un fort déséquilibre.

FIGURE 4.15: Score en fonction du paramètre α.

Notre choix pour le paramètre α s'est fait en consultant l'ensemble des flux sur lesquels nous étudions cette méthode. Nous avons choisi la valeur de 0.6 car elle permet majoritairement un meilleur score.

La figure 4.16 exprime l'impact de la taille de la fenêtre d'évaluation sur la trace réelle et sur un flux de type *MC* avec un fort déséquilibre.



(a) Trace réelle.

(b) Flux de type *MC* avec un fort déséquilibre.

FIGURE 4.16: Score en fonction de la taille de la fenêtre d'évaluation.

L'étude de ce paramètre indique qu'une fenêtre d'évaluation trop courte (10 minutes) génère une plus grande variation de valeurs, signifiant que l'évaluation sera plus sensible aux changements de flux. Il en va de même pour une grande fenêtre (2 heures). Les scores associés aux valeurs entre ces extrémités permettent

une évaluation plus robuste avec une variation du score plus faible.

Notre choix pour la taille de la fenêtre d'évaluation s'est fait en consultant l'ensemble des flux sur lesquelles nous étudions cette méthode. Nous avons choisi la valeur de 60 minutes car elle permet d'obtenir globalement un meilleur score.

4.5.2 Étude des flux d'utilisation normale

La méthode précédente ayant donné de bons résultats pour l'étude des flux d'utilisation normale, nous évaluerons cette méthode sur seulement deux de ces flux : un flux généré de type *MC* avec un faible déséquilibre et la trace réelle, caractéristique de notre utilisation du supercalculateur.

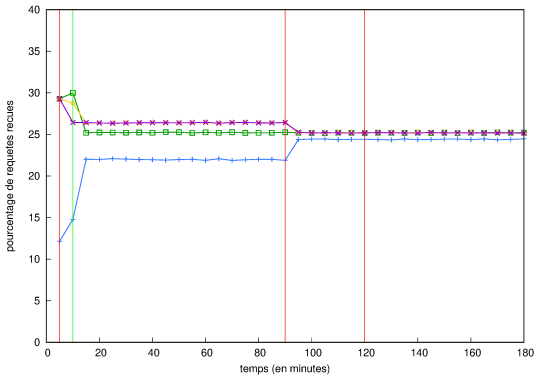
Les sous-figures 4.17a et 4.17e représentent la répartition de la charge des serveurs pour le flux synthétique d'utilisation normale et la trace réelle. Cette analyse s'enrichit avec l'étude des sous-figures 4.18a et 4.18e, affichant le nombre de requête reçues pour chaque serveur et les sous-figures 4.19a et 4.19e, exposant les temps de réponse d'une requêtes. Comme pour la méthode précédente, nous pouvons observer des redistributions dès le début de l'exécution, ce qui signifie que la méthode détecte directement les déséquilibres et ne redemande plus de rééquilibrage tant que le flux de requêtes n'a pas changé. L'analyse des courbes de temps de réponse d'une requête atteste de l'efficacité des redistributions. Pour le cas de la trace réelle, chaque redistribution est considérée comme inutile puisque l'écart de charge est initialement inférieur à 5%, cependant, nous pouvons constater que chaque redistribution a bien un effet sur la répartition de la charge.

Nous n'observons pas de différence significative de comportement entre la méthode *LAD* et la variante *LAD-TW* pour des flux d'utilsation normale. En effet, la variante avec fenêtre temporelle permet une meilleure évaluation des charges lors de changements de flux de requêtes, ce qui n'est pas le cas de ce type de flux. Toutefois, cela nous permet de confirmer que l'ajout d'une fenêtre d'évaluation ne diminue pas les bénéfices apportés par la méthode.

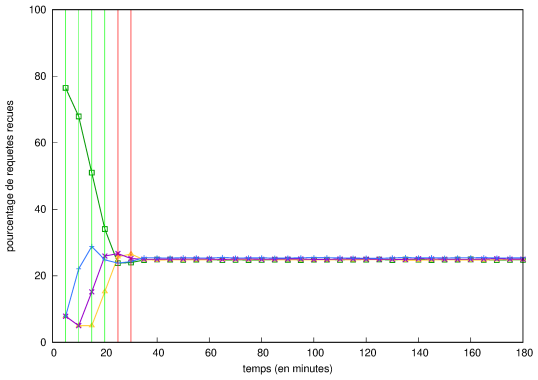
4.5.3 Étude des flux intensifs générés

L'étude des flux de type *MC* et *Pic* générant un fort déséquilibre montre que la méthode *LAD-TW* se comporte de la même manière que la méthode *LAD* sans la fenêtre. Encore une fois, cela nous confirme le bon comportement de la variante avec fenêtre temporelle sur des flux déjà bien gérés par la méthode précédente.

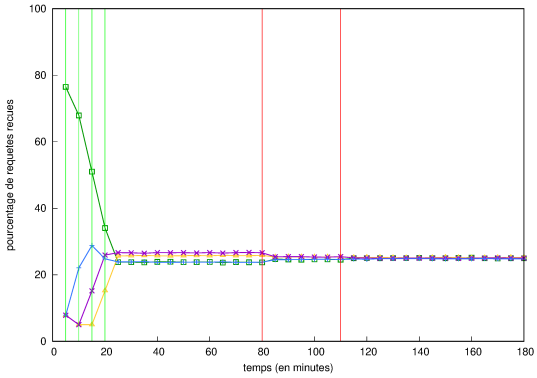
En ce qui concerne le flux de type *On/Off*, nous pouvons constater que le nombre de redistributions inutiles a nettement diminué par rapport à la méthode précédente. La fenêtre d'évaluation permet de mieux gérer les changements de flux, d'autant plus s'ils sont réguliers, comme c'est le cas ici. La méthode anticipe les



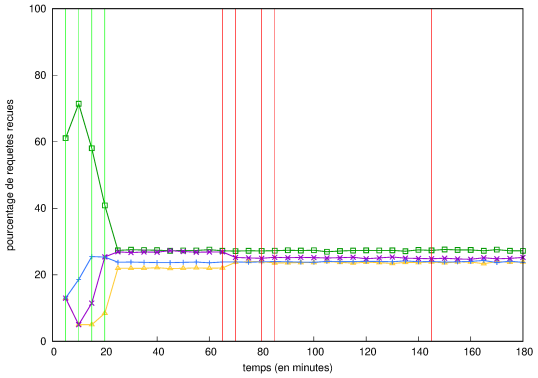
(a) Flux de type *MC* avec un faible déséquilibre.



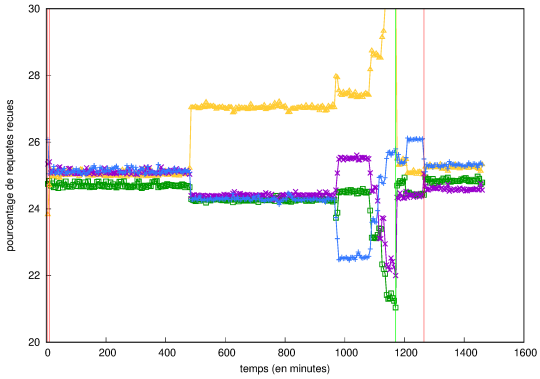
(b) Flux de type *On/Off* avec un fort déséquilibre.



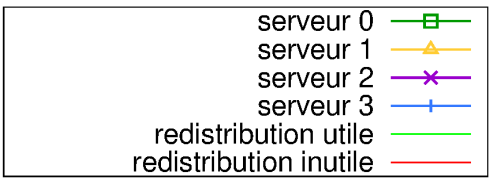
(c) Flux de type *MC* avec un fort déséquilibre.



(d) Flux de type *Pic* avec un fort déséquilibre.

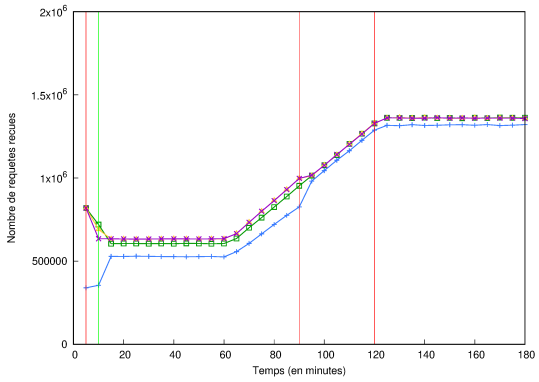


(e) Trace réelle.

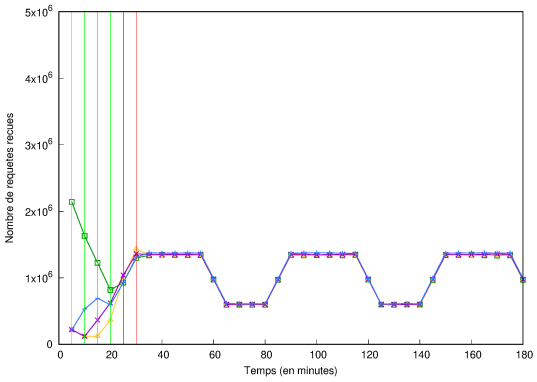


(f) Légende.

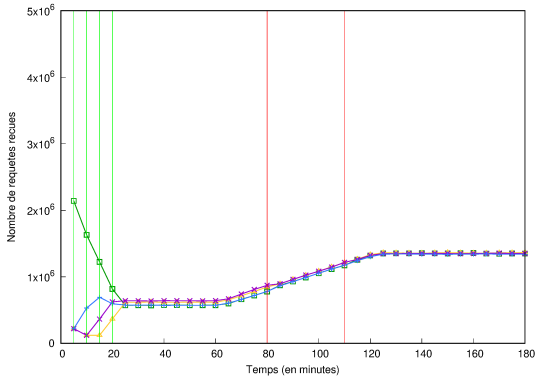
FIGURE 4.17: Répartition de la charge avec la méthode *LAD-TW*.



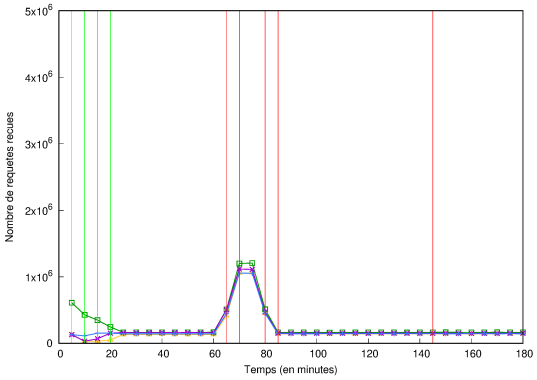
(a) Flux de type *MC* avec un faible déséquilibre.



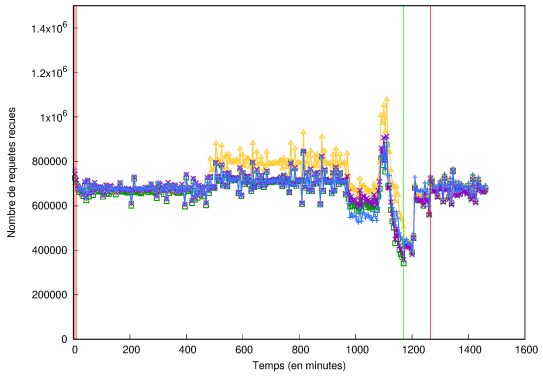
(b) Flux de type *On/Off* avec un fort déséquilibre.



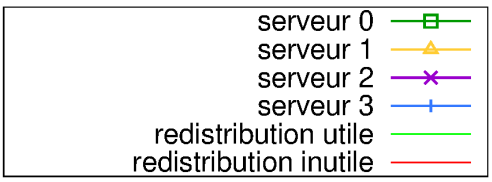
(c) Flux de type *MC* avec un fort déséquilibre.



(d) Flux de type *Pic* avec un fort déséquilibre.

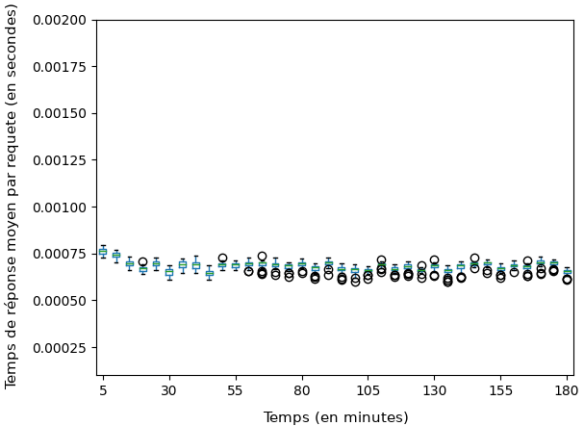


(e) Trace réelle.

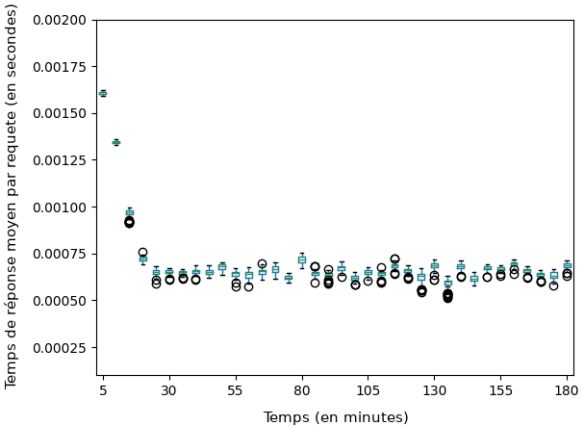


(f) Légende.

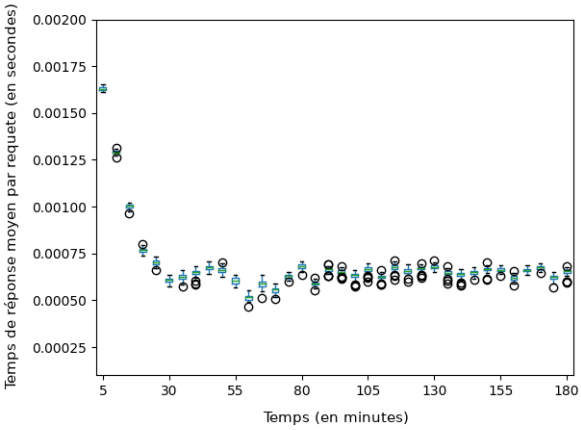
FIGURE 4.18: Charge moyenne avec la méthode *LAD-TW*.



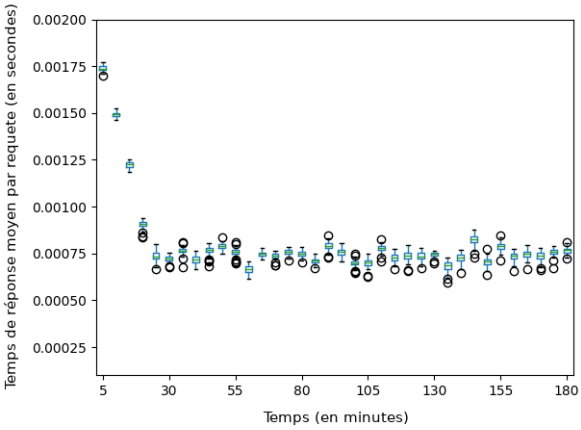
(a) Flux de type *MC* avec un faible déséquilibre.



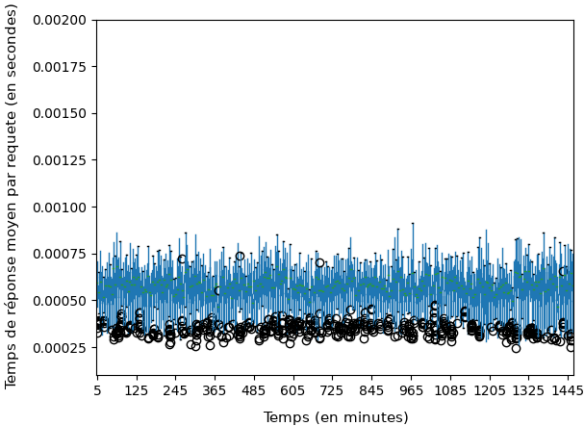
(b) Flux de type *On/Off* avec un fort déséquilibre.



(c) Flux de type *MC* avec un fort déséquilibre.



(d) Flux de type *Pic* avec un fort déséquilibre.



(e) Trace réelle.

FIGURE 4.19: Temps moyen par requête avec la méthode *LAD-TW*.

changements de flux de requêtes en fonction de ceux déjà reçus pour un meilleur équilibrage.

4.5.4 En résumé

Le tableau 4.8 exhibe les scores obtenus par la méthode *LAD-TW* pour l'ensemble des flux testés. Ici encore, les scores liés aux flux d'utilisation normale sont moins bons qu'avec les méthodes précédentes, et cela s'explique de la même manière que précédemment. Toutefois, elle offre des scores similaires pour les flux intensifs et nous pouvons même observer une amélioration pour certains flux de requêtes changeants.

Flux	*MC* faible	*On/Off* fort	*MC* fort	*Pic* fort	Réel
Score obtenu	0.0633	0.0973	0.0995	0.1320	0.0116

TABLE 4.8: Scores de la méthode *LAD-TW*.

4.5.5 Étude de mise à léchelle

Cette méthode a été conçue pour intégrer un service de métadonnée de grande taille, nécessitant une répartition de la charge sur les nombreux serveurs le composant. Il est donc primordiale de vérifier la capacité de mise à l'échelle de cette méthode. De manière générale, la scalabilité est la capacité à s'adapter à un changement d'ordre de grandeur. Dans notre cas, La scalabilité d'une méthode correspond à sa capacité à fournir une distribution équilibrée pour un grand nombre de serveurs.

Pour évaluer cette mise à l'échelle, nous avons exécuter la trace réelle (tronquée à 3h) sur un service de métadonnées contenant de plus en plus de serveurs (jusqu'à 64 serveurs). Pour chaque exécution, nous avons calculé l'écart de charge moyen pour un serveur et l'avons ensuite retranché à la valeur de répartition idéale. La répartition des charges résultant de ces exécutions sont représentées par la figure 4.20. Nous pouvons constater qu'en augmentant le nombre de serveurs, la charge reste bien répartie, ce qui atteste de la bonne scalabilité de la méthode *LAD-TW*.

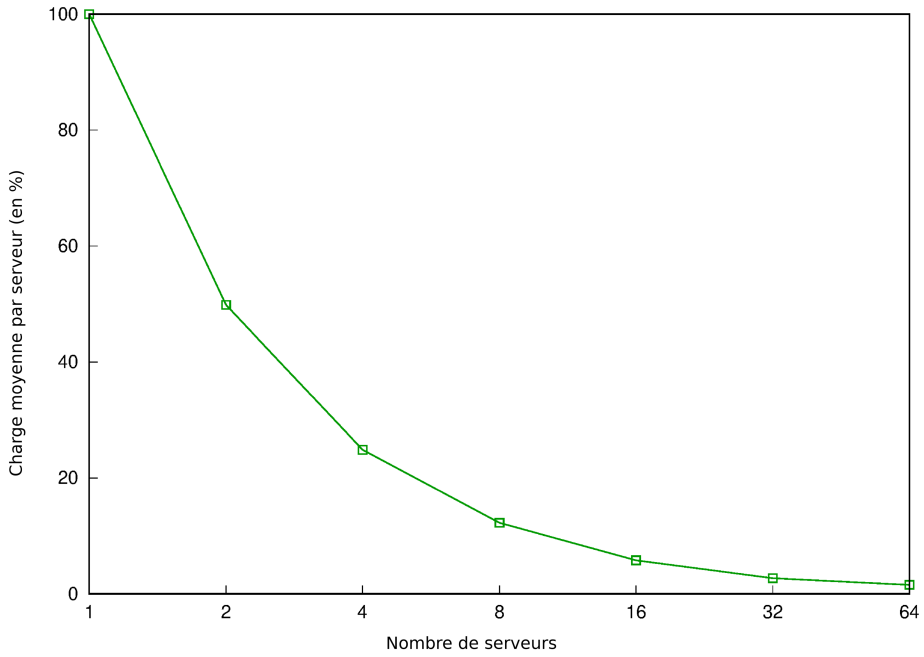


FIGURE 4.20: Charge par serveur selon la taille du service de métadonnées.

4.6 Conclusion

Ce chapitre nous a permis d'évaluer et de comparer équitablement chaque méthode de distribution étudiée dans cette étude. Le tableau 4.9 permet un récapitulatif des scores obtenus pour chacune des méthodes selon les différents flux.

Flux	*On/Off* faible	*On/Off* fort	*MC* faible	*MC* fort	*Chaotique*	*Pic* fort	réel
Static Hashing	0.0498	0.1992	0.0498	0.1992	0.0504	0.1977	**0.0076**
Dynamic Hashing	**0.0476**	0.2131	**0.0476**	0.2131	**0.0190**	0.2116	0.0258
Distribution adaptative à la charge (LAD)	-	0.1525	0.0619	**0.0980**	-	**0.1301**	0.0102
Load-Adaptive Distribution with Temporal Window (LAD-TW)	-	**0.0973**	0.0633	0.0995	-	0.1320	0.0116

TABLE 4.9: Récapitulatif des scores obtenus pour chaque flux de requêtes.

Ainsi, nous pouvons affirmer que la méthode du *Static Hashing* est très efficace pour des flux d'utilisation normale, où il n'est généré qu'un faible déséquilibre, bien qu'une redistribution permettrait un meilleur score global comme nous l'a montré l'étude du *Dynamic Hashing*. Pour ces flux d'utilisation normale, la méthode *LAD* et *LAD-TW* sont moins efficaces, mais la différence de score reste faible. En effet, l'évaluation de la charge est laissée aux serveurs, qui n'ont qu'une vision locale, ce qui provoque des redistributions inutiles.

Si par contre, nous analysons les résultats donnés pour les flux intensifs, nous pouvons déclarer que les méthodes du *Static Hashing* et du *Dynamic Hashing* ne sont pas capables de gérer de tels flux, laissant le fort déséquilibre compromettre le système. La méthode *LAD* et *LAD-TW*, permettent elles, une redistribution de ces fortes surcharges et ce dès la détection de celles-ci. La méthode *LAD-TW* offre même une meilleure gestion des flux changeants comme ceux du type *On/Off*.

Cette évaluation a permis de retrouver, pour les méthodes de références, les limites présentées précédemment et de mettre en évidence les apports de nos méthodes *LAD* et *LAD-TW* pour des flux de requêtes plus soutenus.

Conclusion

La gestion des métadonnées dans un système de stockage objet pour un supercalculateur HPC est une problématique critique. Une des difficultés réside dans la distribution des métadonnées sur les différents serveurs qui constituent le service de métadonnées. Une mauvaise répartition peut engendrer, en cas de trop nombreux accès aux métadonnées, des surcharges sur certains serveurs, ce qui ralentirait significativement le fonctionnement de ceux-ci. C'est pourquoi il est nécessaire d'utiliser une méthode de distribution des métadonnées qui permette un équilibrage des charges. Étant donné que le choix d'une méthode de distribution dépend en grande partie du flux de requêtes auquel le système est soumis, il est important de rappeler que cette thèse se concentre sur des flux typiques du domaine du HPC en prévision du passage à l'exascale. Les travaux réalisés dans cette étude ont visé à obtenir une méthode de distribution équilibrée qui soit en mesure de supporter ces flux de requêtes exascale.

Contributions apportées

Nous distinguons dans cette thèse deux contributions majeures :

- La proposition, l'implémentation et l'évaluation de la méthode *Distribution adaptative à la charge* (*LAD*) : une méthode de distribution effectuant un rééquilibrage quand cela est nécessaire, ainsi que sa variante avec fenêtre temporelle *LAD-TW* ;
- La conception et le développement de *MeDiE* : un outil permettant d'évaluer différentes méthodes de distribution de manière équitable sur différents flux de requêtes, réels ou synthétiquement générés par l'outil lui-même.

La méthode *LAD* est une méthode de distribution de métadonnées dynamique utilisant une fonction de hachage. Elle permet une redistribution des métadonnées quand un déséquilibre de charge devient trop conséquent, tout en limitant le nombre de redistributions inutiles. L'algorithme de distribution utilisé dans cette

méthode tente d'effectuer le moins de transferts possibles, mais si cela est nécessaire, il peut répartir une forte charge sur plusieurs serveurs libres. Cette méthode possède aussi une variante avec fenêtre temporelle *LAD-TW*, qui permet une meilleure évaluation de la charge de chaque serveur en fonction des charges précédemment enregistrées pendant la fenêtre d'observation.

Ces deux méthodes ont été présentées au chapitre 2 puis validées expérimentalement au chapitre 4. Nous avons alors pu voir sur des traces réelles et synthétiques l'apport de ces méthodes en comparaison de méthodes classiques que sont le *Static Hashing* et le *Dynamic Hashing*. Si les méthodes *LAD* et *LAD-TW* n'offrent pas les meilleurs résultats pour des flux d'utilisation normale, ceux-ci restent tout à fait acceptables. Par contre, pour les flux intensifs avec de forts déséquilibres, ces méthodes se sont révélées bien plus efficaces que les méthodes classiques dont les redistributions sont inadaptées et permettent d'obtenir un rééquilibrage durable dès le début de l'exécution. Nous avons aussi vérifié la scalabilité de la méthode *LAD-TW* afin d'attester de sa capacité à gérer un grand nombre de serveurs.

MeDiE est un outil permettant d'évaluer équitablement différentes méthodes de distribution de métadonnées selon différents flux de requêtes. L'évaluation se fait en fonction de l'impact qu'une méthode a sur l'équilibrage de charge des serveurs de métadonnées. Ce calcul se fait à partir de prises de mesures du nombre de requêtes reçues par les serveurs pendant une période et du temps de réponse des requêtes pour les clients. Les fichiers d'entrée pour notre outil sont des traces globales représentant un flux de requêtes sur lequel nous souhaitons évaluer une méthode de distribution.

Notre outil est composé de 3 modules. Un simulateur de service de métadonnées permet l'exécution des interactions entre les clients et les serveurs selon les différentes méthodes de distribution. Un processus de simulation est en charge de la gestion du temps et de la prise des mesures. Un générateur de traces facilite la génération de fichiers de traces globales correspondant à un flux de requêtes selon différentes caractéristiques.

À l'heure actuelle, les méthodes de distribution déjà implémentées dans notre outil sont celles présentées dans ce manuscrit [Bil20]. Toutefois, *MeDiE* est un projet à code source ouvert et sa conception encourage l'ajout de nouvelles méthodes.

Perspectives et travaux futurs

Plusieurs perspectives font suite à ces travaux de thèse à court, moyen et long terme. Certaines d'entre elles consistent à faire évoluer notre outil *MeDiE* et d'autres concernent la recherche d'une meilleure méthode de distribution de métadonnées.

Perspectives à court terme

Des perspectives à court terme sont tout d'abord envisageables dans la recherche d'une meilleure méthode de distribution des métadonnées.

- *L'optimisation de l'algorithme de redistribution :* Nous pourrions simplifier cet algorithme afin de réduire sa complexité en O(N) en utilisant deux pointeurs pour parcourir le tableau des charges en parallèle depuis chaque extrémité. Une idée à développer serait de prendre en compte la popularité des métadonnées lors de la redistribution en déplaçant en priorité les métadonnées fortement utilisées récemment afin de réduire le déséquilibre de charge le plus rapidement possible.
- *La prise en compte d'une fenêtre temporelle dynamique :* Une évolution de la méthode *LAD-TW* serait de faire varier dynamiquement certains paramètres de la méthode comme le paramètre d'oubli α ou la taille de la fenêtre. Ainsi, la méthode s'adapterait aux différents flux auxquels elle est soumise. Par exemple, pour des flux correspondant à un système en pleine évolution, il n'est pas nécessaire d'accorder trop d'importance au passé, et à l'inverse, si ce flux redevient stable, il est important de prendre en compte les accès passés.

La dernière perspective à court terme concerne notre outil.

- *L'Amélioration de la gestion des médias de stockage :* Ajouter de nouvelles implémentations au panel de médias de stockage géré par notre outil permettrait l'usage de *MeDiE* sur un plus grand nombre de systèmes. De plus, l'implémentation actuelle ne permet pas de lier un serveur avec plusieurs médias de types différents. Une deuxième possibilité pourrait être d'adapter la gestion des médias de stockage afin qu'elle prenne en compte plusieurs types de médias pour un même serveur.

Perspectives à moyen terme

Certaines des perspectives à moyen terme concernent la recherche d'une meilleure méthode de distribution.

- *La diminution de l'occupation mémoire coté client :* Utiliser une structure plus légère du type *B-tree* [CLRS09] par exemple permettrait de réduire le coût dû à la table d'index. Il est aussi possible de limiter le nombre de redirections lors de la connexion d'un nouveau client en initialisant la table d'index directement avec la version la plus à jour possible. L'utilisation de mécanismes d'indexation pourrait aussi alléger ce coût et mériterait d'être approfondie.

- *Enrichissement de la méthode LAD-TW :* Nous pourrions prendre en compte de nouveaux mécanismes tel que les systèmes de réplication temporaire comme dans la méthode *WPAR* [LWV07] qui permettraient d'alléger momentanément la charge d'un serveur. De la même manière, incorporer de nouveaux comportements pour l'ajout ou la suppression de serveur fournirait une meilleure tolérance aux pannes.

Notre outil possède lui aussi des possibilité d'évolution à moyens termes.

- *Prise en compte d'autres mesures :* L'analyse serait plus complète gràce à l'ajout de nouvelles prises de mesures tel que le temps nécessaire à chaque redistribution, qui est actuellement en cours d'implémentation. Une mesure du temps d'accès à une requête par serveur permettrait d'observer les latences pour chaque serveur pour en déduire les serveurs surchargés.

Perspectives à long terme

La première perspective à long terme concerne la recherche d'une meilleure méthode de distribution.

- *La fusion des méthodes :* Il pourrait être intéressant de choisir les méthodes de distribution dynamiquement et d'appliquer la plus adaptée aux flux actuels.

La dernière perspective à long terme de ces travaux porte sur l'évolution de l'outil.

- *L'amélioration du réalisme de la simulation :* Il est possible d'intégrer dans *MeDiE*, des mécanismes rendant la simulation d'un service de métadonnées plus réaliste. La prise en compte de la réplication des métadonnées ou encore la gestion des changements de topologies du réseau (ajout ou suppression de serveurs) améliorerait considérablement la représentativité de notre outil, et permettrait également d'élargir son domaine applicatif.

Glossaire

MDS MetaData Server ou serveur de méta-données

Create Read Update Delete (CRUD) Sémantique pour des systèmes de stockage ou les seules opérations disponibles sont la création (*Create*), la lecture (*Read*), la mise à jour (*Update*) et la suppression (*Delete*).

Key Value Store (KVS) Système de stockage où chaque donnée (*value*) est accessible uniquement avec un identifiant (*key*).

Cohérence forte Critère indiquant que l'on voit l'effet de l'action précédente lors de l'action suivante. Il existe deux types de cohérence forte: la *Sequential Consistency* et la *Linearizability*.

Cohérence faible Est considérée cohérence faible, toute cohérence qui n'est pas forte. Parmi les coherences faibles, on peut trouver la *Weakest Consistency* (le système pourra être cohérent un jour) ou la *Eventually Consistency* (le système sera cohérent un jour).

Hot spots (points chauds). C'est un terme employé pour désigner un ensemble de données très fortement accédées causant généralement des ralentissements.

Application Programming Interface (API) Interface par laquelle un logiciel propose des services à d'autres logiciels

JavaScript Object Notation (JSON) Format de données textuelles permettant de représenter de l'information structurée et décrit dans la RFC 8259 et ECMA 404.

Burst (rafale). C'est un terme qui est employé pour parler d'une émission soudaine, très intense et très brève dans notre cas de requêtes.

Timestamp (horodatation). L'horodatage consiste à associer une date et une heure à un événement. Un timestamp est donc la date et l'heure à laquelle s'est exécuté l'évènement.

Table des figures

Liste des tableaux

Bibliographie

[AAB+15] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow : Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[ABPH07] Paulo Sérgio Almeida, Carlos Baquero, Nuno M Preguiça, and David Hutchison. Scalable bloom filters. *Information Processing Letters*, 2007.

[ACZ03] Cristiana Amza, Alan L Cox, and Willy Zwaenepoel. Distributed versioning : Consistent replication for scaling back-end databases of dynamic contentweb sites. In *ACM/IFIP/USENIX International Conference on Distributed Systems Platforms and Open Distributed Processing*, 2003.

[AMS+15] Mark James Abraham, Teemu Murtola, Roland Schulz, Szilárd Páll, Jeremy C Smith, Berk Hess, and Erik Lindahl. Gromacs : High performance molecular simulations through multi-level parallelism from laptops to supercomputers. *SoftwareX*, 2015.

[BB08] Robert Battle and Edward Benson. Bridging the semantic web and web 2.0 with representational state transfer (rest). *Journal of Web Semantics*, 2008.

[BDH+10] Michael Barton, John Dickinson, Greg Holt, Greg Lange, Jay Payne, Will Reese, and Chuck Thier. Openstack swift pro-

ject. `https://www.openstack.org/software/releases/ocata/components/swift`, 2010.

[Bil20] Eloise Billa. Medie : a metadata distribution evaluator for object storage systems. `https://github.com/Billae/MeDiE`, 2020.

[Blo70] Burton H Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 1970.

[BMLX03] Scott A Brandt, Ethan L Miller, Darrell DE Long, and Lan Xue. Efficient metadata management in large distributed storage systems. In *Mass Storage Systems and Technologies, 2003.(MSST 2003). Proceedings. 20th IEEE/11th NASA Goddard Conference on*, 2003.

[Bor08] Dhruba Borthakur. Hdfs architecture guide. *Hadoop Apache Project*, 2008.

[Bra17] Tim Bray. The JavaScript Object Notation (JSON) Data Interchange Format. RFC 8259, 2017.

[Bra19] Peter Braam. The lustre storage architecture. *arXiv preprint arXiv :1903.01955*, 2019.

[Bre12] Eric Brewer. Cap twelve years later : How the" rules" have changed. *Computer*, 2012.

[Cai09] Xuan Cai. Canonical coin systems for change-making problems. In *2009 Ninth International Conference on Hybrid Intelligent Systems*, 2009.

[CDM+12] Jianjun Chen, Chris Douglas, Michi Mutsuzaki, Patrick Quaid, Raghu Ramakrishnan, Sriram Rao, and Russell Sears. Walnut : a unified cloud object store. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, 2012.

[CEA15] CEA. Tera 1000 : Cea completes first milestone towards exascale. `http://www.cea.fr/english/Pages/News/tera1000-cea-completes-first-milestone-towards-exascale.aspx`, 2015.

[CEA16] CEA. Ccrt boosts industrial innovation with a petascale supercomputer from bull. `http://www.cea.fr/english/Pages/News/ccrt-boosts-industrial-innovation-with-a-petascale-supercomputer-from-Bull` `aspx`, 2016.

[CEA19] CEA. Inauguration of joliot-curie, the french supercomputer dedicated to french and european research. `http://www.cea.fr/english/Pages/News/Inauguration-of-Joliot-Curie,-the-French-supercomputer-dedicated-to-French-and-European-research.aspx`, 2019.

[Cen02] EMC Centera. Content addressed storage. *Product description*, 2002.

[Cis19] Cisco. Cisco visual networking index : Cisco visual networking indexglobal mobile data traffic forecast update, 2017–2022. Technical report, 2019.

[Cla07] Michael Clark. Json-c - a json implementation in c. `https://github.com/json-c/json-c`, 2007.

[CLRS09] Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein. *Introduction to algorithms*. MIT press, 2009.

[CPK+18] Badrish Chandramouli, Guna Prasaad, Donald Kossmann, Justin Levandoski, James Hunter, and Mike Barnett. Faster : A concurrent key-value store with in-place updates. In *Proceedings of the 2018 International Conference on Management of Data*, 2018.

[CXZ07] Bin Cai, Changsheng Xie, and Guangxi Zhu. Performance evaluation of a load self-balancing method for heterogeneous metadata server cluster using trace-driven and synthetic workload simulation. In *2007 IEEE International Parallel and Distributed Processing Symposium*, 2007.

[DAI17] Niv Dayan, Manos Athanassoulis, and Stratos Idreos. Monkey : Optimal navigable key-value store. In *Proceedings of the 2017 ACM International Conference on Management of Data*, pages 79–94, 2017.

[DB16] Francois Diakhate and Jean-Baptiste Besnard. Pcocc : Run vms on an hpc cluster. `https://github.com/cea-hpc/pcocc`, 2016.

[DHJ+07] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo : amazon's highly available key-value store. *ACM SIGOPS operating systems review*, 2007.

[DHJ+15] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo. In *Proc. of CS 848 : Modern Database Systems*, 2015.

[DMS+18] Jack J Dongarra, Hans W Meuer, Erich Strohmaier, et al. Top500 list - june 2018 | top500 supercomputer sites, June 2018. https ://www.top500.org/lists/2018/06/. [Online].

[ELSC13] Ifeanyi P Egwutuoha, David Levy, Bran Selic, and Shiping Chen. A survey of fault tolerance mechanisms and checkpoint/restart implementations for high performance computing systems. *The Journal of Supercomputing*, 2013.

[FGM+99] Roy Fielding, James Gettys, Jeffrey Mogul, Henrik Frystyk, Larry Masinter, Paul Leach, and Tim Berners-Lee. Rfc 2616 : Hypertext transfer protocol–http/1.1, june 1999. *Status : Standards Track*, 1999.

[FLP85] Michael J Fischer, Nancy A Lynch, and Michael S Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM (JACM)*, 1985.

[GAA+20] Xavier Gonze, Bernard Amadon, Gabriel Antonius, Frédéric Arnardi, Lucas Baguet, Jean-Michel Beuken, Jordan Bieder, François Bottin, Johann Bouchet, Eric Bousquet, Nils Brouwer, Fabien Bruneval, Guillaume Brunin, Théo Cavignac, Jean-Baptiste Charraud, Wei Chen, Michel Côté, Stefaan Cottenier, Jules Denier, Grégory Geneste, Philippe Ghosez, Matteo Giantomassi, Yannick Gillet, Olivier Gingras, Donald R. Hamann, Geoffroy Hautier, Xu He, Nicole Helbig, Natalie Holzwarth, Yongchao Jia, François Jollet, William Lafargue-Dit-Hauret, Kurt Lejaeghere, Miguel A. L. Marques, Alexandre Martin, Cyril Martins, Henrique P. C. Miranda, Francesco Naccarato, Kristin Persson, Guido Petretto, Valentin Planes, Yann Pouillon, Sergei Prokhorenko, Fabio Ricci, Gian-Marco Rignanese, Aldo H. Romero, Michael Marcus Schmitt, Marc Torrent, Michiel J. van Setten, Benoit Van Troeye, Matthieu J. Verstraete, Gilles Zérah, and Josef W. Zwanziger. The abinit project : Impact, environment and recent developments. *Comput. Phys. Commun.*, 2020.

[GGL03] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The google file system. In *ACM SIGOPS operating systems review*, 2003.

[GVM00] Garth A Gibson and Rodney Van Meter. Network attached storage architecture. *Communications of the ACM*, 2000.

[Hin13] Pieter Hintjens. *ZeroMQ, Messaging for Many Applications*. O'Reilly Media, 2013.

[HISV14] Jürg Hutter, Marcella Iannuzzi, Florian Schiffmann, and Joost VandeVondele. cp2k : atomistic simulations of condensed matter systems. *Wiley Interdisciplinary Reviews : Computational Molecular Science*, 2014.

[HKJR10] Patrick Hunt, Mahadev Konar, Flavio Paiva Junqueira, and Benjamin Reed. Zookeeper : Wait-free coordination for internet-scale systems. In *USENIX annual technical conference*, 2010.

[HM04] RJ Honicky and Ethan L Miller. Replication under scalable hashing : A family of algorithms for scalable decentralized data distribution. In *Parallel and Distributed Processing Symposium, 2004. Proceedings. 18th International*, 2004.

[HZJ+08] Yu Hua, Yifeng Zhu, Hong Jiang, Dan Feng, and Lei Tian. Scalable and adaptive metadata management in ultra large-scale file systems. In *2008 The 28th International Conference on Distributed Computing Systems*, 2008.

[IEE88] IEEE. *IEEE 1003 - IEEE Standard for Information Technology - Portable Operating System Interface (POSIX(R))*, 1988.

[Ins81] Information Sciences Institute. Transmission Control Protocol. RFC 793, 1981.

[Kat97] Jeffrey Katcher. Postmark : A new file system benchmark. Technical report, Technical Report TR3022, Network Appliance, 1997.

[KLL+97] David Karger, Eric Lehman, Tom Leighton, Matthew Levine, Daniel Lewin, and Rina Panigrahy. Consistent hashing and random trees : Distributed caching protocols for relieving hot spots on the world wide web. In *STOC*, 1997.

[KMBL18] Julian Kunkel, George Markomanolis, John Bent, and Jay Lofstead. Io500 full ranked list, supercomputing 2018 (corrected), Nov 2018. http ://io500.org/list/19-01/start. [Online].

[Lam98] Leslie Lamport. The part-time parliament. *ACM Transactions on Computer Systems (TOCS)*, 1998.

[LM10] Avinash Lakshman and Prashant Malik. Cassandra : a decentralized structured storage system. *ACM SIGOPS Operating Systems Review*, 2010.

[LWV07] Wujuan Lin, Qingsong Wei, and Bharadwaj Veeravalli. Wpar : A weight-based metadata management strategy for petabyte-scale object storage systems. In *Storage Network Architecture and Parallel I/Os, 2007. SNAPI. International Workshop on*, 2007.

[LXSZ06] Weijia Li, Wei Xue, Jiwu Shu, and Weimin Zheng. Dynamic hashing : adaptive metadata management for petabyte-scale file systems. In *23rd IEEE/14th NASA Goddard Conference on Mass Storage System and Technologies*, 2006.

[MBO+11] Vilobh Meshram, Xavier Besseron, Xiangyong Ouyang, Raghunath Rajachandrasekar, Ravi Prakash Darbha, and Dhabaleswar K Panda. Can a decentralized metadata service layer benefit parallel filesystems ? In *2011 IEEE International Conference on Cluster Computing*, 2011.

[MCA+16] Pierre Matri, Alexandru Costan, Gabriel Antoniu, Jesús Montes, and María Pérez. *Tỳr : Efficient Transactional Storage for Data-Intensive Applications*. PhD thesis, Inria Rennes Bretagne Atlantique ; Universidad Politécnica de Madrid, 2016.

[Mel16] Mellanox. Accelio – open-source io, message, and rpc acceleration library. `https://github.com/accelio/accelio`, 2016.

[Mes17] P. Messina. The exascale computing project. *Computing in Science Engineering*, 2017.

[MGR03] Mike Mesnier, Gregory R Ganger, and Erik Riedel. Object-based storage. *IEEE Communications Magazine*, 2003.

[MLMK11] Christopher J. Morrone, Bill Loewe, Tyce McLarty, and Ryan Kroiss. Hpc io benchmark repository. `https://github.com/hpc/ior`, 2011.

[MS96] Lily Mummert and Mahadev Satyanarayanan. Long term distributed file reference tracing : Implementation and experience. *Software : Practice and Experience*, 1996.

[NIG+16] Salman Niazi, Mahmoud Ismail, Steffen Grohsschmiedt, Mikael Ronström, Seif Haridi, and Jim Dowling. Hopsfs : Scaling hierarchical file system metadata using newsql databases. *arXiv :1606.01588*, 2016.

[Pan07] Panasas. Object storage architecture. Technical report, Panasas, 2007.

[PJS+94] Brian Pawlowski, Chet Juszczak, Peter Staubach, Carl Smith, Diane Lebel, and Dave Hitz. Nfs version 3 : Design and implementation. In *USENIX Summer*, 1994.

[RFB11] Ioan Raicu, Ian T Foster, and Pete Beckman. Making a case for distributed file systems at exascale. In *Proceedings of the third international workshop on Large-scale system and application performance*, 2011.

[RLA+00] Drew S Roselli, Jacob R Lorch, Thomas E Anderson, et al. A comparison of file system workloads. In *USENIX annual technical conference, general track*, 2000.

[RT+00] Robert B Ross, Rajeev Thakur, et al. Pvfs : A parallel file system for linux clusters. In *Proceedings of the 4th annual Linux showcase and conference*, 2000.

[SB18] Harcharan Jit Singh and Seema Bawa. Scalable metadata management techniques for ultra-large distributed storage systems–a systematic review. *ACM Computing Surveys (CSUR)*, 2018.

[Sco11] Peter Scott. C port of murmur3 hash. `https://github.com/PeterScott/murmur3`, 2011.

[SKK+90] Mahadev Satyanarayanan, James J. Kistler, Puneet Kumar, Maria E. Okasaki, Ellen H. Siegel, and David C. Steere. Coda : A highly available file system for a distributed workstation environment. *IEEE Transactions on computers*, 1990.

[Sko19] Thomas Skordas. Toward a european exascale ecosystem : the eurohpc joint undertaking. *Communications of the ACM*, 2019.

[SKZ+13] Jerome Soumagne, Dries Kimpe, Judicael Zounmevo, Mohamad Chaarawi, Quincey Koziol, Ahmad Afsahi, and Robert Ross. Mercury : Enabling remote procedure call for high-performance computing. In *2013 IEEE International Conference on Cluster Computing (CLUSTER)*, 2013.

[SMK+01] Ion Stoica, Robert Morris, David Karger, M Frans Kaashoek, and Hari Balakrishnan. Chord : A scalable peer-to-peer lookup service for internet applications. *ACM SIGCOMM Computer Communication Review*, 2001.

[Sto12] Michael Stonebraker. Newsql : An alternative to nosql and old sql for new oltp apps. *Communications of the ACM. Retrieved*, pages 07–06, 2012.

[STSM12] Dimokritos Stamatakis, Nikos Tsikoudis, Ourania Smyrnaki, and Kostas Magoutis. Scalability of replicated metadata services in distributed file systems. In *IFIP International Conference on Distributed Applications and Interoperable Systems*, 2012.

[SW14] Brian Stevens and Sage Weil. Red hat to acquire inktank, provider of ceph. https ://www.redhat.com/fr/about/press-releases/red-hat-acquire-inktank-provider-ceph, 2014.

[SXH+11] Ning-Hui Sun, Jing Xing, Zhi-Gang Huo, Guang-Ming Tan, Jin Xiong, Bo Li, and Can Ma. Dawning nebulae : a petaflops supercomputer with a heterogeneous structure. *Journal of Computer Science and Technology*, 2011.

[TBD+17] Houjun Tang, Suren Byna, Bin Dong, Jialin Liu, and Quincey Koziol. Someta : Scalable object-centric metadata management for high performance computing. In *Cluster Computing (CLUSTER), 2017 IEEE International Conference on*, 2017.

[TL18] Mellanox Technologies and Oak Ridge National Laboratory. Reaching the summit with infiniband : Mellanox interconnect accelerates world's fastest hpc and artificial intelligence supercomputer at oak ridge national laboratory (ornl). Technical report, Mellanox Technologies and Oak Ridge National Laboratory, 2018.

[Tre05] Michael Treaster. A survey of fault-tolerance and fault-recovery techniques in parallel systems. *arXiv preprint cs/0501002*, 2005.

[VdSB+18] S S Vazhkudai, B R de Supinski, A S Bland, A Geist, J Sexton, J Kahle, C J Zimmer, S Atchley, S H Oral, D E Maxwell, V G

Vergara Larrea, A Bertsch, R Goldstone, W Joubert, C Chambreau, D Appelhans, R Blackmore, B Casses, G Chochia, G Davison, M A Ezell, E Gonsiorowski, L Grinberg, B Hanson, B Hartner, I Karlin, M L Leininger, D Leverman, C Marroquin, A Moody, M Ohmacht, R Pankajakshan, F Pizzano, J H Rogers, B Rosenburg, D Schmidt, M Shankar, F Wang, P Watson, B Walkup, L D Weems, and J Yin. The design, deployment, and evaluation of the coral pre-exascale systems. 2018.

[vRG10] Robbert van Renesse and Rachid Guerraoui. Replication techniques for availability. In *Replication*. 2010.

[WAN19] Ruibo WANG. Tianhe-3 and the exascale road in china. 1st R-CCS International Symposium, 2019.

[WBM+06] Sage A Weil, Scott A Brandt, Ethan L Miller, Darrell DE Long, and Carlos Maltzahn. Ceph : A scalable, high-performance distributed file system. In *Proceedings of the 7th symposium on Operating systems design and implementation*, 2006.

[WBMM06] Sage A Weil, Scott A Brandt, Ethan L Miller, and Carlos Maltzahn. Crush : Controlled, scalable, decentralized placement of replicated data. In *Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, 2006.

[WFWL09] Juan Wang, Dan Feng, Fang Wang, and Chengtao Lu. Mhs : A distributed metadata management strategy. *Journal of Systems and Software*, 2009.

[WK86] Thomas Williams and Colin Kelley. Gnuplot - a portable, multiplatform, command-line driven graphing utility. `http://www.gnuplot.info/`, 1986.

[WPBM04] Sage A Weil, Kristal T Pollack, Scott A Brandt, and Ethan L Miller. Dynamic metadata management for petabyte-scale file systems. In *Proceedings of the 2004 ACM/IEEE conference on Supercomputing*, 2004.

[WXH+04] Feng Wang, Qin Xin, Bo Hong, Scott A Brandt, Ethan Miller, Darrell Long, and T McLarty. File system workload analysis for large scale scientific computing applications. Technical report, Lawrence Livermore National Lab.(LLNL), Livermore, CA (United States), 2004.

[XAYM14] Quanqing Xu, Rajesh Vellore Arumugam, Khai Leong Yong, and Sridhar Mahadevan. Efficient and scalable metadata management in ebscale file systems. *IEEE Transactions on Parallel and Distributed Systems*, 2014.

[XHL+10] Jin Xiong, Yiming Hu, Guojie Li, Rongfeng Tang, and Zhihua Fan. Metadata distribution and consistency techniques for large-scale cluster file systems. *IEEE Transactions on Parallel and Distributed Systems*, 2010.

[XRZG15] Lin Xiao, Kai Ren, Qing Zheng, and Garth A Gibson. Shardfs vs. indexfs : replication vs. caching strategies for distributed metadata management in cloud storage systems. In *Proceedings of the Sixth ACM Symposium on Cloud Computing*, 2015.

[XXSM09] Jing Xing, Jin Xiong, Ninghui Sun, and Jie Ma. Adaptive and scalable metadata management to support a trillion files. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, 2009.

[XZ12] Wei Xue and Ming Zhu. Efficient dynamic management of distributed metadata. In *Information Engineering and Applications*. 2012.

[YWL+14] Di Yang, Weigang Wu, Zhansong Li, Jiongyu Yu, and Yong Li. Ppms : A peer to peer metadata management strategy for distributed file systems. In *IFIP International Conference on Network and Parallel Computing*, 2014.

[ZCW+12] Qing Zheng, Haopeng Chen, Yaguang Wang, Jiangang Duan, and Zhiteng Huang. Cosbench : A benchmark tool for cloud object storage services. In *2012 IEEE Fifth International Conference on Cloud Computing*, 2012.

[ZJWX08] Yifeng Zhu, Hong Jiang, Jun Wang, and Feng Xian. Hba : Distributed metadata management for large cluster-based storage systems. *IEEE transactions on parallel and distributed systems*, 2008.

[ZRG14] Qing Zheng, Kai Ren, and Garth Gibson. Batchfs : scaling the file system control plane with client-funded metadata servers. In *Parallel Data Storage Workshop (PDSW), 2014 9th*, 2014.

[ZRG+15] Qing Zheng, Kai Ren, Garth Gibson, Bradley W Settlemyer, and Gary Grider. Deltafs : Exascale file systems scale better without dedicated servers. In *Proceedings of the 10th Parallel Data Storage Workshop*, 2015.

université PARIS-SACLAY

ÉCOLE DOCTORALE
Sciences et technologies de l'information et de la communication (STIC)

Titre : Développement d'une méthode de distribution de métadonnées équilibrée pour un flux de requêtes exascale.

Mots clés : Méthode de distribution, Gestion de métadonnées, Équilibrage de charge, Outil d'évaluation, Stockage objet.

Résumé : Dans le domaine du calcul haute performance (HPC), le stockage de plus en plus de données devient un facteur de performance à prendre en compte. Pour cela, l'utilisation des systèmes de stockage objet et des services de métadonnées distribués permet de pallier les limitations induites par la norme POSIX et d'améliorer la concurrence des accès. Toutefois, cela engendre de nouvelles problématiques telles que la distribution des métadonnées et l'équilibrage de charge des serveurs de métadonnées (MDS). Cette thèse a pour objectif de concevoir une nouvelle méthode de distribution des métadonnées permettant un équilibrage de la charge des serveurs sur un flux de requêtes HPC EXASCALE. Dans cette optique, nous avons développé la méthode Distribution adaptative à la charge (ou LAD) effectuant des redistributions de charge seulement quand cela est nécessaire, ainsi que sa variante avec fenêtre temporelle LAD-TW permettant une meilleure évaluation de la charge des serveurs. Afin d'évaluer nos méthodes et de les comparer à celles de l'état de l'art, nous avons conçu et développé MeDiE (pour Metadata Distribution Evaluator), un outil d'évaluation pour les méthodes de distribution des métadonnées. Il permet une étude de la répartition de la charge des différents serveurs, selon plusieurs flux de requêtes caractéristiques. Nous avons ainsi pu observer l'apport des méthodes LAD et LAD-TW par rapport aux méthodes de l'état de l'art testées.

Title : Development of a well-balanced metadata distribution method for an exascale requests flow.

Keywords : Distribution method, Metadata management, Load-balancing, Evaluation tool, Object storage.

Abstract : In the high-performance computing field (HPC), storage systems are an important performance factor. Using object storage systems coupled to distributed metadata services enable to overcome limitations induced by the POSIX norm, and to improve the data access concurrency. But this solution comes with its own drawbacks : it requires to deal with metadata distributions and load-balancing for metadata servers (MDS). This work aims to design a new metadata distribution method enabling to efficiently load balance requests between MDS for exascale HPC request flows. In this way, we develop a method named Load-Adaptive Distribution (or LAD), which rebalances the workload only when it is required, and its extension named Load-Adaptive Distribution with Temporal Window (or LAD-TW), which provides a better workload evaluation. In order to evaluate our methods and compare them with some state-of-the-art methods, we have developed MeDiE, for Metadata Distribution Evaluator, an evaluation tool dedicated to metadata distribution methods. It enables load-balancing analysis on different and representative request flows and allows us to prove the benefit of using the LAD and LAD-TW methods.

Maison du doctorat de l'Université Paris-Saclay,
2ème étage aile ouest, Ecole normale supérieure Paris-Saclay,
4 avenue des Sciences,91190 Gif sur Yvette, France