Placement des réplicas dans un système de gestion de données distribué à large échelle à protocole de cohérence adaptable

Etienne Mauffret

UNIVERSITÉ SAVOIE MONT BLANC

THÈSE

Pour obtenir le grade de

DOCTEUR DE L'UNIVERSITÉ SAVOIE MONT BLANC

Spécialité : **INFORMATIQUE**

Arrêté ministérial : 25 Mai 2016

Présentée par

Etienne Mauffret

Thèse dirigée par **Sébastien Monnet**
et codirigée par **Flavien Vernier**

préparée au sein **LISTIC**
et de **SIE**

Placement des réplicas dans un système de gestion de données distribué à large échelle à protocole de cohérence adaptable

Thèse soutenue publiquement le 20 juin 2022
devant le jury composé de :

Rapporteuse - Anne Benoit
Maîtresse de Conférence - ENS Lyon
Rapporteur - Gabriel Antoniu
Directeur de Recherche Inria Rennes - Bretagne Atlantique
Examinatrice - Christine Morin
Directrice de Recherche Inria Rennes - Bretagne Atlantique
Examinateur - Renaud Lachaize
Maître de Conférence - Université Grenoble Alpes
Directeur de thèse - Sébastien Monnet
Professeur - Université Savoie Mont-Blanc
Co-Directeur de thèse - Flavien Vernier
Maître de Conférence - Université Savoie Mont-Blanc

Remerciements

J'aimerais profiter de cet espace pour remercier chaleureusement toutes les personnes qui m'ont accompagnées pendant ce travail, à la fois long et court, qu'est le doctorat.

Dans un premier temps, j'aimerais adresser des remerciements particuliers à Sébastien et Flavien. On dit souvent que le lien entre le doctorant et ses directeurs est fort et c'est le cas. Sébastien et Flavien se sont assurés que ma thèse se déroule dans les meilleurs conditions. Ils m'ont permis apprendre énormément à leurs côtés, ils ont toujours cru en moi et m'ont témoigné leur soutien au cours de ce travail.

Un grand merci à mon jury, Anne, Gabriel, Christine et Renaud pour avoir pris le temps de relire et évaluer mon travail. Merci également pour leurs retours, leurs questions et nos échanges. Merci à Marc de m'avoir accepté au sein du projet RainbowFS et pour nos échanges toujours très intéressants.

Par la suite, j'ai eu la chance de rencontrer Eddy, Christian et l'equipe AVALON. Merci à eux de m'avoir intégré dans ce cadre de travail exceptionnel ainsi que pour nos nombreuses discussions.

Merci à l'équipe des doctorants et post-doctorants du LISTIC, en particulier Alex, Fanny, Hermann, Hela, Emna, Melissa et Laurane, pour tous les moments que nous avons passés ensembles et d'être des amis avant d'être des collègues.

Je remercie du fond du coeur mes amis, Chatons et Cool Kids, Caro, Antho, Tilina, Sixtine, Eran, Brice, John, Damien, de m'avoir entouré pendant ce travail et de comprendre les implications de mener un travail de doctorat. Merci de votre bienveillance, leur soutien et tous ces moments chaleureux à vos côtés. Je n'ai pas assez de mots pour vous exprimer ma gratitude. En particulier, merci à Julie d'avoir été là, pour son soutien, sa capacité à faire remonter le meilleur et me pousser vers le haut. Pour toutes nos discussions, ses remarques et simplement sa présence, merci. Merci à Aurélien pour son soutient sans faille au fil des années et nos longues soirées. A Ulysse pour toutes ces soirées à échanger et sa gentillesse pure. Merci d'être vous et d'être présent.

J'ai également eu la chance d'être entoure de ma famille, toujours prête à plaisanter et m'encourager. Claire et Tanguy qui ont toujours été là pour moi et sur qui je sais que je peux compter. A Tom pour sa bienveillance et de prendre soin de son petit frère.

Enfin, merci à Lina et Brimaz pour leur compagnie et leurs infatigable encouragements.

Placement des réplicas dans un système de gestion de données distribué à large échelle à protocole de cohérence adaptable

Etienne Mauffret
Thèse dirigée par Flavien Vernier et Sebastien Monnet

Résumé

Grâce au développement des différentes technologies et architectures clouds, de nombreux services s'exécutent à une échelle mondiale tout en manipulant des volumes de données plus importants que jamais. Dans notre société, la manipulation de données occupe une place primordiale, entre autres afin de proposer les services les plus adaptés aux utilisateurs. L'International Data Corporation estime à 385 milliards de dollars l'investissement mondial sur les services de clouds partagés. Afin de répondre à de telles demandes, de nombreux systèmes de gestion de données distribués ont été développés. Ils utilisent la réplication des données pour permettre de faire face à la charge de travail imposée par les utilisateurs tout en étant capable de garantir un service robuste et efficace. Chaque donnée est copiée plusieurs fois et stockée sur un emplacement de stockage différent. Il devient alors nécessaire d'être capable d'estimer le nombre de réplicas idéal et leur place. En

effet, en ayant accès à différents réplicas pour une même donnée, il est possible de traiter certaines requêtes de manière concurrente, ce qui peut entraîner des incohérences entre les états des réplicas. Ces incohérences peuvent être critiques ou non selon les applications et services.

Il existe ainsi de nombreux modèles de cohérences, proposant différentes garanties. La plupart des travaux sur ces modèles étudient les propriétés d'un modèle à la fois ou en proposent une classification (partielle), mais ne se concentrent que sur les propriétés de cohérences. Cependant, les garanties de cohérence sont étroitement liées aux garanties de disponibilité et de tolérance aux fautes. Il est également nécessaire d'adapter certaines stratégies, en particulier celles du placement des réplicas, afin de ne pas détériorer les performances du service.

Certaines applications récentes manipulent différents types de données, avec des besoins différents. Il devient alors nécessaire d'utiliser un système de gestion de données capable de fournir différents niveaux de garanties pour différentes données. Nous détaillons alors les propriétés liées à certains modèles de cohérences couramment utilisés par les systèmes existants, leurs nuances et leurs impacts sur la disponibilité, la tolérance aux fautes et la stratégie de placement des réplicas à adopter. Nous proposons donc ici des études à la fois théoriques et pratiques. En effet, nous étudions dans un premier temps des problèmes fondamentaux des systèmes de gestion de données distribués. Nous nous intéressons ensuite aux choix de conceptions de différents systèmes existants, comme Dynamo, Cassandra ou BigTable.

Ces études nous ont permis de mettre au point CAnDoR, une méthode de placement de réplicas dans un système de gestion de données distribué à protocoles de cohérence adaptables. Chaque modèle de cohérence impose différentes contraintes sur la façon de traiter les réplicas. Nous modélisons donc ces contraintes afin de pondérer différemment l'importance du temps de réponse à l'utilisateur et le temps de propagation d'une requête afin d'estimer un placement efficace en fonction du modèle de cohérence attaché aux données. Ces métriques prennent également en compte le comportement des utilisateurs afin de s'adapter aux utilisateurs les plus actifs. Pour faire face à la complexité de ce problème, nous calculons en priorité le placement des réplicas des données les plus consultées, permettant d'améliorer les performances en réduisant la charge de calcul.

Nous avons mis au point CandorSim, un simulateur pour évaluer les performances d'un système de gestion de données distribué. À l'aide d'études de différents systèmes existants, nous avons paramétré notre simulateur afin de nous approcher d'un contexte réaliste dans lequel nous avons évalué notre méthode de placement. Ces évaluations nous ont permis de mettre en avant l'importance de considérer à la fois le modèle de cohérence et l'activité des utilisateurs pour proposer un service de placement efficace.

Placement of Replica in Large Scale Distributed Data Management Systems with Tunable Consistency Protocols

Etienne Mauffret
Thesis directed by Flavien Vernier and Sebastien Monnet

Abstract

With the development of various cloud technologies and architectures many services are running on a global scale while handling larger volumes of data. In our society, the manipulation of data occupies a primordial place, for example in order to provide the most adapted services to users. The International Data Corporation estimates the global investment in shared cloud services at $385 billion. In order to meet such demands, many distributed data management systems have been developed. They use data replication to cope with the potential load induced by users while maintaining potential user load while still being able to guarantee a robust and efficient service. Each data is copied multiple times and stored on a different storage location. It becomes necessary to be able to estimate the ideal number of replicas, their location and consistency guarantees of consistency. Indeed, by having access to different replicas for the same data it is possible to

process some queries concurrently, which could lead to inconsistencies between the states of the replicas. These inconsistencies may or may not be critical depending on the applications and services.

However, there are many different models of consistency, offering different guarantees. Most of the works on these models study the properties of one model at a time or propose a (partial) classification of these models, while only focusing on the consistency properties. However, consistency guarantees are closely related to availability and fault tolerance guarantees. It is also necessary to adapt strategies, the replica placement strategy for example, in order not to deteriorate the performance of the service.

Some recent applications handle different kinds of data, each piece of data can have different needs. It becomes necessary to use management systems capable of providing different levels of guarantees for different data. We then detail the properties related to some consistency models commonly used by existing systems. We developp their nuances and impacts on availability, fault tolerance and on the replica placement strategy to adopt. We therefore propose here a theoretical studies of the fundamental problems of distributed data management as well as practical studies on the design choices of various existing design choices of various existing systems, such as Dynamo, Cassandra or BigTable.

These studies allowed us to develop CAnDoR, a method for placing replicas in a distributed data management system with adaptable consistency protocols. Each consistency model imposes different constraints on the way replicas are handled. We therefore model these constraints in order to weight differently the metrics. The user response time and the query propagation time are used to compute an efficient placement according to the consistency model attached to the data. These metrics also take into account user behavior in order to adapt to the most active users. To cope with the complexity of this problem, we compute in priority the placement of replicas of the most accessed data. This method allowing us to improve performance by reducing the computational load.

Finally, we have developed CandorSim, a simulator to evaluate the performance of a distributed data management system. Using studies of different existing systems, we have parameterized our simulator in order to modelize a realistic context. This context has been used to have evaluated our placement method. These evaluations allowed us to highlight the importance of considering both the consistency model and the users' activity to propose an efficient placement service.

Table des matières

Table des figures

I

Partie I - Contexte & état de l'art

Sommaire

Chapitre 1

Introduction

1.1 Les systèmes de gestion de données distribués

Ces dernières années ont été les témoins de l'émergence de nombreuses plateformes de gestion de données distribuées dans le *Cloud*, telles Amazon Web Service (AWS S3) [Amaa] ou Microsoft Azure Cosmos DB [Sah20]. Ces systèmes distribués à large échelle ont rapidement joué un rôle essentiel sur le marché du stockage de données, marché étant lui même au centre de nombreuses problématiques de notre société actuelle. Nous pouvons en effet observer la multiplication des applications se basant sur l'utilisation de données : de partage de contenu entre utilisateurs (Facebook, Twitter, TikTok), diffusion de contenu par un créateur (Youtube, Le Monde), utilisation de données par des utilisateurs (GPS, Webmail), services adaptés aux utilisateurs (publicités personnalisées, suggestions de préférences). Certaines de ces applications vont jusqu'à manipuler des petabytes de données quotidiennement [Fac], il faut donc être capable de non seulement stocker de telles quantités de données, mais également être capable de les manipuler *efficacement* afin de rester pertinent face à la concurrence.

L'International Data Corporation estime à 385 milliards de dollars l'investissement mondial sur les services de Clouds partagés avec une croissance annuelle de 21%. Un montant qui devrait atteindre 809 milliards de dollars d'ici 2025 [1].

1. https ://www.idc.com/getdoc.jsp ?containerId=prUS48208321

Il est donc naturel que de très nombreux travaux de recherche se soient intéressés à la problématique du stockage distribué, le stockage centralisé étant inadapté aux masses de données et aux accès géo-distribués auxquels doivent faire face ces applications. Ces travaux proposent des analyses des concepts et problématiques de ce domaine [GL02], [BDF+13], [VV16], nouveaux systèmes de fichiers distribués [TSJ+10], [SKRC10], [GGL03] ou des systèmes de gestion de données distribués dans le *Cloud* [DHJ+07], [ATB+16], [HKJR10] et différentes analyses à leur sujet [JWF+11], [GGSZ09], [HHLD11]. Afin de fournir certaines garanties de disponibilité et pérennité, ces systèmes répliquent les données dans le système : chaque donnée est copiée un certain nombre de fois et ces copies sont stockée sur des noeuds différents. Ce mécanisme peut cependant entraîner un comportement indésirable de la part du service : les différentes copies pourraient se retrouver dans des états différents. C'est ce que l'on appelle la cohérence entre les réplicas. Cette propriété n'est cependant pas binaire et il existe de nombreux modèles différents assurant différentes propriétés entre les réplicas des données.

Lors de la conception d'un service ayant besoin d'un service de stockage de données distribué, les concepteurs déterminent généralement quel modèle de cohérence appliquer et choisissent (ou développent) un système utilisant un protocole permettant de maintenir les propriétés désirées. Cependant, les applications devenant de plus en plus complètes et complexes, il est courant qu'elles utilisent des données soumises à différents types de contrainte. De nouveaux modèles émergent afin de proposer des services capables de s'adapter aux besoins des applications, et non plus l'inverse [SBP+18] ! Certains systèmes proposent de déterminer, pour chaque type de donnée, quel modèle de cohérence appliquer [Sah20]. Le travail effectué dans le cadre de cette thèse se concentre sur la modélisation de ces services, des problématiques qu'ils abordent et l'analyse du placement des données dans de tels contextes. Nous avons également développé un simulateur à événements discrets se basant sur PeerSim [MJ09] afin de proposer une première implémentation de notre approche de placement. Mais avant de rentrer dans le détail, commençons par définir ce qu'est un système de gestion de données distribué.

Nous appelons *système distribué* un ensemble d'éléments capables de communiquer. Ces communications sont utilisées, entre autres, pour permettre aux différents éléments d'accomplir des tâches qu'un système centralisé ne pourrait pas conduire sans difficulté (que ça soit par la nature, la vitesse ou la taille du problème considéré). Les éléments collaborent donc par la communication des calculs effectués localement en s'envoyant les résultats d'opérations. Dans un système de gestion de données distribué, il est également possible que deux éléments s'envoient directement des données. Les communications sont assurées par les *liens de communications*.

Afin de faciliter leur étude, et d'en exhiber certaines propriétés, il est naturel de

représenter les systèmes distribués par des graphes dans lesquels les éléments sont représentés par des nœuds et les liens de communications par des arêtes. Dans le reste de nos travaux, nous appelons principalement ces éléments des *nœuds*, mais nous employons aussi les termes *processeurs* ou *points (de calcul)*.

Il existe trois types d'architecture dans le Cloud, comme définis dans [JWF+11] :

- Publique : conçue spécifiquement pour des besoins à large échelle pour de nombreux utilisateurs simultanés. Tous les composants sont construits sur une infrastructure partagée et les partitions logiques des périphériques de stockage public sont construites à partir des besoins des utilisateurs, en fonction des technologies de virtualisation et de gestion des données disponibles.
- Privé, ou interne : conçue pour un utilisateur spécifique. Contrairement au stockage sur Cloud public, les Clouds internes s'exécutent sur des périphériques dédiés dans des centres de données. Cette approche permet un gain conséquent de performances et de sécurité, au prix d'une perte de capacité de passage à l'échelle.
- Hybride : intégrant un Cloud public et un Cloud privé. Dans la plupart des cas, les stockages sur Cloud hybride sont issus de Clouds privés enrichis de Cloud public pour satisfaire leurs besoins (passage à l'échelle, etc.).

Les architectures Clouds, publiques ou privées, étant très hétérogènes par nature, chaque nœud exécute ses opérations à sa vitesse, indépendamment des autres. On dit alors que le système est asynchrone et décentralisé.

Cette thèse étudie en particulier les systèmes de gestion de données distribués : les systèmes sont composés de différents points de stockage, contenant chacun un ensemble de données. Les utilisateurs accèdent à un de ces points afin d'obtenir un accès aux données qui les intéressent. Plus généralement, nous considérons que les nœuds stockent des copies entières des données, ainsi une opération d'accès à la donnée peut être faite en une seule opération locale.

Le stockage dans les Clouds se faisant à grande échelle, les probabilités qu'un nœud devienne fautif sont élevées. Un nœud fautif peut alors être réparé (et rejoindre le système), être remplacé ou être délaissé. De plus, il n'est pas rare que de nouveaux nœuds rejoignent le système et que d'autres le quittent. Il en est de même pour les liens de communications, pouvant évoluer dans le temps. De tels systèmes sont appelés systèmes dynamiques et il est nécessaire de mettre en place des protocoles se basant sur des algorithmes capables de rester corrects face à ces irrégularités. Dans notre contexte, une faute, une arrivée ou un départ doivent être considérés comme une situation *normale* plutôt que comme une situation exceptionnelle. Afin de permettre l'étude de tels systèmes, les graphes les représentant peuvent être enrichis de fonctions symbolisant leurs évolutions dans le temps, c'est ce que l'on appelle les Time Varying Graph [CFQS11].

1.2 La cohérence dans un système de gestion de données distribué

Lorsque des données sont stockées dans le Cloud, elles sont généralement répliquées, les différentes copies d'une donnée sont appelées réplicas de cette donnée [TVS07], [MN16]. Ce mécanisme apporte de nombreux avantages et défis, que nous explorons dans le Chapitre 2.

Une des premières questions qui peut apparaître lorsque l'on a recours à la réplication de données est la question du placement des réplicas. Il existe différentes approches pour résoudre ce problème, mais le problème associé a été montré NP-Difficile [ST01]. Certains systèmes préfèrent alors avoir recours à un placement aléatoire des réplicas, en estimant que cela n'entraînera qu'une faible dégradation des performances. D'autres méthodes consistent à proposer une méthode approchée [CMM+13] ou de réduire la taille du problème afin de pouvoir effectuer le calcul en un temps raisonnable [PRRR14].

Une autre problématique est de pouvoir s'assurer que toutes les modifications apportées à un réplica soient propagées aux autres réplicas. Dans le cas où des modifications concurrentes sont faites sur différents réplicas d'une même donnée, il est possible de l'état de ces derniers diverge. Ce problème est référé comme le problème de *cohérence entre les réplicas d'une donnée*.

Il existe de nombreux travaux à ce sujet, mais un résultat a particulièrement marqué l'étude (et la conception) des systèmes de gestion de données distribués : la conjecture de Brewer, aussi appelé le théorème du CAP [Bre00]. Ce théorème expose que tout système distribué ne peut offrir que deux propriétés parmi la cohérence, la disponibilité et la tolérance aux partitions. Le nom du théorème est issu du nom anglais de ces propriétés : Consistency, Availability and Partition tolerance.

Il apparaît que pour de nombreuses applications existantes, il est difficile d'affaiblir la propriété de tolérance aux partitions. Ces applications, et les systèmes de gestion de données sous-jacents, doivent donc établir un équilibre entre cohérence et disponibilité. Au cours des discussions informelles, et de la classification des différents systèmes de gestion de données distribués, il est courant de se référer aux systèmes selon les catégories "AP" et "CP", selon le nom des propriétés du théorème du CAP. Un système "AP" est un système assurant fournir une forte disponibilité et une tolérance aux partitions (Availability-Partition tolerance) et fournira donc un modèle de cohérence plus faible que la cohérence forte. À l'opposé, un système dit "CP" promet de maintenir une cohérence forte entre les réplicas, même en cas de partitions dans le système (Consistency-Partition tolerance) mais ne pourra offrir qu'un modèle de disponibilité plus faible, parfois appelé *best-effort availability*.

Cependant, les applications modernes visent à devenir de plus en plus complètes et complexes, entraînant la manipulation d'objets divers avec des propriétés pouvant varier d'un objet à l'autre. Il y a alors un véritable besoin de systèmes de gestion de données capables de manipuler différents équilibres de cohérences et disponibilités tout en étant performants lors d'une exécution à une échelle mondiale. C'est par exemple le cas de Cassandra [LM10] ou AzureDB [Sah20] qui proposent à l'utilisateur de choisir les contraintes de cohérence à appliquer à chaque donnée, individuellement.

Les travaux effectués dans cette thèse se concentrent sur les systèmes de gestion de données non-transactionnels. Dans ces systèmes, les données sont considérées comme indépendantes les unes des autres. Ainsi, une requête portant sur une donnée n'aura pas de conséquence sur les autres données. Il est alors pertinent, afin de simplifier les explications, de considérer les scénarios avec une seule donnée dans le système lorsque nous cherchons à exprimer les propriétés des différents modèles. Nous nous intéressons plus spécifiquement aux systèmes capables d'adapter le modèle cohérence aux besoins de l'utilisateur. Pour cela, nous avons étudié les différentes propriétés des modèles de cohérences et la relation entre ces propriétés et les performances du système afin de proposer une méthode de placement des différents réplicas des données dans des systèmes à modèles de cohérence adaptables.

1.3 Organisation du manuscrit et contributions

Organisation du manuscrit

Les travaux effectués au cours de cette thèse reposent autant sur des études théoriques, telles que celles des problèmes de cohérence, gestion de la connaissance ou du placement des réplicas dans un système distribué ; que sur des applications de ces problèmes à travers la conception de systèmes de gestion de données distribués. Nous avons également conduit une étude de nombreux systèmes de gestion de données existants. Le reste de ce manuscrit s'articule de la manière décrite ci-dessous.

La Partie I présente l'état des connaissances actuelles des systèmes de gestion de donnée distribués. Plus précisément, le Chapitre 2 présente le contexte dans lequel se placent les travaux ici présentés. Nous y explorons les principales problématiques liées aux systèmes de gestion de données distribués. Après avoir présenté trois résultats fondamentaux ayant orienté de nombreux travaux de recherche du domaine, nous détaillons les problématiques étudiées durant cette thèse. Ce chapitre présente également un état de l'art sur les différentes méthodes de placements des données dans un système de gestion de données distribué. Le Chapitre 3 étudie

les différents modèles de cohérence plus en profondeur et propose une étude pédagogique afin d'appréhender les différences entre les trois familles de modèles les plus répandues : les modèles de cohérence forte, causale et à terme. Le Chapitre 4 revient sur la formulation du théorème du CAP et de certaines de ses variantes ainsi que de leurs implications sur les systèmes de gestion de données. Ce chapitre propose également une étude théorique de différents systèmes existants et de leurs choix d'implémentations.

La Partie II détaille les contributions apportées dans cette thèse. Le Chapitre 5 présente CAnDoR, la contribution principale de cette thèse. CAnDoR est une approche permettant de proposer un placement efficace pour les réplicas des données d'un système à protocoles de cohérence adaptables. Le Chapitre 6 explore l'impact des différentes méthodes de construction et maintien de l'historique des événements d'un système distribué et conduit une étude de cas sur un système de gestion de données.

La partie III présente les évaluations des contributions de la thèse. Le Chapitre 7 détaille le fonctionnement de CandorSim, notre extension de PeerSim permettant la simulation d'un système de gestion de donnée simplifié. Ce simulateur a été utilisé afin d'évaluer nos méthodes selon différents scénario et paramètre. Les résultats des évaluations sont présentés dans le Chapitre 8.

Enfin, la partie IV présente le dernier chapitre qui conclut sur les travaux conduits dans cette thèse et explore trois perspectives pour de futurs travaux : l'amélioration de la méthode CAnDoR à travers différents prismes, la mise au point d'un système de recommandation fonctionnel se basant sur CAnDoR pour proposer à des systèmes existants une améliorations des performances via le placement de leurs données, et l'étude de paradigmes centrés sur la donnée, s'affranchissant des systèmes de gestions de données.

Dans le cadre de cette thèse, nous étudions de nombreuses propriétés et définitions sur différentes notions. Afin d'aider la lecture de ce manuscrit, l'Annexe 9.2.3 regroupe les propriétés, définitions et théorèmes utilisés. Nous les rappelons également lorsque nous les évoquons afin d'éviter de devoir se référer à l'annexe dans le cadre de la lecture. Ce manuscrit a été rédigé de manière à pouvoir lire chaque chapitre de manière indépendante, nous demandons donc aux lecteurs d'accepter les répétitions pouvant apparaître d'un chapitre à l'autre.

Contributions scientifiques de la thèse

Cette thèse apporte différentes contributions aux domaines des systèmes de gestion de données distribués.

Étude des modèles de cohérence

Nous présentons dans les Chapitres 3 et 4 une étude didactique des différents modèles de cohérence et des nuances qui les différencient. Bien qu'il existe de

nombreux travaux proposant différents axes d'études des modèles de cohérence (description formelle [VV16], classification 3D [SSAP16]), ces travaux ne précisent que rarement les enjeux de ces modèles sur les systèmes qui les implémentent. Nous pensons également que notre étude permet une bonne compréhension de ces modèles et permettra aux concepteurs de mieux déterminer le modèle de cohérence adapté à leurs besoins. Nous prévoyons de soumettre cette étude à la publication d'une revue à comité de lecture dans un futur proche.

CAnDoR : Consistency Aware Dynamic Data Replication [MVM19]
CAnDoR, détaillée dans le Chapitre 5, est une méthode de placement des réplicas des données dans un système de gestion de données à modèle de cohérence adaptable. Il existe à ce jour de nombreuses méthodes de placement dans les systèmes de gestion de données distribués, certains comme AutoPlacer [PRRR14] ont une approche similaire à celle de CAnDoR. Cependant, il n'apparaît qu'aucune de ces méthodes n'étudie les problèmes de placement en lien avec les problèmes de cohérence des données, la majorité de ces méthodes ayant été conçues pour des systèmes avec un unique modèle de cohérence. À notre connaissance, les systèmes proposant différents modèles de cohérences utilisent une méthode de placement aléatoire [LM10, Sah20]. Les résultats de CAnDoR ont fait l'objet d'une publication à la conférence internationale à comité de lecture NCA19 [2].

Impact de la gestion des événements passés [MVM20]

Nous proposons ensuite, dans le Chapitre 6, un modèle formel simple afin d'étudier les différentes façons de maintenir un historique regroupant les événements ayant eu lieu dans un système, qu'ils soient locaux ou distants. Surprenamment, nous n'avons trouvé que peu de discussions théoriques sur les méthodes pour mettre en place cet historique. Nous étudions l'impact de différentes méthodes à travers un cas d'utilisation spécifique, celui du placement de réplicas. Nous estimons cependant que ce formalisme facilite l'étude de la meilleure méthode pour maintenir et utiliser l'historique des événements. Les résultats de cette étude ont été soumis à la conférence avec comité de relecture NCA20 [3] mais n'ont malheureusement pas été retenus. Une réécriture de l'étude est en cours pour une publication prochaine.

CandorSim

Ce manuscrit détaille également, dans le Chapitre 7, le fonctionnement de CandorSim, une extension de PeerSim pour simuler un système simplifié de gestion de données à modèle de cohérence adaptable. Ce simulateur, que nous avons développé, nous a permis d'évaluer les différents résultats obtenus au cours de cette thèse.

2. https ://www.ieee-nca.org/2019/
3. https ://www.ieee-nca.org/2020/

Travaux de recherches annexes

Cette section présente les travaux annexes à la thèse. Ces travaux sortent du cadre de la thèse, nous ne les détaillerons pas davantage dans ce manuscrit.

Le détecteur de fautes minimal pour résoudre le problème de l'exclusion mutuelle dans un système dynamique inconnu [MJAS19]

Les prémices de la thèse ici présentée ont permis la finalisation des travaux commencés en amont de la thèse. Ces travaux portent sur l'étude du problème de l'exclusion mutuelle dans un système distribué dynamique et anonyme. Nous exposons un détecteur de fautes suffisant pour résoudre le problème et nous prouvons la minimalité de celui-ci. Ces travaux ont fait l'objet d'une publication dans la conférence internationale à comité de lecture ICDCN19 [4].

Étude d'un protocole de consensus sous le paradigme des données autonomes

Nos travaux actuels s'orientent sur l'étude des données autonomes. Ce nouveau paradigme s'affranchit du système de gestion de données et présente alors de nombreux défis. Nos premiers travaux portent sur la conceptualisation de ce paradigme et sur la résolution du problème du consensus dans un environnement de données autonomes. Une première soumission de ces travaux à la conférence internationale à comité de lecture ICDCN22 [5] n'a pas été retenue et nous travaillons actuellement à la soumission prochaine de ces travaux. Une seconde soumission à SRDS22 [6] est en cours de rédaction.

1.4 Conclusion

Dans cette introduction, nous avons vu que de nombreuses applications profitent du Cloud comptuting et des systèmes de gestion de données dans le Cloud pour pouvoir proposer un service à une échelle mondiale, manipulant parfois des petabytes de données quotidiennement [Fac], tout en proposant une qualité de service (SLA) de plus en plus compétitive.

L'utilisation de systèmes de gestion de données dans le Cloud a cependant entraîné l'apparition, ou l'accentuation, de certaines problématiques, parmi lesquelles la tolérance aux fautes, la gestion de la cohérence et de la disponibilité, les différentes stratégies de réplications et de placements de ces réplicas, le maintient des connaissances dans le système ou la surconsommation d'énergie.

Cette thèse, dans le cadre du projet RainbowFS[Rai], aborde différents aspects de la conception d'un système de gestion de données distribué. Un attention parti-

4. https ://events.csa.iisc.ac.in/icdcn2019/index.htm
5. https ://icdcn2022.iiitd.edu.in/
6. https ://srds-conference.org/

culière est portée sur la compréhension des différents modèles de cohérences et leur impact sur la façon de concevoir les différents systèmes de gestions de données, notamment à travers des résultats fondamentaux tels que les horloges de Lamport [Lam78], le FLP85 [Fis83] et théorème du CAP [Bre00] et ses dérivés.

Sommaire

Chapitre

2

Contexte et état de l'art des systèmes de gestion de données distribués

Nous nous proposons, au cours des travaux de cette thèse, d'étudier les systèmes de gestions de données capables d'adapter le modèle de cohérence pour chaque donnée aux différents besoins d'une application. Afin d'être capable de saisir les problématiques liées à ce type de système, nous avons étudié les fondements théoriques des différents problèmes. Cette étude a été couplée à une analyse plus technique. Cette double approche nous semble nécessaire afin de pouvoir apporter des solutions efficaces aux problèmes rencontrés.

Dans ce chapitre, nous développons certaines des problématiques fondamentales des systèmes de gestion de données distribués, telles que les enjeux des systèmes distribués, le problème d'horloge dans un système distribué ainsi que les théorèmes FLP85 Th. 1 et du CAP Th. 2. Nous présentons ensuite les problématiques abordées au sein de ce document ainsi qu'un état de l'art autour de ces problématiques. Enfin, nous discutons de la simulation dans le cadre de la validation des résultats.

2.1 Forces et défis des systèmes de gestion de données distribués

Contrairement aux cas des systèmes de gestion de données centralisés, les données sont ici distribuées sur plusieurs nœuds. Cette approche est née du besoin de

manipuler d'importantes masses de données et/ou d'utilisateurs. La distribution sur différents nœuds offre de nombreux avantages, parmi lesquels :

1. Partage des ressources : la charge de calcul et de stockage ne sont plus concentrés sur un unique nœud, il est alors possible de manipuler une plus grande masse de données et, si les nœuds peuvent effectuer des calculs, de faire des opérations plus coûteuses sur celles-ci.
2. Répartition de la charge d'utilisateur : À l'image du partage des ressources disponibles, tous les utilisateurs ne vont plus accéder à un unique nœud, mais vont pouvoir se répartir sur les différents nœuds, il est alors naturellement possible de faire face à un plus grand nombre d'utilisateurs simultanés.
3. Rapidité : En répartissant la charge des utilisateurs sur différents nœuds, ceux-ci sont capables de traiter chaque requête plus rapidement. De plus, en plaçant stratégiquement les données, il est possible d'offrir une latence plus adaptée à l'utilisation qu'avec une approche centralisée.
4. Fiabilité : Avec l'augmentation de l'espace de stockage disponible, il est possible de *répliquer* les données. Cette méthode consiste à créer plusieurs copies de chaque donnée pour les placer sur différents nœuds. De ce fait, même si un des nœuds stockant une copie devient non fonctionnel, d'autres nœuds peuvent assurer l'accès aux données. Nous discutons en détail dans le Chapitre 3, des conséquences de la réplication sur la conception des systèmes de gestion de données distribués.

Cependant, la distribution des données entraîne également de nouvelles contraintes et de nouveaux défis. En particulier, nous pouvons noter les défis suivants :

1. Fiabilité et sécurité : En augmentant le nombre de nœuds dans le système, les risques qu'au moins un nœud devienne fautif sont accrus. Il en va de même pour les risques que l'un d'eux soit victime d'une attaque. Une attention particulière doit donc être apportée à la gestion des fautes et aux différents problèmes de sécurité.
2. Maintien et transfert de "méta-informations" : Contrairement à un environnement centralisé, dans un système distribué, il est nécessaire de faire circuler des méta-informations : l'emplacement de certaines données, acheminement des utilisateurs, etc. De plus, de nombreux protocoles doivent être mis en place afin que les algorithmes centralisés puissent s'exécuter (et être efficace) en distribué : tolérance aux fautes, communications, etc.
3. Dépense énergétique : L'utilisation de plusieurs nœuds de stockage entraîne de facto un surcoût énergétique, critère essentiel à notre époque. De nombreux travaux de recherche actuels cherchent différentes méthodes pour ré-

duire ce coût. Nous n'avons malheureusement pas eu l'opportunité de nous intéresser directement à ce critère dans les travaux ici présentés.

2.2 Résultats fondamentaux

2.2.1 La relation "happen-before"

Dans [Lam78], Lamport revisite la relation "happen-before" et formalise un des problèmes fondamentaux de l'informatique distribuée et parallèle : le problème d'horloge.

En effet, afin d'étudier la relation entre deux événements, il est instinctif de les ordonner en observant que l'un est arrivé avant l'autre en se basant sur le moment de l'exécution. Dans un système distribué, ces deux événements peuvent se produire sur deux nœuds différents. Cette approche sous-entend donc l'existence d'horloge globale afin que tous les nœuds observent que les événements se produisent dans le même ordre : si chaque nœud dispose de son horloge locale indépendante des autres horloges, il est facile de construire un scénario où deux événements se produisent dans un ordre différent selon le point de vue de l'observateur. Une hypothétique horloge globale devrait de plus être accessible par tous les participants du système, ce qui est irréaliste. Il est donc nécessaire d'avoir recours à l'utilisation d'horloges logiques lorsque l'on souhaite manipuler des événements arrivant sur différents nœuds.

Dans cette thèse, à moins que le contraire ne soit spécifiquement précisé, nous utilisons des horloges logiques pour parler de l'ordonnancement d'événements. De plus, conformément aux travaux de Lamport, nous considérons que tout ensemble d'événements arrivant sur un même nœud peut être traduit en une séquence d'événements consécutifs. En revanche, comme mis en avant dans [Lam78], les événements ayant lieu sur différents nœuds sont sujets à un ordre partiel (et non total). En effet, il est possible qu'un événement e_1 ayant commencé avant un événement e_2 se finisse après celui-ci. Dans ce cas, il est impossible de déterminer à l'aide d'horloge logique quel événement est "arrivé avant" l'autre. Cette observation est la base de l'ordre causal, défini par Lamport [Lam78] :

Définition 1 Causalité. *Un événement e_1 précède causalement un événement e_2, noté $e_1 \rightsquigarrow e_2$, si une des conditions suivantes est vérifiée :*

1. *e_1 et e_2 se produisent sur le même nœud et e_1 se termine avant le commencement de e_2;*
2. *e_1 est l'envoi d'un message par un nœud et e_2 la réception de ce message par un autre nœud;*
3. *il existe un événement e_3 tel que $e_1 \rightsquigarrow e_3 \rightsquigarrow e_2$.*

L'ordre causal est un ordre partiel sur les événements permettant de mettre en avant les événements pouvant avoir un impact entre eux : si deux événements e, e' ne sont pas causalement liés, c'est qu'il n'existe pas de séquence d'événements contenant ces événements dont au moins deux événements e_i, e_j se soient produits sur le même nœud. En d'autres termes, e et e' sont indépendants et l'existence de l'un n'influe pas sur l'existence du second. Cette relation est primordiale dans un système distribué, les nœuds pouvant régulièrement être amenés à effectuer des actions en concurrences sur des objets indépendants.

2.2.2 L'impossibilité de résoudre le consensus

Probablement un des résultats les plus fameux de l'algorithmique parallèle et distribuée, FLP85 [FLP85] (du nom des auteurs Fischer, Lynch, Paterson) est énoncé de la manière suivante :

Théorème 1 (FLP). *Aucun protocole de consensus ne peut être correct en présence d'une faute dans un système asynchrone.*

La preuve de ce théorème tient dans l'existence de configurations dites bivalentes : il s'agit de configurations dans lesquelles il existe au moins un nœud n dont un événement e peut changer le résultat du protocole de consensus. Les auteurs font alors remarquer que dans un système asynchrone où des fautes peuvent arriver, il est possible de construire une exécution *réalisable* où e n'arrive jamais et que le système ne peut pas prendre de décision.

Ce résultat fondamental a marqué les différents domaines des systèmes distribués, le problème du consensus étant un des problèmes fondamentaux. En effet, l'incapacité de garantir que tous les participants peuvent se mettre d'accord sur une valeur non triviale rend la résolution de nombreux problèmes impossible. De plus, le problème du consensus est parfois utilisé pour ordonner d'autres problèmes, comme avec le *consensus number* [Rup00, GKM+19]. Le *consensus number* d'un type T est le nombre maximum de processus pouvant résoudre le consensus en utilisant des objets atomiques du type T et des registres de lecture-écriture.

Il faut cependant de rappeler les hypothèses nécessaires au théorème du FLP85. La force de ce théorème tient dans le peu d'hypothèses nécessaire, lui permettant de s'appliquer à un grand nombre de systèmes distribués : les processus sont supposés asynchrones communiquant à l'aide de messages. Les canaux de communications sont supposés *fair-lossy* ; c'est-à-dire qu'un message émis finira par être reçu (un processus pouvant envoyer plusieurs fois un message émis) et que l'ordre des messages reçus ne correspond pas nécessairement à l'ordre d'émissions. En revanche, tous les messages envoyés seront reçus tant que ni l'émetteur ni le destinataire ne subissent de fautes.

La définition de processus asynchrone utilisé est la suivante : “Aucune hypothèse n’est faite sur la vitesse relative des processus ni sur les délais de traitement des messages. Les processus n’ont pas accès à une horloge synchronisée. Enfin, les processus ne disposent pas de mécanismes pour détecter la panne d’un autre processus. Il est donc impossible pour un processus de déterminer si un autre processus est en panne ou s’il est simplement lent.” De ce fait, s’il est nécessaire de résoudre le problème du consensus pour une application, il faut affaiblir la condition d’asynchronisme. Certains travaux proposent l’utilisation d’une horloge globale, permettant ainsi l’utilisation de mécanisme de *time-out global* [DS82]. Une autre approche consiste à supposer l’existence d’un mécanisme permettant de détecter la panne d’un nœud, on appelle ce mécanisme un détecteur de fautes [CT96]. Cependant, le fonctionnement d’un détecteur de fautes suppose souvent une forme de synchronie dans le système.

Il existe une relation étroite entre le problème du consensus et le stockage de donnée distribué. Les auteurs de [FLP85] prennent comme exemple d’utilisation du consensus le problème de “validation de transaction”. Ce problème central des systèmes de gestion de données distribués stipule que les *data managers* doivent se mettre d’accord sur le fait d’enregistrer ou rejeter une transaction. Une variante de ce problème permet de valider une transaction sur les différents réplicas d’une donnée. Il apparaît alors impossible de résoudre ce problème dans un système de gestion de données distribué asynchrone. Cette observation est renforcée par la conjecture de Brewer.

2.2.3 La conjecture de Brewer

La conjecture de Brewer [GL02], aussi appelé théorème du CAP, est énoncée de la manière suivante :

Théorème 2 (Théorème du CAP). *Tout système distribué ne peut satisfaire que deux des trois propriétés désirables suivantes :*

- *Cohérence;*
- *Disponibilité;*
- *Tolérance aux partitions.*

On appelle ici *cohérence* la garantie de pouvoir maintenir tous les réplicas d’une donnée dans des états équivalents. Comme nous le préciserons dans le Chapitre 3, il s’agit ici de ce que nous appelons désormais la cohérence *forte*. La *disponibilité* demande à tout nœud recevant une requête doit fournir une réponse, sans imposer de borne sur le temps de cette réponse de la part du nœud. Cette propriété est désormais appelée *grande* disponibilité et est souvent nuancée avec une contrainte de latence. Pour être considéré comme tolérant aux partitions, un système distribué

doit être capable de fournir les mêmes services même dans le cas ou un groupe de nœuds devient incapable de joindre un autre groupe de nœuds.

Ce théorème a eu un impact considérable dans la conception des systèmes de gestion de données distribués. Sa formulation originale, ici présentée, est cependant assez peu précise et est très largement critiquée et revue par la communauté scientifique. Comme nous l'avons rapidement évoqué, les concepts de cohérence et disponibilité ont aujourd'hui été précisés pour témoigner davantage de nuances. De plus, lors du déploiement d'une application à échelle mondiale, relâcher la propriété de tolérance aux partitions peut être très dangereux, les partitions pouvant simplement arriver dans le système. Certaines relectures du théorème du CAP mettent en avant ce phénomène en considérant la résistance aux partitions comme une propriété du système et non pas une propriété à atteindre [MAD+11, Aba12]. La conséquence majeure de ce théorème devient alors la nécessité d'établir un compromis entre cohérence et disponibilité, ces deux propriétés n'étant pas binaires [Bre12]. Nous explorons en détail les nuances de ces propriétés et les conséquences de ce théorème sur les systèmes de gestion de donnée distribués dans les Chapitres 3 et 4.

Nous précisons cependant que pour être capable de résoudre le problème de cohérence, à l'image du problème de validation des transactions, il est nécessaire de pouvoir résoudre le problème du consensus. Il faut alors remarquer le lien entre le théorème du CAP et de FLP85 : dans un système asynchrone où des fautes peuvent se produire, il est impossible pour les participants de maintenir une cohérence stricte entre les réplicas d'une donnée. Cette propriété est évidemment désirable, voire fondamentale, pour de nombreuses applications. Comme nous le précisions précédemment, il est alors nécessaire d'affaiblir certaines hypothèses du système, en posant notamment des points de synchronisation.

2.3 Axes d'études et état de l'art

2.3.1 Systèmes de gestion de données distribués à protocole de cohérence adaptable

Les systèmes de gestion de données distribués et les architectures Clouds ont permis le développement d'applications à une échelle mondiale et manipulant des quantités de données inimaginables auparavant. Certaines de ces applications étant utilisées quotidiennement par un très grand nombre de personnes, il est naturel que ce type de service continue d'évoluer et de s'adapter aux utilisateurs.

Nous nous concentrons, au cours des travaux ici présentés, sur des systèmes cherchant à s'adapter au mieux aux utilisateurs, en particulier, les systèmes qui permettent de proposer des protocoles de cohérence adaptables sur les données

afin d'être capable de s'adapter à chaque cas d'utilisation. Ces systèmes ont pour objectif d'être compatibles avec un grand nombre d'applications en étant capables de gérer différemment les données stockées au sein d'un même système, permettant ainsi pour une application d'imposer différentes contraintes sur les données manipulées. De ce fait, cette thèse s'inscrit dans le cadre du projet ANR RainbowFS [Rai]. L'objectif de ce projet est de proposer une approche de stockage distribué permettant de fournir des propriétés de cohérence sur mesure tout en assurant les propriétés de disponibilité et de scalabilité.

Les méthodes existantes implémentent différents modèles de cohérences prédéfinis, mais comme nous l'avons évoqué, il n'existe pas d'unique modèle adapté à tous les besoins. L'approche de ce projet consiste à proposer des outils pour la conception d'une application et des protocoles de cohérences associés. L'approche vise à réduire les conflits entre les conditions de disponibilité et ceux de cohérence : les opérations basiques sont conçues comme asynchrones et la synchronisation n'est utilisée que lorsque cela est nécessaire pour les besoins de cohérence afin de respecter les invariants de l'application.

Dans ce projet, les modèles classiques de cohérence sont décomposés en un ensemble de primitives orthogonales. Le développeur peut composer efficacement avec ces primitives ainsi qu'avec des outils permettant un déploiement sur le Cloud rapide, efficace et correct. Avec cette méthodologie, l'objectif de RainbowFS est de développer un système de fichiers utilisable par les entreprises et pouvant supporter le passage à l'échelle induit par l'utilisation du Cloud, tout en explorant le spectre des différentes sémantiques. Le projet a donc travaillé sur Antidote [AGS19], un système de gestion de données distribué ayant recours au Conflict-Free Replicated Data Type [SPBZ11] afin d'implémenter un modèle de cohérence adapté [SBP+18].

Afin de comprendre les enjeux liés à l'utilisation des différents modèles de cohérence, nous proposons dans le Chapitre 3 une analyse pédagogique des trois principaux modèles de cohérences, la cohérence à terme, la cohérence causale et la cohérence forte, ainsi que des dérivés directs de ces modèles (cohérence causale+, convergence forte à terme, etc.). Malgré l'existence de plus de 50 modèles de cohérence différents [VV16], la majorité des systèmes existants se basent sur un de ces trois modèles ou un dérivé direct.

2.3.2 Stratégies de placement et réplication des données

Nous avons évoqué le principe de réplication des données, très largement répandu de nos jours : chaque donnée est copiée une ou plusieurs fois et chaque copie est stockée sur un nœud différent [TVS07]. Cette technique s'est révélée très efficace dans les systèmes de gestion de données afin d'assurer la pérennité des données et d'offrir de meilleures performances. Cependant, elle intensifie les problématiques soulevées précédemment (gestion de la temporalité entre les copies, capacité à éta-

blir un consensus, gestion de la disponibilité des données et de leur cohérence, etc.). En particulier, il est impossible de maintenir toutes les copies dans un état équivalent sans ralentir de manière conséquente le système. On appelle les différentes copies d'une même donnée les *réplicas* de cette donnée. Ce point, central à la conception d'un système de gestion de données distribué, est abordé en profondeur dans le Chapitre 4. Il est donc nécessaire d'estimer correctement le nombre de réplicas pour chaque donnée. On estime généralement qu'un nombre de réplicas entre 3 et 5 permet de conserver de bonnes performances tout en offrant des garanties de pérennité des données [GBKA11]. Se pose alors la problématique du placement des réplicas. En effet, à chaque requête de la part d'un utilisateur, le système va devoir acheminer la requête jusqu'à un nœud stockant un réplica de la donnée. Puis, le cas échéant, le résultat du traitement doit à son tour être conduit jusqu'à l'utilisateur. Les performances du système, en particulier le délai global pour obtenir la réponse d'une requête, vont donc être impactées par le placement des réplicas. Nous nous intéressons en particulier à minimiser le délai entre une requête émise par un utilisateur et la réponse de la part du système. Cette métrique est appelée la latence pour les accès utilisateurs. Ce problème peut alors être formulé de la manière suivante :

Problème 1 (Placement des réplicas). *Soit un ensemble D de n_d données répliquées RF fois et soit un ensemble S de n_s nœuds de stockage à capacité finie.*

Comment placer les $\sum_{i=0}^{n_d} RF_i$ réplicas des n_d données sur les n_s nœuds de stockage de façon à minimiser les latences médianes pour les accès utilisateurs?

Un algorithme de placement de données cherche, de plus, à optimiser certaines métriques telles que le délai de traitement d'une requête, minimiser ou maximiser la localité des réplicas selon les cas, etc. Ce problème apparaît alors comme une variante du problème de sacs à dos multiple (KBP) et a été démontré comme étant NP-difficile [ST01, GKK+09]. De plus, afin de minimiser le délai de traitement d'une requête, il est souhaitable de se baser sur l'activité des utilisateurs afin de placer les données près des utilisateurs qu'elles intéressent. Certaines applications partent du postulat que les utilisateurs sont uniformément répartis entre les différents nœuds de stockage, rendant l'impact du workload relativement faible. Cette hypothèse est réaliste dans le cas d'applications proposant un service à une échelle mondiale telle que Facebook ou Amazon.

Afin d'éviter une surcharge de calcul, de nombreux systèmes, tels que Cassandra [LM10] ou Dynamo [DHJ+07] ont donc opté pour un placement aléatoire des réplicas. Cette approche ne cherche pas à optimiser le placement des réplicas, mais se révèle efficace si les utilisateurs sont uniformément répartis, tout en entraînant un coût moindre pour déterminer le placement des réplicas.

De nombreux travaux de recherches s'intéressent cependant à proposer une

méthode pour placer efficacement les données de manière automatique, malgré la NP-difficulté du problème. Certains travaux utilisent une variante du Problème d'Allocation des Fichiers (FAP) pour résoudre le problème du placement des réplicas [CH76, LWY93, ZG11]. Les nœuds centralisent les informations sur les requêtes afin de permettre l'utilisation d'un algorithme efficace pour le calcul du placement des réplicas. Afin de pouvoir rester efficaces sur des systèmes à large échelle, certains travaux utilisent un algorithme d'approximation pour calculer ce placement [CMM+13, JBP+13, GM02, BR01].

Cependant, ces méthodes sont centralisées par nature et cherchent à calculer un placement optimal (ou une approximation de ce placement) en considérant les réplicas de l'intégralité des données. Il est possible de décentraliser ce problème par l'observation que certaines données sont moins utilisées que d'autres, et qu'il est donc moins efficace de dépenser du temps de calcul pour déterminer leur emplacement. C'est par exemple le choix fait lors du développement d'AutoPlacer [PRRR14].

Afin de déterminer un placement efficace, AutoPlacer recherche périodiquement les k données accédées le plus fréquemment. Une fois ces k données repérées, les nœuds échangent leurs statistiques afin de déterminer un placement efficace pour ces données. Cette approche se concentre sur les k données les plus accédées. Une étude théorique a été conduite afin de déterminer la bonne valeur de k. Réduire le problème aux k données les plus accédées permet de réduire grandement la complexité de celui-ci, permettant ainsi d'utiliser cette approche sur des systèmes à large échelle.

Le calcul du placement des $r \times k$ réplicas se fait en utilisant la Programmation Linéaire Entière (ILP), en fonction du nombre de lectures, d'écritures ainsi que du coût, local et distant, de ces opérations. L'utilisation de la Programmation Linéaire Entière a été répandue par Dowdy et al. [DF82], puis utilisée dans de nombreux travaux s'intéressant au placement des données [KRS00, LWY93, YHJ13].

Ursa [YHJ13] utilise une approche similaire en essayant de réduire le coût d'une reconfiguration. Afin d'optimiser les dépenses énergétiques, leur fonction de placement essaye cependant de vider certains points de stockage afin de pouvoir les éteindre. Les auteurs garantissent que ce mécanisme ne met pas en danger les propriétés de cohérence ou de disponibilité sans s'étendre longuement sur ce point.

SWORD [QKD13] est un autre système de placement automatique des données. L'objectif de SWORD est de minimiser le délai de traitement d'une requête. Les auteurs ont choisi de permettre au système de créer de nouveau réplica pour certaines données, un système de gestion de données distribué ayant régulièrement plus d'espace de stockage que nécessaire. Cette approche ne convient donc pas à tous les systèmes, la place pouvant être un critère critique. De plus, la création de nouveaux réplicas entraîne un surcoût du maintien de la cohérence et n'est donc

pas adaptée aux systèmes voulant implémenter des protocoles de cohérence forte. Cette approche permet cependant de diminuer le temps de traitement d'une requête tout en augmentant la tolérance aux fautes, en particulier aux partitions. Afin de contre-balancer le surcoût de maintien de la cohérence des données, les auteurs se basent sur un quorum défini selon les activités des nœuds de stockage.

Le Chapitre 5 présente CAnDoR, une méthode de placement des réplicas décentralisée, permettant de déterminer automatiquement un placement efficace pour les réplicas tout en prenant en compte le workload et les contraintes des protocoles de cohérence sur le coût des différentes opérations. Ce travail a été réalisé afin de subvenir aux besoins des applications à protocoles de cohérence adaptables, un coût unique ne pouvant pas être appliqué pour toutes les données et les stratégies de placement pouvant différer selon le protocole de cohérence souhaité.

2.3.3 Maintenir une vision adéquate du système

Afin de pouvoir prendre en compte le workload dans le calcul du placement des réplicas, la façon de maintenir l'historique des événements d'un système pouvait avoir un impact conséquent sur les résultats. En effet, la façon de prendre en compte les événements, en fonction de leur âge ou non, peut entraîner de grandes différences de résultats. Par exemple, si une donnée est très accédée sur une courte période avant de ne plus être accédée, comment faut-il considérer cette période d'activité ? Combien de temps celle-ci doit-elle avoir un impact sur les calculs ?

Or dans un système distribué, un nœud ne traite qu'une partie des événements. Il est alors nécessaire pour établir un workload pertinent que les différents nœuds échangent leurs connaissances, afin que chacun ait une connaissance un peu plus précise du système. Plus généralement, le problème de collecte des connaissances n'est pas trivial dans un système distribué, a fortiori à large échelle. Cependant, les systèmes se doivent de faire face à une constante évolution de leur environnement, la charge de travail, la localité des événements, etc. Pour cela, de nombreuses applications utilisent différentes méthodes pour s'adapter à ces évolutions. Dans [GCG01] Gupta et al. proposent l'utilisation de détecteurs de fautes scalables pour déterminer la configuration actuelle du système, les différents systèmes de placements de données s'adapte notamment selon le comportement des utilisateurs [QKD13, YHJ13, PRRR14].

Ces travaux expriment rarement la façon dont la liste des événements est maintenue par l'application. Dans [PRRR14], Paiva et al. utilisent l'algorithme d'analyse de flux proposé par Metwally et al. [MAA05]. Cet algorithme permet d'établir avec une bonne probabilité la liste des k éléments les plus accédés dans un flux de données. Cette approche, très répandue, s'intéresse donc aux événements les plus récents et s'approche donc du principe de fenêtre glissante : seuls les événements ayant eu lieu dans la dernière fenêtre temporelle sont pris en compte par

l'application.

Une autre approche commune est celle d'un maintien sans failles de l'historique : les différents nœuds gardent une trace de tous les événements dont ils ont connaissance. Cette approche permet de construire des traces d'exécutions robustes à travers le système, mais peut s'avérer très coûteuse en communication et emplacement mémoire.

Bien que cette discussion sur les différentes méthodes semble fondamentale pour de nombreuses applications, nous n'avons pas trouvé d'étude théorique sur les différentes façons de maintenir l'historique des événements. En effet, la plupart des applications semblent utiliser des approches classiques comme l'analyse de flux ou la sauvegarde complète des événements. Dans [Lam78], Lamport présente une réflexion sur les relations entre les événements et Halpern et al. expriment un formalisme pour la représentation de la connaissance et des actions dans un système distribué, mais n'étudie pas l'impact de la date d'une action sur les décisions actuelles [HF89]. Nous proposons donc dans le Chapitre 6 une étude s'intéressant à ce problème. Nous définissons un formalisme simple pour décrire l'historique des événements d'un système et nous étudions, à l'aide de ce formalisme et d'un simulateur, l'impact de la façon de traiter les événements en fonction de leur "âge" sur les décisions d'un système.

2.4 Test et validation dans un monde distribué

La validation

Afin de valider des travaux de recherches, il est important d'effectuer différentes mesures de performances sur des métriques pertinentes. Dans de nombreux cas, une preuve formelle n'est pas suffisante, l'exhaustivité de celle-ci pouvant être difficile à démontrer. Il est alors courant d'avoir recours à des expérimentations afin de valider les résultats. Ces résultats se doivent d'être reproductibles, c'est-à-dire que toute équipe de recherches souhaitant tester l'approche doit pouvoir la mettre en place et obtenir des résultats similaires. Il est donc nécessaire de détailler les différents protocoles d'expérimentations et de fournir le code source des projets lorsque cela est possible.

Il existe deux principales familles d'approches possibles pour conduire des expérimentations, avoir recours à un simulateur ou déployer un prototype, permettant de comparer les différentes approches dans des cas réels d'utilisations.

Cependant, un déploiement peut s'avérer extrêmement complexe et coûteux : dans les cadres d'applications distribuées à large échelle, il faut mobiliser un grand nombre de points de calcul sur des durées pouvant s'étaler dans le temps selon le type de problème considéré.

Dans le cas de bases de données distribuées, il faut que le prototype s'exécute suffisamment longtemps pour constater la sauvegarde des garanties d'accès et d'états des données. Cette opération est donc coûteuse et il est alors d'usage de ne passer en déploiement que lorsque le prototype a déjà fait ses preuves. De plus, comme précisé dans [CRDRB09], les tests sur des bancs d'essai sont limités par la taille de ceux-ci et ont des résultats difficilement reproductibles : de nombreux paramètres et conditions de l'environnement d'internet ne sont pas contrôlables.

Il est alors souhaitable de valider le fonctionnement d'un outil sous certaines hypothèses contrôlables et reproductibles avant de déployer un prototype. Les simulateurs apparaissent ici comme des outils de choix : il est possible de contrôler l'environnement et de nombreuses hypothèses afin de valider un prototype avant de le déployer.

Les familles de simulateurs

Il existe plusieurs grands types de simulations, mais les plus répandues sont la simulation à événements discrets et la simulation continue. Le simulateur met en place différents objets et évalue leur état à certains moments. Nous parlons ici d'objets au sens large : il peut s'agir d'un noeud de calcul, d'une variable, etc.

Lors d'une simulation continue, le temps est découpé en cycles de durée prédéfinie. Lors de chaque cycle, le simulateur évalue l'état du système dans son entièreté. Ainsi les objets simulés évoluent constamment et de manière pseudo-continue. Cette forme de simulation offre l'avantage de s'exécuter sur un temps régulier et maîtrisable, la durée de chaque cycle pouvant être fixée par l'utilisateur. De plus, il est possible d'évaluer les différents éléments à n'importe quel moment, ceux-ci étant réévalués à chaque cycle.

Dans un simulateur à événements discrets, le temps est une notion discrète et est découpé en pas de temps. Les événements sont ordonnancés selon ces pas de temps. On appelle événement une action qui modifie le système. Il peut s'agir de la création d'un objet, d'un message envoyé ou reçu, d'un calcul à effectuer, etc. Le simulateur n'évalue alors que les objets affectés par des événements et uniquement lorsque ceux-ci se "produisent". Ce type de simulation permet donc d'éviter de nombreux calculs lorsque seulement une partie du système est active sur certains instants. Il est cependant nécessaire de stocker les événements dans une file dédiée et de précalculer l'instant ou chaque événement va se produire. Selon le type de simulation, il peut être nécessaire de réévaluer ces temps.

Il faut donc déterminer quel type de simulateur serait le plus adapté pour chaque prototype. Au cours de cette thèse, nous étudions des méthodes de stockage et placements des données dans un système de gestion de données distribué. Ces méthodes ne sont appelées que de manière ponctuelle et vont donc produire des pics d'activités. Nous souhaitons également simuler des utilisateurs effectuant des

requêtes sur ces données. Ici encore, ces actions sont ponctuelles. De plus, le pas de temps souhaitable n'est pas le même pour les utilisateurs et les données. En effet, nous pouvons estimer que les points de stockages doivent évaluer régulièrement si de nouvelles requêtes arrivent. De leur côté, les utilisateurs pourraient avoir une granularité plus large, n'effectuant des requêtes que de temps en temps, à un rythme plus ou moins soutenu, mais peu comparable avec un cycle de vérification des points de stockage. Pour ces raisons, nous avons choisi de nous tourner vers des simulateurs à événements discrets. Ceux-ci se prêtent en effet bien à notre problème : chaque requête est un événement et il est possible de prévoir l'arrivée de celle-ci en maîtrisant les paramètres du réseau. Ce fonctionnement est détaillé dans le Chapitre 7.

Dans [NBL+06], Naicken et al. précisent qu'il existe de nombreux simulateurs ayant fait leurs preuves, mais qu'une grande partie de la communauté scientifique continue de proposer des simulateurs personnels, n'étant jamais réutilisés par la suite ou ne précisant pas le simulateur utilisé. Malheureusement, ce manque de rigueur peut avoir un impact sur la précision des résultats fournis, le réalisme et les performances d'une exécution peuvent varier grandement selon le simulateur utilisé.

Nous avons donc étudié différents simulateurs afin de choisir le plus adapté à notre travail : CloudSim [CRDRB09], iCanCloud [NVPC+12], JBotSim [CL10], OMNet++ [Var01], PeerSim [MJ09], SimGrid [CGL+14]. Notre choix s'est basé sur les publications des simulateurs et sur différentes études de comparaisons des simulateurs présentés dans [AS14], [TXC+15] et [CGL+14]. Dans [AS14], les auteurs étudient 12 simulateurs de Clouds à travers différents critères tels que la plate-forme utilisée, le langage de programmation ou certains paramètres des simulateurs comme le modèle de communication ou l'ordre de grandeur du temps de simulation. L'étude de [TXC+15] se concentre sur 4 simulateurs mais compare ceux-ci plus en profondeur, en détaillant leur fonctionnement et en étudiant les performances de ceux-ci. Enfin, avant de présenter SimGrid, [CGL+14] propose une comparaison de 15 simulateurs de différents domaines tels que le calcul haute performance, le Cloud computing, le Grid computing, le volunteer computing et le peer-to-peer computing. Cette comparaison se base sur différents aspects du réseau, du CPU, du disque et de l'application.

Nos travaux portent sur la gestion des données dans des systèmes de gestion de données à cohérence variable. Il est donc nécessaire de choisir un simulateur permettant une grande scalabilité. Les travaux préliminaires alors effectués nous ont orientés vers un simulateur capable de modéliser différents types de systèmes, du modèle Centre de données aux modèles *fog* avec des données stockées directement chez certains utilisateurs. Ayant effectué des travaux précédant sur PeerSim [MJ09], nous avons choisi de reprendre ce simulateur. En effet, PeerSim semble

adapté à nos travaux, nous permettant de concevoir efficacement nos différents types de nœuds et systèmes, tout en assurant de fonctionner à différentes échelles.

Avec l'avancée de nos travaux, nous nous sommes davantage orientés vers un modèle de centre de données, que nous avons facilement pu modéliser avec notre simulateur. Cependant, il s'est avéré que la non-prise en compte native de la bande passante de PeerSim s'est révélée dommageable pour nos expériences. Nous n'avons malheureusement pas pu convertir notre travail sur un autre simulateur, cette observation étant arrivée tardivement dans nos études. Le chapitre 7 détaille le fonctionnement de CandorSim et les différents choix d'implémentations de celle-ci.

2.5 Conclusion

Le développement des systèmes de gestion de données distribués à large échelle, et dans le cloud, ont permis l'essor de nombreuses applications à échelle mondiale de manipuler des masses de données sans précédent. Dans ce chapitre, nous avons rappelé les principales problématiques des systèmes distribués à large échelle. Nous rappelons ici trois problèmes fondamentaux : la gestion d'horloge [Lam78], le FLP85 exprimant l'impossibilité d'établir un consensus dans un système asynchrone sensible aux fautes [FLP85] et la conjecture de Brewer, précisant l'impossibilité pour un système distribué de satisfaire à la fois cohérence (forte), (grande) disponibilité et résistance aux partitions [GL02]. Ces trois résultats sont au coeur de nombreuses réflexions des systèmes de gestion de données et permettent d'aborder des problématiques tels que : comment résoudre le problème du consensus afin d'être capable de garantir une cohérence forte dans un système ? Quels modèles de cohérences représentent une alternative acceptable dans le cas où être grandement disponible est plus important que de fournir une cohérence forte ? Comment maintenir un ordre causal afin de pouvoir fournir un protocole de cohérence causale ?

Ce chapitre décrit également les problématiques abordées dans nos travaux : comment maintenir efficacement un historique des événements dans un système à large échelle, quel poids donner aux différentes actions en fonction de leur âge et comment utiliser ces connaissances pour déterminer automatiquement un placement efficace des donnes dans un système a protocole de cohérences adaptable ? Enfin, ce chapitre motive l'intérêt d'avoir recours à un simulateur lors de travaux de recherches et présente les simulateurs les plus couramment utilisés.

Sommaire

Chapitre 3

La cohérence dans les systèmes de gestion de données distribués

3.1 La cohérence dans un système de données distribué

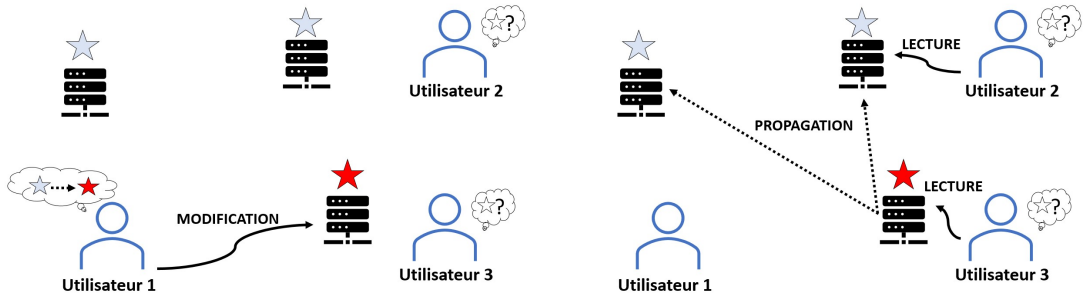
3.1.1 Être ou ne pas être cohérent ?

Au cours de ces dernières années, les évolutions techniques ont permis l'essor des systèmes de gestions de données dans le Cloud. De nombreuses applications profitent de ce type de système afin de proposer des services à large échelle, parfois mondiale, tout en manipulant une masse conséquente de données. Afin de faire face à des charges utilisateurs parfois très importantes et/ou très localisées, les systèmes de gestion de données distribués ont recours à la *réplication* des données. Cette approche consiste à dupliquer une donnée en différents *réplicas*, stockés sur des nœuds physiques différents. Instinctivement, on peut penser que cette méthode apporte 3 grands avantages :

- rapidité de traitement : en ayant accès à différents exemplaires d'une donnée, il est d'autant plus probable qu'un réplica de celle-ci soit proche d'un utilisateur émettant une requête. Cette requête sera alors reçue plus rapidement par un nœud avec un réplica et donc prise en charge par le système ;

- charge d'utilisation : l'existence de différents réplicas permet également de répartir la charge des requêtes entre les différents réplicas et ainsi faire face à une activité plus importante de la part des utilisateurs ;
- robustesse : la distribution des réplicas permet également de minimiser le risque qu'une donnée soit perdue à cause d'une panne, d'une erreur matérielle ou logicielle.

La réplication des données entraîne cependant de nouvelles problématiques. Par exemple, avec un système centralisé, toutes les requêtes des utilisateurs peuvent être traitées séquentiellement, l'état de la donnée après ces traitements est généralement déterministe et prévisible. En revanche, avec l'utilisation de plusieurs réplicas, il est possible (et courant) que les requêtes soient traitées de manière concurrente, chaque réplica pouvant traiter les requêtes en parallèle. Si aucun mécanisme de protection n'est mis en place, il est alors possible de construire des scénarios où les différents réplicas finissent dans des états non équivalents, même dans le cas où toutes les requêtes de modifications sont propagées dans le système. Dans de telles situations, on dit que les réplicas sont *incohérents* entre eux. Le problème de la cohérence entre les réplicas est un problème fondamental des systèmes de gestion de données distribués. Cette situation est illustrée dans les Figures 3.1, 3.2 et 3.3. Dans la Figure 3.1a, l'utilisateur 1 effectue une modification sur un réplica (en changeant la couleur de la donnée). La Figure 3.1b illustre les demandes en lectures de la donnée par les utilisateurs 2 et 3 pendant que le réplica propage la mise à jour.



(a) Modification d'un réplica (b) Lecture des réplicas

FIGURE 3.1 – Exemple simple des problèmes de cohérence

Si les nœuds de stockage délivrent la donnée directement, alors les utilisateurs 2 et 3 auront des résultats différents en consultant différents nœuds de stockage. C'est ce qu'on appelle l'incohérence entre réplicas et cette situation est illustrée dans la Figure 3.2.

À l'inverse, si les nœuds de stockage se synchronisent avant de répondre aux utilisateurs, ils seront capables de fournir la même réponse aux utilisateurs 2 et 3.

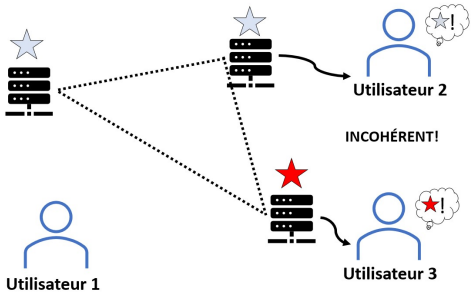
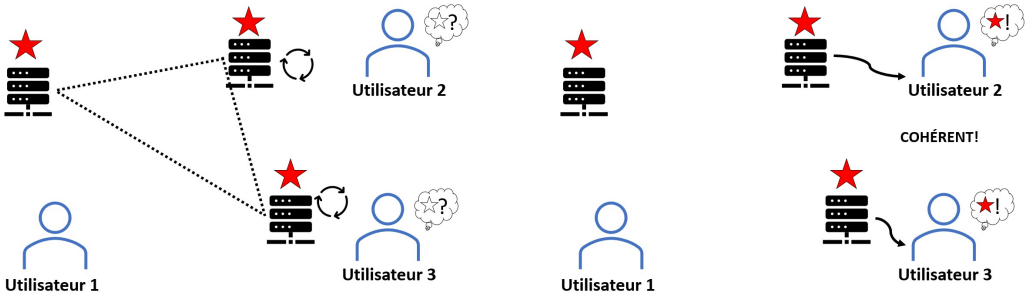


FIGURE 3.2 – Réplicas incohérents

On dit alors que les réplicas sont cohérents (entre eux). La Figure 3.3 illustre ce scénario.



(a) Mise à jour des réplicas (b) Résultats des lectures

FIGURE 3.3 – Réplicas cohérents

3.1.2 50 nuances de cohérence

Le théorème du CAP Th. 2 exprime l'impossibilité pour un système distribué de fournir à la fois cohérence forte, grande disponibilité tout en étant tolérant aux partitions [Bre00]. Bien que la cohérence forte soit un modèle de cohérence instinctif et désirable, de nombreuses applications estiment qu'il est plus important de pouvoir fournir un service fluide et le plus rapide possible [DHJ+07]. Certaines applications, comme Facebook, Amazon, etc., ne manipulent pas de données dont la cohérence entre les réplicas est primordiale. Dans le cas de Facebook, si plusieurs utilisateurs accèdent aux contenus d'un même utilisateur, il est possible que ce contenu s'affiche dans un ordre légèrement différent. Ce comportement n'est pas jugé comme indésirable par l'application, qui peut alors favoriser la vitesse d'exécution à la cohérence de ses données. Amazon estime qu'une augmentation de $100ms$ du délai de traitement entraîne une perte de 1% des ventes [Gre].

La propriété de cohérence n'est cependant pas une propriété binaire et il existe de nombreux modèles différents entre les deux extrêmes de la cohérence forte et la cohérence faible. De nombreux travaux se sont intéressés aux différents modèles de cohérence [VV16, SSAP16, AG96, SN04]. Plus de 50 modèles de cohérence ont été regroupés et classifiés par Viotti et al. [VV16]. Parmi les modèles les plus couramment utilisés par les systèmes existants, nous pouvons noter les modèles de cohérence forte, de cohérence causale et à terme. D'autres études ont recherché quels modèles de cohérence seraient les plus efficaces pour un système souhaitant être toujours capable de traiter les requêtes des utilisateurs, même en cas de partitions [MSL+11]. Selon ces études, les modèles de cohérence causale sont les plus efficaces pour ce type de système.

Dans ce chapitre, nous explorons les différents modèles de cohérence existants. Nous nous intéressons en particulier aux modèles de cohérence forte, causale et à terme, et à leurs dérivés. Ces modèles semblent être les plus répandus dans les systèmes de gestions de données existants, comme l'illustre l'étude comparative, présentée dans le Chapitre 4. Nous proposons notamment un système de notation permettant de comprendre les propriétés de ces modèles et de mettre en avant leurs nuances à l'aide d'un exemple d'exécution simple. Il est important de noter que nous étudions ici les modèles de cohérence et donc les propriétés que nous pouvons en extraire. Nous ne considérons pas ici les *protocoles* de cohérence, qui sont les implémentations de ces modèles.

En étudiant ces différents modèles, il est possible d'essayer de les classifier. Cette démarche n'est cependant pas triviale. Viotti et al. exposent même dans [VV16] que de nombreux modèles sont incomparables entre eux. Nous étudions cette question dans la Section 3.3.

3.2 Étude de différents modèles de cohérence à travers un cas d'exécution

Il existe de nombreuses façons de définir et présenter les différents modèles de cohérence. Comme avancé par Mahajan et al. dans [MAD+11], la plupart des modèles de cohérence peuvent être vus comme un ensemble de restrictions sur l'ordre de traitement des requêtes par les nœuds du système. Nous utilisons cette approche lorsqu'elle est adaptée. Ce n'est cependant pas le cas pour certains modèles de cohérence, en particulier pour la famille des modèles de cohérence à terme. Les modèles de cohérence de cette famille imposent des contraintes sur l'état des réplicas plutôt que sur l'ordre des requêtes. Ces modèles sont détaillés dans la Section 3.2.6.

Intuitivement, une donnée est dite cohérente si, lorsqu'un utilisateur émet une

requête de lecture, tous les réplicas de la donnée sont dans un état équivalent. Nous parlons ici d'états *équivalents* plutôt qu'*égaux* car deux états peuvent légèrement diverger sans que cela ne soit gênant pour l'application. Par exemple, dans le cas de requêtes d'ajouts ou de suppressions d'éléments dans un ensemble, les ensembles $\{a, b, c\}$ et $\{b, c, a\}$ peuvent être considéré équivalent sans être strictement égaux. Une conséquence immédiate de la propriété de cohérence est que si l'un des réplicas est modifié sur un des nœuds, alors tous les réplicas devront finir par prendre en compte cette modification. La modification en question doit alors être propagée à travers le système. Il faut comprendre qu'un nœud peut recevoir (et propager) une opération sans forcément l'appliquer directement. Quand nous précisons ici qu'un nœud reçoit une opération, cela signifie donc juste que ce nœud a connaissance de cette opération. Les moments où cette opération peut être appliquée dépendent du modèle de cohérence.

Il ne faut pas confondre les propriétés de cohérence, qui offrent des garanties sur l'état des réplicas les unes par rapport aux autres avec les propriétés de fraîcheur, qui imposent un nombre maximal de requêtes "de retard". Bien que ces deux types de propriétés soient souvent liés, il est possible d'avoir des réplicas cohérentes entre eux, mais ne considérant pas les dernières mises à jour (par exemple, le cas trivial de système n'appliquant pas les requêtes de modifications) et inversement, tous les réplicas peuvent avoir appliqué toutes les opérations, mais se retrouver dans des états incohérents (nous détaillons cette situation dans la section 3.2.6). Cependant, dans de nombreux cas, lorsqu'on parle de cohérence forte, les données doivent respecter à la fois les propriétés de cohérence et de fraîcheur.

3.2.1 Notations utilisées

Tout le long de ce chapitre, nous utilisons un exemple d'exécution simplifié afin de mettre en avant certaines nuances entre les différents modèles. Nous décrivons dans cette section les notations utilisées dans ce chapitre.

Une application $\mathcal{A}$ est utilisée par n utilisateurs $u_1, \ldots, u_n$. Ces utilisateurs peuvent accéder en lecture et écriture à m données $d_1, \ldots, d_m$. Cependant, comme nous nous intéressons aux systèmes non transactionnels, les données sont indépendantes les unes des autres. Nous pouvons alors simplifier notre système de notation au cas où les utilisateurs n'accèdent qu'à une seule donnée, notée d. Cette donnée est répliquée rf fois sur différents nœuds, identifiés par un identifiant $s_1, \ldots, s_{rf}$. Nous nous référons au réplica détenu par le nœud s_j par d^j. Cette simplification a pour unique but d'alléger les explications et l'exemple d'exécution. Le discours de ce chapitre peut naturellement être généralisé au cas avec m données dans le système.

Pour accéder à une donnée, un utilisateur u_i envoie une requête sur la donnée d à l'application $\mathcal{A}$ (i.e. à un des nœuds du système). Cette requête est alors transmise

à s_j, un des nœuds stockant un réplica de d. Ce nœud va (finir par) envoyer à u_i son réplica d^j. Nous disons alors que u_i fait une requête sur d et que cette requête est traitée par s_j. Nous pouvons noter que dans la plupart des situations, u_i ne sait pas quel nœud va traiter sa requête et qu'une telle connaissance n'est généralement pas requise ni même souhaitable.

Lorsqu'un utilisateur veut effectuer une opération α sur la donnée d, il émet donc la requête $r_\alpha(d)$. Cette requête sera traitée par s_j et sera alors notée $r_\alpha(d^j)$. Une fois traitée, cette requête est propagée aux autres réplicas de d. Nous rappelons que cette opération n'est pas forcément appliquée directement par s_j (i.e. $r_\alpha(d)$ est connu de s_j mais l'état de d^j ne reflète pas cette opération à ce stade). Une opération $r_\alpha(d)$ est appliquée une fois que *suffisamment* de nœuds ont pris connaissance de cette opération. On dit alors que $r_\alpha(d)$ est livrée à l'application et le client peut avoir une réponse (un *ack* ou la valeur de d selon l'opération). Le nombre de nœuds devant avoir connaissance d'une opération pour l'appliquer varie d'un modèle de cohérence à un autre, pouvant aller de 1 dans le cas des modèles de cohérence à terme et jusqu'à rf pour les modèles de cohérence forte.

Soit une séquence successive de requêtes $r_{\alpha_1}, r_{\alpha_2}, \ldots$, l'ordre dans lequel les opérations correspondantes sont appliquées par le nœud s_i est noté $O^i : \langle \alpha_1 \prec \alpha_2 \prec \ldots \rangle$. En toute généralité, ces requêtes ne sont pas commutatives et sont donc sensibles aux conflits. En d'autres termes, soient d^1, d^2 deux réplicas de d détenus par deux nœuds s_1, s_2, tel que d^1, d^2 soient dans un état équivalent, alors noté $d^1 \Leftrightarrow d^2$. Soient deux requêtes $r_{\alpha_1}(d), r_{\alpha_2}(d)$, si s_1 et s_2 appliquent α_1, α_2 dans un ordre différent, alors, à l'issue de ces opérations, il est probable que d^1, d^2 soient dans un état non équivalent.

3.2.2 Exemple avec un scénario d'exécution simplifié

Pour illustrer les discussions tenues dans ce chapitre, nous proposons l'exemple simplifié d'exécution suivant : trois utilisateurs u_1, u_2, u_3 font des requêtes sur une donnée d répliquée deux fois. Nous considérons qu'au début de l'exécution aucune requête sur d n'a été faite au préalable, soit que $d^1 \Leftrightarrow d^2$ et $O^1 = O^2 = \emptyset$. Les utilisateurs u_1, u_2, u_3 ont le comportement suivant :

- u_1 commence par vouloir appliquer α_1 sur d et émet alors la requête $r_{\alpha_1}(d)$. Plus tard dans l'exécution il fera de même pour l'opération α_2 et émettra la requête $r_{\alpha_2}(d)$;
- u_2 veut effectuer l'opération β_1 sur d et émet alors $r_{\beta_1}(d)$;
- u_2 veut effectuer l'opération γ_1 sur d et émet alors $r_{\gamma_1}(d)$.

Ces requêtes sont émises de manière indépendante et parallèle (à l'exception des deux requêtes de u_1). Nous ajoutons les contraintes suivantes à l'exécution :

- La première requête de u_1 et la requête de u_2 doivent être traitées par le même nœud. De plus, β_1 doit être traité avant α_1 par ce nœud. En d'autres termes, le résultat de $r_{\alpha_1}(d)$ peut dépendre du résultat de $r_{\beta_1}(d)$. On dit alors que α_1 dépend causalement de β_1 et cette relation est développée dans la section 3.2.5.
- Selon une horloge globale arbitraire, u_3 émet $r_{\gamma_1}(d)$ *avant* que u_1 n'émette $r_{\alpha_1}(d)$. Cela ne signifie pas que $r_{\gamma_1}(d)$ sera reçu avant $r_{\alpha_1}(d)$ par tous les nœuds.
- Aucun réplica, utilisateur ou lien de communication n'est fautif durant cette exécution.

Nous pouvons illustrer ce scénario par la Figure 3.4.

u_1	$r_{\alpha_1}(d)$	$r_{\alpha_2}(d)$
u_2	$r_{\beta_1}(d)$	
u_3	$r_{\gamma_1}(d)$	

FIGURE 3.4 – Exemple d'exécution du scénario simplifié, du point de vue des utilisateurs

Du point de vue des nœuds s_1, s_2, le scénario est le suivant :

- s_1 reçoit et traite les requêtes $r_{\beta_1}(d), r_{\alpha_1}(d)$ dans cet ordre ;
- s_2 reçoit et traite les requêtes $r_{\gamma_1}(d), r_{\alpha_2}(d)$ dans cet ordre ;

Sauf mention contraire lors des exemples, ces actions sont effectuées de manière indépendante. Une fois qu'une opération a été traitée par un des nœuds, ce dernier envoie le détail de l'opération à l'autre réplica. Nous ajoutons une fois encore une contrainte supplémentaire pour l'exécution de ce scénario :

- selon une horloge globale arbitraire, $r_{\gamma_1}(d^2)$ est traitée *avant* la réception de $r_{\alpha_1}(d)$ par s_1.

Nous illustrons l'exécution d'un point de vue des nœuds par le Figure 3.5.

s_1	$r_{\beta_1}(d^1)$	$r_{\alpha_1}(d^1)$
s_2	$r_{\gamma_1}(d^2)$	$r_{\alpha_2}(d^2)$

FIGURE 3.5 – Exemple d'exécution du scénario simplifié, du point de vue des réplicas

Nous rappelons que nous décrivons ici un exemple particulier et bien que nous précisions l'ordre dans lequel les nœuds reçoivent les opérations, nous ne posons, à ce stade, aucune hypothèse sur la façon dont les requêtes sont appliquées ou délivrées à l'application. Ces paramètres dépendent de la mise en œuvre de $\mathcal{A}$ et du modèle de cohérence choisi, qui est le point de discussion de ce chapitre.

3.2.3 Quelques propriétés

Avant de présenter les différents modèles de cohérence, nous commençons par définir les propriétés qui seront les plus utiles pour la compréhension des spécificités de chaque modèle. Ces définitions sont fortement inspirées du travail de Viotti et al. dans [VV16]. Dans leur étude, les auteurs fournissent une définition formelle de nombreux modèles de cohérence et leurs propriétés. Nous nous concentrons ici sur les définitions essentielles à la compréhension des concepts et nous proposons donc une définition informelle de ces propriétés.

Propriété 1 (Ordre global). *Chaque requête est livrée à l'application selon un ordre unique et global.*

Propriété 2 (Ordre causal). *Chaque requête est livrée à l'application selon un ordre causal, établi d'après la relation de causalité Def. 1.*

Propriété 3 (Temps réel). *Si un nœud s_1 a fini de traiter une requête r_α avant qu'un autre nœud s_2 n'ait commencé à traiter une autre requête r_β, alors tous les nœuds doivent appliquer r_α avant r_β.*

Propriété 4 (PRAM (ou FIFO)). *Si un nœud s_1 a fini de traiter une requête r_α avant de commencer à traiter une autre requête r_β, alors tous les nœuds doivent appliquer r_α avant r_β.*

Propriété 5 (Convergence (à terme)). *Tous les nœuds corrects ayant appliqué les mêmes opérations de mise à jour de la donnée finiront par avoir un état équivalent.*

Propriété 6 (Convergence forte). *Tous les réplicas corrects ayant appliqué les mêmes opérations de mise à jour de la donnée ont un état équivalent.*

Propriété 7 (Réception à terme). *Toutes les opérations émises dans le système finissent par être appliquées sur tous les réplicas des données concernées.*

Nous pouvons noter que certaines de ces propriétés se ressemblent et nous apportons donc quelques précisions. La notion de causalité, approfondie dans la section 3.2.5, permet aux requêtes indépendantes d'être traitées dans des ordres différents sur les réplicas.

Les propriétés PRAM 4 et de temps réel 3 imposent toutes deux que le traitement d'une requête soit fini avant le début du traitement d'une seconde requête. À la différence de la propriété de temps réel, PRAM ne s'intéresse qu'aux requêtes traitées par un même réplica. Dans la pratique, une contrainte de temps réel demande d'avoir des connaissances sur les requêtes traitées par les différents nœuds ainsi qu'un mécanisme d'horloge partagée. Enfin, les propriétés de convergence 5

et 6 imposent toutes deux que les états des réplicas soient équivalents sous certaines conditions. La convergence forte impose que les réplicas soient dans un état équivalent dès l'instant où le même ensemble de requêtes a été appliqué alors que la convergence à terme permet d'avoir des états temporairement non équivalents. Dans la pratique, ces deux propriétés peuvent être obtenues en fournissant un outil de résolution de conflits ou en manipulant uniquement des requêtes dites commutatives.

À l'aide de ces définitions, nous pouvons définir et explorer les relations entre les modèles de cohérence les plus courants : les modèles de cohérence forte, causale et à terme, ainsi que certaines de leurs variations (liénarisable, sériablisables, causale+, etc.).

3.2.4 Modèles de cohérence forte

Définition de la cohérence forte

Les modèles de cohérence forte imposent un ordre total et unique[1] sur les requêtes. Tous les nœuds doivent appliquer donc toutes les requêtes dans le même ordre durant toute l'exécution. La façon la plus courante d'obtenir cette propriété est d'utiliser un consensus entre les différents nœuds [CDG+08], [BBC+11] ou des communications utilisant un broadcast atomique [HKJR10]. Bien que ; techniquement ; ces propriétés n'assurent que des garanties de cohérence et non pas de fraîcheur, dans la pratique la plupart des systèmes fournissent un certain degré de fraîcheur, sans nécessairement le *garantir*.

Lors de travaux plus anciens, les modèles de cohérence *linéarisable* et *séquentielle* ont été indistinctement nommés cohérence forte. Cela est dû au fait que ces deux modèles garantissent un ordre total unique sur les requêtes. Cependant la cohérence linéarisable impose une contrainte de temps réel sur les nœuds, là où la cohérence séquentielle n'impose qu'une contrainte PRAM. Il est à noter que la propriété PRAM 4 est strictement plus faible que la propriété de temps réel 3, tous systèmes garantissant une cohérence linéarisable fournit également une cohérence séquentielle entre les réplicas.

Cependant, cette nuance est souvent théorique et difficilement observable du point de vue de l'utilisateur. En effet, pour s'assurer que les requêtes respectent une propriété de temps réel 3 sans respecter la propriété PRAM 4, il est nécessaire de savoir quand chaque requête commence et termine son traitement sur chacun des réplicas. Avoir de telles connaissances revient à disposer d'une horloge globale et synchronisée pour tous les nœuds, ce qui a été établi comme impossible par Lamport dans [Lam78]. De ce fait, il n'est pas possible de s'assurer, d'un point de vue extérieur, qu'un système respecte une propriété de temps réel ou PRAM.

1. commun à tous les nœuds

Dans [BK14], Bailis et al. définissent également la cohérence *sérialisable*. Ce modèle de cohérence renforce les principes de la cohérence linéarisable en l'appliquant aux transactions. Lors de cette thèse, nous nous sommes cependant concentrés sur les systèmes non transactionnels, nous n'explorons donc pas davantage les modèles de cohérence transactionnels.

Nous posons les définitions suivantes pour les modèles de cohérence linéarisable et séquentielle :

Définition 2 (Cohérence linéarisable). *Un modèle de cohérence est dit linéarisable s'il respecte les propriétés d'ordre global et temps réel.*

Définition 3 (Cohérence séquentielle). *Un modèle de cohérence est dit séquentiel s'il respecte les propriétés d'ordre global et PRAM.*

Étude à travers un exemple simplifié

Afin de souligner les différences théoriques de ces deux modèles, nous utilisons l'exemple présenté dans la Section 3.2.1 et illustré par les Figures 3.4 et 3.5. Nous proposons d'étudier comment les requêtes sont traitées et appliquées sur les réplicas dans cet exemple.

u_1	$r_{\alpha_1}(d)$ $r_{\alpha_2}(d)$
u_2	$r_{\beta_1}(d)$
u_3	$r_{\gamma_1}(d)$

FIGURE 3.4 – Exemple d'exécution du scénario simplifié, du point de vue des utilisateurs

s_1	$r_{\beta_1}(d^1)$ $r_{\alpha_1}(d^1)$
s_2	$r_{\gamma_1}(d^2)$ $r_{\alpha_2}(d^2)$

FIGURE 3.5 – Exemple d'exécution du scénario simplifié, du point de vue des réplicas

Dans cette exécution, s_1 reçoit la requête $r_{\beta_1}(d^1)$, commence le traitement de celle-ci (sans l'appliquer pour le moment) et l'envoie à s_2. Ce dernier fait de même avec $r_{\gamma_1}(d^2)$. Ces requêtes sont alors concurrentes et les deux nœuds doivent se mettre d'accord sur l'ordre dans lequel celles-ci doivent être appliquées, et nous pouvons exhiber deux ordres possibles à ce stade :

$$O^1 \Leftrightarrow O^2$$
$$\forall i, O^i : \langle \gamma_1 \prec \beta_1 \rangle \textbf{ ou } \langle \beta_1 \prec \gamma_1 \rangle$$

Plus tard, l'utilisateur 1 émet les requêtes α_1 et α_2. Ces requêtes devront être appliquées dans cet ordre par tous les réplicas. s_1 reçoit la requête $r_{\alpha_1}(d^1)$, la traite et la propage à s_2. Cette requête sera forcement traitée après la fin du traitement de β_1 et devra alors être appliquée après par tous les réplicas, que ça soit par la propriété temps réel 3 ou PRAM 4. Par symétrie, γ_1 doit précéder α_2 dans tous les ordres d'applications. Nous obtenons alors les restrictions suivantes sur ces ordres :

$$O^1 \Leftrightarrow O^2$$
$$\forall i, O^i : \langle \alpha_1 \prec \alpha_2 \rangle \textbf{ et } \langle \beta_1 \prec \alpha_1 \rangle \textbf{ et } \langle \gamma_1 \prec \alpha_2 \rangle$$

Selon la contrainte fournie dans la Section 3.2.2, en considérant une horloge globale, le traitement de γ_1 est fini avant le début du traitement de α_1. Un système garantissant un modèle de cohérence linéarisable doit donc s'assure que γ_1 soit appliquée avant α_1 par tous les réplicas, par la propriété de temps réel 3. Nous pouvons alors ajouter la contrainte suivante sur l'ordre d'application :

$$O^1 \Leftrightarrow O^2$$
$$\forall i, O^i : \langle \gamma_1 \prec \alpha_1 \rangle$$

Un système fournissant un modèle de cohérence séquentielle n'est pas sujet à cette contrainte, la propriété PRAM 4 n'ayant qu'une visibilité "par nœud". La composition de ces observations nous permet d'établir qu'un modèle de cohérence linéarisable impose que tous les réplicas appliquent les requêtes dans le même ordre et que cet ordre peut être :

$$O^1 \Leftrightarrow O^2$$
$$\forall i, O^i : \langle \gamma_1 \prec \beta_1 \prec \alpha_1 \prec \alpha_2 \rangle \textbf{ ou } \langle \beta_1 \prec \gamma_1 \prec \alpha_1 \prec \alpha_2 \rangle$$

Si le système garantit une cohérence séquentielle, l'ordre $\langle \beta_1 \prec \alpha_1 \prec \gamma_1 \prec \alpha_2 \rangle$ est également acceptable. Nous obtenons alors les ordres suivants :

$$O^1 \Leftrightarrow O^2$$
$$\forall i, O^i : \langle \gamma_1 \prec \beta_1 \prec \alpha_1 \prec \alpha_2 \rangle \textbf{ ou } \langle \beta_1 \prec \gamma_1 \prec \alpha_1 \prec \alpha_2 \rangle \textbf{ ou } \langle \beta_1 \prec \alpha_1 \prec \gamma_1 \prec \alpha_2 \rangle$$

Nous pouvons donc bien remarquer, par cet exemple, que ces deux modèles sont très proches, voire indissociables s'il est impossible d'établir les relations de temps réel. Nous attirons l'attention du lecteur sur le fait que cet exemple décrit un scénario où les nœuds s'accordent sur l'ordre dans lequel les requêtes peuvent être appliquées. Cependant, il est possible que lors de la synchronisation, les nœuds concluent que certaines requêtes ne doivent pas être appliquées. Par exemple, si la propagation de γ_1 est trop longue et que s_1 à déjà fini de traiter β_1 (ou même

α_1), certains protocoles de synchronisation vont déterminer que γ_1 ne doit pas être appliqué car issu d'une trop vieille version. Tous les réplicas appliqueront donc un même ordre n'incluant pas γ_1. Ce cas de figure peut se produire pour chaque requête, nous ne décrivons donc pas ici tous les états possibles des réplicas mais l'ordre que vont suivre les nœuds pour appliquer les requêtes validées par l'ensemble des nœuds. Les nœuds appliquant le même ensemble d'opération avec le même ordre, les réplicas seront donc bien dans un état équivalent à la fin de l'exécution.

Conséquences de l'utilisation des modèles de cohérence forte

Les modèles de cohérence forte sont considérés comme les plus intuitifs pour travailler [Bre12]. En effet, grâce à la propriété d'ordre global 1, du point de vue de l'utilisateur les nœuds se comportement de la même façon que si le système ne disposait que d'une seule copie de chaque donnée. Cela permet aux utilisateurs de manipuler les données sans se soucier des problèmes de cohérence. Cependant, cette robustesse est coûteuse pour le système. En effet, pour assurer que tous les réplicas soient dans un état équivalent, il est nécessaire de faire des synchronisations plus ou moins régulièrement dans le système (selon le protocole de cohérence). Avant d'appliquer une opération sur un réplica, chaque nœud doit s'assurer qu'aucune autre opération n'ait déjà été appliquée par un nœud. Comme nous l'avons souligné dans la section précédente, il est également possible que certaines requêtes soient rejetées lors de la synchronisation des nœuds. Il peut être également souhaitable de vérifier que le réplica d'un nœud est la version la plus récente possible avant de l'envoyer à un utilisateur. Certains protocoles imposent les synchronisations à chaque lecture, écriture ou opérations, selon les attentes sur la répartition des requêtes.

Une synchronisation entre rf réplicas est coûteuse, car elle demande de faire plusieurs vagues de communications afin que tous les nœuds soient sûrs que tous les réplicas aient le même ordre pour les différentes opérations. Et il est connu que le problème de consensus entre nœuds asynchrones ne peut pas être traité facilement [Fis83], [FLP85], [GHM+00]. D'autant que nous le sachions, lorsqu'un système affirme fournir une cohérence forte, il ne précise généralement pas s'il s'agit de cohérence linéarisable ou séquentielle. De ce fait, et suite aux observations précédentes, pour le reste de cette thèse, nous utiliserons également la dénomination de cohérence forte.

3.2.5 Modèles de cohérence causale

Définition de la cohérence causale

La gestion de données par des modèles de cohérence dites *fortes* permette de facilement manipuler les réplicas des données. Nous avons vu que cela se fait au prix d'une synchronisation, ce qui est coûteux pour le système et génère donc au mieux de la latence et au pire une indisponibilité de la donnée.

Afin de réduire cette latence, tout en permettant de maintenir de fortes garanties sur la cohérence entre les réplicas d'une donnée, de nombreux travaux ont proposé des moyens de rendre le système plus "souple". En particulier, les systèmes *causaux* se sont avérés efficaces et simples d'utilisation. Ils se basent grandement sur la relation "happen-before" présentée par Lamport dans l'article [Lam78]) et que nous avons décrite dans la Section 2.2.1. En admettant qu'une notion de temps soit possible, cette relation détermine qu'un événement e_1 est arrivé avant un autre événement e_2, si e_2 commence *après* que e_1 soit terminé.

Bien que cette relation puisse sembler triviale, il est important de rappeler deux contraintes des systèmes distribués :

- il est impossible de mettre en place une horloge partagée, globale et infiniment précise. Chaque nœud d'un système distribué s'exécute à sa propre vitesse et peut donc avoir sa propre notion de temps.
- il est possible qu'un événement e_1 commence sur un nœud avant un autre événement e_2 sur une autre machine, mais se termine après celui-ci.

Il apparaît alors qu'établir un ordre total sur la date des événements est impossible dans un système distribué.

De ce fait, ce type de propriété ne peut pas fournir un ordre total sur les événements, mais uniquement un ordre partiel. Lamport défini alors la relation de causalité, permettant de définir si un évément précède causalement un autre événement. Cette relation a été présenté dans le Chapitre 2, et nous la rappelons ici :

Définition 1 Causalité. *Un événement e_1 précède causalement un événement e_2, noté $e_1 \rightsquigarrow e_2$, si une des conditions suivantes est vérifiée :*

1. *e_1 et e_2 se produisent sur le même nœud et e_1 se termine avant le commencement de e_2;*
2. *e_1 est l'envoi d'un message par un nœud et e_2 la réception de ce message par un autre nœud;*
3. *il existe un événement e_3 tel que $e_1 \rightsquigarrow e_3 \rightsquigarrow e_2$.*

Dans leurs travaux présentés dans [HW90], Herlihy et al. introduisent l'*ordre (potentiellement) causal*[2] dans un contexte de système distribué, que nous rappelons ici :

Propriété 2 (Ordre causal). *Chaque requête est livrée à l'application selon un ordre causal, établi d'après la relation de causalité Def. 1.*

Instinctivement, un système qui gère les données avec une cohérence causale entre les réplicas remplace la propriété d'ordre global 1 des protocoles de cohérence forte par la propriété d'ordre causal 2. Cet ordre est plus léger à maintenir que l'ordre total : un nœud détermine que les requêtes qu'il traite dépendent causalement des requêtes qu'il a précédemment traitée, permettant de vérifier la condition 1 de la relation causale. Cette information est transmise lors de la propagation des message permettant d'établir les conditions 2 (réception de la requête) et 3 (transitivité) de la relation causale. Cette approche permet donc d'éviter les phases de synchronisation sans fournir de résultats "absurdes" : par la relation causale, seules les requêtes indépendantes les unes des autres peuvent être appliquées dans des ordres différents.

Dans [BSW04], Brzezinski et al. montrent que pour assurer une cohérence causale, il est nécessaire de fournir les propriétés de garantie de sessions. Ces garanties ont été définies par Terry et al. dans [TDP+94] et permettent d'isoler certaines garanties sur l'états des réplicas. Intuitivement, ces propriétés assurent que lors d'une "session" de travail (i.e. une suite d'opérations d'un utilisateur sur un même nœud), les états successifs du réplica suivront les requêtes de l'utilisateur.

Propriété 8 (Monotonic Reads). *Les requêtes de lectures successives doivent renvoyer une série d'ensembles croissants de requêtes.*

Propriété 9 (Reads your Writes). *Les requêtes de lectures sont traitées par un nœud qui a appliqué toutes les requêtes d'écriture de la même source.*

Propriété 10 (Monotonic Writes). *Les requêtes d'écritures sont traitées par un nœud qui a appliqué toutes les requêtes d'écriture de la même source.*

Propriété 11 (Writes follows Reads). *Les requêtes d'écritures traitées effectuées sur un nœud ayant appliqué une requête de lecture sur une même donnée doivent être appliquées sur une version au moins aussi récente de la donnée.*

Suite à la démonstration de la conjecture de Brewer par Gilbert et al. dans [GL02], et le désir de proposer des services toujours plus réactifs, de nombreux

2. Bien que la formulation correcte soit "potentiellement causal", pour des soucis de simplicité nous utiliserons le terme "causal" et ses dérivés dans la suite de ce manuscrit.

travaux ont été menés sur les modèles de cohérence causale. En effet, il est possible de fournir de telles garanties tout en gardant le système disponible à tout moment, même en cas de partition. Tant que le système souffre d'une partition, il existe (au moins) deux groupes de nœuds incapables d'établir des communications. Soient deux opérations différentes faites sur chaque groupe, ces deux requêtes ne pourront vérifier aucune des conditions de la causalité et pourront donc être appliquées sans risque de transgresser l'ordre causal. Mahajan et al. affirment que les modèles de cohérence causale sont les modèles les plus "forts" pouvant être implémentés dans un système grandement disponible et dont les réplicas convergent [MAD+11]. Cette analyse fait écho aux restrictions de compromis du théorème du CAP. Les auteurs affirment donc que s'il est possible de garantir la convergence des données, les systèmes garantissant un accès (aussi rapide que possible) aux données, devraient implémenter un protocole de cohérence permettant d'assurer une cohérence causale entre les réplicas. Ils affirment également que la plupart des systèmes implémentant un protocole de cohérence causale implémentent en fait un protocole de cohérence causale en temps réel. Ce protocole assure donc aussi bien les propriétés de causalités 2 que la propriété de temps réel 3 telle que défini dans la Section 3.2.3.

Lloyd et al. présentent dans [LFKA11] le modèle de cohérence causale+, parfois appelé cohérence causale convergente. Le principe de ce modèle est d'ajouter la propriété de convergence forte 6, présentée dans la section 3.2.3 :

Propriété 6 (Convergence forte). *Tous les réplicas corrects ayant appliqué les mêmes opérations de mise à jour de la donnée ont un état équivalent.*

De plus, la propriété d'ordre causale 2 assure que toute paire de requêtes causalement liée a été appliquée dans le même ordre par les deux nœuds. Ce modèle garantit donc que les potentiels conflits issus de requêtes concurrentes finiront par être résolus et que l'état des réplicas va converger. Les auteurs de l'article avancent que leur modèle est moins fort que le modèle de cohérence causale en temps réel. Cependant, ce dernier ne fournit pas de garanties de convergences, rendant les deux modèles théoriquement incomparables. Viotti et al. affirment dans [VV16] que les modèles de cohérence causale en temps réel pourraient théoriquement être enrichis pour fournir des propriétés de convergence. À notre connaissance, il n'existe pas de travaux de recherches disponibles traitant de cet axe. Nous pouvons toutefois noter que Mahajan et al. incluent lors de [MAD+11] la convergence comme une propriété intrinsèque du système [3].

Bien qu'il existe de nombreuses nuances théoriques aux modèles de cohérence causale, nous nous concentrons ici sur le modèle de cohérence causale classique et des deux variantes présentées précédemment afin de détailler leurs différences.

3. *always available, one-way convergent system*

En effet, lorsqu'un système implémente un modèle de cohérence causale, il s'agit généralement d'un des modèles présentés précédemment. Nous pouvons définir les modèles de cohérence causale de la façon suivante :

Définition 4 (Cohérence causale). *Un modèle de cohérence est dit causal s'il respecte la propriété d'ordre causal.*

Définition 5 (Cohérence causale en temps réel). *Un modèle de cohérence est dit causal en temps réel s'il respecte les propriétés d'ordre causal et de temps réel.*

Définition 6 (Cohérence causale+). *Un modèle de cohérence est dit causal+ s'il respecte les propriétés d'ordre causale et de convergence forte.*

Étude à travers un exemple simplifié

Afin de souligner les différences théoriques de ces deux modèles, nous utilisons et rappelons l'exemple présenté dans la Section 3.2.1 et illustré par les Figures 3.4 et 3.5. Nous proposons d'étudier comment les requêtes sont traitées et appliquées sur les réplicas dans cet exemple.

u_1	$r_{\alpha_1}(d)$ $r_{\alpha_2}(d)$
u_2	$r_{\beta_1}(d)$
u_3	$r_{\gamma_1}(d)$

FIGURE 3.4 – Exemple d'exécution du scénario simplifié, du point de vue des utilisateurs

s_1	$r_{\beta_1}(d^1)$ $r_{\alpha_1}(d^1)$
s_2	$r_{\gamma_1}(d^2)$ $r_{\alpha_2}(d^2)$

FIGURE 3.5 – Exemple d'exécution du scénario simplifié, du point de vue des réplicas

Nous pouvons déduire de ce scénario et de l'ensemble des contraintes, les relations causales suivantes :

1. $r_{\alpha_1}(d) \rightsquigarrow r_{\alpha_2}(d)$;
2. $r_{\beta_1}(d^1) \rightsquigarrow r_{\alpha_1}(d^1)$;
3. $r_{\gamma_1}(d^2) \rightsquigarrow r_{\alpha_2}(d^2)$.

Si l'application $\mathcal{A}$ fournit des garanties de cohérence causale, s_1 peut traiter et appliquer les requêtes $r_{\beta_1}(d^1)$ et $r_{\alpha_1}(d^1)$ dès leur réception et les propager à s_2 en

indiquant la relation de causalité. Le nœud s_2 se comporte de la même manière et propage $r_{\gamma_1}(d^2)$ puis $r_{\alpha_2}(d^2)$ en indiquant le lien de causalité entre les deux. Nous pouvons donc en déduire que l'ordre d'application des réplicas doit tenir compte des restrictions suivantes :

$$O^1 : \langle \beta_1 \prec \alpha_1 \rangle$$
$$O^2 : \langle \gamma_1 \prec \alpha_2 \rangle$$

Les requêtes $r_{\alpha_1}(d)$ $r_{\alpha_2}(d)$ étant toutes deux issues de u_1, l'ordre d'application doit également respecter la causalité entre ces opérations :

$$\forall i, O^i : \langle \alpha_1 \prec \alpha_2 \rangle$$

Nous ne pouvons pas établir quand les propagations d'un nœud vers l'autre seront reçues par ce dernier. Après réception des propagations, un nœud peut choisir d'appliquer immédiatement les requêtes ou d'attendre, tout en respectant l'ordre causal. Nous obtenons alors les ordres suivants :

$$O^1 : \langle \beta_1 \prec \gamma_1 \prec \alpha_1 \prec \alpha_2 \rangle \textbf{ ou } \langle \gamma_1 \prec \beta_1 \prec \alpha_1 \prec \alpha_2 \rangle \textbf{ ou } \langle \beta_1 \prec \alpha_1 \prec \gamma_1 \prec \alpha_2 \rangle$$
$$O^2 : \langle \beta_1 \prec \gamma_1 \prec \alpha_1 \prec \alpha_2 \rangle \textbf{ ou } \langle \gamma_1 \prec \beta_1 \prec \alpha_1 \prec \alpha_2 \rangle \textbf{ ou } \langle \beta_1 \prec \alpha_1 \prec \gamma_1 \prec \alpha_2 \rangle$$

Contrairement aux ordres décrits pour les modèles de cohérence forte, il n'y a pas de garanties d'équivalence entre O^1 et O^2. En effet, avec un modèle de cohérence causal, chaque nœud est susceptible d'ordonner les requêtes concurrentes différemment (ici (β_1, γ_1) et (α_1, γ_1).

Si l'application souhaite garantir une cohérence causale en temps réel, alors les nœuds doivent respecter la contrainte de temps réel entre α_1 et γ_1, ce qui restreint les ordres possibles à :

$$O^1 : \langle \beta_1 \prec \gamma_1 \prec \alpha_1 \prec \alpha_2 \rangle \textbf{ ou } \langle \gamma_1 \prec \beta_1 \prec \alpha_1 \prec \alpha_2 \rangle$$
$$O^2 : \langle \beta_1 \prec \gamma_1 \prec \alpha_1 \prec \alpha_2 \rangle \textbf{ ou } \langle \gamma_1 \prec \beta_1 \prec \alpha_1 \prec \alpha_2 \rangle$$

Nous pouvons remarquer que même avec ce modèle, la convergence des réplicas n'est pas garantie. En effet, il est possible d'avoir $O^1 : \langle \beta_1 \prec \gamma_1 \prec \alpha_1 \prec \alpha_2 \rangle$ et $O^2 : \langle \gamma_1 \prec \beta_1 \prec \alpha_1 \prec \alpha_2 \rangle$ ce qui pourrait conduire à des réplicas incohérents entre eux (on rappelle que β_1 et γ_1 ne sont pas nécessairement commutatives).

Afin d'assurer que les réplicas soient dans un état équivalent à la fin de l'exécution, $\mathcal{A}$ doit fournir une garantie de convergence et donc implémenter une cohérence causale+ (en temps réel ou non). Pour cela, il est nécessaire de disposer d'une règle de résolution de conflits : en cas de requêtes concurrentes ((β_1, γ_1) et (α_1, γ_1) dans notre exemple), chaque nœud utilise la même règle menant à un état équivalent des réplicas. Par abstraction, nous pouvons considérer que les nœuds ordonnent

les requêtes concurrentes dont ils ont connaissance. Par exemple, les nœuds pourraient décider que les requêtes traitées par s_1 sont prioritaires sur celles traitées par s_2 en cas de concurrences. Une autre approche serait d'établir au préalable un ordre sur les requêtes (par exemple, une addition sera toujours traitée avant une multiplication concurrente, un ajout sera traité avant une suppression dans un set, etc.). Selon le critère choisi, les nœuds peuvent suivre un ordre différent mais qui mènera à un résultat équivalent :

$$O^1 \Leftrightarrow O^2$$
$$\forall i, O^i : \langle \beta_1 \prec \gamma_1 \prec \alpha_1 \prec \alpha_2 \rangle \textbf{ ou } \langle \gamma_1 \prec \beta_1 \prec \alpha_1 \prec \alpha_2 \rangle (\textbf{ ou } \langle \beta_1 \prec \alpha_1 \prec \gamma_1 \prec \alpha_2 \rangle)$$

Il est important de noter que la convergence n'est assurée que sur les périodes où s_1, s_2 ont appliqué le même ensemble d'opérations. Tant que certaines requêtes sont en transit, il est possible (voire probable) que les réplicas soient dans des états non équivalents. Les applications utilisant un modèle de cohérence causale+ étant toujours disponible, les utilisateurs peuvent donc avoir accès à des données dont les réplicas sont dans des états temporairement incohérents.

Conséquences de l'utilisation des modèles de cohérence causale

À la grande différence des modèles de cohérence forte, les modèles de cohérence causale imposent que les données soient accessibles à tout moment, y compris en cas de perturbations dans le système. En particulier lors des partitions, les utilisateurs doivent être capables de faire des requêtes sur les réplicas accessibles. Dans cette situation, chaque nœud va poursuivre son exécution normalement, toute opération faite dans une autre partition est considérée comme concurrente puisqu'elle ne peut satisfaire aucune des propriétés de la relation causale.

Si l'application utilise un modèle de cohérence causale+, alors lorsque la partition se termine, les nœuds doivent gérer les conflits afin d'atteindre des états équivalents. Nous rappelons que la propriété de convergence forte 6 ne s'applique qu'aux réplicas sur lesquels ont été appliquées les mêmes opérations. De ce fait, tant qu'une partition est présente, les réplicas de différentes partitions peuvent avoir des états non équivalents sans transgresser la propriété.

De nombreux travaux ([LFKA11, MAD+11, BG13]) présentent les modèles de cohérence causale comme les *meilleurs* compromis entre cohérence et disponibilités tout en étant résistant aux partitions. Il est cependant nécessaire de maintenir un ordre causal ce qui peut augmenter le nombre de messages échangés (ou leur taille) au sein de l'application. De plus, comme nous l'avons exposé lors de l'étude à travers l'exemple, il est possible que les réplicas soient dans des états divergents si aucune résolution de conflits n'est fournie. C'est en particulier le cas lorsque deux partitions se regroupent et que les nœuds se propagent les requêtes reçues lors de la période durant laquelle la partition était présente. Afin de pouvoir maintenir

des garanties sur la convergence des réplicas, différents travaux se sont portés sur les modèles de cohérence à terme et les outils de résolutions de conflits, comme les *Conflit-Free Replicated Data Type* [SPBZ11].

3.2.6 Modèles de cohérence à terme

Définition de la cohérence à terme

Initialement conçu pour permettre la mise à jour de données dans des systèmes à faible latence, les modèles de cohérence à terme imposent deux conditions :

Propriété 5 (Convergence (à terme)). *Tous les nœuds corrects ayant appliqué les mêmes opérations de mise à jour de la donnée finiront par avoir un état équivalent.*

Propriété 7 (Réception à terme). *Toutes les opérations émises dans le système finissent par être appliquées sur tous les réplicas des données concernées.*

La combinaison de ces deux propriétés permet de s'assurer qu'à terme, toutes les répliques d'une donnée seront dans un état équivalent.

Lors des travaux publiés dans [KJBH+88], Kawell et al. décrivent de la façon suivante les modèles de cohérence à terme : "Les modifications apportées à un réplica finiront par être propagées aux autres réplicas. Si toutes les activités de mise à jour cessent, après un certain temps, tous les réplicas d'une donnée vont converger vers un état logique équivalent." Nous pouvons noter l'absence de borne sur le temps nécessaire à la convergence des réplicas ou sur le nombre d'états intermédiaires avant d'atteindre un état équivalent. De plus, bien que la propriété de convergence 5 soit une propriété contraignante, les modèles de cohérence à terme n'imposent aucun ordre sur l'application des requêtes tant que les différents réplicas convergent vers un état équivalent. De ce fait, il existe différentes façons d'implémenter un modèle de cohérence à terme. Par exemple, chaque nœud stockant un réplica de la donnée peut retourner la dernière version connue et appliquer directement les requêtes de mises à jour, tout en effectuant une version non bloquante du consensus avec les autres nœuds. Une autre approche est de fournir un outil de résolution de conflits comme un ordre de priorité qui serait appliqué sur les requêtes concurrentes. Il est classique d'utiliser des concepts tels que "add-win" : une opération d'ajout dans un ensemble est toujours traitée en priorité dans le cas d'une concurrence avec une opération de suppression. Il existe de nombreuses autres approches pour déterminer un ordre, la seule contrainte étant l'existence d'un ordre total en cas de conflits.

Dans [SPBZ11], Shapiro et al. définissent un nouveau type de données : les *Conflict-Free Replicated Data Type*. Ce type de données permet de maintenir une faible latence sur les réponses aux utilisateurs, même en cas de fautes dans le système. Les différentes façons d'implémenter les résolutions de conflits ont permis

aux auteurs de mettre en avant une seconde propriété de convergence, dite convergence forte Prop. 6. Cette propriété impose que tous nœuds corrects ayant appliqué les mêmes requêtes sont dans un état similaire. Cette propriété se distingue de la *convergence à terme* en interdisant les états intermédiaires lorsque deux nœuds ont appliqué les opérations. Une des conséquences de la convergence forte est de pouvoir assurer que, si aucune opération n'est en suspens dans le système, alors tous les réplicas sont dans un état équivalent.

Bien que les modèles de convergence à terme aient été présentés vers la fin des années 80, ils ne sont devenus populaires qu'avec l'avènement des systèmes de gestion de donnée distribués *grandement disponible* tel que décrit dans [Vog09]. Dans [BG13], Bailis et al. étudient les conséquences de tels modèles de cohérence. Ils en concluent que l'absence d'ordre (total ou causal) permet théoriquement d'atteindre de bonnes performances dans toutes situations, mais que le manque de garanties sur la *fraîcheur* et l'état des réplicas peut rendre leur utilisation dangereuse. Cependant, de nombreuses applications ont opté pour de tels modèles. En effet, Dynamo, le système de gestion de données distribué d'Amazon, implémente un modèle de cohérence à terme. Cassandra, le système de gestion de données distribué de Facebook, se base également sur ce type de modèle. Ce choix de design semble justifié par deux observations majeures détaillées dans [DHJ+07] :

- Malgré le manque de garanties théoriques sur la vitesse de convergence, cette dernière s'avère élevée en pratique. En effet, dans de nombreux cas, les applications à large échelle utilisent des centres de données pour stocker les données. Ces centres sont capables d'établir des communications très rapidement, ce qui permet de réduire le nombre de requêtes concurrentes dans la pratique : les messages sont en transit moins longtemps, la probabilité de traiter une requête pendant le transit d'une autre est donc réduite. De la même façon, il est possible de résoudre rapidement les conflits lorsque de telles requêtes se produisent.
- Les applications qui implémentent des modèles de cohérence à terme, ne sont pas dépendantes de la cohérence entre les réplicas d'une donnée. Par exemple, l'ordre des messages affiché par Facebook peut être temporairement différent si cela permet de publier chaque message plus rapidement. Après quelque temps, l'ordre des messages sera le même pour tous les utilisateurs.

Selon les analyses effectuées dans [DHJ+07], ces deux observations sont illustrées par le fait que le panier d'achats d'Amazon peut temporairement être incorrect (objets disparus ou non supprimés) dans le cas de modifications concurrentes. L'étude affirme que ce cas de figure ne se produit que dans 0.06% des situations d'ajouts et suppressions concurrentes dans le panier d'achats. Decandia et al. affirment, de plus, que ce scénario n'est pas vraiment pénalisant, car les utilisateurs doivent

valider le panier d'achats avant la fin de la transaction, permettant de corriger de potentielles erreurs.

Nous pouvons proposer les définitions suivantes pour les modèles de cohérence à terme couramment utilisés :

Définition 7 (Cohérence à terme). *Un modèle de cohérence est dit à terme s'il respecte la propriété de convergence et de réception à terme.*

Définition 8 (Cohérence forte à terme). *Un modèle de cohérence est dit fort à terme s'il respecte la propriété de convergence forte et de réception à terme.*

Et l'exemple ?

Comme les modèles de cohérence à terme n'imposent ni d'ordre sur les requêtes ni de borne sur le nombre d'états intermédiaires, l'utilisation de notre exemple de scénario simplifié présenté dans la Section 3.5, et illustré par le Figure 3.4 ne nous semble donc pas justifiée. Nous rappelons simplement que si l'application fournit un modèle de cohérence à terme, alors s_1 et s_2 devront être dans un état équivalent avant la fin de l'exécution. Dans le cas où l'application fournit un modèle de cohérence forte à terme alors, tant que s_1 et s_2 ont reçu les mêmes opérations, leurs états sont équivalents.

Conséquences de l'utilisation des modèles de cohérence à terme

Comme nous l'avons évoqué à de nombreuses reprises, les modèles de cohérence à terme n'imposent que la convergence des différents réplicas dans le temps. Il peut alors être difficile de travailler avec les réplicas des données. En effet, si les modèles de cohérence forte sont les modèles les plus proches d'une donnée centralisée, et donc les modèles les plus intuitifs d'utilisation, les modèles de cohérence à terme sont ceux qui s'en éloignent le plus. Ces modèles n'imposant pas de borne sur la durée nécessaire, ni à la réception des requêtes ni à la convergence des différents réplicas, il est théoriquement impossible de savoir si le résultat d'une requête est équivalent selon les nœuds ou non.

C'est pourquoi ces modèles ne doivent être utilisés que dans des applications dont la manipulation de données temporairement incohérentes n'est pas critique. Cependant, comme évoqué par l'étude de [DHJ+07], dans la pratique ces modèles se révèlent rapides, et les cas de réplicas incohérents suffisamment rares pour que l'utilisation de modèle de cohérence à terme soit adaptée.

Les modèles de cohérence à terme sont généralement présentés comme des modèles de cohérence classiques. Il est cependant important de souligner que ces modèles sont différents d'un point de vue conceptuel. En effet, ces modèles n'imposent pas d'ordre ou de garantie à un instant donné, mais uniquement des propriétés qui seront vérifiées *à terme*. Comme nous l'avons évoqué, le plus souvent ces propriétés

sont obtenues en fournissant un outil de résolution de conflits au système. De ce fait, il est possible d'enrichir un modèle de cohérence avec une propriété de convergence 5. Cette particularité est soulignée par Mahajan et al. dans [MAD+11], où ils considèrent alors la convergence comme une propriété du système et non une garantie du modèle de cohérence.

3.2.7 Modèles de cohérence "alternatifs"

Nous n'avons décrit ici que les trois familles de modèles de cohérence les plus courants : les modèles de cohérence forte, causale et à terme. Cependant, comme détaillé dans l'article [VV16], il existe plus de 50 modèles de cohérence différents. Cependant, certains de ces modèles sont peu utilisés dans le pratique, nous ne les présentons donc pas ici. Il existe d'autres approches au problème de la cohérence, comme la gestion par quorum ou la cohérence probabiliste, que nous présentons ici. Ces modèles se démarquent des modèles précédents par leur approche et sont donc naturellement incomparables à ces derniers.

Cohérence par quorum

Certains systèmes basent leurs modèles de cohérence sur l'utilisation de quorums [Apa], [Vol] ou [Red]. Pour chaque requête, l'application attend une réponse de la part d'un certain nombre de nœuds avant de fournir le résultat à l'utilisateur. Dans ces systèmes, pour chaque donnée, des tailles de quorum d'écritures et de lectures sont déterminées.

Lorsqu'une opération d'écriture est émise par un utilisateur, la requête est envoyée à un quorum d'écriture. De la même façon, les requêtes de lectures sont traitées par un quorum de lecture. Il est possible de mettre en avant certaines propriétés du système en fonction de la taille des quorums. Nous considérons ici qu'une donnée est répliquée rf fois. On note qw la taille du quorum d'écriture et qr la taille du quorum de lecture.

Si un système cherche à fournir des garanties de cohérence forte entre les réplicas, il faut alors s'assurer que $qw + qr > rf$. De cette façon, les quorums s'intersectent et toute requête de lecture sera faite par un nœud ayant déjà traité la dernière opération d'écriture. À l'inverse, il est possible de poser $qw = qr = 1$ afin de fournir les meilleurs délais de traitements pour les requêtes, sans offrir de garanties sur l'état des réplicas. On considère généralement que les systèmes utilisant une cohérence par quorum de cette façon disposent d'un outil de résolution de conflits et fournissent une cohérence à terme [LM10]. Les systèmes existants proposent aux utilisateurs de fixer eux-mêmes ces paramètres, mais utilisent des valeurs par défaut. Par exemple, Cassandra utilise $qw = qr = 1$ par défaut [Apa] alors que Project Voldemort utilise $qw = qr = \lceil \frac{rf}{2} \rceil$ [Vol].

Une telle vision de la cohérence permet d'adapter la gestion des différents réplicas précisément selon le besoin des utilisateurs. Cependant, les paramètres R, W étant fixés par ceux-ci, il est nécessaire d'avoir une profonde connaissance des implications de ces paramètres sur la cohérence et des besoins de l'application. De plus, cette approche n'offre que peu de garanties quand $qw + qr \leq rf$. Une étude probabiliste de ces garanties est menée dans [BVF+12].

Modèles de cohérence probabilistes

Afin de mieux comprendre le comportement des systèmes utilisant un modèle de cohérence par quorum, Bailis et al. proposent une étude probabiliste de la cohérence [BVF+12]. Cette étude se concentre sur les systèmes où l'intersection entre les deux quorums, de taille respective qw et qr, n'est pas garantie. Plus précisément, la garantie que ces quorums ne s'intersectent pas est de p_s avec :

$$p_s = \frac{\binom{rf-qw}{qr}}{\binom{rf}{qr}}$$

Lorsqu'une requête de lecture est émise, celle-ci a donc une probabilité $1 - p_s$ de refléter la dernière opération d'écriture. Les auteurs s'interrogent ensuite sur l'état des réplicas quand celui n'est pas à jour. Ils définissent alors la cohérence Probalistic Bounded Staleness. Ce modèle de cohérence utilise les définitions de la *k-fraîcheur*, la *t-visibilité* et la *⟨k, t⟩-fraîcheur*.

Définition 9 (Cohérence k-fraîche). *Un modèle de cohérence est dit k-frais, avec une probabilité* $1 - p_{sk}$*, si, pour tout quorum de lecture, au moins une valeur est issue d'une des k dernière requêtes d'écriture. Avec* p_{sk} *la probabilité calculée par :*

$$p_{sk} = \left(\frac{\binom{RF-qw}{qr}}{\binom{RF}{qr}} \right)^k$$

Définition 10 (Cohérence t-visible). *Un modèle de cohérence est dit t-visible, avec une probabilité* $1 - p_{st}$*, si, pour tout quorum de lecture, au moins une valeur est issue d'une requête d'écriture traitée il y a t unités de temps. Avec* $P_w(c, t)$ *la fonction de cumul des densités des c nœuds ayant reçu une mise à jour après t et* p_{st} *la probabilité calculée par :*

$$p_{st} = 1 - \left(\frac{\binom{RF-qw}{qr}}{\binom{RF}{qr}} + \sum_{c \in [qr, RF)} \frac{\binom{RF-c}{qr}}{\binom{RF}{qr}} . [P_w(c+1, t) - P_w(c, t)] \right)$$

Définition 11 (Cohérence $\langle k, t \rangle$-*fraîcheur*). *Un modèle de cohérence est dit* $\langle k, t \rangle$-frais*, avec une probabilité* $1 - p_{skt}$*, si, pour tout quorum de lecture, au moins une*

valeur est issue d'une des k dernière requêtes d'écriture à condition que la lecture commence t unités de temps après les k dernières requêtes d'écriture. Avec p_{skt} la probabilité calculée par :

$$p_{skt} = 1 - \left(\frac{\binom{RF-qw}{qr}}{\binom{RF}{qr}} + \sum_{c \in [qr, RF)} \frac{\binom{RF-c}{qr}}{\binom{RF}{qr}} . [P_w(c+1,t) - P_w(c,t)] \right)^k$$

Les auteurs recommandent cependant de favoriser l'utilisation de *k-fraîcheur* ou de *t−visibilité* plutôt que de la $\langle k,t \rangle$*-fraîcheur*, ce dernier étant plus difficile à utiliser dans les raisonnements. Grâce à leur définition, Bailis et al. mettent en avant qu'avec une donnée répliquée 3 fois et des quorums de taille 1, la probabilité qu'une requête de lecture retourne ne prenne en compte aucunes des 3 requêtes d'écritures les plus récentes est de $0.\overline{703}$. Cette probabilité passe à $0.\overline{962}$ si un des deux quorums est de taille 2.

Dans [RTN+17], Rahman et al. revisitent le théorème du CAP sous le point de vue probabiliste. Afin de proposer une version probabiliste du théorème, les auteurs proposent une version probabiliste des trois propriétés. Nous étudions cette version du théorème, et les définitions probabilistes des propriétés de latences et de partitions dans le Chapitre 4. Les auteurs proposent une définition probabiliste de la cohérence, appelé (t,p)*-cohérence probabiliste*. Cette définition utilise la notion de *t-fraîcheur*.

Propriété 12 (t-fraîcheur). *Une requête de lecture est dite* t-fraîche *si elle retourne un état qui prend en compte une opération d'écriture ayant eu lieu depuis $\tau(R,t)$ unités de temps avec :*

- *$\tau(R,t)$ la date de la dernière écriture si celle-ci est plus vieille que t unités de temps;*
- *$\tau(R,t) = \tau(R) - t$* [4] *si au moins une écriture a été appliquée au cours des t unités de temps précédant la requête de lecture.*

Plus instinctivement, une requête de lecture est dite *t-fraîche* si celle-ci prend en compte une requête d'écriture plus récente que t unités de temps (ou la plus récente opération d'écriture si celle-ci est plus vieille). Il est alors possible de proposer le résultat suivant pour le modèle de cohérence (t,p)*-probabiliste* :

Définition 12 (Cohérence (t,p)-probabiliste). *Un modèle de cohérence est dit (t,p)-probabiliste si le rapport de requête de lecture t-fraîche est d'au moins $(1-p)$.*

La probabilité p peut alors être vue comme la probabilité qu'une requête de lecture retourne une valeur n'étant pas à jour. Selon Rahman et al., cette définition de la cohérence probabiliste est compatible avec la définition de la *t-visibilité*

4. avec $\tau(R)$ la date de la requête de lecture

proposé dans [BVF+12]. La (t, p)-*cohérence probabiliste* est utile, toujours selon les auteurs, pour les scénarios d'enchère, ou une cohérence forte n'est pas souhaitable (de par la synchronisation qu'elle impose) alors que la t-*fraîcheur* permet aux utilisateurs d'attendre moins de temps entre deux enchères.

3.3 Classification des modèles de cohérence

Une fois les définitions et implications des différents modèles de cohérence posées, de nombreux travaux ont essayé d'en proposer une classification. Une motivation pour proposer une classification des modèles est de pouvoir fournir un outil précis pour aider les concepteurs à choisir le modèle de cohérence le plus adapté à leur besoin. En ne considérant que la cohérence, il est possible de voir l'ordre causal comme un affaiblissement de l'ordre total, tel que défini dans la Section 3.2.3. Nous pouvons alors poser :

$$\text{cohérence forte} \succ \text{cohérence causale}$$

De plus, nous pouvons remarquer que la propriété de temps réel 3 est plus forte que la propriété PRAM 4 puisque cette dernière est une réduction de la propriété de temps réel au point de vue d'un seul réplica. Notre classification devient alors :

$$\begin{aligned}&\text{cohérence linéarisable} \succ \text{cohérence séquentielle}\\ &\succ \text{cohérence causale en temps réel} \succ \text{cohérence causale}\end{aligned}$$

Nous rappelons que bien que les modèles de cohérence linéarisable soient théoriquement plus forts que les modèles de cohérence séquentielle, dans la pratique il est souvent impossible de déterminer quel modèle est implémenté par un système. Les propriétés de temps réel 3 et de convergence 5 permettent d'affirmer que les modèles de cohérence causale en temps réel et causal+ sont tous deux plus fort que les modèles de cohérence *simple*. Cependant, comme nous l'avons souligné dans la Section 3.2.5, et comme l'ont affirmé Viotti et al. dans [VV16], ces deux modèles sont incomparables. En effet, chacun des modèles garantit une propriété que l'autre ne peut pas garantir. Notre classification devient alors :

$$\begin{aligned}&\text{cohérence linéarisable} \succ \text{cohérence séquentielle}\\ &\succ \text{cohérence causale en temps réel} \,||\, \text{cohérence causale+}\\ &\succ \text{cohérence causale}\end{aligned}$$

Lors des travaux présentés dans [SSAP16], Shapiro et al. proposent un repère 3D afin de comparer les modèles de cohérence. Ce repère 3D tente de pallier l'impossibilité de proposer un ordre total entre les modèles de cohérence. Les auteurs

définissent trois axes permettant de placer les modèles de cohérence sur ces axes selon certains critères. Les trois axes ne sont cependant pas indépendants les uns des autres. Shapiro et al. proposent les définitions suivantes pour les axes :

- Ordonnancement des opérations : cet axe permet de définir l'ordre que doivent respecter les nœuds pour appliquer les requêtes, allant d'un ordre total commun à tous les nœuds à un ordre indéfini propre à chaque nœud ;
- Visibilité des requêtes : cet axe permet de définir la relation entre écriture et lecture, en particulier, quelles requêtes d'écriture doivent être prises en compte (visible) par une requête de lecture ;
- Composition des objets : cet axe défini les garanties de couple entre les différentes requêtes et données. Cet axe permet notamment de valider les liens entre les deux premiers axes.

Cette classification permet alors d'aborder différents modèles de cohérence et de représenter les apports et contraintes de chacun de ces modèles. Comme nous l'avons vu dans ce chapitre, l'axe de l'ordonnancement des requêtes est régulièrement utilisé pour définir les différents modèles. En particulier, les modèles de cohérence forte imposent un ordre total et commun à tous les nœuds, se plaçant à une extrémité de l'axe. Les modèles de cohérence à terme, en revanche, n'imposent pas d'ordre sur les requêtes et se situent donc à l'autre extrémité de l'axe. L'axe de visibilité des requêtes, lié à l'axe précédent, est utile pour définir les nuances entre les différents modèles du point de vue de l'utilisateur. Les modèles de cohérence forte doivent assurer que toutes les requêtes d'écritures doivent être prises en compte lors des requêtes de lectures. En revanche, les modèles de cohérence à terme n'imposent généralement que la prise en compte des écritures traitées sur le même nœud. Le troisième axe permet de formaliser les dépendances entre les deux axes. Il est également utile pour manipuler la composition d'objet, mais cette notion est attachée aux systèmes transactionnels et n'entre donc pas dans le cadre de cette thèse.

Dans [VV16], Viotti et al. présentent un peu plus de 50 modèles de cohérence différents et proposent une définition formelle pour chacun d'entre eux. Cette définition s'accompagne d'une explication rapide, mais les auteurs ne s'attardent pas sur les différences entre les modèles ni des conséquences de ceux-ci sur un système de gestion de données. Les auteurs proposent également un graphe "exhaustif" pour classifier ces modèles. Ce graphe permet de mettre en avant 8 grandes "familles" de modèles : faible, garanties de sessions, fork-based, causal, par objet, synchronisée, staleness based et linéarisable. Cependant, de nombreux modèles n'entrent pas directement dans ces familles. Nous pouvons également observer que si les modèles d'une même famille sont comparables entre eux, dans la plupart des cas, deux modèles de cohérence sont incomparables. Nous nous sommes inspirés

des définitions et explications de ces travaux afin de construire l'analyse présentée dans ce chapitre.

Adve et al. proposent un tutoriel sur les différents problèmes les plus communs lors de leurs utilisations dans les systèmes à mémoire partagée les plus courants [AG96]. Dans [SN04], Steinke et al. présentent une théorie unifiée de la cohérence dans les modèles à mémoire partagée, en se basant sur une composition de propriétés fondamentales. Friedman et al. fournissent une autre proposition de description de différents modèles, selon leurs restrictions respectives [FVC03].

Ces travaux permettent d'avoir une vue d'ensemble des différents modèles de cohérence pouvant exister, ils ne présentent cependant que les propriétés de cohérence et ne considèrent pas la disponibilité des données gérées par de tels modèles. Dans la Section 3.2, nous avons étudié les familles de modèles de cohérence les plus répandus dans les systèmes de gestion de données distribués existants et le Chapitre 4 propose de discuter des conséquences de ces modèles dans un système de gestion de données distribués et donc des interactions entre les modèles de cohérence, la disponibilité et les différentes fautes.

3.4 Conclusion

Ce chapitre détaille le principe de réplication et de cohérence entre les différents réplicas d'une donnée. Il existe de très nombreux modèles de cohérence différents dont la plupart sont incomparables entre eux. Or, chacun de ces modèles impose différentes contraintes sur les données, et en particulier sur l'ordre dans lequel les requêtes peuvent être appliquées. Ces contraintes peuvent imposer la présence d'autres mécanismes dans le système comme la possibilité d'une synchronisation (et d'un consensus) ou de résoudre les conflits de requêtes concurrentes. Avant de pouvoir étudier le théorème du CAP et ses conséquences sur les systèmes de gestion de données distribués, il est donc nécessaire de bien comprendre les différences entre les principaux modèles de cohérence.

Nous proposons ici une explication pédagogique sur les nuances entre les trois familles de modèles les plus répandus : les modèles de cohérence forte (linéarisable, séquentielle), les modèles de cohérence causale (causale, causale+, causale en temps réel) et les modèles de cohérence à terme (convergence, forte convergence). Nous observons notamment les différences sur l'ordre de traitement des requêtes selon le modèle de cohérence. Nous utilisons un exemple simple permettant de décrire les différents résultats accessibles en appliquant le même scénario avec les modèles de cohérence évoqués. Nous pensons que cette approche permet de mettre en avant certaines nuances parfois peu intuitives et de mieux appréhender les effets des différents modèles de cohérence sur les systèmes de gestion de données. La littérature propose de nombreuses analyses des différents modèles

de cohérence [VV16, SSAP16, AG96, SN04, FVC03]. Mais nous avons cependant déterminé que peu d'entre eux s'attardent sur les relations entre ces modèles de cohérence et les systèmes de gestion de données distribués, en particulier au vu du théorème du CAP.

De ce fait, lors de l'utilisation d'un système de gestion de données distribués, l'utilisateur doit être capable de maîtriser les enjeux des différents modèles de cohérence et des stratégies qui les accompagnent (réplications, placement des réplicas, etc.). Nous proposons donc dans le Chapitre 4 d'étudier les liens entre les modèles de cohérence et les propriétés des systèmes de gestions de données distribués.

Sommaire

Chapitre 4

Systèmes de gestion de données distribués et propriétés garanties

4.1 Conjecture de Brewer et Systèmes de gestion de données distribués

L'émergence des architectures large échelle et des *clouds* a permis le développement de nombreuses applications à l'échelle mondiale, telle Facebook, Amazon. Ces applications produisent d'immenses quantités de données quotidiennement, de l'ordre des pétabytes [Fac]. Ces données, pouvant être largement sollicitées par les utilisateurs, sont stockées dans des clouds et doivent être accessibles à tout moment, peu importe la localisation de l'utilisateur [1]. Afin de maintenir un tel service, les applications utilisent des systèmes de gestion de données géodistribuées. Idéalement, l'utilisation de ces systèmes devrait permettre l'accès rapide, et à tout moment, de n'importe quelle donnée dans son état le plus récent, et ce, malgré la présence potentielle de partitions dans le système sous-jacent.

Nous avons cependant déjà exprimé, dans le Chapitre 2 que l'utilisation de systèmes de gestion de données distribués a entraîné de nouveaux défis : fiabilité, transfert de méta-informations, dépenses énergétiques, consensus, etc. Parmi ces défis, la conjecture de Brewer a particulièrement marqué les travaux de recherches

1. Tout en respectant les droits des lieux impliqués

et de conception de systèmes de gestion distribués. Présenté dans [FB99] et formalisé dans [Bre00], Brewer propose la conjecture conjecture de Brewer, ou théorème du CAP, illustré dans la Figure 4.1 et dont nous rappelons l'énoncer :

Théorème 2 (Théorème du CAP). *Tout système distribué ne peut satisfaire que deux des trois propriétés désirables suivantes :*

- *Cohérence;*
- *Disponibilité;*
- *Tolérance aux partitions.*

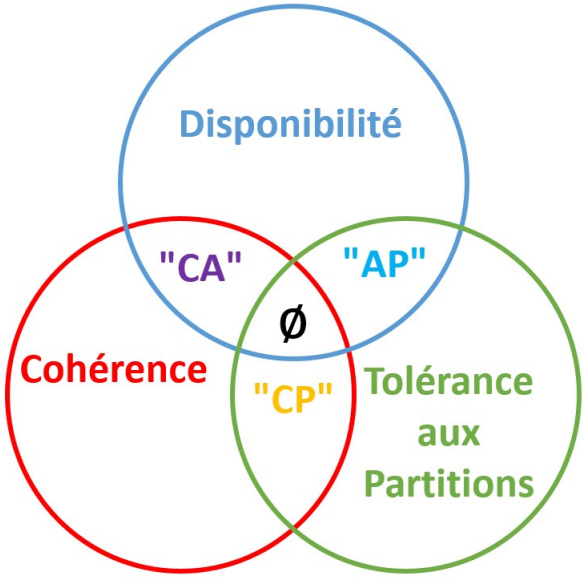


FIGURE 4.1 – Illustration du théorème du CAP

Précisons que les concepts de cohérence et disponibilité font ici référence à la plus forte version de ces concepts, soit la cohérence linéarisable et la grande disponibilité respectivement. La conjecture de Brewer fut démontrée par Gilbert et Lynch dans [GL02]. Les auteurs utilisent les définitions suivantes pour leur démonstration :

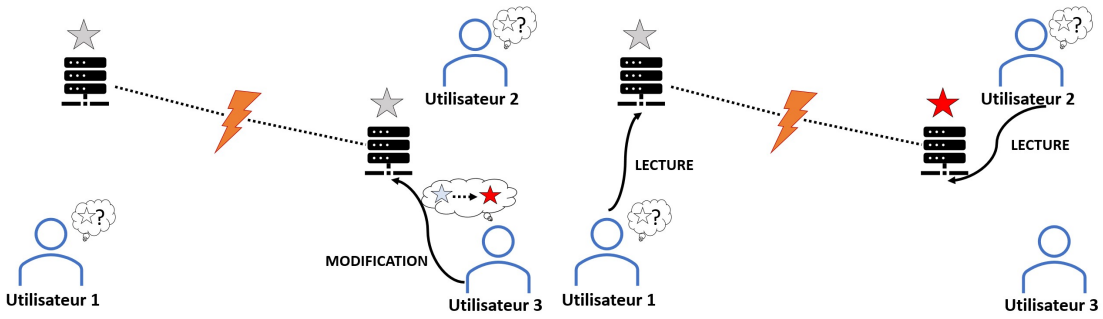
Définition 13 (Cohérence ([GL02])). *Il existe un ordre total sur toutes les opérations de façon à ce que chaque opération puisse être exécuté en un instant.*

Définition 14 (Disponibilité ([GL02])). *Un système est dit disponible s'il garantie que toutes requêtes reçu par un nœud non fautif résulte en une réponse.*

Définition 15 (Partition ([GL02])). *Un système est partitionné s'il existe deux groupes de nœuds tels que tous les messages d'un groupe vers l'autre sont perdus.*

Les auteurs proposent une démonstration formelle, basée sur le principe suivant : soient deux nœuds n_1 et n_2 dans deux partitions différentes du système, i.e. tous les messages de n_1 vers n_2 sont perdus. Si une requête d'écriture est reçue par n_1, ce dernier ne peut pas la propager à n_2. Il est alors impossible que les requêtes de lecture traitées par n_2 prennent en compte l'écriture de n_1. Dans ce cas, soit les nœuds manipulent des réplicas dans des états non équivalents soit la requête d'écriture (ou de lecture selon le protocole) ne peut pas aboutir, rendant le système indisponible.

Les Figures 4.2 et 4.3 illustrent cette situation : l'utilisateur 3 modifie la donnée dans la Figure 4.2a puis les utilisateurs 1 et 2 font une requête de lecture sur cette donnée dans la Figure 4.2b. La Figure 4.3a illustre le comportement d'un système disponible, même lors des partitions. Ce système ne peut pas propager en compte l'écriture de l'utilisateur 3 et va donc fournir des résultats incohérents aux utilisateurs. La Figure 4.3b à l'inverse représente un système garantissant une cohérence (linéarisable) entre les réplicas. Dans le cas de partitions, ce système est contraint de mettre les requêtes des utilisateurs "en attente", le système est alors indisponible.

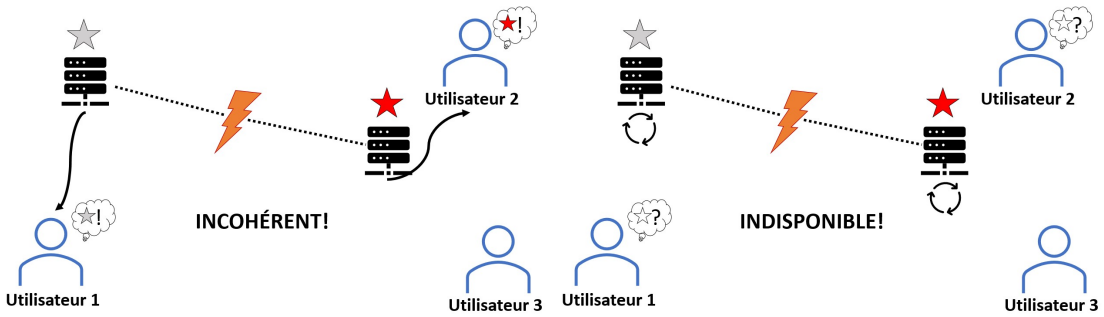


(a) Modification d'un réplica (b) Lecture des réplicas

FIGURE 4.2 – Illustration de l'impossibilité du CAP

Au fil des années, de nombreux travaux, autant théoriques que pratiques, se sont basés sur ce théorème et la plupart des systèmes de gestion de données distribués sont positionnés en fonction des propriétés garanties :

- Disponible et tolérant aux Partitions "AP" : les données sont toujours accessibles, même en cas de partitions ;
- Cohérent et tolérant aux Partitions "CP" : tous les réplicas d'une même donnée sont toujours dans des états équivalents, y compris pendant une partition ;
- Cohérent et Disponible "CA" : les données sont toujours disponibles et leurs réplicas sont dans des états équivalents. Le bon fonctionnement du système n'est cependant pas assuré lors des partitions.



(a) Système disponible (AP) (b) Système cohérent (CP)

FIGURE 4.3 – Illustration de l'impossibilité du CAP

Cependant, par sa formulation imprécise, ce théorème a été l'objet de nombreuses critiques, plus ou moins justifiées. En effet, l'énoncé du théorème ne donne pas de définition précise de la disponibilité et considère une donnée comme disponible si cette dernière est accessible en un temps borné non précisé. La définition d'une partition n'est pas non plus clairement établie. Enfin, de nombreux travaux semblent considérer ces propriétés comme binaires : une donnée est accessible ou non, à jour ou non et le système fonctionne ou non lorsqu'une partition est présente. Brewer précise dans [Bre12] que ces propriétés doivent plus être vues comme des spectres que comme des propriétés binaires. De fait, de nombreux travaux essayent de classifier les différents modèles de cohérence ou de disponibilité afin de pouvoir développer des systèmes de gestion de données distribués les plus adaptés aux besoins des applications.

Dans ce chapitre, nous rappelons rapidement les définitions de certains modèles de cohérence avant de proposer une discussion autour la disponibilité et la résistance aux fautes dans les systèmes de gestion de données distribués. Nous étudions ensuite les compromis entre cohérences, disponibilité et tolérance aux fautes et les variations du théorème du CAP Th. 2. Ce chapitre se clôt par un tour d'horizon de 18 systèmes de gestion de données de données distribués, en étudiant leurs choix de conception et les conséquences de ceux-ci.

Il est important de noter que nous nous concentrons ici sur le cas des systèmes de gestion de données dits NoSQL, à savoir que les données accédées ne sont pas liées par des relations causales. Notre objectif ici est de présenter une vision pédagogique des différences entre les principaux modèles de cohérence et leurs conséquences sur les systèmes de gestion de données.

Lors de cette thèse et des travaux présentés, nous étudions des données répliquées sur différents nœuds d'un système de gestion de données. Ces répliques (ou réplicas) sont gérées par certains protocoles de cohérence afin de garantir des propriétés sur leurs états respectifs. Pour des soucis de légèreté d'expressions, il nous

arrivera de parler de données cohérentes (respectivement fortement cohérentes, cohérentes à terme, etc.) en lieu et place de données gérées par un protocole de cohérence (respectivement un protocole de cohérence forte, à terme, etc.).

4.2 Analyse et définitions des propriétés de la conjecture de Brewer

4.2.1 Un rapide retour sur les problèmes de cohérence dans un système de gestion de données distribué

Afin de permettre aux utilisateurs d'accéder aux données dans de nombreux cas et le plus rapidement possible, les systèmes de gestion de données répliquent les données. Chaque donnée est répliquée un certain nombre de fois dans le système et placée à différents endroits stratégiques. Cependant, cette approche peut conduire à de nouvelles difficultés : selon leur emplacement et les opérations qu'elles reçoivent, les différents réplicas d'une même donnée peuvent se trouver dans des états différents. On dit alors que les réplicas sont incohérents entre eux ou que la cohérence de la donnée est compromise.

Afin de décrire les différentes situations possibles, de nombreux modèles de cohérence ont été exposés au travers de nombreux travaux de la communauté scientifique [SSAP16], [VV16] et présentés dans le Chapitre 3. On appelle modèle de cohérence un ensemble de propriétés offrant différentes garanties sur les états respectifs des données et leurs répliques. Les modèles les plus courants sont les modèles de cohérence forte, causale et à terme (et leurs variations respectives). Dire qu'un système fournit un modèle de cohérence revient à dire que ce système implémente des mécanismes permettant de garantir que les propriétés du-dit modèle sont toujours vérifiées. De par la nature hétérogène des différents modèles, la plupart d'entre eux se retrouvent être incomparables entre les uns et les autres. Le Chapitre 3 approfondit ces notions. Il est important de comprendre que l'implémentation de ces mécanismes a des conséquences sur les autres performances du système comme discuté au cours de ce chapitre, et qu'il est donc important de choisir les modèles de cohérence les plus adaptés au service développé. Nous rappelons très rapidement les modèles de cohérence les plus populaires.

Modèles de cohérence forte

Définition 2 (Cohérence linéarisable). *Un modèle de cohérence est dit linéarisable s'il respecte les propriétés d'ordre global et temps réel.*

Propriété 1 (Ordre global). *Chaque requête est livrée à l'application selon un ordre unique et global.*

Propriété 3 (Temps réel). *Si un nœud s_1 a fini de traiter une requête r_α avant qu'un autre nœud s_2 n'ait commencé à traiter une autre requête r_β, alors tous les nœuds doivent appliquer r_α avant r_β.*

Les modèles de cohérence forte assurent que, à tout instant de l'exécution, tous les réplicas d'une même donnée sont dans un état similaire. Comme leur nom l'indique, ces modèles sont "forts" et donc coûteux. De fait, il requiert une forme de synchronie pour maintenir de telles propriétés, généralement un protocole de consensus ou de broadcast atomique [CDG+08], [HKJR10]. Ces modèles sont les plus intuitifs pour travailler puisqu'ils sont conceptuellement équivalents à manipuler une seule copie de chaque donnée.

Modèles de cohérence causale

Définition 4 (Cohérence causale). *Un modèle de cohérence est dit causal s'il respecte la propriété d'ordre causal.*

Propriété 2 (Ordre causal). *Chaque requête est livrée à l'application selon un ordre causal, établi d'après la relation de causalité Def. 1.*

Les modèles de cohérence causale sont plus "faibles" : afin de proposer un service plus rapide, la propriété d'ordre total 1 est remplacée par la propriété d'ordre causal 2. Ces modèles se basent sur la relation *happen-before*, définie dans [Lam78] et assurent que les états des différents réplicas respectent les relations causales entre les opérations : si une opération est "arrivée avant" une autre, alors tous les réplicas rendront compte de cet ordonnancement. En revanche, aucune garantie n'est donnée sur les opérations concurrentes. Ce modèle est plus léger à maintenir, car seul un ordre causal et non total est nécessaire. Cependant, il est tout de même possible d'obtenir des réplicas dans des états incohérents entre eux.

Modèles de cohérence à terme

Définition 7 (Cohérence à terme). *Un modèle de cohérence est dit à terme s'il respecte la propriété de convergence et de réception à terme.*

Propriété 5 (Convergence (à terme)). *Tous les nœuds corrects ayant appliqué les mêmes opérations de mise à jour de la donnée finiront par avoir un état équivalent.*

Propriété 7 (Réception à terme). *Toutes les opérations émises dans le système finissent par être appliquées sur tous les réplicas des données concernées.*

Les modèles de cohérence à terme permettent de garantir que les différents réplicas finiront dans des états équivalents et que toutes les requêtes d'écriture seront appliquées. Ces propriétés étant les seules fournies par ces modèles, ces

derniers sont simples à maintenir et permettent au service d'être très réactifs aux différentes requêtes. Cependant, aucune garantie n'est faite sur les différents états intermédiaires, possiblement nombreux et incohérents entre eux. Il est possible d'établir des propriétés probabilistes pour émettre des précisions sur l'état des réplicas [BVF+12].

4.2.2 Disponibilité d'une donnée et latence d'accès

Lors de la mise en place d'un système de gestion de données distribué, la disponibilité et la latence d'accès sont deux autres métriques primordiales. En effet, il est souhaitable que toutes les données soient accessibles en permanence et avec une faible latence. De plus, la latence moyenne d'un système peut être assimilée à sa vitesse et semble donc être un moyen naturel de comparer deux systèmes. Ce critère est aujourd'hui un positionnement important pour les applications. Amazon estime, par exemple, qu'une augmentation de $100ms$ dans le délai des requêtes entraîne une perte de 1% des ventes [Gre].

Cependant, comme l'avance Brewer dans le théorème du CAP [Bre00], aucun système distribué ne peut garantir une cohérence forte *et* une haute disponibilité tout en étant tolérant aux partitions. Brewer évoque ici la cohérence forte, et de manière plus nuancée les contraintes de cohérence dans [Bre12]. Nous avons vu dans le Chapitre 3 que chaque modèle de cohérence induit différentes contraintes et a donc un impact différent sur la latence possible. Les modèles de cohérence causale, par exemple, demandent également de maintenir un ordre causal, ce qui entraîne davantage de communications entre les nœuds que les modèles de cohérence à terme. Les modèles de cohérence causale peuvent donc avoir un impact plus important sur la latence.

La disponibilité d'une donnée est définie comme le fait que cette donnée soit accessible ou non. Il est donc possible de voir cette dernière comme un cas particulier de latence infinie. Cependant, de nombreux travaux évoquent d'avantages les problèmes de disponibilité que de latences. Ce fait est probablement dû à la formulation du théorème du CAP Th. 2, qui ne traite que de la disponibilité, et que dans le cas où une donnée est accessible, la plupart des services affirment fournir "la meilleure latence possible", aussi appelée latence au mieux.

La définition de la disponibilité, telle que décrite par Gilbert et Lynch dans [GL02] pose cependant différents soucis, comme souligné par Kleppmann dans [Kle15]. La définition de Gilbert et Lynch semble correspondre à une version différente de la disponibilité que la définition utilisée par Brewer dans les articles [Bre00], [FB99]. En effet, dans ces travaux, Brewer et al. semblent considérer qu'une donnée est disponible si les utilisateurs peuvent toujours contacter "des réplicas". Cette vision est centrée sur le point de vue des utilisateurs : un utilisateur peut toujours avoir accès à la donnée qu'il désire.

En revanche, la définition de Gilbert et Lynch est centrée sur le point de vue des réplicas : tout nœud non fautif *doit* répondre aux requêtes qu'il reçoit. Si ces définitions semblent équivalentes (un utilisateur doit pouvoir contacter un réplica, qui doit répondre), elles ne sont pas identiques. En particulier, comme expliqué par Kleppmann, la notion de nœud non fautif n'est pas clairement définie par Gilbert et Lynch [Kle15]. Il semble que les auteurs réfèrent ici des nœuds qui ne tombent pas en panne pendant le traitement de la requête. Cependant, même dans le cas d'un nœud non fautif, il est possible que les liens de communications ne délivrent pas les messages correctement. Dans le cas extrême où de nombreux liens de communications deviendraient inactifs et entraîneraient l'isolement d'un groupe de nœud, alors ce groupe pourrait fonctionner normalement sans être capable de contacter les autres nœuds. C'est ce qu'on appelle une partition dans un système. Dans de tels scénarios, nous pouvons considérer un groupe de nœuds non fautifs trop peu nombreux pour pouvoir établir un consensus entre les réplicas d'une donnée. Ces nœuds peuvent recevoir des requêtes, mais être incapables d'établir un consensus pour déterminer si la version est à jour et ne doivent donc pas renvoyer de réponse à l'utilisateur, rendant ainsi la donnée indisponible.

Un autre problème, cette fois commun aux deux définitions, est qu'elles ne posent aucune borne sur le temps de réponse. En l'absence de toute considération pour la latence, et en supposant que toute faute (venant d'un nœud ou d'un lien de communication) finira par être corrigée (bug corrigé, matériel réparé ou remplacé, etc.), la réponse pourrait être envoyée à un moment et donc satisfaire les propriétés de disponibilités. Cependant, dans la pratique, les mécanismes de time-out empêchent de telles situations. Kleppmann décrit cette propriété comme une propriété purement de *liveness*, mais pas une propriété de sécurité [Kle15]. Il expose également que d'un point de vue extérieur, aucune donnée ne peut être considérée comme non disponible sans mécanisme de time-out (ou assimilé), car la réponse de la requête pourrait simplement arriver "plus tard".

À la lumière de ces considérations, il semble aujourd'hui incorrect de définir la notion de disponibilité sans évoquer la notion de latence. La latence est un paramètre qui peut cependant être difficile à évaluer de manière objective. Un même scénario d'exécution peut mener à des résultats différents : de nombreux facteurs, parfois externes au service, entrent en jeu lors de l'exécution dans le monde réel. Les mesures de latences de différents services doivent alors toujours être prises avec du recul. De ce fait, la plupart des systèmes aujourd'hui utilisés prétendent être "hautement disponible" et proposent un niveau de qualité de service [2] en ce qui concerne la disponibilité et la latence d'une donnée. Ces qualités de service stipulent que, sous certaines conditions, un seuil de latence sera atteint dans un certain nombre de cas. Par exemple, de nombreux services affirment fournir une

2. Service Level Agreement - SLA

latence en deçà d'un seuil 99% du temps, dans toute situation (i.e. en présence de partitions ou non). Chaque service se basant sur un système de gestion de données distribué doit prendre en compte les différentes contraintes liées à ses modèles de cohérence. De ce fait, les différents niveaux des qualités de service peuvent parfois être incomparables. En effet, chaque service va essayer de minimiser le goulot lié à son modèle de cohérence : un service fournissant des données fortement cohérentes doit réduire le temps nécessaire pour construire un ordre total alors qu'un service fournissant des données cohérentes à terme, doit s'assurer que l'utilisateur obtiendra une réponse le plus rapidement possible. De plus, les qualités de services doivent rendre compte des différentes situations où un ensemble de nœuds et/ou de lien de communications sont fautifs.

4.2.3 Impact des fautes dans un système de gestion de données distribué

La prise en compte des différentes fautes pouvant intervenir dans un système distribué est essentielle lors de la mise en place de services à large échelle. À l'image des modèles de cohérence ou de la notion de disponibilité dans un système, il n'existe pas de définition universelle de la notion de faute : il existe de très nombreux types de fautes. Par exemple, une faute peut concerner un nœud ou un lien de communication, être groupée ou isolée, être temporaire ou définitive, être une absence de réaction ou un comportement inattendu, etc. Il est donc toujours nécessaire de préciser le type de fautes considéré lors de la présentation de travaux. Dire qu'un service est tolérant aux fautes revient à dire qu'il fonctionne normalement quand des fautes du type attendu interviennent. On précise parfois le nombre de fautes que le système peut supporter avant de perdre ses garanties.

Le plus souvent, on considère les types de fautes suivants :

- omission : un nœud ou un lien de communication omet un ou plusieurs messages.
- panne (franche) : un nœud ou un lien de communication cesse de fonctionner et n'effectue donc plus aucune action.
- comportement byzantin : un nœud peut effectuer des actions non désirées.
- partition : l'ensemble des nœuds est séparé en plusieurs groupes et toute communication entre ces groupes échoue.

Chacun de ces types peut être temporaire ou permanent, selon le problème considéré. Il est à noter que les définitions d'omission et de panne franche peuvent paraître similaires en cas de fautes temporaires. Dans ce cas, les omissions entraînent la perte de certains messages alors qu'une panne va entraîner la perte de tous les messages sur un laps de temps.

Nous proposons d'étudier l'impact de ces différents types de fautes sur un système de gestion de données distribué.

Les partitions dans un système distribué

Comme nous l'avons évoqué précédemment, il est possible qu'une ou plusieurs partitions appairassent dans le système. Dans ce cas, les nœuds sont séparés en différents groupes qui ne peuvent plus communiquer entre eux. Dans une telle situation, il est clair que deux réplicas dans deux partitions différentes ne peuvent pas être à la fois disponibles et être cohérents entre eux. Malheureusement, les partitions peuvent juste apparaître dans le système résultant de phénomènes extérieurs (problèmes matériels, encombrement réseau par d'autres services, etc.). Lorsqu'une application doit s'exécuter à une échelle suffisamment large, la probabilité qu'une partition apparaisse dans le système devient non négligeable et un service fonctionnel doit être capable de s'exécuter "normalement" (i.e. ne pas retourner de résultats imprévisibles).

Cependant, en pratique, aucune partition n'est permanente. En effet, les liens de communications fautifs finissent par être réparés ou remplacés. Selon les besoins de l'application, les données peuvent donc être mises en attente si elles sont trop peu nombreuses pour maintenir une majorité de réplicas. On dit alors que les nœuds dans les partitions concernées sont indisponibles le temps de la partition.

Nœuds et liens de communication fautifs

Nous considérons dans un premier temps les fautes de type panne franche et omission. Lorsqu'un nœud devient fautif, il arrête de répondre aux autres nœuds (soit parce que les messages sont omis, soit parce qu'il est en panne). Lorsqu'un lien de communication devient fautif, alors les messages qu'il transporte sont perdus. Un nœud peut devenir indisponible s'il est fautif ou si tous les liens de communications permettant de l'atteindre sont fautifs. Du point de vue des autres nœuds du système, il est impossible de dissocier un nœud fautif d'un nœud injoignable à cause des liens de communications. Pour les mêmes raisons, il est impossible de dissocier p nœuds fautifs d'une partition de p nœuds. Il est donc nécessaire que les nœuds se comportent de la même façon lorsqu'une faute apparaît, qu'elle soit liée à un nœud ou à un ensemble de liens de communication. Ici encore, le passage à l'échelle du système implique la croissance de la probabilité qu'au moins un nœud (ou un lien de communication) soit fautif. Les algorithmes et mécanismes déployés doivent donc fonctionner correctement quand ce genre de faute se produit.

Enfin, les fautes byzantines caractérisent les comportements indésirables des nœuds (ou lien de communication). Un nœud byzantin peut envoyer des messages déformés, ou reproduire n'importe quel autre type de faute. Il peut donc être très difficile de les observer de l'extérieur. La considération de comportement byzantin

peut rendre le fonctionnement de nombreux service beaucoup plus difficile. De nombreux travaux, dont les nôtres, se concentrent sur des méthodes tolérantes aux fautes de type panne/omission et partitions.

4.3 Discussion autour du théorème du CAP

4.3.1 Le choix du modèle de cohérence

Les différents modèles de cohérence présentés dans le Chapitre 3 sont les modèles les plus couramment implémentés dans des systèmes utilisés par des applications existantes. Une rapide description de ces modèles est rappelée en Section 4.2.

Si une application a besoin de fournir des données fiables et à jour, alors le système de gestion de données sous-jacent doit être capable de fournir de telles garanties et implémenter un protocole de cohérence forte. C'est par exemple le cas de Zookeeper [HKJR10], utilisé entre autres par Kafka afin de monitorer l'état des nœuds du système [KNR+11]. En effet, lors d'une requête sur l'index d'un objet, il est préférable de souffrir d'une légère latence plutôt que de courir le risque d'obtenir un indice erroné. À l'opposé, certaines applications peuvent se contenter de proposer des données seulement partiellement à jour. C'est par exemple le cas de Facebook, où l'ordre des différents postes importe peu, tant que l'application permet l'utilisation la plus agréable, et donc rapide, pour les utilisateurs. Pour ce genre de service, il est primordial d'être toujours disponible et aussi réactif que possible. Il est donc plus approprié pour ces services de fournir une cohérence à terme ou causale.

De récents travaux étudient la possibilité pour un système de fournir différents niveaux de cohérence, en fonction des besoins de chaque donnée. En effet, les applications récentes étant de plus en plus complexes, il n'est pas rare de voir des applications avoir besoin de maintenir certaines données avec de fortes garanties tout en laissant d'autres données disponibles à tout moment. Dans [LPC+12], Li et al. présentent la *cohérence rouge/bleue*. Cette approche consiste à proposer un modèle de cohérence forte (rouge) quand cela est nécessaire et de relâcher cette cohérence pour un modèle de cohérence à terme (bleue) quand cela est possible. Plus précisément, les auteurs font une distinction entre les opérations rouges, sujettes à un ordre total, et les opérations bleues, effectuées en local et propagées sans synchronisation. Ce modèle a été repris dans les travaux par Gotsman et al. qui proposent une preuve formelle de la préservation d'invariant en utilisant les contraintes de ces modèles [GYF+16]. Bien que cette approche ne semble pas avoir été directement utilisée dans un système existant, Azure CosmosDB [Mic] propose une approche similaire en laissant l'utilisateur choisir le type de cohérence associé à chaque donnée.

4.3.2 Pourquoi faire un choix ? Discussion autour du théorème du CAP

Depuis la présentation de la Conjecture de Brewer, la plupart des systèmes de gestion de données sont présentés en fonction des propriétés garanties parmi les trois proposées par le théorème du CAP. Commençons par rappeler l'énoncé du théorème du CAP :

Théorème 2 (Théorème du CAP). *Tout système distribué ne peut satisfaire que deux des trois propriétés désirables suivantes :*

- *Cohérence;*
- *Disponibilité;*
- *Tolérance aux partitions.*

Il est important de remarquer que Brewer utilise ici les versions les plus fortes de chaque propriété (cohérence linéarisable, grande disponibilité et résistant aux partitions). De nombreux travaux semblent utiliser ce théorème avec des propriétés "binaire", c'est à dire, sous la forme "Si les données doivent être toujours disponibles, alors il est impossible de maintenir des réplicas cohérents entre eux". Cependant, nous avons défini précédemment de nombreuses nuances sur ces propriétés, et Brewer insiste dans un second article [Bre12], que ces propriétés doivent être vues comme des spectres et non pas comme des propriétés binaires, i.e. qu'une donnée peut être plus ou moins disponible et les répliques remplir certaines garanties de cohérence (à terme, causales, etc.). Mahajan et al. montrent dans [MAD+11] qu'il est possible d'implémenter un protocole de cohérence causale en temps réel dans un système où les données sont toujours disponibles, y compris en présence de partitions. Il montre également qu'aucun modèle de cohérence "plus fort" [3] que celui-ci ne peut être implémenté dans un tel système.

Il existe de nombreuses discussions autour du théorème du CAP et du choix des trois propriétés par Brewer. Notamment sur la nature de ces propriétés. En effet, dans [MAD+11], Mahajan et al. présentent la résistance aux partitions comme une propriété du système, à l'opposé de la cohérence et de la disponibilité, présentées comme des propriétés désirables. En effet, dans un système distribué, les fautes ont lieu de manière incontrôlée et les services espérant s'exécuter à une échelle mondiale doivent être capables de faire face à ce genre de scénario. Il est également possible de défendre qu'un système garantissant d'être toujours disponible et d'avoir des réplicas cohérents entre eux, mais ne résistant pas aux partitions n'ait pas de sens. En effet, lorsqu'une partition apparaît, sans tolérance aux partitions, le système ne pourra plus garantir la disponibilité et/ou la cohérence entre les réplicas. Nous

3. Il convient de rappeler que de nombreux modèles de cohérence sont incomparables.

considérons donc, à partir de ce point, uniquement des systèmes de gestion de données offrant une tolérance aux partitions, et devant donc faire un choix sur le compromis entre cohérence et disponibilité.

Mahajan et al. proposent donc dans [MAD+11] une variante du théorème du CAP, appelée le théorème du *Consistency, Availability, Convergence [CAC]*, en utilisant la définition de la convergence suivante :

Théorème 3 (Théorème du CAC). *Aucun modèle de cohérence plus fort que la cohérence causale ne peut être implémenté dans un système toujours disponible et convergent.*

Diak et al. affirment dans [DNS13] que tout système devrait être capable de maintenir la disponibilité des réplicas si aucune partition n'est présente dans le système. Il devient alors nécessaire de faire deux choix pour le système : entre disponibilité et cohérence en cas de partitions et entre latence et cohérence sinon. Ils représentent une variante du théorème, initialement proposé par Abadi et al. dans [Aba12] : le *Partition : Availability-Consistency ; Else : Latency-Consistency (PACELC)*.

Théorème 4 (Théorème du PACELC). *Tout système distribué doit, s'il y a une partition (P) dans le système, établir un compromis entre disponibilité (A) et cohérence (C); sinon (E), quand le système s'exécute normalement, le système doit établir un compromis entre latence (L) et cohérence (C).*

Cette version du théorème propose une vue plus précise et nuancée des choix à faire lors de la mise en place d'un système de gestion de données distribué. Les auteurs soulignent notamment le fait que dans la plupart des designs, les mêmes décisions de compromis sont prises dans les deux cas (i.e. la plupart des systèmes sont PC/EC ou PA/EL). En effet, si l'utilisateur souhaite conserver des garanties de cohérence en cas de partitions, alors il est naturel d'avoir des garanties similaires lorsque le système s'exécute normalement. Cependant, il existe des systèmes, comme PNUTS [CRS+08], présentés comme PC/EL. En temps normal, le système favorise la latence à la cohérence, mais ne fait pas de compromis supplémentaire vis-à-vis de la cohérence en cas de partitions.

Il est également possible d'aborder ce théorème, et les notions de cohérence sous le prisme probabiliste. Dans leurs travaux présentés dans [RTN+17], Rahman et al. définissent le *Probabilistic Consistency, Availability and Partitions Tolerance (PCAP)*. Pour cela, les auteurs posent les définitions probabilistes suivantes :

Propriété 12 (t-fraîcheur). *Une requête de lecture est dite* t-fraîche *si elle retourne un état qui prend en compte une opération d'écriture ayant eu lieu depuis* $\tau(R, t)$ *unités de temps avec :*

- $\tau(R,t)$ *la date de la dernière écriture si celle-ci est plus vieille que* t *unités de temps;*
- $\tau(R,t) = \tau(R) - t$ [4] *si au moins une écriture a été appliquée au cours des* t *unités de temps précédant la requête de lecture.*

Propriété 13 (t-latence). *Une requête de lecture satisfait la* t-latence *si elle retourne en moins de* t *unités de temps.*

Nous rappelons la définition de la cohérence $(t,p)-probabiliste$, présentée dans le Chapitre 3.

Définition 12 (Cohérence (t,p)-probabiliste). *Un modèle de cohérence est dit* (t,p)*-probabiliste si le rapport de requête de lecture* t*-fraîche est d'au moins* $(1-p)$.

Nous introduisons également les définitions de latence et partition probabiliste :

Définition 16 (Latence (t_α,p)-probabiliste). (t_α,p)*-latence : un système atteints une* (t_α,p)*-latence si la proportion de lectures satisfaisant la* t_α*-latence est d'au moins* $(1-p)$.

Définition 17 (Partition (t_α,p)-probabiliste). (t_α,p)*-partition : une exécution souffre d'une* (t_α,p)*-partition si la proportion de chemin d'un réplica à un autre avec une latence supérieur à* t_α *est supérieure à* p.

Le théorème du PCAP peut alors être énoncé de la manière suivante :

Théorème 5 (Théorème du PCAP). *Si* $t_c+t_\alpha < t_p$ *et* $p_\alpha+p_c < p_p$*, il est impossible d'implémenter un objet en lecture/écriture en présence d'une* (t_α,p)*-partition tout en achevant la* (t_c,p)*-cohérence et la* (t_α,p)*-latence.*

Toutes ces discussions et variations autour du théorème du CAP exposent que lors de la mise en place d'un système de gestion de données distribué, il est important de comprendre les relations entre les différentes propriétés du système, et qu'il ne faut pas simplement choisir entre la cohérence des réplicas et leur disponibilité. Nous proposons donc d'étudier les différents choix faits par certains systèmes existants et leurs conséquences et raisons quand cela est possible.

4.4 Étude de systèmes de gestion de données distribués existants

De nombreux services à large échelle se basent sur une utilisation de données et utilisent donc un système de gestion de données distribué. Comme nous l'avons

4. avec $\tau(R)$ la date de la requête de lecture

détaillé au cours des deux derniers chapitres, chaque service a ses propres besoins et va donc avoir recours à différents protocoles et compromis entre la cohérence des réplicas et latence d'accès à ces réplicas.

Il existe différents travaux proposant une comparaison technique de certains systèmes [BCK+10, HHLD11, Cat11]. Notre approche diffère de ces travaux en proposant une comparaison des choix de développement effectués par ces systèmes de gestion de données distribués. Nous étudions ici les choix sur le degré de réplication, la méthode utilisée pour stocker les données et propager les mises à jour et le compromis entre cohérence et disponibilité/rapidité. Comme nous l'avons exprimé dans la Section 4.2.2, il est difficile de comparer directement les latences des différents systèmes et nous n'avons malheureusement pas pu mettre en place des tests comparatifs pertinents pour les différents systèmes étudiés. Nous conduisons donc ici une étude théorique de ces systèmes. Les études [BCK+10, HHLD11, Cat11] proposent des comparaisons de performances de ces systèmes.

Nous nous sommes concentrés sur les systèmes de gestion de données suivants :

- Antidote [1]
- AzureDB [2]
- BigTable [3]
- Cassandra [4]
- CoackroachDB [5]
- Depot [6]
- Dynamo [7]
- FaunaDB [8]
- HyperTable/HBase [9]
- Megastore [10]
- MongoDB [11]
- OpenSwift [12]
- Redis [13]
- Scalaris [14]
- Scatter [15]
- Spanner [16]
- Voldemort Project [17]
- Zookeeper [18]

4.4.1 Stockage des données

Le premier critère qui nous intéresse ici est la façon dont chacun de ces systèmes conserve les données sur les différents nœuds. Il existe trois approches principalement utilisées parmi les différents systèmes existants : basées sur les ensembles *Clé-Valeur*, sur les *documents* et *en colonnes*.

L'approche basée sur les ensembles Clé-Valeur

Dans ce type de système, la valeur d'une donnée est associée à une clé. Cette approche permet de maintenir une structure de donnée simple et efficace, en particulier pour les bases de données relationnelles. Ces systèmes peuvent supporter d'importantes masses de données et une forte concurrence de requêtes. Cette approche a été privilégiée par :

- Antidote [1]
- CoackroachDB [5]
- Depot [6]
- Dynamo [7]
- Redis [13]
- Scalaris [14]
- Voldemort Project [17]
- Zookeeper [18]

L'approche basée sur les documents

Les systèmes avec une approche basée sur le stockage par document utilisent une structure similaire, mais avec une valeur sémantique, stockée au format JSON ou XML. Cette approche permet notamment d'utiliser des attributs secondaires lors des requêtes. C'est la méthode ayant été retenue par :

- FaunaDB [8]
- MongoDB [11]
- OpenSwift [12]

L'approche basée en colonnes

Cette dernière méthode consiste à stocker les données dans différentes colonnes, qui sont elles utilisées via leurs indices. Ce mécanisme permet la réduction d'entrées et sorties sur les colonnes appropriées, conduisant à une meilleure gestion des données pour les systèmes. Les systèmes suivants ont adopté cette approche afin de stocker leurs données :

- BigTable [3]
- Cassandra [4]
- HyperTable/HBase [9]
- Megastore [10]
- Spanner [16]

Nous n'avons pas trouvé d'informations sur la méthode de stockage utilisée par Scatter [15]. Azure [2] affirme pouvoir maintenir les trois approches, selon les besoins des utilisateurs.

4.4.2 Propagation des mises à jour

Lorsqu'une donnée est modifiée, il est nécessaire de propager la modification entre les différents réplicas. Il existe pour cela deux approches : basées sur les *opérations* ou les *états*.

Approche basée sur les opérations

Comme son nom l'indique, cette approche consiste à échanger les opérations nécessaires pour traiter les différentes requêtes. Lorsqu'un réplica reçoit ces opérations, ainsi que l'aval de l'application (en concordance avec les contraintes de cohérence), il applique ces opérations sur l'objet qu'il stocke. Cette approche permet notamment de ne communiquer que les opérations effectuées et ne pas faire de transit

des données en elles-mêmes, permettant ainsi de réduire la taille des communications. De plus, cette approche limite également les risques liés à l'écoute du réseau : les données ne sont pas directement envoyées d'un nœud à un autre, il est donc beaucoup plus difficile de pouvoir retrouver les données pour un membre extérieur au système. Les systèmes suivants implémentent une telle approche pour la propagation des mises à jour.

- BigTable [3]
- HyperTable/HBase [9]
- Redis [13]
- Voldemort Project [17]

Approche basée sur les états

Cette approche consiste au contraire à partager le nouvel état d'une donnée plutôt que les opérations nécessaires pour obtenir cet état. Cette méthode est particulièrement adaptée à certains types de données tels que des ensembles. En effet, il peut être coûteux d'effectuer certaines opérations sur ces ensembles, il convient alors de ne les effectuer qu'une fois et de transférer directement le résultat. Cette approche est utilisée par les systèmes suivants :

- Antidote [1]
- AzureDB [2]
- Cassandra [4]
- CoackroachDB [5]
- Depot [6]
- Dynamo [7]
- FaunaDB [8]
- Megastore [10]
- MongoDB [11]
- OpenSwift [12]
- Scalaris [14]
- Scatter [15]
- Spanner [16
- Zookeeper [18]

4.4.3 Facteur de réplication

Le facteur de réplication indique le nombre de réplicas de chaque donnée devant être présent dans le système. L'intérêt premier de disposer de plusieurs réplicas d'une donnée est de s'assurer que celle-ci ne sera pas perdue en cas de faute. Le facteur de réplication doit donc être adapté aux risques de fautes dans le système. Il faut cependant considérer qu'un facteur de réplication trop élevé peut entraîner des pertes de performance, que ça soit au sujet de la latence ou du nombre de versions différentes d'une même donnée.

De ce fait, la plupart des systèmes permettent à l'utilisateur de fixer lui-même le degré de réplication pour chaque donnée. Cependant, une étude menée par Glendenning et al. dans [GBKA11] montre, qu'en pratique, un facteur de réplication entre 3 et 5 permet d'atteindre une résistance aux fautes supérieures à 99% tout en maintenant de bonnes performances.

4.4.4 Compromis entre cohérence et disponibilité/rapidité

Grande disponibilité/faible latence et modèles de cohérence causale ou à terme

Bien que les modèles de cohérence forte soient les modèles les plus instinctifs, Decandia et al. exposent dans [DHJ+07] que certains services n'ont pas besoin de propriété aussi forte et peuvent affaiblir leur modèle de cohérence afin de proposer un service plus rapide et toujours disponible. Il base leur exemple sur le site de vente *Amazon* (qui utilise le système de gestion de données Dynamo) : il apparaît plus important de fournir une expérience d'utilisation fluide et toujours disponible, quitte à ce que des incohérences apparaissent parfois (anomalies dans les paniers d'achats en cas de modification concurrente). Linden rapporte qu'Amazon estime qu'une latence de $100ms$ supérieure entraîne une chute de 1% des ventes [Gre]. Il en va de même pour les réseaux sociaux : l'ordre des messages et le nombre de réactions peuvent temporairement différer d'un utilisateur à l'autre sans compromettre l'expérience d'utilisation. Dans ces deux exemples, l'utilisation de modèles de cohérence forte permettrait d'éviter les incohérences soulignées au prix d'une légère latence à chaque opération. Les concepteurs de ces services ont considéré que cette latence impactait davantage l'expérience d'utilisation et les services correspondants utilisent donc des systèmes de gestion de données distribués proposant des modèles de cohérence causale ou à terme. C'est notamment le cas des systèmes suivant :

- Antidote [1]
- Dynamo [7]
- OpenSwift [12]

Modèle de cohérence forte et disponibilité/latence "au mieux"

Cependant, de nombreux services optent pour un système de gestion de données de données utilisant de la cohérence forte. En effet, ces modèles ont le double avantage de permettre une manipulation intuitive des données et de proposer un service avec des données fiables. Ces modèles de cohérence sont particulièrement adaptés aux services avec peu de modifications des données (services GPS par exemple) ou aux services manipulant des données sensibles (comme des services de transaction bancaires). Les systèmes suivants proposent une gestion des données avec un modèle de cohérence forte :

- BigTable [3]
- HyperTable/HBase [9]
- Megastore [10]
- MongoDB [11]
- Redis [13]
- Scalaris [14]
- Scatte [15]
- Spanner [16]
- Zookeeper [18]

Autres approches

Certains services, devenant de plus en plus complexes et ayant de multiples objectifs, peuvent avoir des besoins différents pour chaque donnée. Nous pouvons donc observer l'émergence de systèmes permettant à l'utilisateur de choisir le modèle de cohérence à associer à chaque donnée. Nous appelons ces systèmes des systèmes à cohérence multiples. En particulier, Azure [2] permet de sélectionner un modèle de cohérence parmi 9 allant de la cohérence forte à la cohérence à terme.

Cassandra [4] et Voldemort Project [**17**] proposent une approche différente : plutôt que de donner le choix d'un protocole de cohérence, l'utilisateur peut associer à chaque donnée un nombre de réponses nécessaire pour valider l'opération, formant ainsi des quorums de lecture et écriture. En choisissant le nombre approprié de réponses, il est alors possible d'atteindre des garanties similaires à celles de la cohérence forte ou à terme. Ce paradigme, appelé cohérence par quorum, a été détaillé dans la Section 3.

- AzureDB [**2**]
- Cassandra [**4**]
- Voldemort Project [**17**]

4.5 Conclusion

Ce chapitre revient sur la formulation de la conjecture de Brewer après avoir rappelé les définitions considérées par les auteurs à travers les concepts de cohérence, disponibilité et tolérance aux partitions. Nous proposons également une discussion autour des différentes critiques autour de cette conjecture et nous en présentons les variantes les plus pertinentes. La version probabiliste PCAP [BVF+12], détaille les cas ou des versions probabilistes des propriétés sont abordées. Le CAC [MAD+11] qui considère la tolérance aux partitions comme une propriété du système et la remplace donc par une autre propriété désirable dans l'énoncé : la convergence, ou le fait que chaque réplica finisse par atteindre un état équivalent. Le PACELC [Aba12] établit que le système ne devrait pas sacrifier la cohérence ou la disponibilité lors qu'aucune partition n'est présente dans le système. Cette version du théorème précise donc qu'en cas de partitions, le système doit trouver un compromis entre cohérence et disponibilité, et un compromis entre latence et cohérence lorsqu'aucune partition n'est active. Brewer revisite également sa conjecture afin de revenir sur l'impact de celle-ci dans le domaine des systèmes de gestion de données distribués et précise qu'aucune de ces propriétés n'est binaire, mais qu'il est préférable de les considérer comme un spectre [Bre12]. Certaines études [VV16, SSAP16] proposent des modèles plus approfondis afin d'explorer les différentes possibilités des modèles de cohérence.

Lors de la conception d'un système de gestion de données distribués, il est important d'être conscient des garanties que celui-ci doit offrir afin de sélectionner le bon niveau pour chacune des propriétés désirables. De plus, l'étude comparative de 18 systèmes de gestion de données distribués existants permet de mettre en avant d'autres facteurs importants : la nature des données stockées, la façon de propager les mises à jour, etc. De plus, ces facteurs sont décorrélés. Il faut donc se poser la question de la nature de l'application afin de proposer le système le plus adapté possible.

Nous pouvons cependant observer que certains systèmes, à l'image d'Azure CosmosDB, proposent de laisser le choix à l'utilisateur plutôt que de lui imposer un modèle. En effet, CosmosDB propose de stocker les données sous forme de document, ou de mettre en place un système basé sur des ensembles de clé-valeur ou de documents. Nous pensons que de plus en plus de systèmes évolueront en ce sens : cela permet l'utilisation d'un unique système de gestion de données pour un ensemble de services, pouvant être complexe et avoir de nombreux besoins différents. Cependant, ce genre d'approche reste récent et peu de travaux sont disponibles à ce sujet. Nous nous concentrons dans cette thèse à la mise en place de mécanismes adaptés à ce type de systèmes, en particulier sur les problèmes de placements soulevés par ceux-ci.

II

Partie II - Contributions

Sommaire

Chapitre

5

CAnDoR : Placement dynamique des données avec prise en compte des contraintes de cohérence

5.1 Le placement dans un système de gestion de données distribué

Comme nous l'avons évoqué dans les précédents chapitres, de nombreuses applications profitent de l'évolution des architectures clouds (de calcul et/ou de stockage) pour offrir des services à une échelle mondiale. Certaines de ces applications, comme les réseaux sociaux [Fac], [Twi], les moteurs de recherches [Dan], les marketplaces [Amab], etc. manipulent d'importants volumes de données. Les utilisateurs peuvent accéder à ces services depuis n'importe quel point du monde, à condition d'avoir une connexion internet. Afin d'assurer un accès rapide pour toute localisation des utilisateurs et pour éviter la perte de données, la plupart de ces applications répliquent les données ; chaque donnée est répliquée et stockée sur différents nœuds de stockage [TVS07].

Chaque application utilise ces données de manières distinctes et propose un ensemble de services différents aux utilisateurs. Par exemple, pour les services de publications de nombreux réseaux sociaux, sites journalistiques, etc., les données sont écrites par leur auteur puis consultées par des utilisateurs. Une donnée ne

peut être éditée que par son créateur (ou n'est pas éditable dans certains cas). Dans d'autres cas, les données sont éditées et consultées par différents utilisateurs, comme c'est, par exemple, le cas des *wikis* [LC01] ou du travail collaboratif. D'autres applications encore, proposent de manipuler des données *personnelles* : seul le créateur de la donnée (et éventuellement un groupe restreint) peut accéder à celle-ci, comme, par exemple, pour les applications bancaires, les drives de stockages, etc.

Selon les besoins de l'application, et des utilisations envisagées lors du développement, deux données distinctes d'une même application peuvent être gérées par des protocoles de cohérences différents. Différents modèles de cohérence ont été présentés dans le Chapitre 3. Chaque modèle est accompagné de contraintes qui lui sont propres. Considérons l'exemple ou une donnée est répliquée 3 fois et où 3 utilisateur veulent accéder à cette donnée. Cet exemple est illustré par les Figures 5.1,5.2 et 5.3. L'utilisateur 1 va d'abord modifier la donnée (Figure 5.1) puis les utilisateurs vont lancer une requête de lecture de la donnée (Figures 5.2a et 5.3a). Si la donnée est gérée avec un modèle de cohérence forte entre ses réplicas, les utilisateurs 2 et 3 doivent attendre la synchronisation entre les réplicas (Figure 5.2a). Une fois cette synchronisation terminée, les nœuds répondent aux utilisateurs et ceux-ci ont la même vision de la donnée, peu importe le réplica accédé (Figure 5.2b). À l'inverse, dans un modèle de cohérence à terme, les nœuds vont répondre aux requêtes le plus rapidement possible quitte à fournir des versions différentes de la donnée selon le réplica (Figure 5.3a). En revanche, la cohérence à terme assure que les réplicas finiront par atteindre un état équivalent (Figure 5.3b).

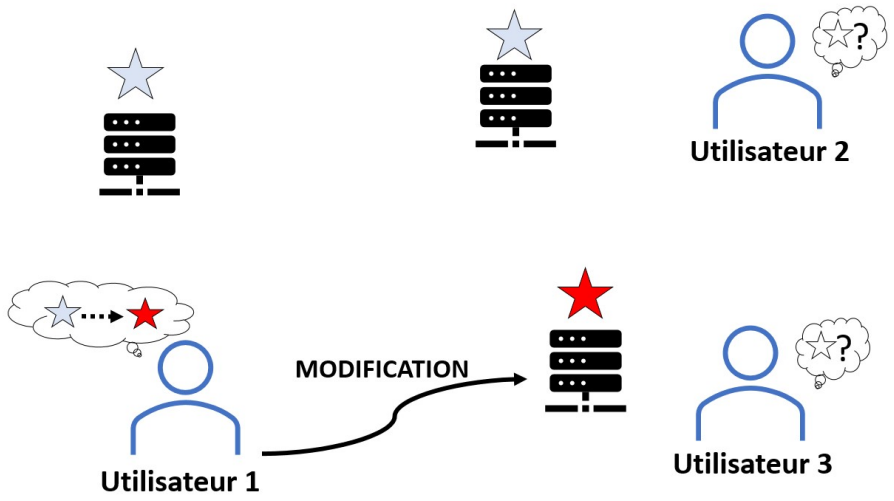
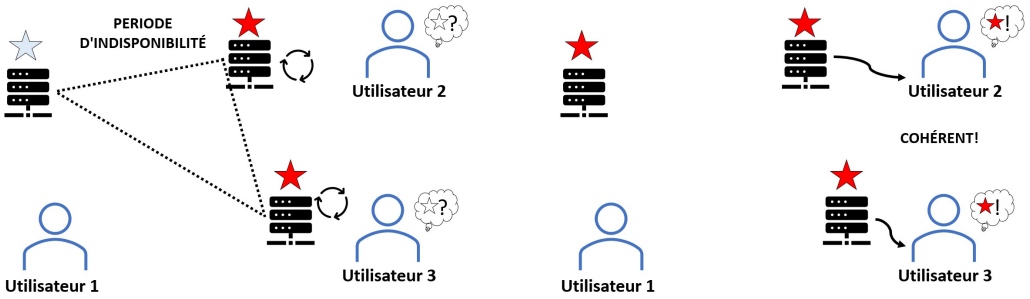
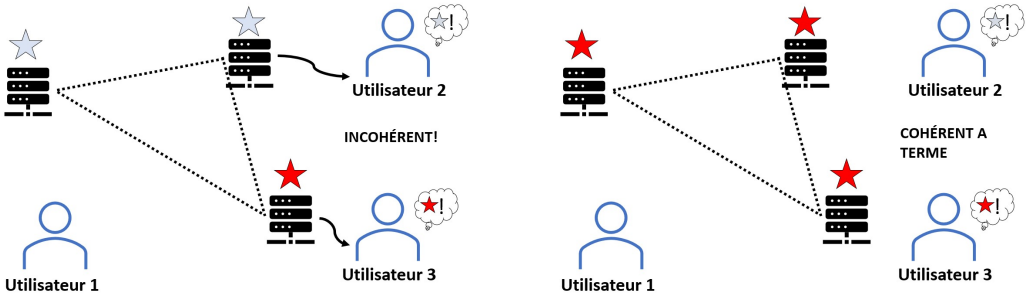


FIGURE 5.1 – Modification d'un réplica



(a) Modification d'un réplica (b) Lecture des réplicas

FIGURE 5.2 – Exemple avec un modèle de cohérence fort



(a) Modification d'un réplica (b) Lecture des réplicas

FIGURE 5.3 – Exemple avec un modèle de cohérence à terme

De nombreux systèmes de stockage de données récents, comme Cassandra [Apa] ou Azure CosmosDB [Mic], permettent aux développeurs de déterminer pour chaque donnée le protocole de cohérence à lui appliquer. Comme nous l'avons détaillé dans le Chapitre 2, il existe de nombreuses méthodes de placement dynamique des données, cependant ces méthodes ne prennent pas en compte les différents modèles de cohérence pouvant s'appliquer au sein d'un même système, et donc les contraintes que ces modèles peuvent imposer sur les données. Par exemple, pour une donnée gérée par un protocole de cohérence forte, le système doit effectuer des consensus entre les réplicas. Afin de permettre à l'utilisateur d'accéder le plus rapidement à une donnée, il faut que ce consensus se fasse le plus rapidement possible. Il faut alors placer les réplicas de manières a minimiser le temps d'un consensus. À l'inverse, pour une donnée avec un modèle de cohérence à terme, placer les réplicas de manière à couvrir une large zone peut fournir de meilleures performances pour les utilisateurs.

Dans la suite de ce chapitre, nous commençons par la présentation de la modélisation utilisée dans nos travaux avant de représenter le problème du placement

des réplicas. Nous présentons ensuite CAnDoR (*Consistency Aware Data Replication*), une méthode de placement dynamique de données capable de s'adapter aux modèles de cohérence utilisés pour maintenir des garanties sur les données et des accès utilisateurs. De cette façon, le placement permet d'offrir de bonnes performances dans les systèmes de gestion de données distribués à modèles de cohérence adaptables. Après avoir décrit le fonctionnement de cette approche, nous discutons des paramètres et de l'impact des fautes sur les calculs.

5.2 Modélisation et problème étudié

CAnDoR est une méthode de calcul de placement des réplicas dans un système de gestion de données distribuées. Cette méthode prenant en compte les contraintes liées aux modèles de cohérence, elle est davantage adaptée aux systèmes capables de fournir plusieurs modèles de cohérence. Cette section décrit le modèle utilisé dans ce chapitre. Nous rappelons que les différents éléments du système n'ont pas accès à une mémoire partagée, les variables utilisées dans ce chapitre sont donc des représentations locales. Nous utilisons la notation $\tilde{}$ pour spécifier les connaissances globales et exactes. Ces notations ne sont utilisées que pour faciliter la compréhension et ne sont jamais utilisées lors des calculs.

5.2.1 Modélisation

Nous considérons un système composé d'un ensemble $S = \{s_1, \ldots, s_{n_s}\}$ de n_s nœuds de stockage et $U = \{u_1, \ldots, u_{n_u}\}$ un ensemble de n_u nœuds utilisateurs. Les nœuds de stockage sont capables de communiquer entre eux et de répondre aux utilisateurs. Les utilisateurs ne peuvent contacter que les nœuds de stockage (ils ne peuvent pas communiquer entre eux). Afin de communiquer, un nœud (de stockage ou utilisateur) doit envoyer un message à travers un lien de communication ou emprunter un chemin de liens de communication si cela est possible. Les liens existants et le temps de communication pouvant évoluer dans le temps, nous associons au système une fonction $\tilde{C} : \mathcal{T} \times S \cup U \times S \cup U \mapsto \mathbb{N} \cup \perp$. Cette fonction associe à un instant t une matrice donnant le temps pour une communication entre deux nœuds et $\perp$ si ces deux nœuds ne peuvent pas communiquer. Ainsi si s_1 envoie un message à u_3 à l'instant t, celui-ci mettra $\tilde{C}(t, s_1, u_3) = \tilde{\lambda}$ ms à arriver.

CAnDoR étant prévu pour s'exécuter dans un système existant, nous supposons que chaque nœud de stockage a accès à une table de routage et donc à un chemin pour contacter un autre nœud, lorsque cela est possible [1]. Un nœud s a également accès à $C^s : \mathcal{T} \times S \cup U \times S \cup U \mapsto \mathbb{N} \cup \perp$ une approximation du temps nécessaire

1. S'il n'y a pas de partition dans le système.

pour contacter les autres nœuds. Un nœud s est donc capable d'estimer le temps pour contacter un autre nœud s_2 en calculant $C^s(t, s, s_2) = \lambda$ ms.

5.2.2 Problème du placement des réplicas

Nous nous intéressons ici au problème du placement des réplicas. Nous disposons d'un ensemble $D = \{d_1, \ldots, d_{n_d}\}$ de n_d données disponibles dans le système. Chaque donnée d_i est répliquée RF_i fois avec un modèle de cohérence entre les réplicas et à un poids p_i. Nous cherchons à placer ces réplicas parmi les n_s nœuds de stockage. Chaque nœud de stockage s_i a une capacité c_i pour stocker ces données. On note $\mathcal{R}^d$ l'ensemble des nœuds de stockage ayant un réplica de d.

Les n_u utilisateurs peuvent effectuer des requêtes de lecture ou d'écriture sur les données. Les données étant soumises à des contraintes de cohérence différentes, les opérations d'écriture et/ou de lecture peuvent entraîner un processus de synchronisation entre les nœuds. Nous nous intéressons en particulier à minimiser le délai entre une requête émise par un utilisateur et la réponse de la part du système. Cette métrique est appelée la latence pour les accès utilisateurs.

Le problème du placement des données dans un système distribué a été très étudié [GM02], [BR01], [PRRR14], [QKD13]. Dans [ST01] et [GKK+09], les auteurs définissent ce problème comme une variante du problème de sacs à dos multiple (KBP) et a été démontré comme étant NP-difficile et peut être posé de manière suivante :

Problème 1 (Placement des réplicas). *Soit un ensemble D de n_d données répliquées RF fois et soit un ensemble S de n_s nœuds de stockage à capacité finie.*

Comment placer les $\sum_{i=0}^{n_d} RF_i$ réplicas des n_d données sur les n_s nœuds de stockage de façon à minimiser les latences médianes pour les accès utilisateurs?

Un descriptif des méthodes courantes pour résoudre ou approximer ce problème est donné dans la Section 2.3.2. Cependant aucune de ces méthodes ne considère le paradigme récent des systèmes proposant différents différents modèles de cohérence pour les données. Or différents modèles de cohérence imposent différentes contraintes sur le placement des réplicas. Nous considérons donc une variante du problème sous la forme suivante :

Problème 2 (Placement des réplicas à modèles de cohérence variables). *Soit un ensemble D de n_d données répliquées RF fois et avec des modèles de cohérence différent. Soit un ensemble S de n_s nœuds de stockage à capacité finie.*

Comment placer les $\sum_{i=0}^{n_d} RF_i$ réplicas des n_d données sur les n_s nœuds de stockage de façon à minimiser les latences médianes pour les accès utilisateurs?

5.3 CAnDoR : Consistency Aware Data réplication

5.3.1 Description de l'approche

Comme nous l'avons précisé dans la Section 3.2.1, le problème du placement Prob. 1 est un problème NP-Difficile. Afin de permettre de mettre en place une méthode de placement automatique des réplicas sans surcharger le système de calculs, nous avons choisi d'aborder le problème "par donnée". Nous commençons par donner une description informelle du fonctionnement de CAnDoR avant de détailler l'approche utilisée.

Nous considérons ici les systèmes non transactionnels où les requêtes sont atomiques : les résultats d'une requête ne dépendent pas des autres requêtes et il n'y a pas de relation entre les différentes données. De cette façon, pour chaque donnée, un nœud s_l est élu comme "responsable" de la donnée. Sous certaines conditions, ce nœud va calculer quel placement serait optimal pour cette donnée. Pour cela, il va évaluer l'espérance des performances de différents ensembles de nœuds pouvant accueillir un réplica de la donnée. À l'issue de ce calcul, le nœud va estimer si le déplacement des réplicas n'est pas trop coûteux pour le gain espéré et va, le cas échéant lancer une procédure de migration. A l'issue de cette migration, un nouveau nœud responsable s'_l peut être déterminé.

Cette approche permet de réduire grandement le nombre de calculs à effectuer par les nœuds : au lieu de calculer le placement de $\sum_{i=0}^{n_d} RF_i$ réplicas, un nœud ne calcul que le placement des données dont il est responsable. De plus, ces calculs ne se font pas nécessairement en même temps, un nœud peut ne calculer le placement que de certaines données dont il est responsable. Nous évoquons dans la Section 5.3.5 quand un tel calcul est lancé. Intuitivement, celui-ci est provoqué en fonction d'un timer local et d'un outil de détection de popularité.

Choisir le nœud responsable d'une donnée

Nous ne détaillons pas ici de méthode précise pour déterminer un nœud s_l responsable des données, mais nous donnons simplement quelques observations. Il est à noter que ce nœud va devoir effectuer les calculs de placement en fonction des métadonnées et des accès utilisateurs. Selon les systèmes, il peut être donc judicieux de choisir spécifiquement un nœud stockant un réplica de la donnée, afin de collecter les accès à la donnée plus facilement ou un nœud capable d'effectuer des calculs sans dégrader les performances du système. Idéalement, s_l devrait pouvoir effectuer efficacement les calculs tout en pouvant échanger rapidement avec les réplicas. Enfin, la charge des responsabilités devrait se faire en fonction des capacités de calcul des nœuds.

Dans la suite de ce chapitre, nous avons considéré que tous les nœuds disposaient de la même capacité de calcul, nous sélectionnons donc un des réplicas arbitrairement pour être responsable. Si ce nœud ne stocke plus de réplica de la donnée après une migration, il détermine un nouveau responsable parmi les nœuds.

Bien que nous n'ayons pas directement étudié ces systèmes, nous pensons que si certaines données sont liées, alors le nœud responsable doit être le même afin d'optimiser le placement des données liées.

5.3.2 Calcul de l'ensemble des nœuds d'accueil potentiels

Le nœud s_l responsable de la donnée d_i va chercher à déterminer quel ensemble de RF_i nœuds permettra d'obtenir les meilleures performances pour le système. Pour cela, il est nécessaire de construire une liste des ensembles de RF_i nœuds potentiels. Nous utilisons actuellement une méthode exacte sur les nœuds ayant la capacité de recevoir la donnée : pour chaque nœud de stockage s_j, s_l vérifie si s_j peut stocker un réplica de d_i, de poids p_i : $c_j - p_i \geq 0$ avec c_j la place restante sur s_j. La liste des ensembles potentiels est alors construite à partir de l'ensemble des combinaisons de RF_i nœuds.

Cette approche présente, bien sûr, le défaut de manipuler une liste en $\mathcal{O}(\binom{n_s}{RF_i}) = \mathcal{O}(n_d!)$, ce qui représente une charge de calcul conséquente pour le nœud responsable.

5.3.3 Phase de propagation et temps de réponse

Lorsqu'une donnée est répliquée dans un système de stockage distribué, il est important de fournir deux niveaux de garanties à l'utilisateur :

1. garanties de disponibilité : ces garanties indiquent à l'utilisateur à quel point un réplica sera disponible. Idéalement, à chaque instant et pour chaque donnée, au moins un réplica devrait être accessible et avec un délai de traitement minimal (garanties de hautes disponibilités).
2. garanties de cohérence : ces garanties concernent les différences de versions entre les réplicas. Idéalement, à tout moment, tous les réplicas d'une même donnée devraient être à jour et dans un état équivalent (garantie de cohérence forte).

Dans la pratique, et comme expliqué dans le Chapitre 4, il est impossible de fournir à la fois des garanties de hautes disponibilités et de cohérence forte. Plus précisément, le théorème du CAP Th. 2 et ses variantes, détaillent qu'il est nécessaire de considérer un compromis entre disponibilité, cohérence et tolérance aux partitions selon les besoins de l'application [GL02]. Or, nous considérons ici les

systèmes de gestion de données distribués et de tels systèmes ne peuvent pas se passer de la tolérance aux partitions. Ces systèmes vont donc chercher le “meilleur” compromis entre disponibilité et cohérence.

La plupart des applications commencent par poser la contrainte la plus forte (un besoin de haute disponibilité ou de cohérence forte) et proposent de faire au mieux pour la seconde contrainte. Par exemple, un service bancaire distribué doit fournir des informations fiables à ses utilisateurs. Les réplicas des données doivent alors être dans des états équivalents en permanence et le service fournira une latence aussi faible que possible dans ces conditions. On dit que les données de ce service sont gérées par un modèle de cohérence forte et de disponibilité au mieux. À l'inverse, une application comme Facebook préfère garantir à ses utilisateurs un service aussi rapide que possible au détriment de quelques incohérences temporaires entre 2 utilisateurs. On dit que les données de ce service sont gérées par un protocole de cohérence à terme et avec une grande disponibilité. Cependant, toutes les applications cherchent à proposer le plus petit délai de traitement possible en respectant ces contraintes. Le placement des données est alors primordial à la qualité d'un service : des réplicas mal placés peuvent entraîner de mauvaises performances malgré le respect des protocoles.

La cohérence entre les réplicas est assurée par un algorithme de propagation de l'information. On appelle *phase de propagation* la période d'exécution de cet algorithme. Dans le cas de modèle de cohérence forte, cette phase est bloquante : les nœuds ne répondent pas aux utilisateurs tant qu'il n'est pas garanti que tous les réplicas soient dans un état équivalent. Le temps de propagation est illustré par la Figure 5.4. Dans cet exemple, l'utilisateur 1 effectue une requête de mise à jour d'une donnée. On appelle phase de propagation la période durant laquelle cette requête est en transit du nœud 1 vers les nœuds 2 et 3. A la fin de cette phase, tous les nœuds ont connaissances de la requête de l'utilisateur 1.

La disponibilité d'une donnée peut se mesurer par une étude sur le temps de réponse du système à une requête, c'est ce qu'on appelle *le temps de réponse*, illustré par la Figure 5.5. Dans cet exemple, l'utilisateur 1 effectue une requête de mise à jour d'une donnée et l'utilisateur 2 effectue une requête de lecture. Les nœuds peuvent attendre ou non la fin de la propagation pour répondre à l'utilisateur 2. Le temps de réponse correspond à la période pendant laquelle l'utilisateur 2 doit attendre avant d'obtenir une réponse de la part d'un nœud. Les temps de communication peuvent être victimes d'événements extérieurs au système, les faisant ponctuellement varier de manière anormale. Nous nous intéressons donc ici au quatre-vingt-quinzième centile des temps des délais de traitement de requêtes.

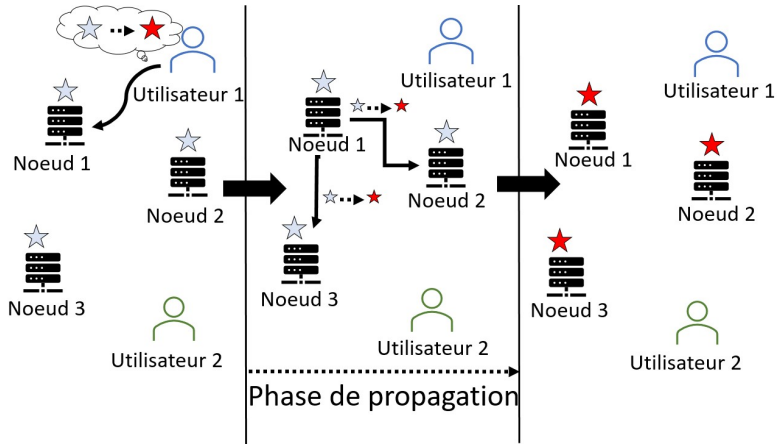


FIGURE 5.4 – Illustration de la phase de propagation

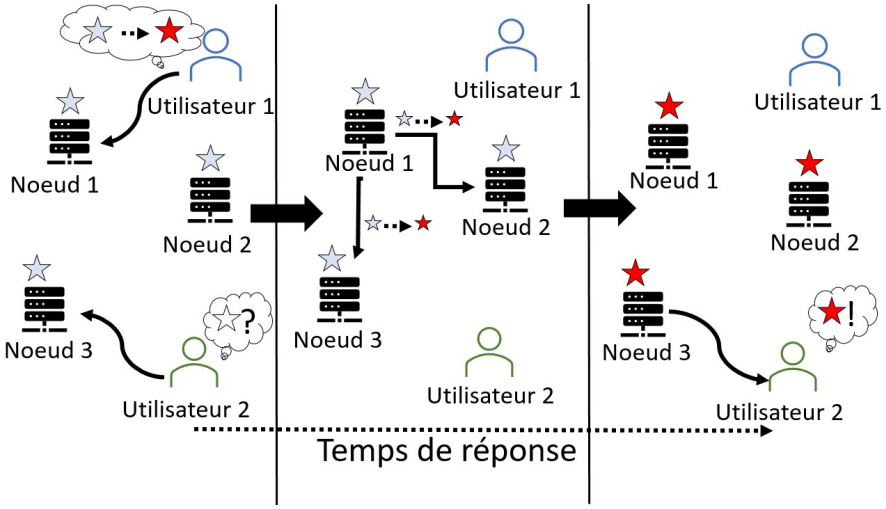


FIGURE 5.5 – Illustration du temps de réponse

Estimation de la durée de la phase de propagation

Lorsqu'un utilisateur u émet une requête α sur une donnée d, cette requête est reçue par un nœud s stockant un réplica de d. s doit alors envoyer cette requête aux autres nœuds stockant un réplica de d. On appelle phase de propagation la période durant laquelle cette requête est transmise aux autres réplicas et on note t_p sa durée.

Dans un système distribué, les communications se faisant en parallèle, le temps pour propager la requête à tous les nœuds est majoré par le temps nécessaire le plus long pour contacter un nœud. Certains protocoles de propagation peuvent demander plusieurs vagues de messages [Lyn96]. Nous posons k le nombre de mes-

sages à envoyer pour valider une propagation. Formellement, si s veut propager α aux autres nœuds stockant un réplica de d $s_i \in \mathcal{R}^d$ à un instant t, on obtient le temps de propagation d'un nœud $t_p(t, s, \mathcal{R}^d)$ suivant :

$$t_p(t, s, \mathcal{R}^d) = k. \max_{\substack{s_i \in \mathcal{R}^d \\ s_i \neq s}} [C^{s_l}(t, s, s_i)]$$

Afin de déterminer si un ensemble de nœuds est un bon ensemble pour maintenir les réplicas, nous intéressons au temps de propagation pour chaque nœud de l'ensemble. En effet, tous les nœuds sont susceptibles de recevoir des requêtes et de donc devoir les propager. Le temps de propagation d'un ensemble $t_p(t, \mathcal{R}^d)$ est le temps nécessaire pour que tous les nœuds de l'ensemble puissent propager une requête. Une fois encore, ce temps est donné par le temps maximum des temps de propagation :

$$t_p(t, \mathcal{R}^d) = \max_{s_i \in \mathcal{R}^d} [t_p(t, s_i, \mathcal{R}^d)]$$

Estimation de la durée du temps de réponse

Lorsqu'un utilisateur u émet une requête α sur une donnée d, il attend une réponse de la part du système. Typiquement, si α est une requête d'écriture, u attend un message de validation ou le nouvel état d'un réplica de d. Si α est une requête de lecture, u attend l'état actuel d'un des réplicas de d. Selon le modèle de cohérence suivi par la donnée, cette réponse peut survenir après la propagation de la requête. On pose b_d un coefficient indiquant si les nœuds doivent terminer la propagation avant de répondre à une requête sur d.

Il existe différents protocoles d'implémentation du traitement d'une requête, par exemple les nœuds peuvent communiquer entre eux et envoyer une seule réponse à l'utilisateur ou, dans le cas d'un modèle par quorum, l'utilisateur attend une réponse de la part d'un quorum de lecture ou d'écriture. Posons q le nombre de messages qu'attend u pour valider sa requête de lecture (dans le cas ou les nœuds communiquent entre eux et envoient une seule réponse, $q = 1$). Le temps de réponse d'une requête de l'utilisateur u à un instant t est noté $t_r(t, u, \mathcal{R}^d)$ et correspond au temps nécessaire pour recevoir q messages. En notant $min_{(q)}(X)$ la $q^{\text{ième}}$ plus petite valeur de l'ensemble X on peut calculer le temps de réponse avec la formule suivante :

$$t_r(t, u, \mathcal{R}^d) = b_d.t_p(t, \mathcal{R}^d) + \min_{\substack{(q) \\ s_i \in \mathcal{R}^d}} [C^{s_l}(t, u, s_i)]$$

Afin d'évaluer la performance d'un ensemble pour répondre aux requêtes, il faut considérer que les requêtes peuvent être émises par n'importe quel utilisateur. On note w_i le nombre de requêtes émises par l'utilisateur u_i. Il est possible de

considérer deux métriques pour l'évaluation des performances d'un ensemble de nœuds pour traiter ces requêtes :

- considérer tous les utilisateurs comme équivalent, on estime alors le temps nécessaire pour répondre à tous les utilisateurs et on retient le temps maximal :
$$t_r(t, \mathcal{R}^d) = \max_{u_i \in U}[t_r(t, u_i, \mathcal{R}^d)]$$
- favoriser les utilisateurs en fonction de leur activité. On calcule alors la somme des temps de réponse pour chaque utilisateur en les pondérant par une fonction de l'activité f. Cette fonction est à déterminer par le concepteur du système et témoigne de l'importance donnée aux utilisateurs. On peut alors estimer le temps de réponse d'un ensemble de nœuds par :
$$t_r(t, \mathcal{R}^d) = \sum_{u_i \in U}[f(w_i).t_r(t, u_i, \mathcal{R}^d)]$$

5.3.4 Pondérer les estimations

L'estimation des temps de propagation et de réponse va permettre au nœud s_l responsable d'une donnée d'estimer quels ensembles sont de bons candidats pour stocker les réplicas. Afin de déterminer l'ensemble le plus efficace possible, il convient cependant de considérer les contraintes liées au modèle de cohérence imposé à la donnée ainsi qu'à l'activité des utilisateurs. La Figure 5.6 illustre un exemple simple de problème de placement. Nous disposons d'une donnée à répliquer 2 fois parmi 3 nœuds de stockage potentiels. Les temps de communication sont indiqués sur la figure. La Figure 5.7a illustre le cas où les réplicas sont placés proche les uns des autres. De cette façon, la propagation des opérations sera rapide, avec une estimation de 4 ms. Les utilisateurs 2 et 3 auront également un accès très rapide à un réplica, estimé à 4 ms et 2 ms respectivement. En revanche l'utilisateur 1 souffre d'un temps d'accès plus long, estimé à 127 ms. La Figure 5.7b illustre une stratégie différente : les réplicas sont espacés afin que tous les utilisateurs aient un délai estimé à moins de 50 ms. La stratégie à choisir pour le placement va dépendre de différents paramètres du système : modèle de cohérence entre les réplicas ou activité des utilisateurs ainsi que de l'objectif du système.

Contraintes du modèle de cohérence

Dans la plupart des cas, lorsqu'une donnée est soumise à un modèle de cohérence forte, le système doit effectuer des synchronisations lors de la modification d'un réplica. Durant cette synchronisation, les utilisateurs ne peuvent plus accéder à la donnée, sous peine de violer les garanties de cohérence. Afin de réduire le temps de ce blocage, il est nécessaire de réduire le temps d'une phase de propagation.

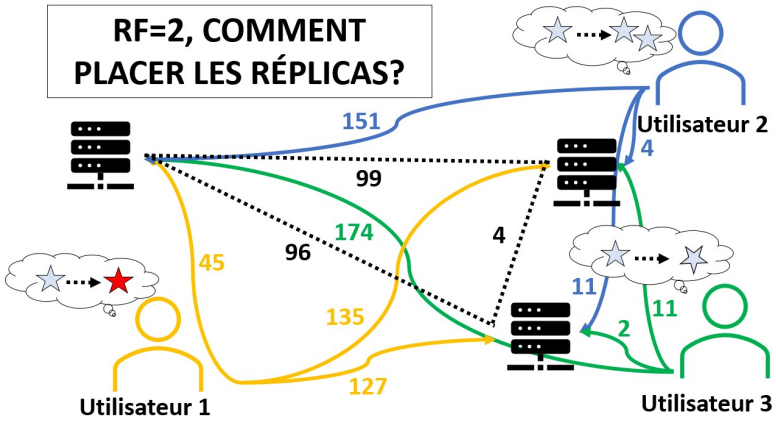
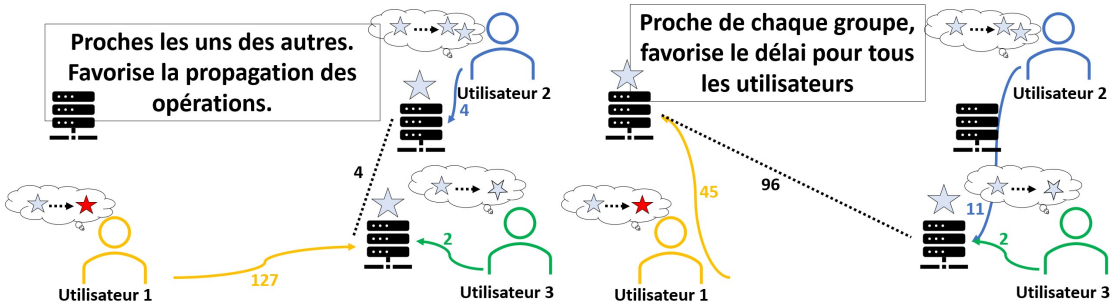


FIGURE 5.6 – Exemple de placement de réplicas



(a) Modification d'un réplica (b) Lecture des réplicas

FIGURE 5.7 – Exemple de l'impact du placement des réplicas

À l'inverse, une donnée gérée par un protocole de cohérence à terme cherchera à minimiser le temps de réponse aux utilisateurs, au détriment de la durée de propagation, pouvant ainsi entraîner des différences entre réplicas.

Pour répondre à ces besoins, nous posons deux coefficients c_c, c_a pour symboliser l'importance du temps de propagation et de réponse respectivement. Pour des raisons de lisibilité, nous considérons ici que ces coefficients sont des pourcentages, soit $c_c + c_a = 100$. Ces coefficients sont propres à chaque donnée et ne devraient être modifiés que si les garanties de cohérences et disponibilités sont mises à jour. Ces deux coefficients représentent donc les contraintes liées au modèle de cohérence et vont permettre de moduler la distance entre les nœuds stockant un réplica : plus c_c est élevé, plus un poids fort sera donné au temps de propagation. Ce temps est réduit en "approchant" les réplicas les uns des autres. De manière symétrique, plus c_a est élevé, plus le poids du temps de réponse sera élevé et les nœuds choisis auront tendance à être proche d'un maximum d'utilisateurs actifs.

Activité des utilisateurs

En plus des coefficients c_c, c_a qui représentent les contraintes de cohérence, les algorithmes de CAnDoR utilisent également des indicateurs d'activité des utilisateurs pour s'assurer de placer les réplicas au plus proche des utilisateurs actifs. Pour cela, nous nous sommes basés sur deux principes :

- plus un utilisateur est actif, plus un réplica doit être proche de cet utilisateur ;
- plus une donnée est accédée en écriture, plus les réplicas doivent être proches les unes des autres.

Le premier principe permet simplement de diminuer les temps d'accès pour les utilisateurs les plus actifs, et donc ceux profitant au plus d'un gain de performance. Le second principe est issu de l'observation suivante : plus une donnée est accédée en écriture, plus il y aura de phases de propagation. Il sera alors souhaitable de rapprocher les réplicas afin de minimiser le temps passé à propager des informations dans le système. Ce principe permet, par exemple, de disperser des réplicas près des utilisateurs lorsque celles-ci sont accédées principalement en lecture, même si elles sont gérées par un protocole de cohérence forte. Inversement, dans un système à cohérence à terme, si les données sont principalement accédées en écriture, ce principe permet de rapprocher les réplicas et donc de limiter les différences entre les réplicas.

Nous posons donc les ratios $\frac{h_w}{h_w+h_r}$ et $\frac{h_r}{h_r+h_w}$ avec h_w, h_r respectivement le nombre de requêtes en lecture et en écritures contenues dans l'historique des événements. La durée des phases de propagations et les temps de réponse aux utilisateurs sont donc pondérés par les coefficients suivants :

$$\begin{cases} w_p & = c_c \cdot \frac{h_w}{h_r+h_w} \\ w_r & = c_a \cdot \frac{h_r}{h_r+h_w} \end{cases}$$

5.3.5 Calcul du placement

Afin de déterminer quel ensemble de nœuds doit stocker les réplicas d'une donnée d, s_l parcourt la liste des ensembles de $\mathcal{LR}$ de façon à déterminer l'ensemble $\mathcal{R}^*$ qui permet d'obtenir les meilleures estimations de performances. L'estimation de performances d'un ensemble $\mathcal{R}$, notée $E(\mathcal{R})$, est calculée en sommant le carré des temps de propagation et de réponse. Afin de respecter les observations précédentes, le temps de propagation est pondéré par w_p et le temps de réponse par w_r.

$$E(\mathcal{R}_i) = w_p.t_p(\mathcal{R}_i)^2 + w_r.t_r(\mathcal{R}_i, U)^2$$
$$\mathcal{R}^* = \min_{\mathcal{R}_i \in \mathcal{LR}} [E(\mathcal{R}_i]$$

Une fois que s_l a déterminé $\mathcal{R}^*$ l'ensemble des nœuds devant stocker un réplica de d, s_l estime si la migration est pertinente ou si celle-ci est trop coûteuse. Pour cela, s_l calcule la performance estimée de l'ensemble $\mathcal{R}_{cur}$ stockant actuellement les réplicas. Si le gain de performances $g(\mathcal{R}^*)$ est suffisamment grand par rapport au coût de la migration, alors s_l déclenche une migration des réplicas et détermine un nouveau nœud responsable de d.

$$g(\mathcal{R}^*) = E(\mathcal{R}^*) - E(\mathcal{R}_{cur})$$

Déclenchement du calcul de $\mathcal{R}^*$

Afin de déterminer l'ensemble $\mathcal{R}^*$ des nœuds devant stocker un réplica de la donnée d_i, s_l parcourt la liste des ensembles potentiels. Cette liste étant actuellement de taille $\mathcal{O}(\binom{n_d}{RF_i})$, ce calcul peut être coûteux. Il convient donc de ne pas le déclencher trop régulièrement. Notre méthode repose actuellement sur un timer local indiquant la date du dernier calcul de placement et d'un indicateur de popularité. Si une de ces deux valeurs atteint un seuil prédéterminé, alors s_l commence un calcul de placement. Ce mécanisme permet de calculer en priorité le placement des données les plus populaires dans un système, ces données étant celles influençant le plus les performances du système.

5.3.6 Impact des fautes sur CAnDoR

CAnDoR a été développé pour s'exécuter au sein d'un système de gestion de données distribué. Nous avons donc supposé que des fautes pouvaient intervenir, mais que celle-ci était prise en charge par le système : c'est-à-dire que le système est capable de détecter les fautes et que les nœuds sont prévenus des fautes des autres nœuds. Nous étudions dans cette section l'impact des fautes de type panne sur un des nœuds stockant un réplica de d.

Faute d'un réplica non responsable

Lorsqu'un nœud tombe en panne, si celui-ci n'était pas responsable de la donnée d, nous considérons que le nœud responsable de d finira par être notifié de cette faute. Dans ce cas, une étape de réplication est lancée afin de retrouver le bon nombre de réplicas. Comme un nouveau réplica a été ajouté au système, le nœud responsable doit déterminer s'il faut relancer un calcul de placement. Il utilise pour cela ses compteurs locaux. S'il n'apparaît pas nécessaire de relancer un calcul de placement, le nœud responsable va envoyer le nouveau réplica sur le nœud le proche du nœud fautif.

Afin d'affiner cette approche, le nœud responsable pourrait disposer d'outils permettant de déterminer si le nœud fautif dispose d'un voisinage pouvant le rem-

placer sans détériorer les performances ou si cette faute doit forcement faire l'objet d'un nouveau calcul de placement.

Faute du réplica responsable

Si le nœud responsable d'une donnée d tombe en panne, alors il faut déterminer un nouveau nœud responsable. Comme nous estimons que les fautes sont prises en charge par le système, les autres nœuds stockant un réplica de d finiront par être prévenus. Un nouveau responsable doit alors être déterminé entre eux. Une fois qu'un nouveau responsable est mis en place, nous utilisons le même mécanisme que précédemment.

Les calculs de CAnDoR, tels que décrit ici, sont centralisés sur le nœud responsable. Cependant, ces calculs correspondent au problème de recherche d'une valeur minimale parmi des valeurs calculée indépendamment. Il existe de nombreuses méthodes efficaces pour paralléliser ce problème. Ainsi chaque nœud portant un réplica de d pourrait effectuer une recherche sur une partie des ensembles potentiels et regrouper les résultats. Cette approche permet de minimiser l'impact d'une faute du nœud responsable si celui-ci effectuait un calcul de placement. En effet, seuls ses calculs seraient perdus dans ce cas. Cette méthode ne devrait cependant pas améliorer les performances du service dans les autres situations : les nœuds doivent communiquer davantage pour partager leurs résultats. Et dans la plupart des systèmes existants, chaque nœud physique devrait être responsable de plusieurs données. Ce mécanisme ne permet donc pas d'alléger sa charge de calcul.

5.4 Conclusion

Nous avons présenté, dans ce chapitre, CAnDoR, une méthode de placement automatisée des différents réplicas des données. Cette méthode utilise une approche inspirée de la programmation linéaire entière, couramment utilisée pour le problème de placement des données. Afin de faire face à la NP-difficulté du problème [ST01], nous abordons le problème du point de vue de la donnée.

Plutôt que d'essayer de placer tous les réplicas de toutes les données à un instant précis, pour chaque donnée dans le système, un noeud est élu responsable de cette donnée et va calculer comment ses réplicas doivent être placés. Ce calcul prend en compte à la fois le modèle de cohérence de la donnée et l'activité des utilisateurs. En effet, les différentes analyses des modèles de cohérences nous ont permis de déterminer que les différents modèles de cohérence peuvent mener à différentes stratégies de placement (nombre de réplicas, proximité entre les réplicas et les utilisateurs, localité des réplicas, etc.). Le nœud responsable de la donnée calcule des estimations des performances des ensembles pouvant accueillir les réplicas en

estimant les temps de propagation et de réponse de ces ensembles et en pondérant ces estimations selon différents facteurs.

Le calcul de placement se fait sur une base périodique, mais peut être retardé ou provoqué selon le nombre de requêtes faites sur la donnée. De cette façon, une donnée peu intéressante pour les utilisateurs ne consommera que peu de puissance de calcul, qui sera allouée aux placements des données effectivement accédées dans le système.

Cette approche permet donc de déterminer un placement des réplicas permettant de réduire le temps de réponse des requêtes dans le système. Il reste cependant différentes pistes à explorer pour affiner les calculs, notamment le calcul de l'ensemble des nœuds de stockage potentiels ou de la concurrence des opérations pour le placement de données différentes. Enfin, CAnDoR se basant sur l'activité des utilisateurs pour ses calculs, il est important d'être capable d'établir une vue pertinente du présent et de l'activité des utilisateurs en fonction de leur action passées. Nous proposons donc dans le chapitre suivant un étude de la gestion de la mémoire et différentes méthodes pour utiliser de manière pertinente les informations sur les événements passés.

Sommaire

Chapitre

6

Observation d'un système : Quel poids donner au passé ?

6.1 Introduction

De nos jours, il n'est pas rare que des applications aient recours à un système de gestion de données distribué dans le cloud. Ces systèmes font face à un environnement continuellement changeant et hétérogène. L'architecture matérielle évolue rapidement avec des arrivées et départs (volontaires ou non) de machines. La charge de travail est également souvent hautement dynamique en termes de fréquence, de localité, de type, etc. Lors de la conception de CAnDoR, la méthode de placement des données dans un système large échelle présentée dans le Chapitre 5, nous avons été confronté à de telles problématiques. La stratégie de prise en compte des requêtes passées et les positions des nœuds du système peut avoir une influence sur les résultats obtenus : accorder une trop d'importance à de vieux événements peut conduire le placement à défavoriser les utilisateurs récents ; à l'inverse ne considérer que les événements les plus récents peut empêcher CAnDoR de s'adapter à certains comportements. Nous avons alors mené une étude analytique sur les différentes stratégies de prise en compte des événements passés. Nous pensons que cette analyse s'applique à tout mécanisme d'adaptation d'un système dynamique, ce chapitre propose alors une approche abstraite de la gestion des événements, détachée d'un service spécifique.

De plus, lorsqu'un service s'exécute sur une architecture hétérogène, chaque point de calcul dispose d'une puissance et d'une capacité qui lui sont propres. Il est donc nécessaire de s'adapter à ces différences de caractéristiques en étant capable, par exemple, de redistribuer les tâches à accomplir ou de les assigner au fur et a mesure.

Ce mécanisme permet également de faire face à des arrivées et départs des points de calcul. De plus, certaines applications, comme Facebook, doivent faire face à des comportements très différents de la part de leurs utilisateurs en fonction de l'heure de la journée et de la localité de ceux-ci. Les applications très populaires doivent aussi faire face aux phénomènes de *buzz* : une donnée, ou un ensemble de données, devient le centre d'intérêt d'un grand nombre d'utilisateurs, pour une durée généralement courte. Ces phénomènes sont imprévisibles et peuvent conduire à une saturation des serveurs ou des canaux de communications.

De plus il n'est pas possible de poser d'hypothèses sur la vitesse, la capacité ou même la durée de vie des différents éléments du système. Les différents nœuds du système exécutent les différentes tâches de manière asynchrone sans avoir de connaissance globale précise du reste du système. Seule une connaissance globale approximative est obtenue au fil de l'exécution et des échanges avec les autres participants.

Adaptation

Malgré cela, les systèmes distribués se doivent de continuer à offrir un service fiable et de qualité. Pour ce faire, il est souvent nécessaire d'adapter la configuration du système afin de correspondre au mieux à l'environnement courant. Cette adaptation peut être très différente d'un service à l'autre, et peut alors avoir des coûts tout aussi différents. Par exemple, un service de détection de fautes cherche à déterminer quels nœuds sont présents dans le système, et parmi eux ceux qui ne sont pas fautifs. Pour cela il va maintenir une vision locale du système en se basant sur le temps entre deux messages, leur nombre, etc. En revanche, un système de stockage distribué doit adapter le nombre et le placement des réplicas des données en fonction de la volatilité (tolérance aux fautes), de la localité des accès, de leur fréquence, de leur type (lecture/écriture), du protocole de cohérence entre ces données, etc. Sans cela, de nombreux accès risquent de souffrir de mauvaises performances. Aussi, de nombreux systèmes distribués s'adaptent à leur environnement changeant : [YYLC10], [GCG01], [SM03], etc.

Surveillance

Afin de s'adapter, un système doit avoir une connaissance de son environnement. Idéalement, il devrait connaître les futures évolutions de la plate-forme et la future utilisation du service. Comme cela est évidemment impossible, les services

se contentent d'utiliser les informations passées. En pratique, chaque nœud d'un système distribué maintient en général une vision partielle de son environnement et extrapole le futur en se basant sur l'observation du passé. Ce comportement repose sur l'observation que les utilisateurs ont souvent des "habitudes" et que celles-ci évoluent progressivement dans le temps, et rarement de manière radicale.

C'est, par exemple, ce que font les politiques de gestion de cache, conserver les données récemment accédées, car elles estiment qu'elles risquent d'être utilisées à nouveau. C'est également le cas de certains détecteurs de fautes avancés qui estiment l'heure d'arrivée des prochains heartbeats de chaque nœud surveillé en fonction des précédentes réceptions [HDYK04], [BMS02], etc.

Il existe un grand nombre de méthodes pour surveiller l'environnement et potentiellement partager la connaissance entre les nœuds. La question se pose alors de savoir quelle connaissance prendre en compte. Faut-il prendre en compte tout le passé ? Seulement le passé récent ? Donne-t-on plus d'importance aux événements récents ? Prend-on en compte la fréquence des événements ? Dans ce chapitre, nous proposons une représentation formelle de la manière dont le passé est pris en compte pour construire l'historique des événements, et des différentes stratégies d'utilisation de cet historique.

6.2 Construction et utilisation de l'historique des événements

6.2.1 Représentation de la mémoire

Soit $\mathcal{S}$ un service dynamique opérant sur un système à large échelle. Ce service s'exécute sur une période de temps $\mathcal{T} = [0; t_\infty]$. Chaque nœud du système participant à l'exécution du service dispose d'une instance locale de ce service. Nous considérons par la suite le point de vue d'un nœud, et donc, les exécutions *locales* de $\mathcal{S}$.

Chaque instance locale de $\mathcal{S}$ doit construire un historique local des événements. Cet historique est composé de l'ensemble des événements dont le nœud a connaissance et des poids qui leur sont associés. Habituellement, un événement e est défini par un triplet (a_e, s_e, t_e) composé de l'action effectuée a_e, la source de cette action s_e et la date à laquelle cette action a été effectuée $t_e \in \mathcal{T}$. Selon les besoins du service, certaines composantes peuvent être ajoutées ou modifiées, en particulier, il est possible de considérer un destinataire d_e pour les messages. Dans la suite de ce chapitre, nous considérons, sauf mention contraire, des événements définis par le triplet présenté. Afin de déterminer le poids de chaque événement dans l'historique, $\mathcal{S}$ utilise une fonction de pondération, notée $p_\mathcal{S}$, qui fournit une valeur entre 0 et 100 en fonction du temps passé depuis l'événement. Intuitivement, ce

poids correspond au pourcentage de prise en compte par le service et permet de définir la stratégie d'utilisation de l'historique dans les différents calculs de $\mathcal{S}$, en particulier lorsque celui-ci doit déterminer s'il faut, ou non, lancer une adaptation de la configuration du système.

Nous nous intéressons particulièrement à trois stratégies d'utilisation de l'historique. Pour cela, et afin d'illustrer nos propos, nous considérons trois services qui ne diffèrent que par leur stratégie d'utilisation de l'historique, et donc de leur fonction de pondération :

1. $\mathcal{S}_{IT}$: un service utilisant l'historique de manière intemporelle, c'est à dire utilisant tous les événements sans considérer leur date ;

2. $\mathcal{S}_{FT}$: un service utilisant l'historique en se basant sur une fenêtre glissante, c'est à dire utilisant seulement les événements ayant eu lieu récemment ;

3. $\mathcal{S}_{EE}$: un service utilisant l'historique en considérant des événements évanescents, dont le poids diminue avec le temps.

Lorsque l'un de ces services apprend l'existence d'un événement e à un instant t_e, il l'ajoute à son historique. t_e désigne ici la date de l'événement, et non la date à laquelle le service apprend son existence. Lors de l'utilisation de l'historique, à un instant $t \in \mathcal{T}$, il pondère les informations de l'événement e par le résultat de $p^t_{\mathcal{S}}(t_e)$.

Nous étudions les différences de ces systèmes sous différents scénarios simplifiés de la part des sources d'événements du système. Nous considérons que les sources peuvent avoir deux types de comportements : actif ou passif sur une période, i.e. être régulièrement à l'origine d'événements ou non sur cette période. Selon la nature du système, il peut être nécessaire de faire la distinction d'activité pour une cible précise. Par exemple, un utilisateur peut être à l'origine de requêtes pour une donnée d_1 mais ne jamais faire de requêtes vers une autre donnée d_2, dans ce cas, il est précisé que cet utilisateur est actif pour d_1 et passif pour d_2. Nous utilisons ce type de comportement pour décrire les scénarios présentés dans le Chapitre 7 :

1. Comportement stable : les sources (ou utilisateurs) ont un comportement constant durant la totalité de l'exécution. C'est-à-dire qu'une source active pour une cible à un instant quelconque t l'est (et l'a été) à tout instant de l'exécution. Ce scénario illustre en particulier l'utilisation d'un service par des utilisateurs fixes et est représenté par la Figure 6.1 où l'utilisateur 1 est la seule source active ;

2. Comportement en alternance : deux groupes de sources changent de comportements à intervalle régulier. À chaque nouvel intervalle, le groupe actif

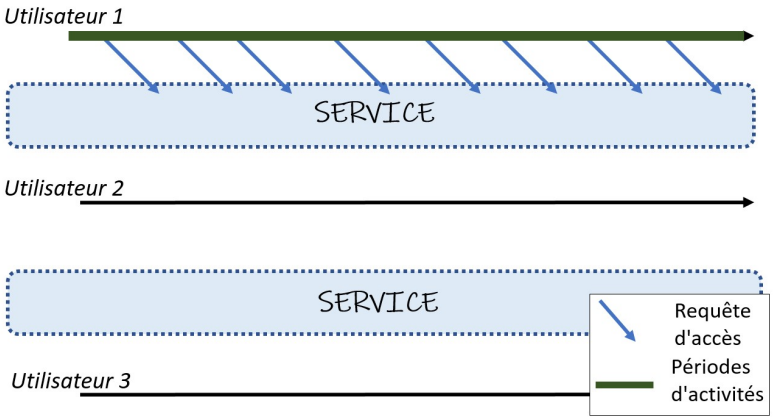


FIGURE 6.1 – Illustration du scénario 1

(pour une cible) devient inactif et inversement. Certaines sources peuvent rester inactives sur la durée de l'exécution. Ce scénario est illustré par la Figure 6.2 où les utilisateur 1 et 2 alternent les périodes d'activité pendant que l'utilisateur 3 reste inactif ;

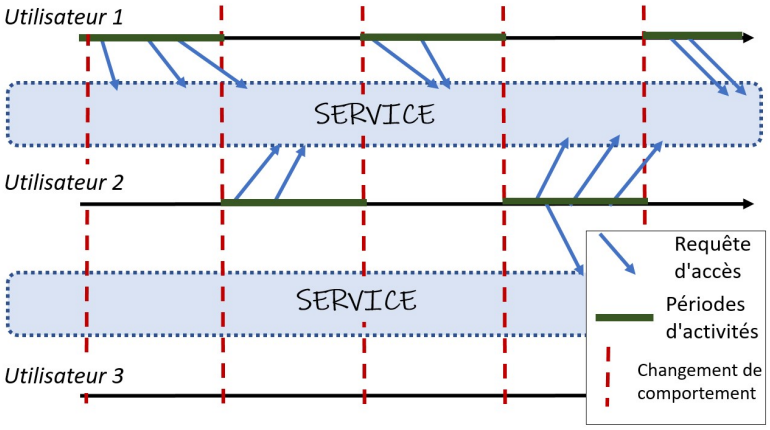


FIGURE 6.2 – Illustration du scénario 2

3. Comportement en alternance déséquilibrée : deux groupes de sources sont alternativement actifs sur des intervalles de temps propre à chaque groupe. Ce scénario représente, par exemple, une activité d'un groupe d'utilisateur utilisant un service à deux locations différentes, comme un lieu de travail et un domicile et est illustré par la Figure 6.3 où les utilisateur 1 et 2 alternent les périodes d'activités pendant que l'utilisateur 3 reste inactif ;

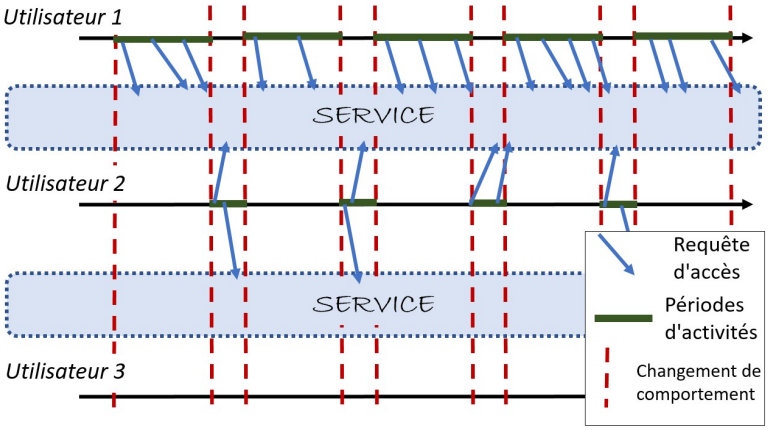


FIGURE 6.3 – Illustration du scénario 3

4. Comportement irrégulier : les sources n'ont pas d'habitude particulière et peuvent changer de comportement à tout moment. Ce scénario est illustré par la Figure 6.4.

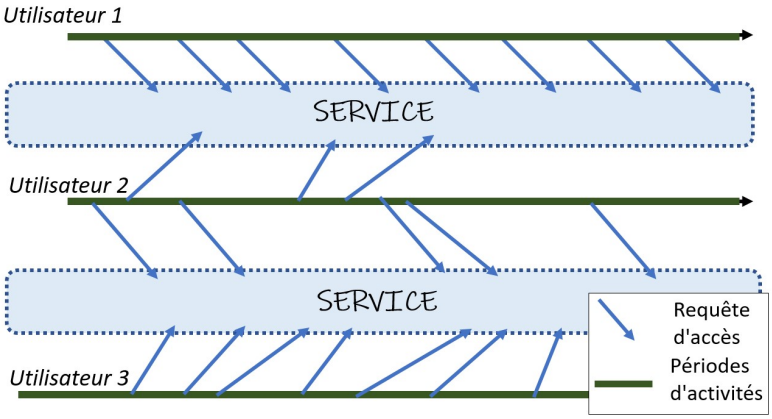


FIGURE 6.4 – Illustration du scénario 3

Afin d'illustrer les propos de ce chapitre nous illustrons l'historique des événements avec la droite de la Figure 6.5. Cette vue représente l'historique des événements au temps t_k. Nous utiliserons ce schéma pour détailler le poids donné aux événements entre e_i et e_{i+7} en fonction des différentes stratégies d'utilisation.

FIGURE 6.5 – Représentation des événements au cours du temps entre 0 et t_k

6.2.2 Utilisation intemporelle de l'historique

Concept de la stratégie intemporelle

Une stratégie intuitive d'utilisation de l'historique des événements est de simplement considérer tous les événements avec un poids équivalent. Ainsi, si le service cherche à connaître, par exemple, le nombre de requêtes de lecture d'un objet par les utilisateurs, il suffit de compter le nombre d'événements correspondant à ce profil. Cette approche ne considère pas "l'âge" des événements en question et utilise donc la fonction constante comme fonction de pondération : $\forall t_i, \in \mathcal{T} | t_i < t, p^t_{\mathcal{S}_{IT}}(t_i) = 100$, avec t la date actuelle et t_i la date de l'événement e_i. Ce fonctionnement est illustré sur la Figure 6.6. Cette stratégie est particulièrement adaptée lorsque le service cherche à établir des statistiques objectives du passé, comme la durée de vie moyenne d'un nœud ou le nombre de requêtes faites sur une donnée. Ainsi c'est une stratégie à considérer pour un traitement *post-mortem* des événements : étudier ce qu'il s'est passé lors de l'exécution.

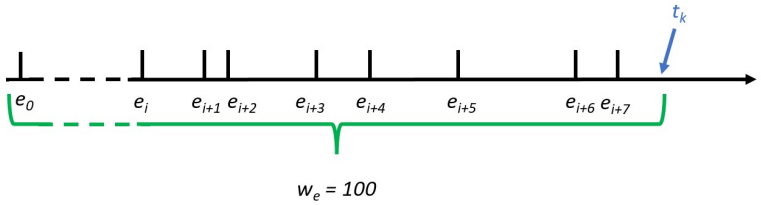


FIGURE 6.6 – Poids des événements au temps t_k avec une stratégie intemporelle

Lors d'une utilisation sur un service en cours, la stratégie intemporelle présente l'inconvénient majeur de pouvoir souffrir d'une forte inertie. En effet, par la nature de la stratégie, en donnant un poids égal à chaque événement du système, les traitements sont incapables de différencier un événement récent d'un événement datant du début de l'exécution. Cette particularité rend impossible l'adaptation rapide du système. Lors du calcul pour déterminer si le système doit, ou non, changer de configuration, seul le nombre d'événements ayant eu lieu aura une influence. Cette stratégie privilégie donc fortement les sources ayant été les plus

actives depuis le début de l'exécution sans considérer l'activité effective au moment du calcul. Ainsi une source ayant été très active au début de l'exécution sera davantage mise en avant, au contraire d'une source active depuis peu.

Intérêt de la stratégie intemporelle

Cette stratégie peut s'avérer adaptée si les sources ne changent pas ou peu de comportements, comme décrit lors des scénarios à comportement régulier ou en alternance déséquilibrée. En effet dans ces deux scénarios, certaines sources sont actives sur la majorité de l'exécution, et seront donc privilégiées lors des calculs, à tout moment de l'exécution. Par exemple, dans le cas d'un service de placement dynamique de données, les données seront placées près des utilisateurs *globalement* plus actifs et non pas des utilisateurs les plus actifs au moment du calcul pour déterminer un nouveau placement. Ceci permet d'avoir un placement robuste aux petites variations d'utilisations et aux fautes temporaires lors de l'exécution, comme illustré sur la Figure 6.7.

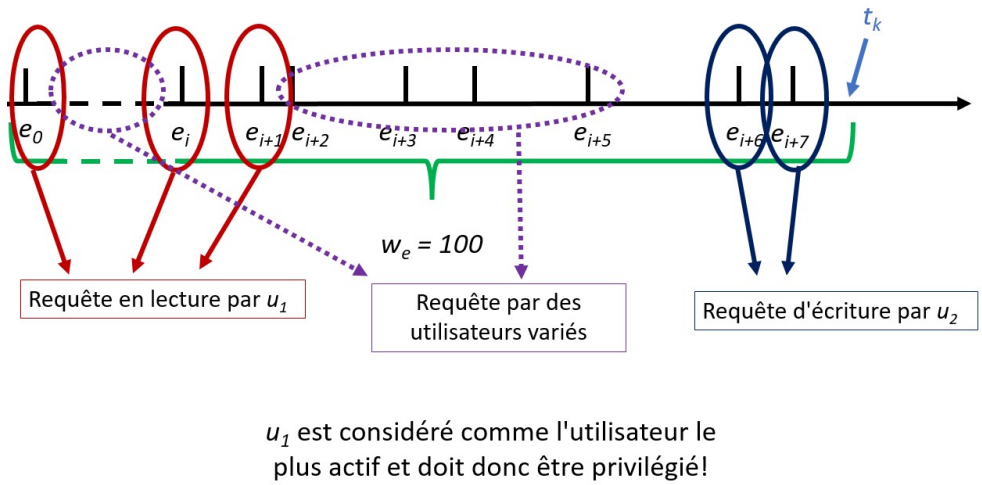


FIGURE 6.7 – Conséquence d'une stratégie intemporelle

Dans le cas d'applications de gestion de données, chaque utilisateur accède généralement à un ensemble restreint de données qui lui est propre et chaque donnée est accédé par un ensemble restreint d'utilisateurs. Si de telles applications s'exécutent sur un système stable avec des utilisateurs statiques, alors l'exécution sera proche du scénario de comportement régulier. L'exécution peut ressembler davantage aux scénarios de comportement en alternance, déséquilibré ou non, si des fautes temporaires sont possibles ou si les utilisateurs accèdent à leurs données depuis différents points d'accès ou sous différents identifiant. Dans le cas d'un

service de placement de données, il peut être préférable de différencier des accès distants par un même utilisateur.

Limite de la stratégie intemporelle

Il est important de noter que si cette stratégie semble adaptée aux applications gérant des données personnelles, si les utilisateurs utilisent différents points d'accès de manière équivalente, le service va donner un poids identique à chaque point d'accès, ce qui peut mener à une mauvaise décision pour tous. $\mathcal{S}_{IT}$ aura tendance à cherche un barycentre des différentes sources, proposant ainsi une qualité de service "moyenne" pour toutes les sources à tout moment de l'exécution, la où des configurations permettraient une qualité de service adaptée à l'utilisation courante.

De plus, comme précisée précédemment, l'approche intemporelle ne permet pas de s'adapter rapidement, ce qui peut être dommageable dans différentes situations. Par exemple, si un utilisateur change définitivement de point d'accès, ou si un nœud en panne est remplacé plutôt que réparé, alors le service aura besoin d'une longue période d'activité afin de déterminer une configuration avantageant les nouveaux points d'accès. Ce cas peut également être illustré par la Figure 6.7.

Enfin un service utilisant une stratégie intemporelle doit garder une trace de tous les événements étant survenus durant l'exécution. Ce besoin peut conduire à des soucis de stockage, l'historique pouvant devenir volumineux dans des systèmes à haute activité ou s'exécutant sur de longues périodes. Cette stratégie apparaît donc peu adaptée à un service dynamique s'exécutant à large échelle : dans de tels systèmes, le grand nombre de participants induit une dynamicité inévitable ainsi qu'un grand nombre d'événements. Elle est cependant utile pour un traitement *post-mortem* de l'historique des événements, qui peut alors être stocké et manipulé dans des nœuds dédiés.

6.2.3 Utilisation de l'historique reposant sur une fenêtre glissante

Concept de la stratégie à base de fenêtre glissante

Lorsqu'un service s'exécute sur un système à large échelle et nécessite une reconfiguration dynamique lors de l'exécution, il est préférable d'utiliser des stratégies plus fines que des stratégies intemporelles. Une stratégie considérée dans de nombreux cas est la stratégie se basant sur une fenêtre glissante. Dans cette stratégie, un intervalle de temps est déterminé et seuls les événements ayant eu lieu dans cet intervalle sont considérés lors des calculs du service. Cet intervalle est appelé *fenêtre glissante* et sa taille τ peut être fixé en début d'exécution ou adapté dynamiquement. Sa fonction de pondération $p^t_{\mathcal{S}_{FT}}$ doit donc rendre 100 pour les

événements récents et 0 pour les autres, plus formellement :

$$\forall t_i \in \mathcal{T}, t_i < t, p^t_{\mathcal{S}_{FT}}(t_i) = \begin{cases} 100, & \textbf{si } (t - t_i) \leq \tau \\ 0, & \textbf{si } (t - t_i) > \tau \end{cases}$$

avec t la date actuelle et t_i la date de l'événement e_i.

Rapidité d'adaptation

Cette stratégie se concentre davantage sur le présent et supprime l'impact des événements passés, ce qui permet au service de s'adapter rapidement à chaque changement. En effet, lorsque le service accède à l'historique, seuls les événements ayant eu lieu dans la dernière fenêtre glissante sont pris en compte. Ainsi si les sources ont changé de comportement, le résultat des calculs favorisera les sources actives récemment. Contrairement aux stratégies intemporelles, les sources inactivent ne perturbent donc plus les calculs pour déterminer les configurations optimales pour le comportement actuel des sources. De la même façon, les pannes des nœuds seront rapidement prises en compte par le service et un nœud inopérant n'aura plus d'impact. Le fonctionnement de cette stratégie est illustré par la Figure 6.8

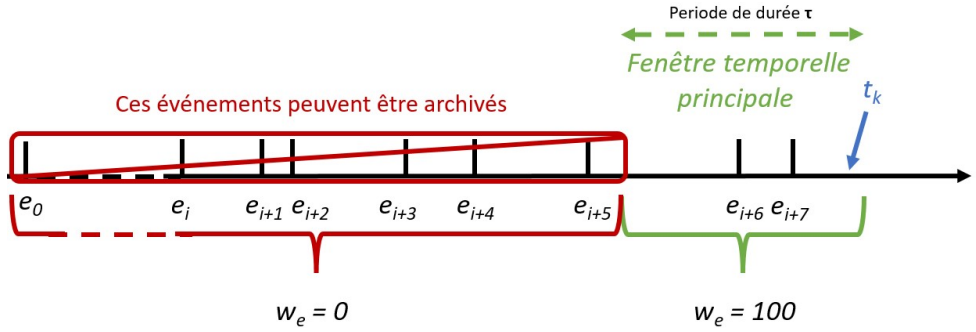


FIGURE 6.8 – Poids des événements au temps t_k avec une stratégie basée sur une fenêtre glissante

Cette rapidité d'adaptation a cependant un coût : Si certaines sources deviennent inactives pour une courte période, mais suffisamment longtemps pour permettre de sortir de la fenêtre glissante, alors le service ne considère plus les sources, comme illustré par la Figure 6.9. Si ce comportement est souhaitable dans certains cas, il peut être dommageable dans d'autres. Il est donc nécessaire pour le développeur d'être capable de configurer correctement la taille de la fenêtre.

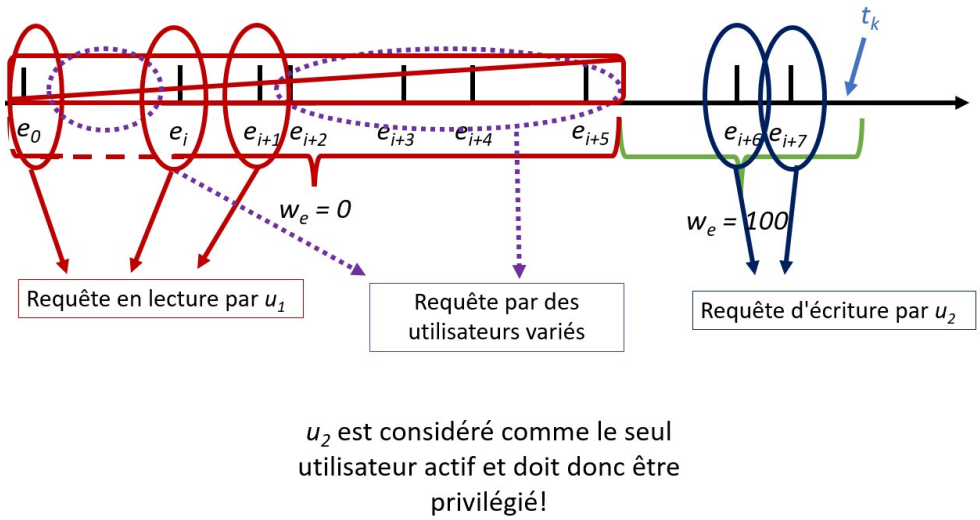


FIGURE 6.9 – Conséquence d'une stratégie basée sur une fenêtre glissante

Taille de la fenêtre glissante

Comme précisé précédemment, la taille de la fenêtre peut être fixe ou dynamique. Une taille fixe demande au concepteur d'être capable d'évaluer une taille raisonnable *a priori* et nécessite donc une expertise de sa part. Une fenêtre trop courte mènera à ignorer des événements qui devraient avoir un impact dans les calculs, menant à de nombreuses adaptations, potentiellement moins efficaces. À l'opposé, une fenêtre trop longue ; à l'image de la stratégie intemporelle, réduira la capacité d'adaptation du service, incitant ce dernier à rester sur la configuration courante. Ce choix peut se faire via une étude d'exécution passée ou en évaluant les scénarios attendus lors de l'exécution. Une fenêtre de taille variable ne demande pas de connaissance *a priori* du système, mais nécessite cependant une capacité de traitement supplémentaire : le service doit être capable de déterminer de lui-même une taille satisfaisante pour la fenêtre glissante. Nous détaillons certaines méthodes dans la Section 8.2.1. Ce traitement peut demander des ressources précieuses pour le système et un temps de calcul supplémentaire. Bien que plus coûteux, une gestion dynamique de la taille de la fenêtre est parfois essentielle, notamment si le taux d'activité peut évoluer avec le temps.

Lors de la mise en place d'une stratégie basée sur une fenêtre glissante, le concepteur doit également prendre en compte le coût des adaptations du service. En effet, cette stratégie favorise de nombreuses adaptations de la part du système : seuls les événements récents étant pris en compte, il est probable que l'ensemble des événements diffère d'une fenêtre à une autre, conduisant ainsi à des résultats différents lors des calculs du service. Ces différences de résultats entraînant généra-

lement un changement de configuration. Si dans certains services une adaptation peut avoir un coût négligeable, comme pour certains détecteurs de fautes ou il suffit de mettre à jour une liste, d'autres services, comme les services de placement de données, ont besoin de mettre en place des migrations de données parfois volumineuses. Le coût de la migration n'est alors pas négligeable et chaque adaptation doit être suffisamment impactante pour compenser ce coût. Il est donc important que le concepteur du service soit capable d'évaluer aussi bien le coût d'une adaptation que la configuration adaptée de la fenêtre. Or, ce travail supplémentaire est parfois conséquent lors de la mise en place d'un service qui demande déjà de nombreux défis de conceptions.

Conséquence de l'utilisation d'une fenêtre glissante

Toutefois, si le service utilise exclusivement des stratégies se basant sur des fenêtres temporelles, celui-ci sera capable de proposer rapidement des adaptations, mais pourra également manipuler un historique de taille réduite. Nous avons évoqué précédemment que l'utilisation de stratégies intemporelles pouvait conduire à des historiques de taille non négligeables. À l'inverse, une utilisation de fenêtre glissante autorise la suppression d'éléments de l'historique. En ne gardant que les événements les plus récents et avec une fenêtre correctement dimensionnée, la taille de l'historique reste de taille raisonnable et est facilement manipulable. L'analyse *post-mortem* de l'historique est cependant rendue impossible par la suppression d'événements. Il est donc nécessaire de déterminer si une telle analyse est souhaitée, et, lorsque c'est le cas, coupler cet allègement de l'historique a une méthode de sauvegarde des événements.

L'utilisation de stratégies à base de fenêtres temporelles permet donc une adaptation rapide et efficace tout en autorisant la manipulation d'un historique de taille réduite. En revanche, cette stratégie requiert une expertise de la part du concepteur, afin de la configurer correctement, et ne permet pas la prise en compte de tous les événements lors des calculs, ce qui peut mener à de mauvaises adaptations en oubliant des informations importantes. De plus, les stratégies se basant sur des fenêtres temporelles ont tendance à augmenter le nombre de changements de configuration, ce qui peut s'avérer coûteux pour le service.

Double fenêtre glissante

Afin de modérer le nombre de changements de configuration, certains services utilisent une "double fenêtre glissante", illustrée par la Figure 6.10. Cette approche consiste à réduire le poids, généralement le diviser par 2, des événements étant sorti de la première, mais de ne les fixer à 0 qu'après une seconde période. Formellement,

pour deux périodes τ_1, τ_2 telles que $\tau_1 < \tau_2$:

$$\forall t_i \in \mathcal{T} | \, t_i < t, \; p^t_{\mathcal{S}_{FT}}(t_i) = \begin{cases} 100, & \textbf{si } (t - t_i) \leq \tau_1 \\ 50, & \textbf{si } \tau_1 < (t - t_i) \leq \tau_2 \\ 0, & \textbf{si } \tau_2 < (t - t_i) \end{cases}$$

avec t la date actuelle et t_i la date de l'événement e_i.

Cette approche présente cependant des inconvénients similaires à l'utilisation de fenêtres temporelles simples : une fois les fenêtres dépassées, les événements sont purement oubliés, ce qui peut mener à de mauvaises adaptations ou de nombreuses migrations.

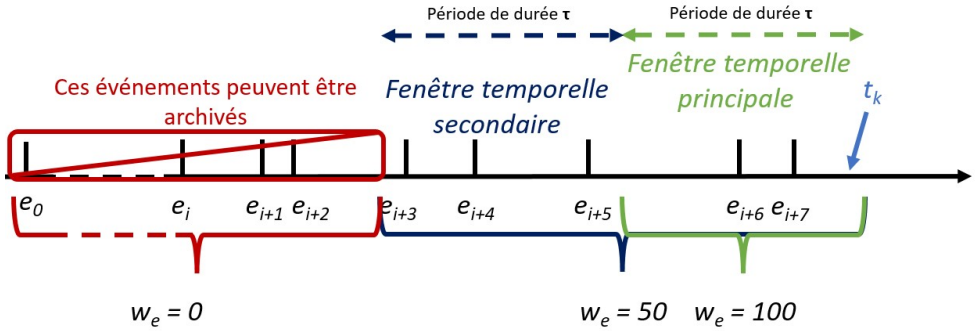


FIGURE 6.10 – Poids des événements au temps t_k avec une stratégie basée sur une double fenêtre glissante

6.2.4 Utilisation d'événements évanescents

Concept de la stratégie à base d'événements évanescents

De nombreux services ont besoin d'avoir cette rapidité d'adaptation, mais ont tout de même besoin d'avoir accès aux événements peu récents afin d'affiner leurs résultats, en particulier pour permettre d'effectuer un apprentissage sur les habitudes d'utilisation ou de comportements des sources. Il convient alors d'utiliser une stratégie utilisant des événements évanescents. Ce type de stratégie consiste à utiliser une fonction de pondération $p^t_{\mathcal{S}_{EE}}$ qui donne une valeur décroissante avec le temps. Il existe de nombreuses fonctions pouvant modéliser des événements évanescents, mais nous avons identifiés trois critères récurrents pour que celles-ci permettent une dégradation progressive de l'historique :

1. $p^t_{\mathcal{S}_{EE}}$ doit être défini par morceaux sur l'ensemble du temps de l'exécution. $\forall t_i \in \mathcal{T} | t_i < t, p^t_{\mathcal{S}_{EE}}(t - t_i)$ est défini. Il n'est cependant pas nécessaire

que $p^t_{\mathcal{S}_{EE}}$ soit continue sur son ensemble. Cette contrainte impose qu'un poids puisse être appliqué à tout moment de l'exécution pour tout événement.

2. $p^t_{\mathcal{S}_{EE}}$ doit être décroissante sur son ensemble. $\forall t_i, t_j \in \mathcal{T}, |t_i \leq t_j \leq t \Rightarrow p^t_{\mathcal{S}_{EE}}(t - t_i) \leq p^t_{\mathcal{S}_{EE}}(t - t_j)$. Cette contrainte impose que plus un événement est ancien, plus son poids est faible.

3. $p^t_{\mathcal{S}_{EE}}$ ne doit pas être constante sur son ensemble. $\forall t_i \in \mathcal{T}, t_i < t, \exists t_j \in \mathcal{T}, t_j < t \mid p^t_{\mathcal{S}_{EE}}(t - t_i) \neq p^t_{\mathcal{S}_{EE}}(t - t_j)$. Il est toutefois possible que $p^t_{\mathcal{S}_{EE}}$ soit localement constante, c'est en particulier le cas pour des fonctions décroissantes constantes par morceaux.

Dans de nombreux cas, il est souhaitable de ne jamais complètement oublier un événement, celui-ci peut alors avoir un poids négligeable, mais non nul. Il faut donc ajouter une quatrième contrainte :

4. $p^t_{\mathcal{S}_{EE}}$ doit être une fonction de $\mathcal{T}$ dans $\mathbb{R}+$. Comme précisé précédemment, cette contrainte permet de ne jamais oublier un événement passé.

Pour la discussion qui suit, et pour nos exemples, nous posons la fonction suivante : $\tilde{p}^t_{\mathcal{S}_{EE}}(t_i) = \frac{100}{2^{t-t_i}}$. L'utilisation d'une fonction continue permet de donner un poids précis pour chaque instant, mais peut demander de nombreux calculs de la part du service. Dans la pratique, nous recommandons l'utilisation d'une fonction continue par morceaux, comme des fonctions décroissantes constantes par morceaux. Ce type de fonction utilise généralement une période de temps τ pour définir la taille d'un intervalle sur laquelle la fonction est constante. Nous utilisons alors la fonction $p^t_{\mathcal{S}_{EE}}(t_i) = \frac{100}{2^{\delta}}$, $\delta = \lfloor \frac{t-t_i}{\tau} \rfloor$, illustrée sur la Figure 6.12. La Figure 6.11 illustre la différence de précisions entre $\tilde{p}^t_{\mathcal{S}_{EE}}$ et $p^t_{\mathcal{S}_{EE}}$ pour différentes périodes τ.

À l'image des fonctions utilisées par les stratégies se basant sur des fenêtres temporelles, cette période peut être fixe ou dynamique.

Cette fonction consiste à diviser périodiquement par 2 le poids des événements passés. Cette opération peu coûteuse permet de réduire rapidement le poids des événements tout en gardant une trace, comme le montre la Figure 6.13. De cette façon, le service est capable de proposer des adaptations rapidement sans omettre d'événements, ce qui permet de réduire le nombre de changements de configuration.

Taille de l'historique

Cependant, manipuler des événements évanescents nécessite généralement de conserver une trace de tous ces événements en mémoire ce qui peut s'avérer coûteux. Comme pour les stratégies intemporelles, il est donc parfois nécessaire de mettre

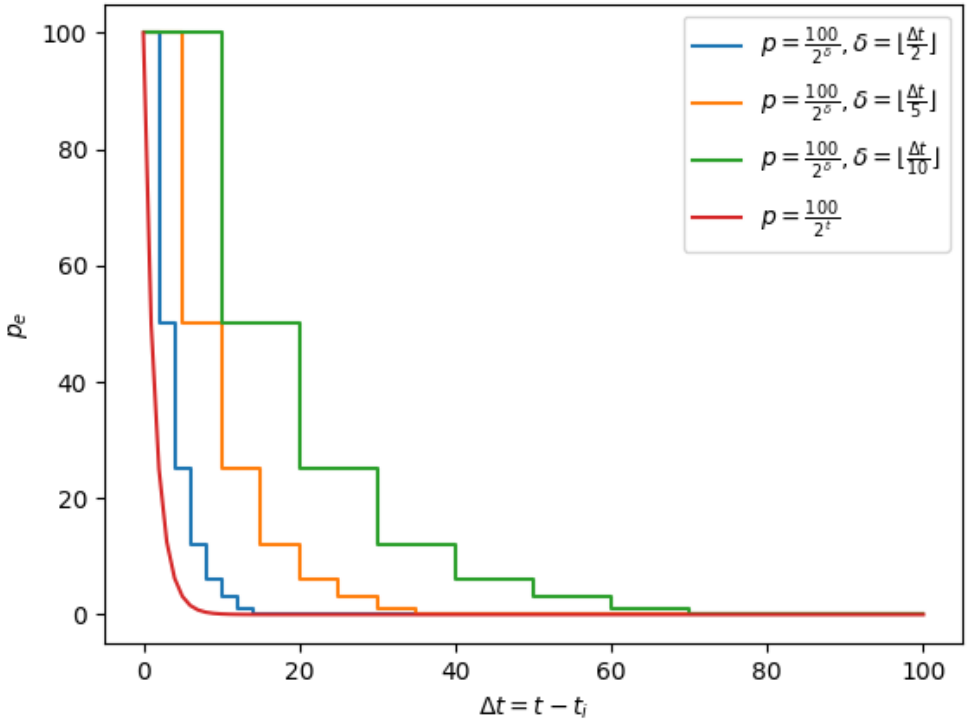


FIGURE 6.11 – Discrétisation du temps

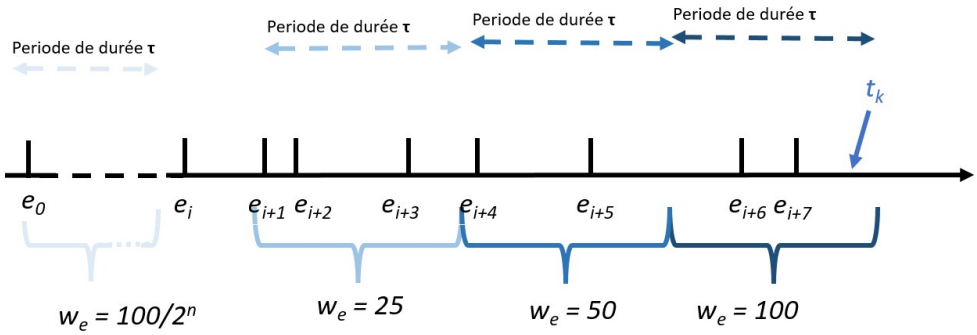


FIGURE 6.12 – Poids des événements au temps t_k avec une stratégie d'événements évanescents

en place des méthodes de regroupement des événements ou de prévoir des emplacements de stockage suffisant. De telles méthodes sont toutefois plus simples à mettre en place dans le cas d'événements évanescents, car ces derniers ne requièrent pas, par nature, une information complète. Si les capacités mémoires des nœuds

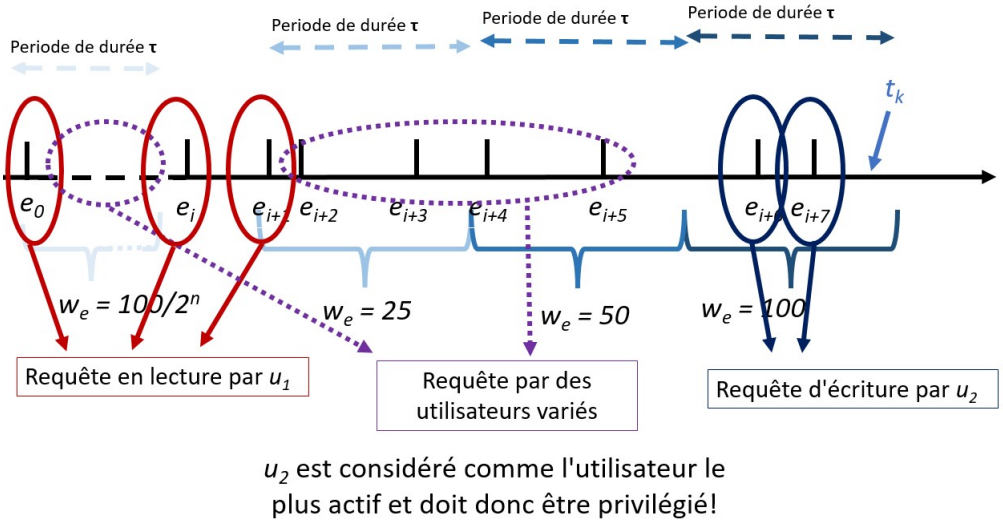


FIGURE 6.13 – Conséquence d'une stratégie basé sur d'événements évanescents

exécutant $\mathcal{S}_{EE}$ sont limitées, comme c'est usuellement le cas, il est alors pertinent de ne pas respecter la contrainte 4. Dans le cas de la fonction $p^t_{\mathcal{S}_{EE}}(\delta t) = \frac{100}{2}^{\lfloor \frac{\delta t}{\tau} \rfloor}$, par exemple, il est possible de fixer un nombre limite de périodes après lequel un événement est oublié. Une méthode efficace pour y parvenir est de considérer que lorsque le poids d'un événement est inférieur à un seuil, cet événement peut être supprimé de l'historique. Cette approche ne permet cependant pas de borner la taille de l'historique.

Les stratégies basées sur des événements évanescents sont donc adaptées pour des services capables d'avoir une charge de calculs supplémentaire, afin de permettre l'utilisation d'une fonction de pondération adaptée. Une capacité mémoire suffisante pour maintenir un historique de taille non borné est également souhaitable pour assurer une adaptation sans perte d'événement. Ces stratégies permettent alors une adaptation rapide et efficace, tout en gardant une trace des événements passés pour pouvoir mettre en place des stratégies robustes dans le temps.

Stratégie de résurgences d'événement évanescent

Enfin, il est possible d'utiliser des stratégies se basant sur des événements évanescents, mais en relâchant la contrainte 2, qui impose que $p_{\mathcal{S}_{EE}}$ soit décroissante. Afin d'assurer un bon fonctionnement de ces stratégies, il convient d'utiliser une fonction qui ne relâche cette contrainte que ponctuellement. Par exemple, lorsqu'un événement est similaire à un ancien événement (source et action identique par exemple), il peut s'avérer efficace d'augmenter le poids de l'événement le plus ancien. Ce type de stratégie est notamment à l'origine des listes de type *Least Fre-*

quently Used et ses variations. Ces stratégies sont particulièrement efficaces, mais leurs mises en place demandent une expertise de la part du concepteur afin de ne pas fournir de résultats contre-productifs.

6.3 Adaptation du service

6.3.1 Déclencher une reconfiguration

Afin de déterminer si un service doit changer de configuration, ce dernier doit estimer le profit d'un tel changement. Ce changement se fait par le biais d'une adaptation α. Le service doit commencer par calculer des estimations du coût $c(\alpha)$ et du gain $g(\alpha)$ de cette adaptation.

L'évaluation du coût dépend fortement du type de service : certains détecteurs de fautes doivent simplement mettre à jour un timer local, mais un service de placement de données doit mettre en oeuvre une migration de données. Nous supposons donc que le service est capable d'estimer un tel coût c(α).

Le gain $g(\alpha)$ d'une adaptation α s'obtient en calculant la différence de performance entre l'état courant σ_{cur} et l'état du système après l'adaptation σ_α. Cette valeur est estimée grâce à une fonction d'évaluation de performance μ, qui fournit un indicateur en fonction d'un événement e et d'une configuration σ. Cette mesure est un indicateur de la performance du service et dépend donc fortement du service. Il peut indiquer, par exemple, la fiabilité d'un détecteur ou la latence médiane d'accès à une donnée. Le service ne pouvant pas calculer la mesure de performance des événements à venir, il doit se baser sur les événements présents dans l'historique. Nous proposons la formule suivante pour calculer ce gain :

$$g(\alpha) = \sum_{e \in \mathcal{H}_S} (p_S(t - t_e)(\mu(\sigma_\alpha, e) - \mu(\sigma_{cur}, e)).$$

Cette approche consiste à calculer la différence de performance entre chaque événement connu du service. Chaque mesure de performance doit être pondérée par le poids donné à l'événement correspondant selon la stratégie d'utilisation.

Une fois que le service a estimé le coût et le gain d'une adaptation, celui-ci peut déterminer s'il doit ou non mettre en place cette adaptation. En particulier, si $g(\alpha) - c(\alpha) > 0$, alors le service devrait estimer que cette adaptation est bénéfique pour le service.

Cette modélisation illustre l'importance de la stratégie d'utilisation de l'historique. Une stratégie intemporelle associe un poids identique à chaque événement, sans tenir compte de sa date. En conséquence, le calcul du gain ne sera alors pas affecté par la date des événements et entraînera un placement favorisant les sources les plus actives dans le temps. Ce comportement peut amener a réduire le nombre de configuration.

6.4 Conclusion

Avant d'être capable d'établir une méthode de placement dynamique des données, nous nous sommes intéressés à la façon d'exploiter un historique des événements dans un système distribué. À notre surprise, peu de travaux ont été menés sur ce problème, et dans la plupart des cas, ce problème semble implicite et les différents auteurs utilisent des méthodes prédéfinies telles que la fenêtre glissante ou une stratégie intemporelle. Nous avons donc traité dans ce chapitre du problème d'adaptation des systèmes distribués dynamiques, en particulier des stratégies d'utilisation de l'historique des événements. Ces stratégies sont essentielles afin que les services dynamiques puissent s'adapter efficacement aux changements d'état du système ou de comportement des utilisateurs. Nous proposons dans la Section 6.2 une représentation de la mémoire des événements passés et des stratégies de son utilisation. Trois stratégies sont approfondies : les stratégies intemporelles, basées sur des fenêtres temporelles et à événements évanescents. Nous avons également exploré différentes méthodes pour paramétrer les stratégies basées sur des fenêtres temporelles et à événements évanescents.

Comme nous l'avons évoqué dans la Section 6.2.4, il existe de nombreuses fonctions de pondérations et nous souhaitons dans la suite de nos travaux nous concentrer sur l'approfondissement d'autres fonctions de pondérations, en particulier dans le cadre des stratégies à événements évanescents ou des stratégies de résurgences d'événements évanescents.

III

Partie III - Évaluations

Sommaire

Chapitre

7

CandorSim : un simulateur pour le placement dynamique de données à cohérence adaptable

7.1 Introduction

Concevoir et mettre en place des algorithmes robustes à large échelle peut s'avérer être une tâche difficile. De nombreux travaux de recherches se sont heurtés à différents problèmes : passage à l'échelle, interaction imprévue, timing des communications, etc. Prévoir le comportement d'un grand nombre de nœuds est compliqué et il est donc courant d'avoir recours à la simulation pour s'assurer du fonctionnement d'un programme avant de le déployer sur de vrais systèmes. Nous avons utiliser un simulateur afin de pouvoir conduire des expérimentations de CAnDoR, la méthode de placement de réplicas présentée dans le Chapitre 5.

L'intérêt de la simulation ne s'arrête cependant pas à la vérification et validation de programme. Comparée à un émulateur ou un déploiement, la nature légère d'un simulateur lui permet d'effectuer plus de calcul, plus rapidement. Les simulateurs, une fois correctement configurés, sont donc un choix pertinent pour la prévision et l'estimation différents résultats. Il convient cependant d'utiliser un simulateur efficace, validé par la communauté scientifique et de le paramétrer correctement afin d'obtenir des résultats exploitables.

Dans le cadre des travaux menés dans cette thèse, nous avons développé CandorSim, un simulateur pour le calcul automatique du placement des réplicas dans un système de gestion de donnée distribué. Ce simulateur a été développé par dessus PeerSim [MJ09] afin de nous permettre d'évaluer le comportement de CAnDoR. Nous l'avons principalement utilisé afin d'étudier le comportement de nos programmes et de mener des expériences pour vérifier leur efficacité. Nous pensons qu'avec un léger remaniement notre simulateur pourrait devenir un outil d'aide à la conception (et au maintien) de systèmes de gestion de données distribués. En effet, il est possible d'adapter le simulateur à la configuration d'un système précis et ainsi obtenir une estimation de comment placer les réplicas des différentes données afin d'améliorer les performances du système. Ce chapitre détaille le fonctionnement de CandorSim.

7.2 Fonctionnement de CandorSim

CandorSim est un simulateur pour le placement dynamique de données développé durant cette thèse. Il a été construit au-dessus de PeerSim [MJ09], un simulateur reconnu. CandorSim est un simulateur centralisé à événements discrets : tous les événements qui arrivent pendant l'exécution simulée sont ajoutés à une unique file d'événements.

Lorsqu'un événement (par exemple un envoi de message) est créé, le simulateur calcule quand cet événement va se produire et l'ajoute à la file à la bonne position. Seules les dates où un événement a lieu sont simulées, permettant ainsi d'économiser du temps et des calculs par rapport à une simulation complète. L'objectif de CandorSim est de simuler un système de gestion de données dans lequel les données sont répliquées et stockées sur différents nœuds. Comme nous travaillons sur les systèmes de gestion de données distribués à modèles de cohérence adaptables, chaque donnée est associée à un modèle de cohérence. Ce modèle indique les contraintes attachées à la donnée. Des utilisateurs ont la possibilité d'effectuer des requêtes d'accès en lecture ou écriture sur ces données. Pour cela, ils envoient un message à un nœud dont ils ont l'adresse avec la requête et attendent une réponse.

CandorSim est organisé en quatre types de classes :

- les nœuds : SNode, CandorNode, UserNode
- les données : Data
- les messages : Message, MessageAck, MessageData, MessageRequest
- les classes utilitaires : Action, HWTransport, Initializer, MyLogOutputFormatter

Un UserNode a connaissance d'un groupe de CandorNodes auprès duquel il peut effectuer des requêtes sur les données, ce mécanisme est une abstraction de la

couche applicative du système de gestion de donnée. Les CandorNodes peuvent échanger entre eux, répondre aux utilisateurs et stockent des données. Afin d'établir un échange entre 2 nœuds, ces derniers créent des messages.

7.2.1 La classe SNode

Le rôle de la classe SNode est de regrouper les méthodes et paramètres communs aux CandorNode et UserNode ainsi que d'abstraire la couche applicative. Tout nœud participant à la simulation, qu'il s'agisse d'un CandorNode ou d'un UserNode, hérite de SNode. Ils ont ainsi accès à la liste des identifiants des données stockées et à une table indiquant quels nœuds du système contacter pour accéder à chaque donnée. Les SNodes ont également une primitive permettant de communiquer avec un nœud connu. Cette primitive se base sur une matrice de communication notée $\mathcal{C}$.

$\mathcal{C}$ est une matrice de taille $N \times (N+M)$ avec N le nombre de nœuds de stockage et M le nombre d'utilisateurs. $\mathcal{C}_{ij}$ indique le temps médian de communication entre les nœuds i et j. On note δ_{ij} la variance de ce temps de communication. Pour des raisons de simplification, nous considérons dans nos expériences que tous les liens de communications du système disposent de la même variance, notée δ. Il est toutefois possible d'affecter une variance différente à chaque lien de communications. Lors de l'envoi d'un message du nœud i vers j, afin de déterminer le temps nécessaire pour le transit du message, le simulateur tire aléatoirement un nombre dans l'intervalle $[\mathcal{C}_{ij} - \delta, \mathcal{C}_{ij} + \delta]$.

PeerSim ne propose pas de prise en compte native de la bande passante et notre travail portant à l'origine sur des données de taille négligeable, cette primitive de communication ne prend pas encore en compte la charge du réseau lors de l'envoi de message et la manière dont le transit de données impacte les latences. Nous prévoyons de travailler à une meilleure représentation du réseau.

7.2.2 La classe CandorNode

Les CandorNodes représentent les nœuds du système de gestion de données distribué. Ils ont pour rôle de stocker des réplicas de données et de répondre aux requêtes des utilisateurs. Pour cela, chaque nœud a accès à une table indiquant quels nœuds contacter pour chaque donnée.

Lorsqu'un nœud de stockage reçoit une requête, il vérifie s'il possède bien la donnée concernée. Si ce n'est pas le cas, il transmet la requête à un nœud qui devrait stocker un réplica de la donnée, selon ses connaissances locales. Si le nœud possède un réplica de la donnée concernée, il traite la requête puis envoie le résultat à l'initiateur de la requête.

De plus, afin d'assurer le fonctionnement de la stratégie de placement dynamique, chaque nœud de stockage maintient un historique des événements d'accès aux données dont il stocke une réplique. Cet historique est mis à jour en échangeant les statistiques d'utilisation avec les autres nœuds stockant un réplica des données présentes sur ce nœud. Nous avons détaillé les méthodes de construction de l'historique des événements dans le Chapitre 6.

Les CandorNodes sont capables de calculer de nouveaux placements pour les réplicas. Cette méthode utilise les statistiques d'accès de la part des utilisateurs ainsi que des contraintes liées aux modèles de cohérence. Le fonctionnement de cette méthode est décrit dans le Chapitre 5, traitant du placement dynamique des données. Cette méthode s'accompagne des primitives nécessaires aux déplacements d'une donnée : transfert de données, mise à jour de la table de correspondance du système et suppression d'un réplica local.

Nous avons développé CandorSim afin d'étudier CAnDoR, une méthode de placements de réplicas, présenté dans le Chapitre 5 et les stratégies du maintien de l'historique, telles que présentées dans le Chapitre 6. Ces méthodes visent à être utilisées au sein d'un système de gestion de données existant. Ainsi nous faisons abstraction de la gestion des noeuds du système. CandorSim ne gère donc pas directement les arrivées et départs de noeuds. De plus, nous fournissons une table de communication permettant à tout noeud du système d'estimer le temps de communication avec un autre noeud (dont il connaît l'identité).

7.2.3 La classe UserNode

Les utilisateurs, ou UserNodes, ont la capacité de créer des données et d'envoyer des requêtes d'accès en lecture ou en écriture sur une donnée. Les UserNodes n'ont pas connaissance des autres utilisateurs et ne peuvent donc pas communiquer directement avec eux.

Le comportement des utilisateurs peut suivre des traces d'exécutions fournies au simulateur ou suivre un scénario d'exécution parmi une présélection. Nous avons implémenté 5 scénarios pour nos expériences, dans lesquels les utilisateurs peuvent être *actifs* ou *au repos*. Un utilisateur actif envoie régulièrement des requêtes au système, alors qu'un nœud au repos ne le fait que de manière occasionnelle. Cette particularité a été choisie pour ajouter du bruit lors du calcul des statistiques d'utilisation et représente des accès opportuns (ou depuis un emplacement inhabituel). Les scénarios types sont classés en 4 catégories :

1. Comportement stable : les utilisateurs ont un comportement constant durant la totalité de l'exécution. C'est-à-dire qu'une source active pour une cible à un instant quelconque t l'est (et l'a été) à tout instant de l'exécution. Ce scénario illustre en particulier l'utilisation d'un service par des utilisateurs

fixes et est représenté par la Figure 7.1 où l'utilisateur 1 est la seule source active ;

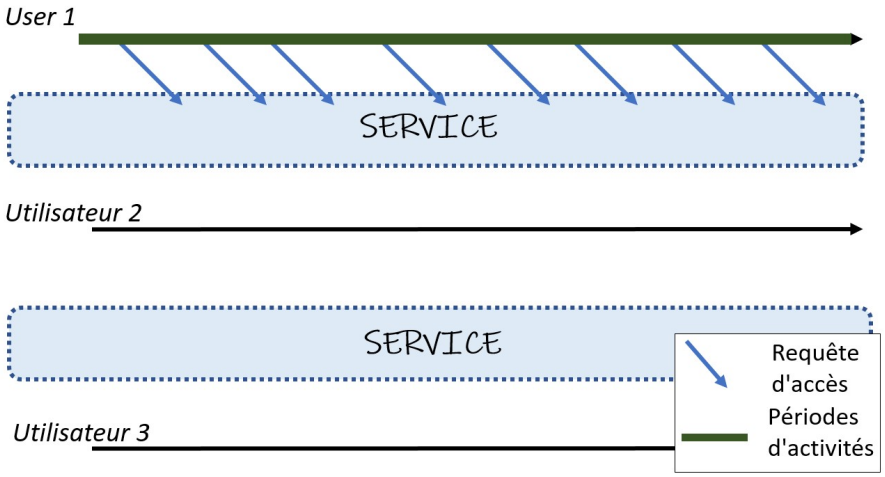


FIGURE 7.1 – Illustration du scénario 1

2. Comportement en alternance : 2 groupes d'utilisateurs changent de comportements à intervalle régulier. À chaque nouvel intervalle, le groupe d'utilisateurs actif devient inactif et inversement. Certains utilisateurs peuvent rester inactifs sur la durée de l'exécution. Ce scénario est illustré par la Figure 7.2 où les utilisateur 1 et 2 alternent les périodes d'activités pendant que l'utilisateur 3 reste inactif. ;

3. Comportement en alternance déséquilibrée : 2 groupes d'utilisateurs sont alternativement actifs sur des intervalles de temps propre à chaque groupe. Ce scénario représente, par exemple, une activité d'un groupe d'utilisateur utilisant un service à 2 localisations différentes, comme un lieu de travail et un domicile et est illustré par la Figure 7.3 où les utilisateur 1 et 2 alternent les périodes d'activités pendant que l'utilisateur 3 reste inactif ;

4. Comportement irrégulier : les utilisateurs n'ont pas d'habitude particulière et peuvent changer de comportement à tout moment. Ce scénario est illustré par la Figure 7.4.

Dans un système déplaçant les données en fonction de l'activité des utilisateurs, le changement de comportement de la part d'un utilisateur est parfois accompagné d'un pic de latence moyenne. En effet, un utilisateur nouvellement actif ne sera que peu considéré par le système dans un premier temps alors que les utilisateurs

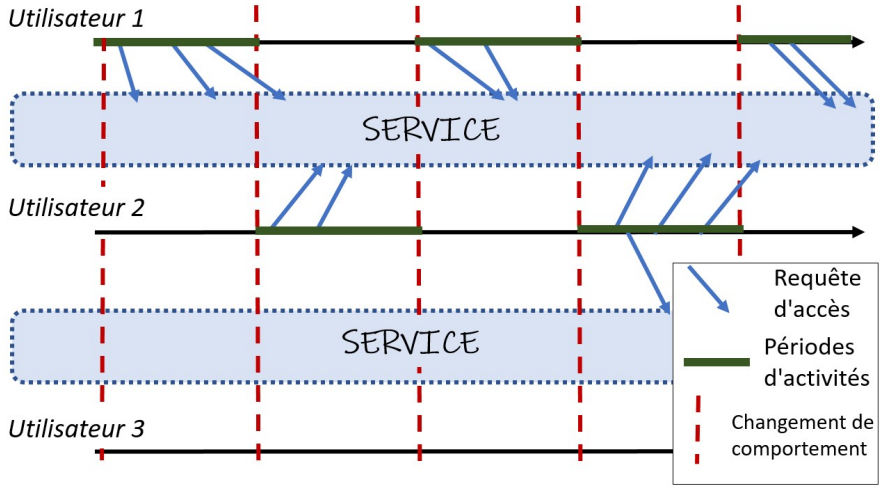


FIGURE 7.2 – Illustration du scénario 2

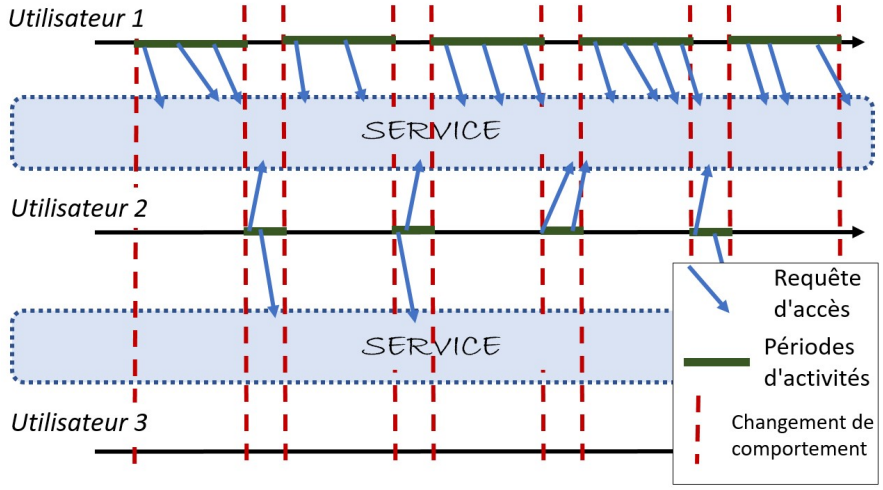


FIGURE 7.3 – Illustration du scénario 3

inactifs depuis peu profitent d'un placement adapté à leur ancienne activité. Une métrique d'évaluation pertinente des méthodes dynamique est donc la rapidité à retrouver un placement reflétant l'activité effective des utilisateurs.

Nous estimons que ces scénarios permettent de simuler les comportements les plus courants des utilisateurs, mais davantage de scénarios pourraient être développés dans le futur. Il est possible d'ajuster les paramètres, comme la taille des groupes, le nombre d'utilisateurs, les fréquences d'accès, etc. De cette façon, il est possible de simuler un comportement proche des situations attendues. Nous travaillons actuellement à ajouter la capacité pour les utilisateurs de suivre un

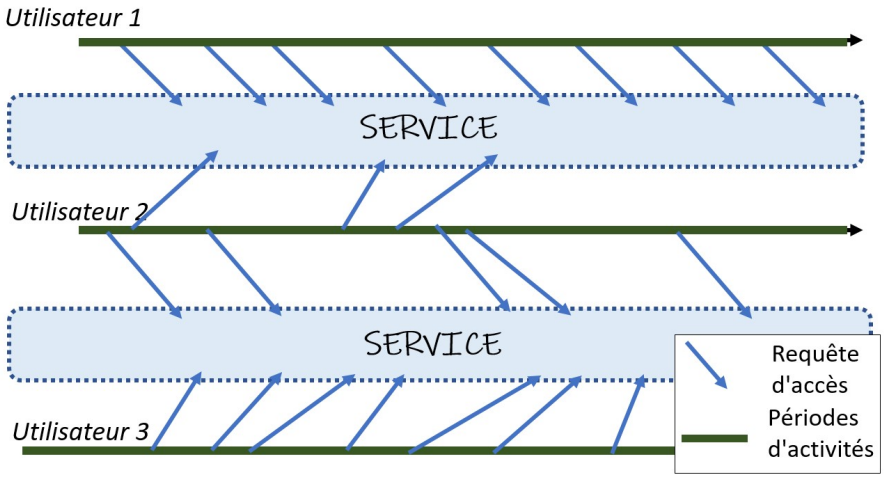


FIGURE 7.4 – Illustration du scénario 4

fichier de log pour déterminer les actions à prendre. Cette fonctionnalité permettrait d'utiliser le simulateur comme un outil d'aide à la conception et au maintien de systèmes de gestion de données distribués. En effet, en utilisant les log qu'un système a subits lors d'une exécution, notre simulateur pourrait rejouer ce scénario et ainsi indiquer comment améliorer les performances du système.

7.2.4 La classe Data

La classe Data est utilisée pour simuler les réplicas de données ainsi que les informations attachées à ces derniers. Les données, manipulées par les différents CandorNodes, sont définies par leur identifiant unique et leur méta-données, parmi lesquelles :

- le protocole de cohérence utilisé pour la gestion de la donnée ;
- le nombre de réplicas de la donnée ;
- la liste des nœuds qui stockent des réplicas ;
- une copie de l'historique d'utilisation lié à ce réplica.

Comme nous le décrivons dans le Chapitre 6, sur l'utilisation de l'historique, il est parfois préférable de mettre à jour les données de l'historique, voire supprimer des informations. Afin de faciliter le traitement, ces méthodes sont directement liées aux données.

7.2.5 La classe Message

Les messages utilisés par le simulateur sont séparé en différent type afin de pouvoir leur associer les paramètres adéquats. Toutes les classes héritent de la classe Message, qui contient les champs indispensables : le type de message, l'ID de la source et l'ID du message. Il existe à ce jour trois classes enfants : MessageAck, MessageData et MessageRequest permettant respectivement de notifier un utilisateur, d'envoyer une donnée à un nœud (utiliser pour une opération de lecture et pour une migration) et de faire une requête de lecture ou écriture sur une donnée.

7.2.6 Les classes utilitaires

Afin d'assurer le bon fonctionnement du simulateur, nous avons également mis au point des classes utilitaires. Cette section comporte les classes LogOutputFormatter, Initializer, HWTransport et Action. Ces classes permettent respectivement de specifier le format des fichiers log, d'initialiser certains paramètre pour la simulation et de la couche de transport et de définir le comportement des utilisateurs.

7.3 Les paramètres utilisés pour nos simulations

Au cours de nos expérimentations présentées dans le chapitre suivant, nous utilisons CandorSim afin d'étudier les performances de différentes approches. Nous détaillerons lors de ces chapitres les approches et paramètres impliqués lors de l'évaluation. Cependant certains paramètres, tels que le nombre de clusters, la taille des clusters, le nombre d'utilisateurs, le nombre d'utilisateurs actifs, la matrice de communication, etc. seront fixés pour toutes les évaluations, sauf mention contraire, afin de pouvoir comparer correctement les différents résultats. Nous justifions dans cette section les valeurs données à ces paramètres.

7.3.1 Nombre d'exécutions et choix des métriques

Lors d'une exécution, les utilisateurs notent le temps de latence entre l'envoi de leur requête et la réception de la réponse du nœud de stockage. Périodiquement, chaque utilisateur actif écrit dans un fichier de trace d'exécution le quatre-vingt-quinzième centile des temps d'accès de la période précédente. En d'autres termes, notre métrique principale est la latence à quatre-vingt-quinzième centile des utilisateurs actifs. En effet, nous avons principalement travaillé dans l'objectif de fournir un système de placement de données capable d'améliorer les temps d'accès des utilisateurs. Dans la suite de nos travaux, nous prévoyons de nous intéresser à d'autres métriques, en particulier la réduction de dépenses énergétiques par un

placement stratégique des données. Chaque exécution étant sujette à des variations dans les temps de communication, comme expliqué ci-dessous, et les temps effectifs des communications, nous effectuons chaque exécution 50 fois et nous nous intéressons à la médiane des résultats de chaque exécution.

7.3.2 Nombre de données simulées

Lors des travaux ici présentés, nous étudions la gestion de données dans le cadre de requêtes dites atomiques. De ce fait, le résultat de chaque requête est indépendant des autres requêtes. Nous considérons également que les réplicas sont des réplicas "complets" des données, nous n'implémentons donc pas de méthodes de codes correcteurs [DDH10]. Ces choix nous permettent d'aborder le problème de placement "par donnée". Nos méthodes ne permettant actuellement pas de prendre en compte des relations causales entre différentes données, le placement de chaque réplica d'une donnée ne prend en compte que les autres réplicas de la même donnée. Ainsi, sauf mentions contraires, les utilisateurs n'accèdent qu'à une seule donnée au cours de l'exécution. Nous souhaitons lors de futurs travaux étendre nos méthodes afin de prendre en compte des relations entre les données et adapter le placement en fonction de celles-ci.

7.3.3 Matrice de communications

La matrice de communications, telle que définie dans la Section 7.2.1, joue un rôle primordial dans l'évaluation des performances et il est important que celle-ci soit le plus réaliste possible afin que les simulations soient pertinentes. Pour établir les temps de communications médians entre les nœuds, nous nous sommes basés sur les travaux des études [Aga18] et [Pop19]. Notre matrice de communications est organisée de la manière suivante : $n = \frac{N}{c}$ clusters de taille c sont organisés et M utilisateurs peuvent contacter n'importe quel nœud de n'importe quel cluster.

Les communications au sein d'un même cluster sont généralement extrêmement rapides, avec une latence comprise entre 1 et 3 millisecondes. Les temps de communication entre 2 clusters distincts, semblables à ceux entre un cluster et un utilisateur, varient entre 30 millisecondes et 300 millisecondes. Les temps de communications entre 2 couples distincts étant indépendants, les valeurs sont générées aléatoirement selon un tirage uniforme au début de l'exécution. Les temps de latence pour contacter 2 nœuds d'un même cluster sont proches, mais sujets à une légère variance. La Figure 7.5 illustre un exemple de communication. Dans cet exemple, nous disposons de 2 clusters de 2 nœuds et de 2 utilisateurs. Les distances entre deux nœuds sont les suivantes, toutes sujettes à une variance de $\delta = 3$ ms :

- utilisateur 1-cluster 1 : 70 ms

- utilisateur 1-cluster 2 : 191 ms
- utilisateur 2-cluster 1 : 100 ms
- utilisateur 2-cluster 2 : 52 ms
- cluster 1-cluster 2 : 42 ms

La matrice 7.1 correspond à la matrice utilisée par CandorSim pour un tel environnement. Nous pouvons noter que cette matrice est composée d'une matrice triangulaire de taille $N \times N$, CandorSim utilise donc cette propriété pour manipuler efficacement ce genre de matrices.

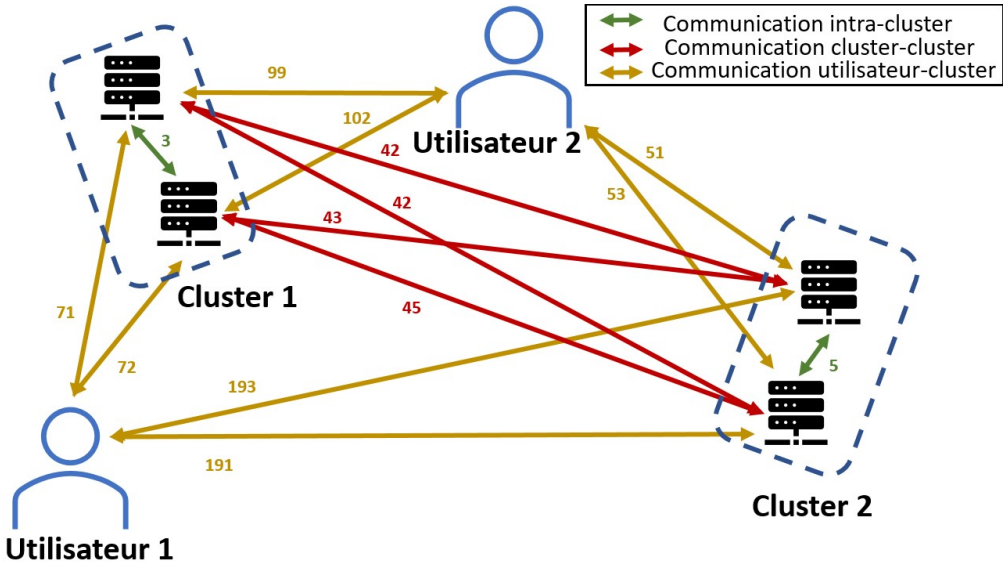


FIGURE 7.5 – Exemple de communication selon CandorSim

$$\left[\begin{array}{c|cccccc} & c_{11} & c_{12} & c_{21} & c_{22} & u_1 & u_2 \\ \hline c_{11} & - & 3 & 42 & 42 & 71 & 99 \\ c_{12} & 3 & - & 43 & 45 & 72 & 102 \\ c_{21} & 42 & 43 & - & 2 & 191 & 51 \\ c_{22} & 42 & 45 & 2 & - & 193 & 53 \end{array}\right] \Leftrightarrow \left[\begin{array}{cccccc} - & 3 & 42 & 42 & 71 & 99 \\ - & - & 43 & 45 & 72 & 102 \\ - & - & - & 2 & 191 & 51 \\ - & - & - & - & 193 & 53 \end{array}\right] \quad (7.1)$$

Il est également possible de fournir un fichier de configuration de la matrice afin de tester les performances sur un système précis.

Il est à noter que certains clusters maintenus par un même fournisseur peuvent être reliés par des moyens de communication privilégiés, permettant d'atteindre des latences entre entre 6 et 20 millisecondes. Toutefois, ce type de liens de communications étant particulier, nous ne les considérons pas lors de nos expériences.

7.3.4 Taille des clusters

En étudiant les différents temps de communication, il apparaît que les communications au sein d'un même cluster sont quasiment négligeables par rapport aux autres types de communications. Cette particularité joue un rôle très impactant lorsque les méthodes essayent de déterminer un placement des réplicas en favorisant la communication entre les nœuds du système. Ces méthodes sont détaillées dans le Chapitre 5, traitant des stratégies de placements dynamiques.

De plus, pour des raisons de tolérance aux fautes, il est souvent préférable de ne pas placer 2 réplicas d'une même donnée dans un même cluster. En effet, en cas de fautes du cluster, de types panne ou partitions du système, avoir 2 réplicas dans un même cluster peut être contre-productif. Toutefois, certaines applications peuvent désirer placer plusieurs réplicas dans un même cluster. CandorSim laisse cette possibilité en donnant la possibilité de dimensionner les clusters. Nous recommandons actuellement de dimensionner les clusters à la taille du nombre de réplicas à placer au sein d'un même cluster.

Au final, CandorSim est capable de gérer les communications intra-cluster. Ces communications étant négligeables, et afin de ne pas placer 2 réplicas d'une même donnée dans le même cluster, ces derniers sont représentés dans nos simulations par un seul CandorNode. Cette approche nous permet également d'effectuer des tests sur de plus gros systèmes tout en gardant un temps d'exécution raisonnable. Cette démarche explique également le choix d'écarter des clusters différents, mais proches physiquement, ayant donc des temps de communications très faibles. La proximité physique de 2 clusters peut présenter les mêmes soucis que de placer 2 réplicas dans un même cluster, en particulier dans le cas de fautes corrélées ou liées à des facteurs extérieurs aux systèmes (environnementaux par exemple). Il est cependant possible de souhaiter placer plusieurs réplicas dans un même cluster afin de permettre un équilibrage des charges. CandorSim ne permettant actuellement pas de simuler l'encombrement du réseau, nous n'utilisons pas ce mécanisme dans les chapitres suivants.

7.3.5 Nombre de clusters

Le nombre de clusters est également un paramètre impactant lors de l'évaluation des stratégies. Une donnée est généralement répliquée 3 à 5 fois dans les systèmes géodistribués [GBKA11]. Les méthodes de placement doivent donc déterminer des ensembles de 3 ou 5 noeuds capables de stocker des réplicas des données.

Un faible nombre de clusters entraîne un ensemble restreint de possibilités pour le placement des réplicas. Plus le nombre de clusters augmente, plus les possibilités de répartitions sont grandes, ce qui a deux conséquences majeures :

- Les méthodes de placements sont capables de trouver de meilleures répartitions pour les réplicas.
- Le temps de calcul de ces méthodes, et donc de la simulation, augmente.

Nous nous intéressons d'abord au temps de calcul nécessaire pour une exécution en fonction du nombre de clusters, détaillé sur la Figure 7.6. Ces temps de calculs dépendants de l'architecture matérielle utilisée, nous nous intéressons ici à la tendance asymptotique des temps de calcul. Comme nous pouvions nous y attendre, la méthode utilisée étant exacte et le nombre de triplets de nœuds de stockage évoluant de manière exponentielle avec le nombre de clusters, le temps de calcul d'une exécution évolue également de manière exponentielle. Nous étudions ensuite les performances de nos méthodes en fonction du nombre de clusters. Nous avons donc effectué un comparatif d'un même scénario avec un nombre de clusters allant de 10 à 300. Les résultats de cette comparaison sont illustrés dans la Figure 7.7. Nous avons choisi d'évaluer un scénario dans lequel 2 groupes de 10 d'utilisateurs échange les périodes d'activité. Les nœuds de stockage disposent ici d'une méthode pour calculer un placement dynamique de réplicas. Cette méthode, détaillée dans le Chapitre 5, est une méthode exacte et va donc considérer l'ensemble des ensembles de 3 nœuds de stockage.

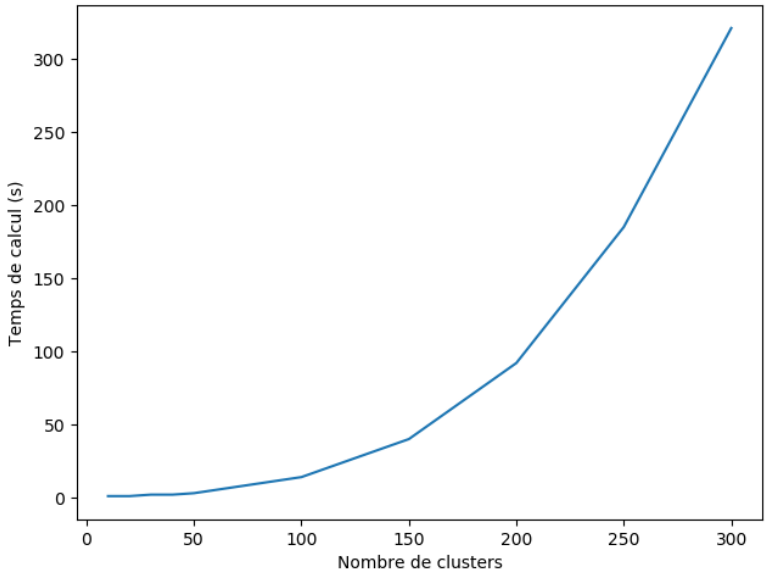


FIGURE 7.6 – Évaluation du temps de calcul en fonction du nombre de clusters

Nous pouvons observer en Figure 7.7 que les performances semblent bien s'améliorer avec un nombre croissant de clusters. Cette amélioration est toutefois très faible une fois 100 clusters atteints. Lors de ces évaluations préliminaires, nous

pouvons constater des pics de latences réguliers. Ceux-ci sont la conséquence d'un changement d'activité de la part des utilisateurs, tel que détaillé dans la Section 7.2.3. Ce changement de comportement a besoin de temps pour être pris en compte par le système. Le Chapitre 6 décrit les différentes façons de s'adapter à ce changement. Le temps de latence du quatre-vingt-quinzième centile évolue entre :

- 175 et 200 millisecondes à 10 clusters,
- 150 à 175 (avec un pic à 200 ponctuel) millisecondes pour 20, 30 et 40 clusters,
- 140 à 160 (avec un pic à 175 ponctuel) millisecondes pour 50 clusters
- 130 à 150 (avec un pic à 175 ponctuel) millisecondes pour 100, 150, 200, 250 et 300 clusters.

Ces différences de performances étant minimes au-delà de 100 et le temps de calcul grandissant de manière exponentielle, nous utilisons une base de 100 clusters pour la suite de nos expériences.

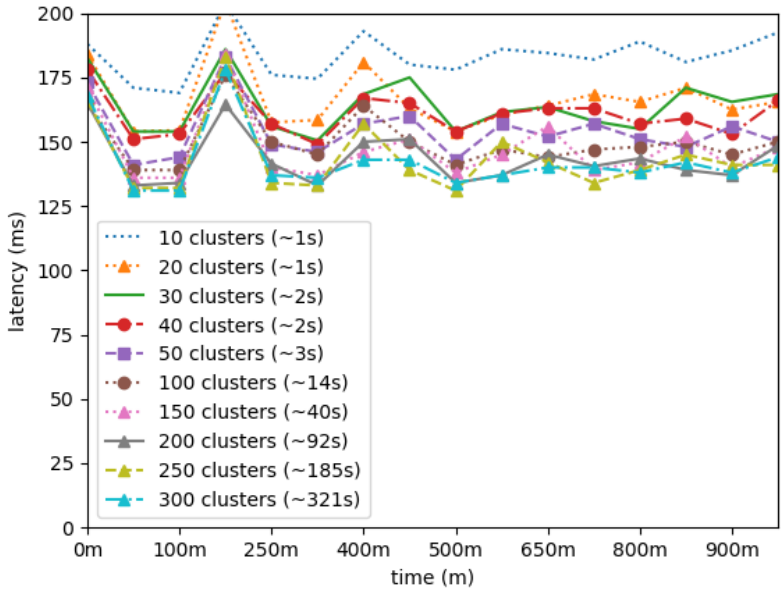


FIGURE 7.7 – Évaluation des performances en fonction du nombre de clusters

7.3.6 Nombre d'utilisateurs et taille des groupes actifs

Lors des évaluations de performance, nous considérons les latences au quatre-vingt-quinzième centile des utilisateurs actifs. Le nombre d'utilisateurs inactifs n'a donc que peu d'impact sur les performances du système. Nous avons choisi de simuler un

ensemble de 100 utilisateurs. Sur ces 100 utilisateurs, dans la plupart des scénarios envisagés pour nos évaluations, seul un groupe est actif à la fois. Ce groupe est alors celui évalué par les métriques présentées. Nous nous intéressons ici à la taille de ce groupe. Comme lors de l'étude du nombre de clusters, nous commençons par étudier l'impact du nombre d'utilisateurs actifs sur le temps de calcul d'une exécution. Les résultats sont présentés dans la Figure 7.8.

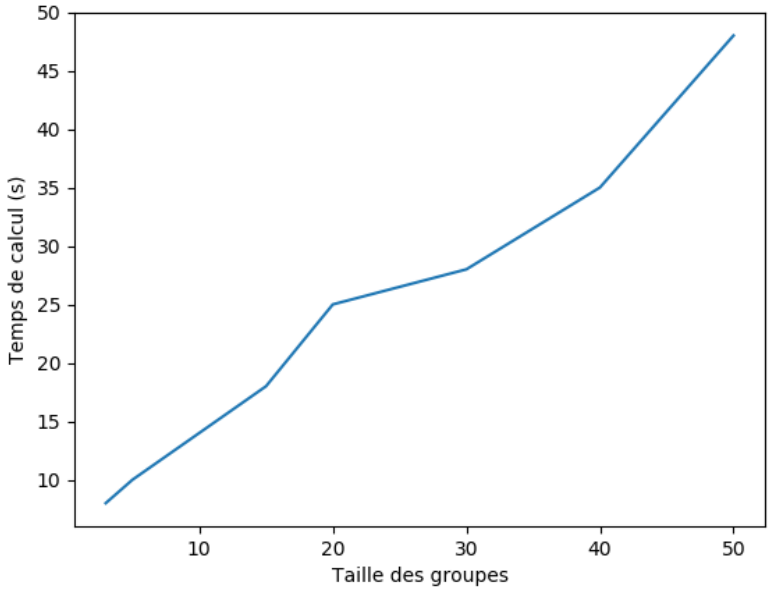


FIGURE 7.8 – Évaluation du temps de calcul en fonction de la taille des groupes d'utilisateurs

Nous pouvons y observer que le temps de calcul augmente linéairement avec le nombre d'utilisateurs actifs. Cette tendance s'explique par le fait que la méthode testée effectue une comparaison pour chaque utilisateur actif. Nous étudions ensuite la différence de performances de la méthode en fonction du nombre d'utilisateurs (voir Figure 7.9).

La donnée étant répliquée 3 fois, les expériences avec 3 ou 5 utilisateurs produisent de très bons résultats : chaque réplica peut être placé à proximité d'un utilisateur actif. À l'opposé, pour des groupes de plus de 20 utilisateurs actifs simultanément, les performances se dégradent. En effet, lorsqu'un grand nombre d'utilisateurs essaye d'accéder à une même donnée depuis de nombreuses localisations, il n'est pas possible d'avoir un placement efficace pour tous les utilisateurs. Les performances pour des groupes de $10, 15$ et 20 utilisateurs sont similaires. Nous utiliserons principalement des groupes de 10 utilisateurs dans les chapitres

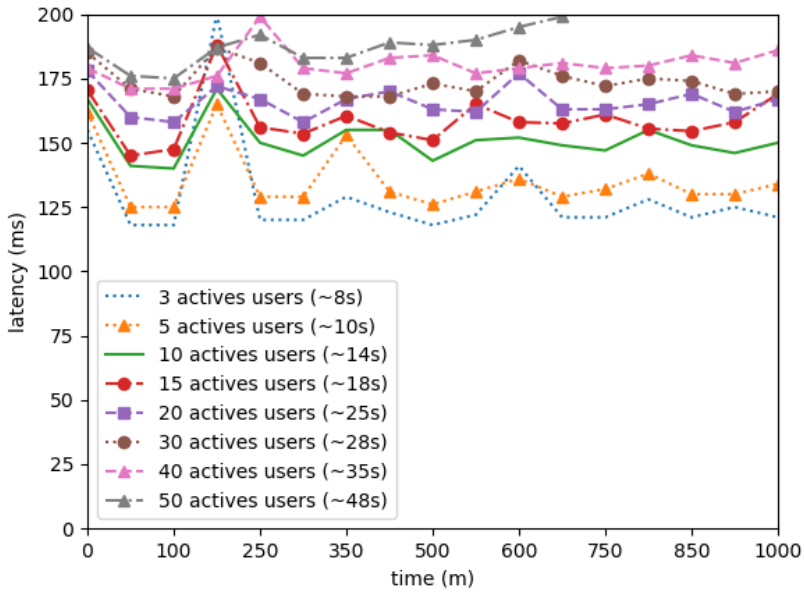


FIGURE 7.9 – Évaluation des performances en fonction de la taille des groupes actifs

suivants.

7.3.7 Quelques exécutions

Enfin lors des évaluations de performances, nous avons considéré dans un premier temps deux approches statiques :

1. Un placement statique aléatoire : lors de la création d'une donnée, l'application choisit au hasard autant de nœuds de stockage que de réplicas souhaités et envoie les réplicas à ces nœuds. Le placement n'évolue pas pour la suite de l'exécution. Cette méthode de placement est rapide à mettre en place et peu coûteuse. Si l'hypothèse est que les accès des utilisateurs sont géodistribués uniformément, un placement aléatoire devrait permettre des performances proches de méthodes de placements basés sur une prévision du workload. De nombreux systèmes utilisent une méthode de placement aléatoire [DHJ+07, LM10].
2. Un placement statique adapté à la cohérence : Lors de la création d'une donnée, l'application calcule un placement qui semble efficace en fonction de la cohérence associée à la donnée. Cette méthode se base sur les observations

décrites dans le Chapitre 5 : une donnée gérée avec de la cohérence forte demande des synchronisations bloquantes, les réplicas des données seront donc proches les uns des autres. À l'inverse, une donnée gérée avec de la cohérence à terme doit être délivrée aux utilisateurs le plus rapidement possible, les réplicas seront donc espacés afin de répondre rapidement aux utilisateurs. Cette considération est peu coûteuse en calcul et devrait permettre des résultats plus performants qu'un placement aléatoire. Le placement n'évolue pas pour la suite de l'exécution.

Nous avons évalué ces 2 approches avec une donnée gérée avec de la cohérence forte et avec une donnée avec de la cohérence à terme. Le résultat de cette étude est illustré dans la Figure 7.10.

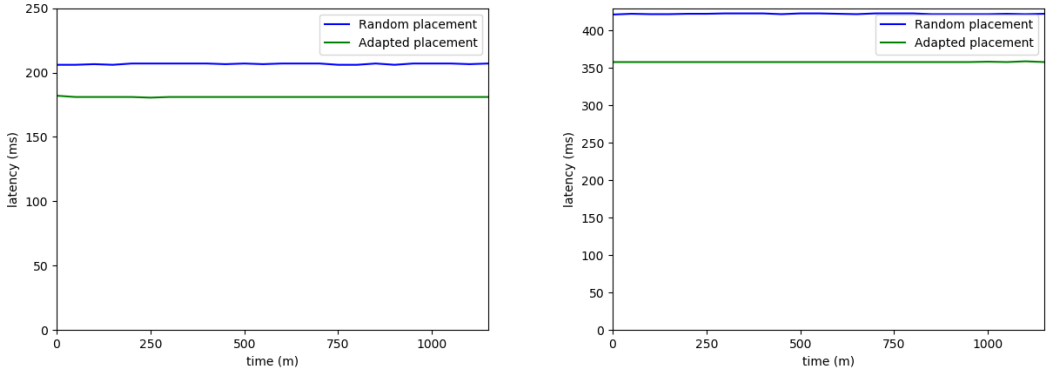


FIGURE 7.10 – Comparatifs de méthodes statiques avec une donnée à cohérence à terme (à gauche) et forte (à droite)

Lors d'une simulation incluant une donnée gérée avec de la cohérence à terme, le placement adapté est plus efficace, il permet une latence au quatre-vingt-quinzième centile de 190 millisecondes contre une latence quatre-vingt-quinzième centile de 220 millisecondes pour un placement aléatoire. La même tendance peut être observée dans le cadre d'une donnée gérée avec de la cohérence forte, avec des latences plus élevées, de 360 millisecondes pour un placement adapté et 425 millisecondes pour un placement aléatoire. Dans la suite de nos travaux, nous nous utiliserons donc uniquement un placement statique adapté comme base de référence, celui étant toujours plus efficace qu'un placement aléatoire. L'algorithme utilisé est celui décrit dans cette section.

7.4 Conclusion

Afin de valider les différents résultats que nous avons obtenus, nous avons développé une extension de PeerSim, un simulateur répandu. Cette extension, nommée CandorSim, permet de simuler un système de gestions de données simplifié dans lequel des utilisateurs peuvent effectuer des requêtes de différents types sur des données et évaluer le délai pour obtenir une réponse à ces requêtes.

Ce chapitre détaille le fonctionnement de ce simulateur et présente les différents choix d'implémentations ainsi que les paramètres utilisés pour nos expériences. Le choix de ces paramètres s'est basé sur différentes études, notamment pour déterminer une estimation réaliste des temps de communications entre les acteurs du système [Aga18], [Pop19], [GBKA11]. Différentes options sont disponibles afin de paramétrer le simulateur et nous prévoyons la possibilité de lire des traces d'exécution d'un véritable système afin de déterminer quels placements auraient été optimaux dans le temps de l'exécution ainsi que le calcul de version des données.

CandorSim met en place différents types de nœuds : les CandorNodes et le UserNodes. Ces derniers peuvent envoyer des requêtes aux CandorNodes et calculer le temps avant d'avoir une réponse (ack ou la donnée selon le type de requête). Les CandorNodes sont organisés en cluster, avec des temps de communications privilégiés au sein d'un même cluster. Ils peuvent traiter les requêtes des utilisateurs et calculer le placement des réplicas parmi les CandorNodes. Ce calcul est déclenché en fonction du temps et de la popularité d'une donnée. L'hypothèse de requêtes atomiques et de système de gestion de données non transactionnel nous permet d'utiliser un environnement avec une donnée simulée : chaque donnée est indépendante des autres, son placement ne se fait donc pas en fonction des autres données. Nous réduisons également la taille des clusters à un : les communications au sein d'un même cluster sont extrêmement rapide (de l'ordre de la milliseconde), les différences de performances entre placer un réplica sur différents nœuds du même cluster sont donc négligeable. De plus pour des raisons de sécurité, il peut être préférable d'éviter de placer deux réplicas dans le même cluster. Conformément à l'étude de [GBKA11], nous considérons entre 3 et 5 réplicas pour les données. La matrice de communications est construite selon les résultats de [Aga18], [Pop19]. Nous utilisons ce simulateur dans la suite de nos travaux afin d'estimer les gains de performances liés à nos approches.

Sommaire

Chapitre 8

Évaluations de CAnDor et de l'utilisation de l'historique à l'aide de CandorSim

8.1 Évaluations de la méthode CAnDoR

Afin d'évaluer les performances apportées par CAnDoR, présentée dans le Chapitre 5, nous avons conduit deux études. La première se concentre sur la paramétrisation des coefficients c_a et c_c. Ces coefficients doivent être déterminés en fonction du modèle de cohérence reliant les réplicas. Afin de pouvoir conseiller les développeurs lors de la mise en place d'un système de gestion de données distribué, nous effectuons ici des simulations avec différentes valeurs pour ces paramètres. La seconde étude porte sur les différences de performances de CAnDoR en fonction du comportement des utilisateurs.

Ces deux études ont été réalisées en utilisant CandorSim, le simulateur développé pendant nos travaux de thèse et présenté dans ce chapitre. Nous utilisons les paramètres tel que décrits dans la Section 7.3 : nous simulons 100 nœuds de stockage et 100 utilisateurs. L'utilisateur de l'historique de la part de nœuds de stockage se fait à l'aide la fonction basée sur des événements évanescents $p^t_{\mathcal{S}}(t_i) = \frac{100}{2^\delta},\ \delta = \lfloor \frac{t-t_i}{\tau} \rfloor$. Afin de rendre les résultats plus explicites, nous ne simulons qu'une seule donnée à la fois dans le système. Nous estimons que tous les

nœuds de stockage ont des capacités et puissances similaires. La liste des nœuds potentiels est donc calculée sur l'ensemble des nœuds de stockage et un nœud responsable est déterminé aléatoirement parmi les nœuds stockant un réplica de d. Les temps de communication entre les nœuds se basent sur les études [Aga18] et [Pop19] :

- communications entre un nœud de stockage et un utilisateur : estimation entre 30 et 300 millisecondes ;
- communications entre deux nœuds de stockage : estimation entre 30 et 180 millisecondes.

8.1.1 Configuration des paramètres de contraintes

Cette première étude porte sur l'impact des différentes valeurs du couple de coefficients $(c_c; c_a)$. Pour cela nous effectuons un balayage des différentes valeurs en faisant varier c_c de 10 à 90 par pas de 10, c_a étant obtenu par le calcul $c_a = 100 - c_c$. Ce couple de coefficients est utilisé pour pondérer les temps de propagation et de réponse obtenus par les calculs de CAnDoR.

Nous présentons ici les performances de CAnDoR lorsque les utilisateurs suivent le scénario suivant, présenté dans le Chapitre 7 :

- Comportement en alternance : deux groupes d'utilisateurs changent de comportements à intervalle régulier. À chaque nouvel intervalle le groupe d'utilisateurs actif devient inactif et inversement. Certains utilisateurs peuvent rester inactifs sur la durée de l'exécution. Ce scénario est illustré par la Figure 8.1 où les utilisateurs 1 et 2 alternent les périodes d'activités pendant que l'utilisateur 3 reste inactif.

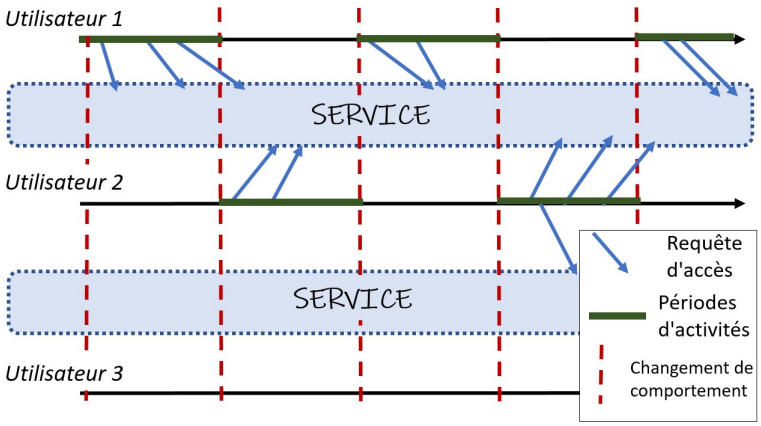


FIGURE 8.1 – Illustration du scénario 2

Donnée gérée sans propagation bloquante

Nous commençons par évaluer l'impact du couple de coefficients $(c_c; c_a)$ lors de l'accès à une donnée gérée sans propagation bloquante. C'est par exemple le cas de donnée dont les réplicas sont soumis aux modèles de cohérence à terme ou causal. Dans ce cadre, lorsqu'un nœud de stockage reçoit une requête sur une donnée dont il a un réplica, il traite directement la requête puis envoie sa réponse à l'utilisateur. La propagation de l'information se fait dans un second temps, de manière non bloquante. Intuitivement, les données gérées par un protocole de cohérence à terme devraient avoir une faible valeur de c_c (proche de 0) et une forte valeur de c_a (proche de 100) afin d'obtenir de bonnes performances. En effet, une valeur de c_a proche de 100 indique à CAnDoR de se concentrer sur le temps de réponse aux utilisateurs.

Nous présentons, dans un premier temps, sur la Figure 8.2, les performances obtenues avec les valeurs suivantes : $(c_c; c_a) = (10; 90)$ (en bleu marine sur la figure), $(c_c; c_a) = (50; 50)$ (en cyan sur la figure), $(c_c; c_a) = (90; 10)$ (en orange sur la figure). Comme nous pouvons nous y attendre, une forte valeur de c_a permet d'obtenir de meilleures performances : en posant $(c_c; c_a) = (10; 90)$, nous observons un gain allant jusqu'à 11% comparé à $(c_c; c_a) = (50; 50)$ et des résultats 20% plus efficaces qu'en utilisant $(c_c; c_a) = (10; 90)$. Nous pouvons observer que CAnDoR offre de bonnes performances en comparaison d'une méthode de placement statique. En effet, avec un coefficient c_a suffisamment élevé, notre méthode permet de détecter les utilisateurs actifs et de placer les données de manière à les favoriser lors du déplacement des réplicas.

Nous avons ensuite effectué le balayage complet des valeurs de $(c_c; c_a)$. Les résultats de ces évaluations sont présentés sur la Figure 8.3. Ces résultats confirment la tendance d'améliorations des performances avec une forte valeur de c_a. Pour davantage de lisibilité, la Figure 8.4 se concentre sur les performances des valeurs fortes de $(c_c; c_a)$: $c_c \in [10; 30]$ puis $c_c \in [70; 90]$.

En étudiant les performances obtenues pour des valeurs de c_a allant de 70 à 90 (Figure 8.4a), nous pouvons observer que les performances obtenues sont proches les unes des autres tout en permettant une meilleure latence en augmentant la valeur de c_a. Les meilleures performances obtenues ici sont lorsque $(c_c; c_a) = (10; 90)$. Dans ce cas, CAnDoR ne considère que très peu la durée d'une propagation d'information. Cependant, en allongeant la durée de propagation, le risque de délivrer des copies dans des états non équivalents à deux utilisateurs augmente. Nous projetons dans de futurs travaux d'effectuer une étude approfondie sur les répercussions des coefficients sur l'état des réplicas et les propriétés de fraîcheur.

L'étude des valeurs de c_a a également révélé qu'en prenant des valeurs trop faibles, à partir de $c_a < 20$ (Figure 8.4b), alors l'approche CAnDoR devient moins efficace qu'un placement statique. En particulier, lorsque $c_a = 10$, le temps médian

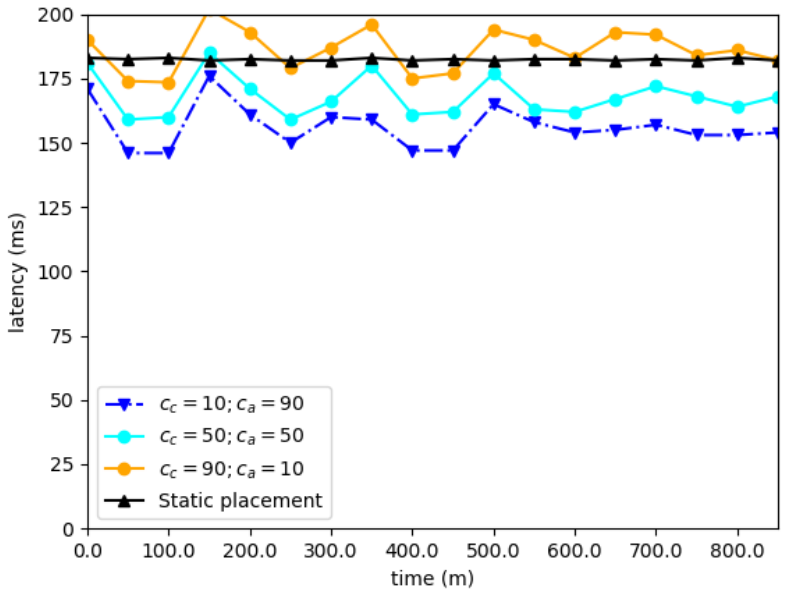


FIGURE 8.2 – Balayage des valeurs de c_c pour une donnée gérée sans propagation bloquante

pour traiter une requête varie entre 175 et 200 ms, contre un temps stable à 180 ms pour un placement statique. Ceci illustre l'importance d'un paramétrage adéquat du couple $(c_c; c_a)$. Dans le cadre de données gérées par un protocole de cohérence à terme, nous recommandons l'utilisation d'une forte valeur de c_a tel que $(c_c \leq 30; c_a \geq 70)$.

Donnée gérée par un protocole de propagation bloquante

La seconde partie de cette étude porte sur l'impact du couple $(c_c; c_a)$ lorsque la donnée est gérée par un protocole de propagation bloquante, par exemple pour garantir des propriétés de cohérence forte. Lorsqu'un nœud de stockage reçoit une requête d'un utilisateur, il contacte les autres nœuds stockant un réplica de la donnée. Une fois un consensus atteint, le nœud envoie sa réponse à l'utilisateur. La phase de propagation est ici bloquante, au sens où tant qu'un consensus n'est pas atteint, l'utilisateur doit attendre. Intuitivement, nous nous attendons à obtenir de meilleures performances avec une valeur de c_c élevée (proche de 100) afin de diminuer le temps nécessaire à obtenir la propagation.

Nous présentons, dans un premier temps, sur la Figure 8.5, les performances obtenues avec les valeurs suivantes : $(c_c; c_a) = (10; 90)$ (en bleu marine sur la figure), $(c_c; c_a) = (50; 50)$ (en cyan sur la figure), $(c_c; c_a) = (90; 10)$ (en orange sur

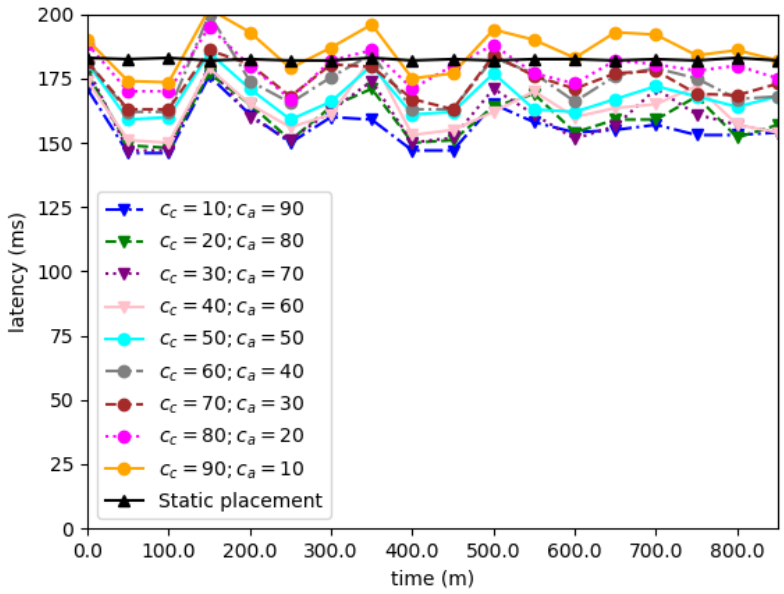
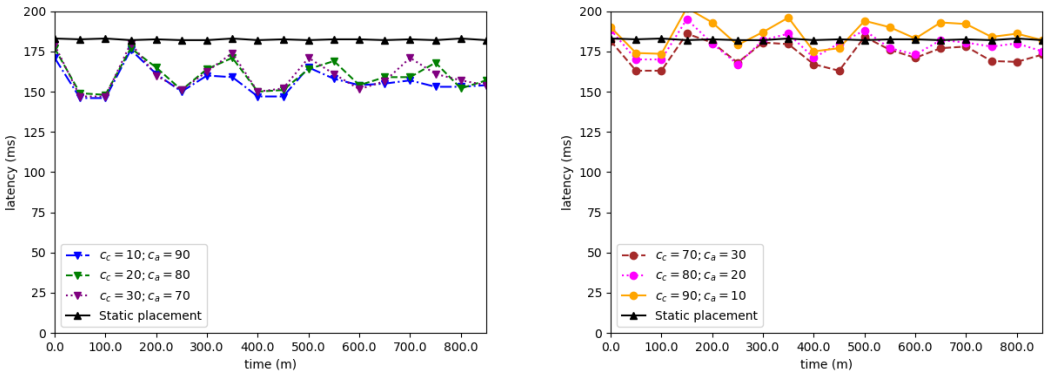


FIGURE 8.3 – Balayage des valeurs de c_c pour une donnée gérée sans propagation bloquante



(a) Balayage des valeurs de $c_c \in [10; 30]$ (b) Balayage des valeurs de $c_c \in [70; 90]$

FIGURE 8.4 – Balayage des valeurs remarquables de c_c

la figure). Nous pouvons y observer que les résultats sont très proches, mais que les meilleures performances sont obtenues pour $(c_c; c_a) = (50; 50)$. Ces résultats soulignent l'importance de considérer le temps de réponse, afin d'être capable de placer les réplicas de façon à propager rapidement les requêtes tout en offrant de

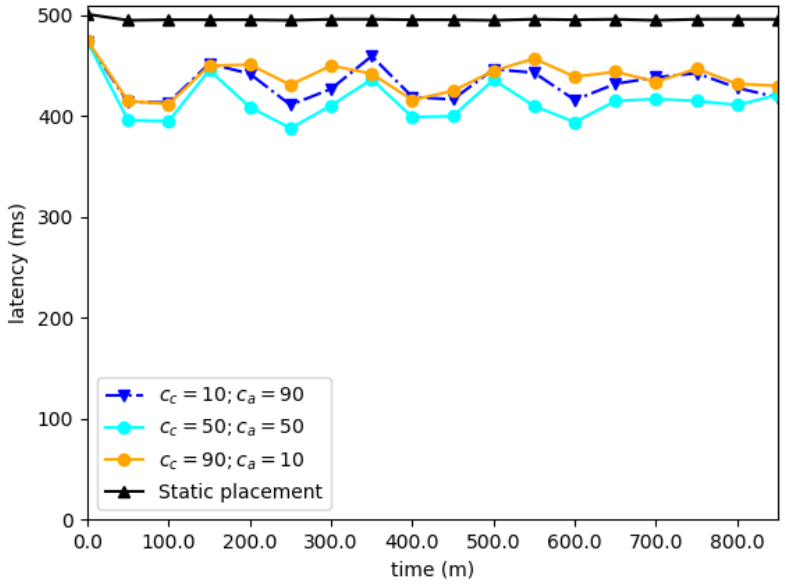


FIGURE 8.5 – Balayage des valeurs de c_c pour une donnée gérée avec une propagation bloquante

bons délais de réponse. Cette particularité montre également l'importance d'un placement dynamique dans le cadre de données gérées avec un protocole de cohérence forte.

L'étude du balayage complet des valeurs de $(c_c; c_a)$, illustré sur les Figures 8.6 et 8.7 confirme que les meilleures performances sont obtenues pour $c_c \in [40; 60]$. Il est intéressant de remarquer une certaine symétrie entre les performances pour les valeurs de c_c en dehors de cet intervalle. Ces résultats sont explicables par le protocole de cohérence utilisé lors de nos évaluations. Lors de celles-ci, il y a peu de mises à jour concurrentes, les phases de consensus sont alors achevées rapidement, en un seul échange de messages entre les nœuds de stockage. Les temps de communication entre deux nœuds de stockage étant proches du temps de communication entre un nœud de stockage et un utilisateur, la moitié des messages de la phase de propagation est envoyée entre un nœud de stockage et un utilisateur. Cette proportion devrait donc alors se refléter dans le rapport $\frac{c_c}{c_c+c_a}$.

Conclusion de l'étude sur les coefficients

Cette première étude permet de mettre en avant l'importance de correctement évaluer les contraintes de cohérence et des temps de délai. Nous avons vu qu'un mauvais paramétrage des coefficients $(c_c; c_a)$ peut mener à une perte de perfor-

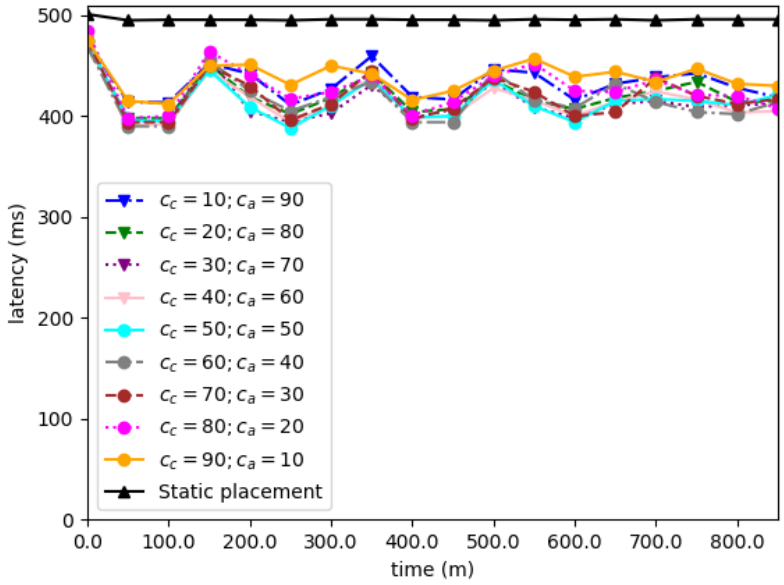
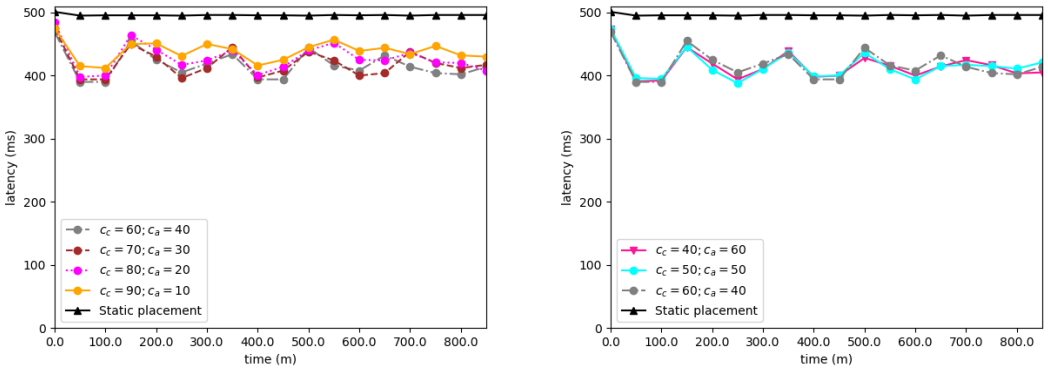


FIGURE 8.6 – Balayage des valeurs de c_c pour une donnée gérée avec une propagation bloquante



(a) Balayage des valeurs de $c_c \in [60; 90]$ (b) Balayage des valeurs de $c_c \in [40; 60]$

FIGURE 8.7 – Balayage des valeurs remarquables de c_c

mance. De plus, nous pouvons observer que ces paramètres doivent être configurés différemment selon le modèle de cohérence appliqué aux données. Nous conseillons d'utiliser de fortes valeurs de c_a ($c_c \leq 30; c_a \geq 70$) lorsque le service n'utilise pas de propagation bloquante entre les réplicas. Si le service a recours à un protocole

de propagation bloquant, nous recommandons d'utiliser des valeurs reflétant le nombre de messages échangés. Dans notre cas, avec un protocole de propagation en 1 phase, nous observons que le couple $(c_c = 50; c_a = 50)$ permet d'obtenir les meilleurs résultats.

8.1.2 Performances de CAnDoR face à différents scénarios d'utilisation

La seconde étude menée consiste en l'évaluation des performances de CAnDoR face à différents scénarios d'utilisation. Ces scénarios ont été présentés dans le Chapitre 7, mais nous les rappelons ici :

1. Comportement stable : les utilisateurs ont un comportement constant durant la totalité de l'exécution : un utilisateur actif à un instant t_1 l'est à tout instant de l'exécution. Ce scénario illustre en particulier l'utilisation d'un service par des utilisateurs fixes et est représenté par la Figure 8.8 où l'utilisateur 1 est la seule source active ;

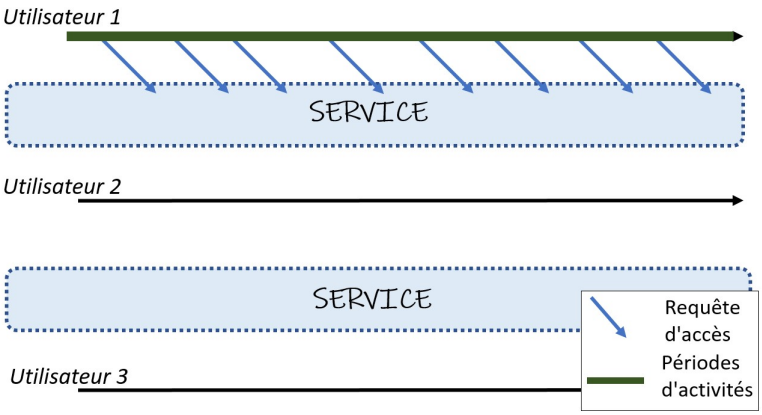


FIGURE 8.8 – Illustration du scénario 1 : un seul utilisateur actif

2. Comportement en alternance : deux groupes d'utilisateurs changent de comportements à intervalle régulier. À chaque nouvel intervalle le groupe d'utilisateurs actif devient inactif et inversement. Certains utilisateurs peuvent rester inactifs sur la durée de l'exécution. Ce scénario est illustré par la Figure 8.9 où les utilisateurs 1 et 2 alternent les périodes d'activités pendant que l'utilisateur 3 reste inactif ;
3. Comportement en alternance déséquilibrée : deux groupes d'utilisateurs sont alternativement actifs sur des intervalles de temps propre à chaque groupe. Ce scénario représente, par exemple, une activité d'un groupe d'utilisateurs

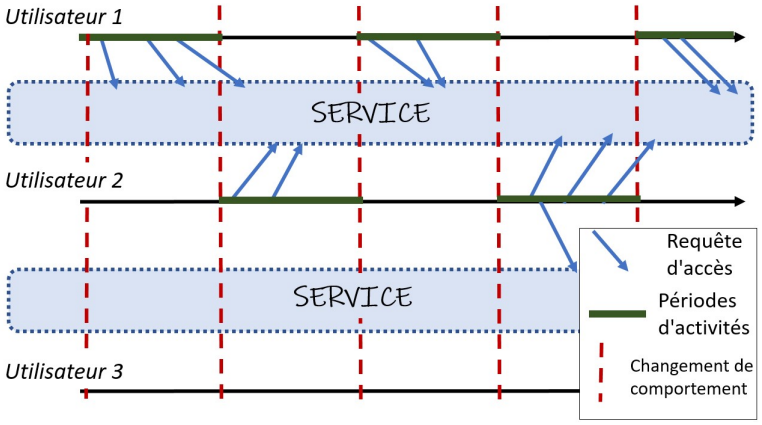


FIGURE 8.9 – Illustration du scénario 2 : alternance périodique et équilibrée

utilisant un service à deux locations différentes, comme un lieu de travail et un domicile et est illustré par la Figure 8.10 où les utilisateurs 1 et 2 alternent les périodes d'activités pendant que l'utilisateur 3 reste inactif ;

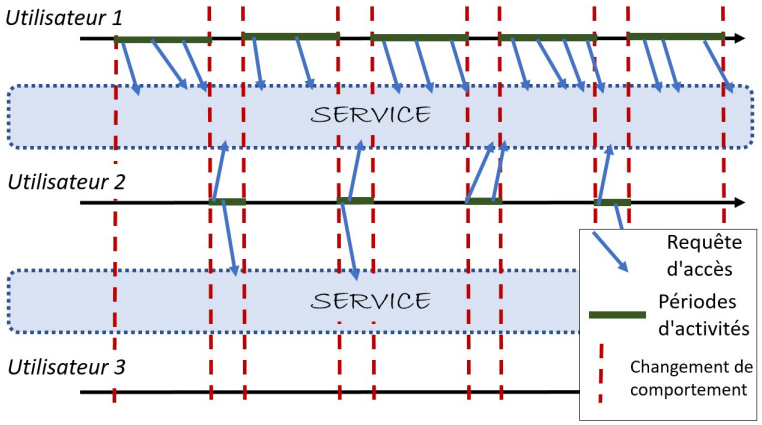


FIGURE 8.10 – Illustration du scénario 3 : alternance périodique et déséquilibrée

4. Comportement irrégulier : les utilisateurs n'ont pas d'habitude particulière et peuvent changer de comportement à tout moment. Ce scénario est illustré par la Figure 8.11.

Pour chacun de ces scénarios, nous évaluons nos algorithmes en manipulant des données gérées par des protocoles de cohérence forte et à terme. Nous utilisons les résultats de l'étude précédente, les coefficients $(c_c; c_a)$ sont alors fixés à $(50; 50)$ pour le protocole de propagation bloquante et $(10; 90)$ pour le protocole de propagation non bloquante. Les autres paramètres des simulations restent inchangés.

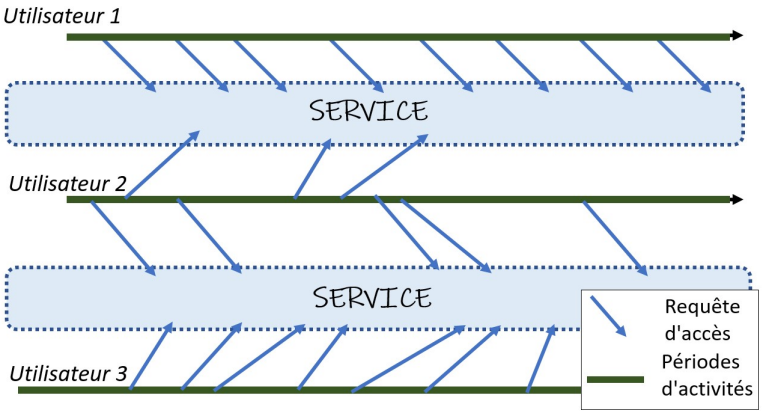


FIGURE 8.11 – Illustration du scénario 4 : comportement aléatoire

Donnée gérée par un protocole de propagation non bloquant

Les résultats des évaluations de CAnDoR en utilisant un protocole de propagation non bloquant pour les scénarios $1, 2, 3$ et 4 sont illustrés par la Figure 8.12.

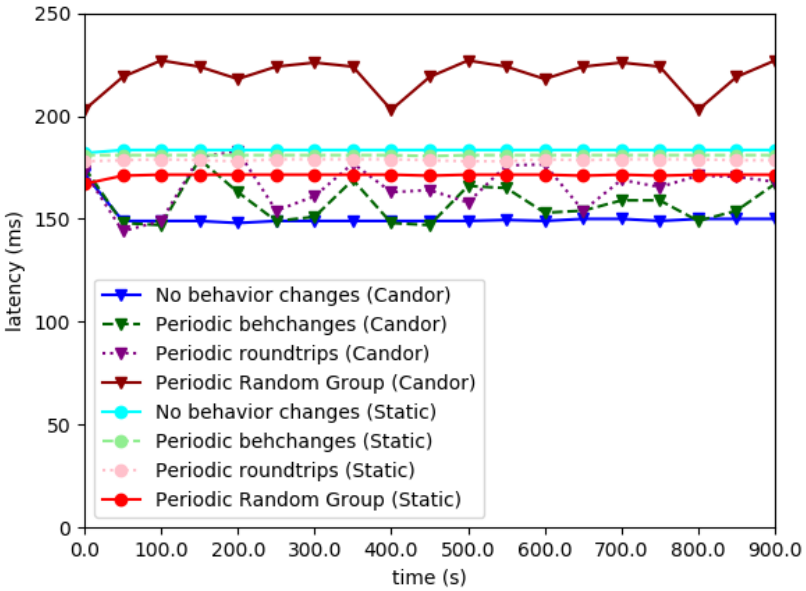
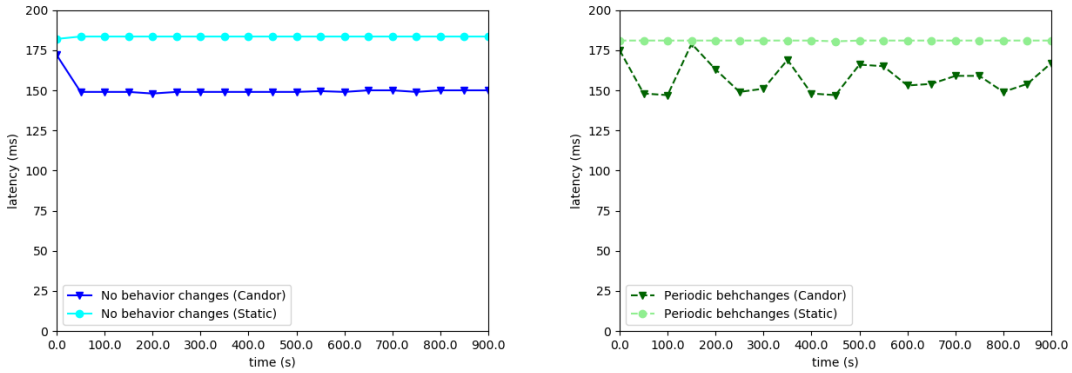


FIGURE 8.12 – Performances de CAnDoR suivant les différents scénarios

Nous pouvons y observer que notre approche permet d'améliorer les performances du service lorsque le comportement des utilisateurs est prévisible. Dans

les situations où les utilisateurs actifs ne changent pas ou peu, comme illustré dans la Figure 8.13a, les différents placements proposés par CAnDoR permet une diminution de l'ordre de 20% du délai de réponse, passant de 185 ms à légèrement moins de 150 ms. Lorsque les utilisateurs changent de comportement, tout en étant prévisible, comme illustré par les Figures 8.13b et 8.14a, les placements proposés CAnDoR sont rapidement mis à jour et permettent des gains de performances allant jusqu'à 20% ici encore. Nous pouvons toutefois noter que lorsque les utilisateurs changent de comportement, le service met un temps à se réadapter. L'analyse menée dans le Chapitre 6 nous a permis de réduire ce temps d'adaptation. Durant ces périodes, le service à des performances similaires aux placements statiques, permettant de valider l'utilisation de CAnDoR dans ces scénarios.



(a) performances de CAnDoR en suivant le premier scénario

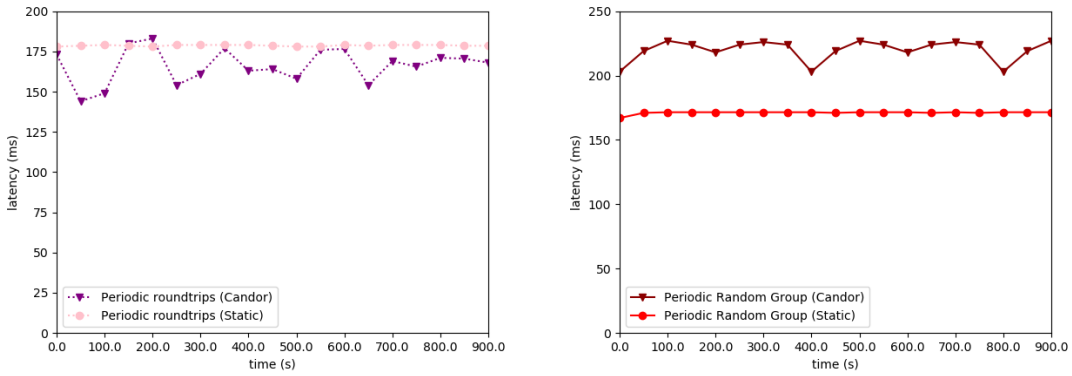
(b) performances de CAnDoR en suivant le deuxième scénario

FIGURE 8.13 – Performances de CAnDoR en suivant le premier et deuxième scénario

Cependant si le comportement des utilisateurs est imprévisible et décorrélé dans le temps, représenté par la Figure 8.14b, nous pouvons remarquer que les placements de CAnDoR ne permettent pas d'obtenir de bons résultats. En effet, les délais de réponses obtenus dans ce cas dépassent les 220 ms, étant systématiquement pire que les résultats obtenus avec un placement statique. Cette différence de performances s'explique par les tentatives d'adaptations de CAnDoR. Ces dernières se basent actuellement sur les actions passées des utilisateurs, n'étant donc pas adaptées à des situations où les utilisateurs sont imprévisibles.

Donnée gérée par un protocole de propagation bloquante

Les résultats des évaluations de CAnDoR en utilisant un protocole de cohérence forte pour les scénarios 1, 2, 3 et 4 sont illustrés par la Figures 8.15. Nous pouvons



(a) performances de CAnDoR en suivant le troisième scénario

(b) performances de CAnDoR en suivant le quatrième scénario

FIGURE 8.14 – Performances de CAnDoR en suivant le troisième et quatrième scénario

remarquer que les gains de performances suivent une logique similaire au service utilisant un protocole de propagation non bloquant.

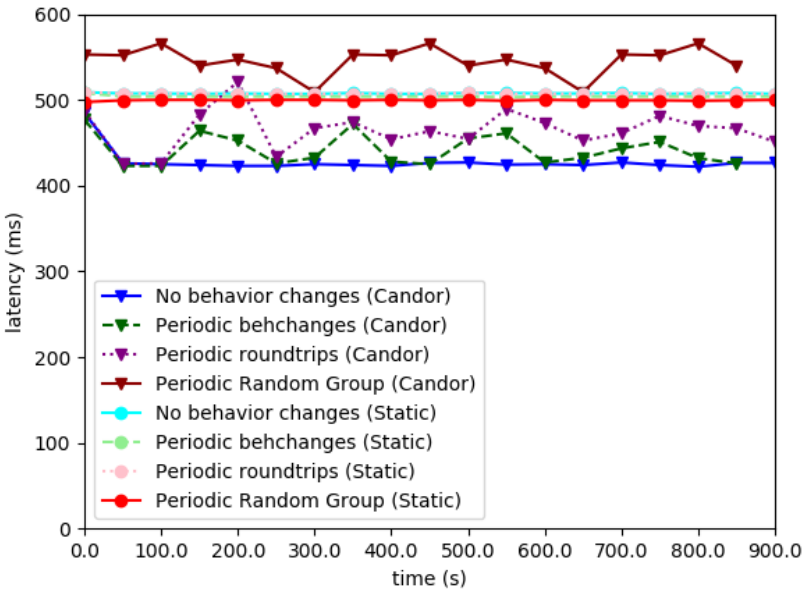
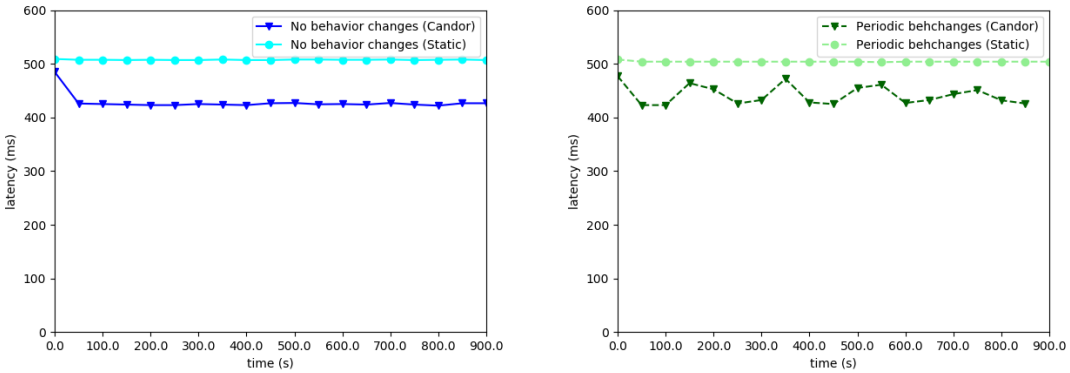


FIGURE 8.15 – Performance de CAnDoR suivant les différents scénarios

En particulier, nous pouvons observer que, lorsque le comportement des uti-

lisateurs est stable, décrit par la Figure 8.16a , l'approche CAnDoR permet une amélioration des performances de l'ordre de 20 % par rapport à un placement statique avec une latence médiane de 410 ms contre 500 ms.



(a) Performance de CAnDoR en suivant le premier scénario

(b) Performance de CAnDoR en suivant le deuxième scénario

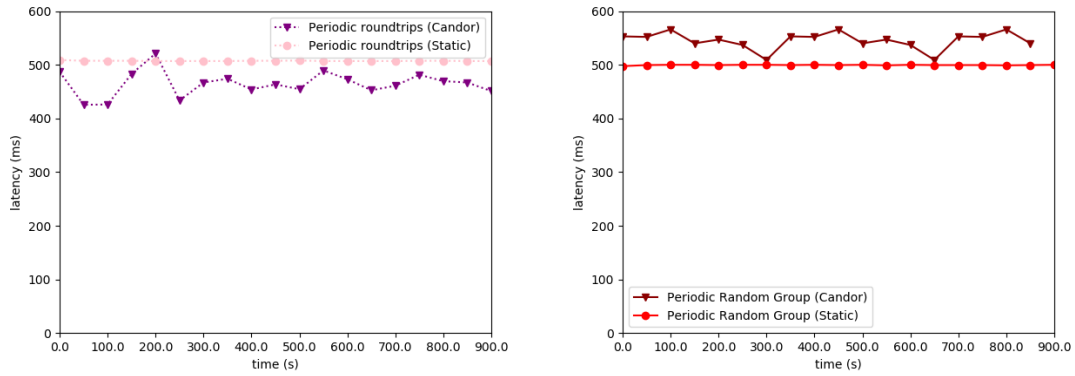
FIGURE 8.16 – Performance de CAnDoR en suivant le premier et deuxième scénario

Lorsque les utilisateurs suivent un comportement en alternance, représenté sur les Figures 8.16b et 8.17a nous pouvons observer un gain de performances jusqu'à 15% avec des pics de latence lors d'un changement de groupe actif. Dans le cas de comportement en alternance équilibré, ces pics de latence restent toujours inférieurs à la latence obtenue avec un placement statique. Une fois encore, ces pics s'expliquent par le temps d'adaptation de nos méthodes. La connaissance des groupes actifs ne peut être établie avant que ceux-ci soient actifs. Nous pensons qu'utiliser une méthode de détection des comportements plus complexes permettrait d'obtenir de meilleurs résultats dans ces scénarios.

En revanche lorsque les utilisateurs suivent un comportement irrégulier, CAnDoR ne permet pas d'obtenir de bonnes performances. Dans ce cas, un placement statique s'avère plus efficace. En effet, malgré l'utilisation d'une stratégie se basant sur des événements évanescents, CAnDoR va essayer d'adapter le placement des réplicas en fonction des utilisateurs précédemment actifs, qui sont décorrélés des utilisateurs qui seront actifs. De fait notre approche s'avère inefficace dans ce type de scénario.

Conclusion de l'étude sur les différents scénarios

Cette seconde étude nous permet d'observer que CAnDoR permet d'améliorer les performances du service tant que les utilisateurs sont prévisibles. Nous n'avons eu



(a) Performance de CAnDoR en suivant le troisième scénario

(b) Performance de CAnDoR en suivant le quatrième scénario

FIGURE 8.17 – Performance de CAnDoR en suivant le troisième et quatrième scénario

recours qu'aux 4 scénarios détaillés précédemment, mais nous pensons que ceux-ci permettent de recouvrir une large partie des cas d'utilisation. Une méthode de détection et prévision des comportements plus complexe que de simplement utiliser le nombre de requêtes devrait permettre d'améliorer davantage les performances, notamment en évitant les pics de latences.

Nous projetons également d'étudier les méthodes permettant de réduire la liste des ensembles potentiels afin de réduire la charge de calcul nécessaire pour déterminer un placement. En particulier, si un ensemble de nœuds ne représente pas un bon ensemble, alors il devrait être possible de détecter les ensembles similaires et de les ignorer pour le reste des calculs.

8.2 Étude de cas des différentes stratégies d'utilisations de l'historique des événements

Nous présentons dans cette section une étude portant sur les différentes stratégies d'utilisations de l'historique, telles que présentées dans le Chapitre 6. Cette étude a pour objectif de déterminer leur impact sur l'utilisation de CAnDoR. Le besoin de comprendre la meilleure façon d'utiliser l'historique des événements s'est fait sentir lors de nos travaux sur le placement dynamique des données. Nous utiliserons donc un tel service pour l'étude de cas mais nous pensons toutefois que ces résultats peuvent se généraliser aux services ayant besoin de construire une vue cohérente de l'activité d'un système.

Dans cette section nous comparons donc trois variantes de ce service : $\mathcal{C}_{IT}$ une version de CAnDoR utilisant son historique de manière intemporelle, $\mathcal{C}_{FT}$ reposant sur l'utilisation de fenêtre glissante et $\mathcal{C}_{EE}$, utilisant des événements évanescents. La fonction de pondération utilisée par $\mathcal{C}_{EE}$ est la fonction $p^t_{\mathcal{S}_{EE}}(t_i) = \frac{100}{2^\delta},\ \delta = \lfloor \frac{t-t_i}{\tau} \rfloor$. $\mathcal{C}_{FT}$ et $\mathcal{C}_{EE}$ ont tous deux une période dynamique, mise à jour via le nombre de requêtes reçues, comme détaillé dans la Section 8.2.1.

Chacun de ces trois services a été évalué lorsque les sources suivent les scénarios décrits dans la Section 6.2.1, que nous rappelons ici :

1. Habitudes stables : les sources ont un comportement constant durant la totalité de l'exécution : une source active pour une cible à un instant t_0 l'est à tout instant de l'exécution ;

2. Habitudes en alternance : deux groupes de sources changent de comportements à intervalle régulier. À chaque nouvel intervalle le groupe de sources actif (pour une cible) devient inactif et inversement. Certaines sources peuvent rester inactives sur la durée de l'exécution ;

3. Habitudes en alternance déséquilibrée : deux groupes de sources sont alternativement actifs sur des intervalles de temps propre à chaque groupe ;

4. Habitudes irrégulières : les sources n'ont pas d'habitude particulière et peuvent changer de comportement à tout moment.

Comme détaillé dans le Chapitre 7, nous utilisons ici 100 clusters capables de stocker des réplicas de données et des groupes de 10 utilisateurs parmi 100 émettent des requêtes sur une donnée. Nous nous intéressons à la médiane des temps d'accès de ces utilisateurs pour chaque situation. Nous utilisons $\mathcal{P}$, un service de placement statique adapté à la cohérence des données comme cas de référence.

Habitudes stables

Dans ce scénario, les utilisateurs actifs sont déterminés au début de l'exécution et restent actifs pour le restant de l'exécution. Le service de placement s'intéressant principalement au ratio de requête émises par un utilisateur sur le nombre total de requêtes reçues, l'utilisation de différentes stratégies n'a que peu d'impact : ce ratio est globalement similaire que l'on considère tout ou une partie de l'historique. Nous pouvons effectivement observer sur la Figure 8.18 que les trois stratégies produisent un résultat similaire. De plus, comme le montre les résultats, ces méthodes sont très efficaces comparées à un placement statique, qui ne peut pas déterminer à l'avance quels seront les utilisateurs actifs. En effet les trois stratégies effectuent une migration des données au bout de quelques minutes pour passer d'une

latence médiane de 175 millisecondes (placement des réplicas lors de leur création) à une latence médiane de 130 millisecondes. Ce placement reste stable dans le temps. $\mathcal{C}_{FT}, \mathcal{C}_{EE}$ changent occasionnellement de placement si un utilisateur est ponctuellement plus actif.

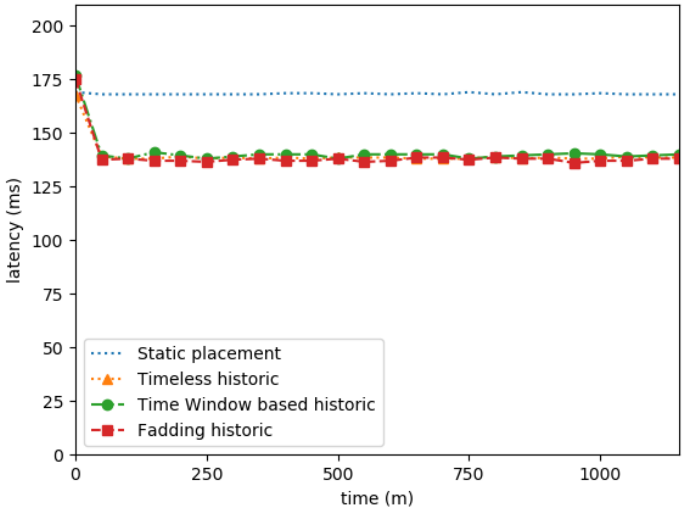


FIGURE 8.18 – Études des différentes stratégies d'utilisation de l'historique avec le scénario 1

Habitudes en alternance

Dans ce scénario, les sources sont divisées en trois groupes :

1. groupe A : ce groupe est actif une période sur deux et est constitué de 10 utilisateurs ;
2. groupe B : ce groupe est actif une période sur deux, quand le groupe A ne l'est pas, et est constitué de 10 utilisateurs ;
3. groupe C : ce groupe n'est jamais actif. Il est constitué des 80 utilisateurs restants.

L'exécution est divisée en périodes d'environs 250 minutes chacune, le groupe A et B alternant les périodes d'activité. La Figure 8.19 illustre les résultats obtenus avec ce scénario. Nous pouvons constater qu'après chaque changement de comportement, $\mathcal{C}_{FT}, \mathcal{C}_{EE}$ ont ponctuellement un pic de latence (environ 160 millisecondes) mais propose rapidement un placement permettant d'offrir de bonnes performances

au groupe B. $\mathcal{C}_{IT}$ en revanche, met beaucoup plus de temps à proposer des placements efficaces lors d'un changement groupe (plus de 100 minutes pour le premier changement). Nous pouvons noter qu'au fil du temps, et du nombre de changements de groupe, le pic de latence est plus faible, mais plus long à corriger. Cela vient directement du fait que ce service donne la même importance à tous les événements. Après suffisamment de temps, un changement d'activité n'impactera que peu les ratios de requêtes émises et $\mathcal{C}_{IT}$ propose alors un placement médian aux deux groupes sans considérer celui qui est actif.

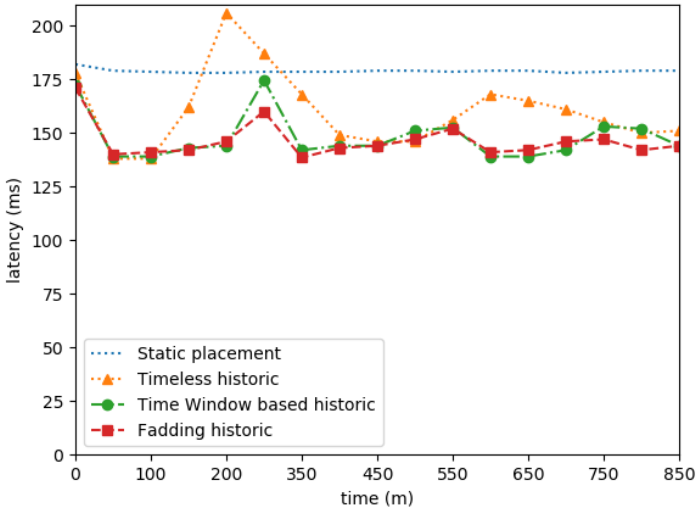


FIGURE 8.19 – Études des différentes stratégies d'utilisation de l'historique avec le scénario 2

Habitudes en alternances déséquilibrée

Dans ce scénario, les sources sont divisé en trois groupes. Ces groupes ce comportent de la même façon que les groupes présentés dans la section précédente à la différence notable que les temps d'activité de chaque groupe n'est pas le même.

L'exécution est divisée en périodes d'environ 250 minutes pour le groupe A et de 50 minutes pour le groupe B alternant les périodes d'activité. Les résultats de ce scénario sont représentés sur la Figure 8.20. Nous pouvons y observer que $\mathcal{C}_{IT}$ permet d'obtenir de bonne performances lorsque le groupe A est actif mais de mauvaise performances lors de l'activité du groupe B. En donnant le même poids à toutes les requêtes, $\mathcal{C}_{IT}$ avantage grandement le groupe A, qui a de plus longues périodes d'activité. $\mathcal{C}_{FT}$ et $\mathcal{C}_{EE}$ permettent d'obtenir de bonnes performances, bien que légèrement moins efficaces que dans les scénarios précédents. En effet, à chaque

changement de groupe actif, il faut peu de temps à ces services pour s'adapter et proposer une configuration efficace.

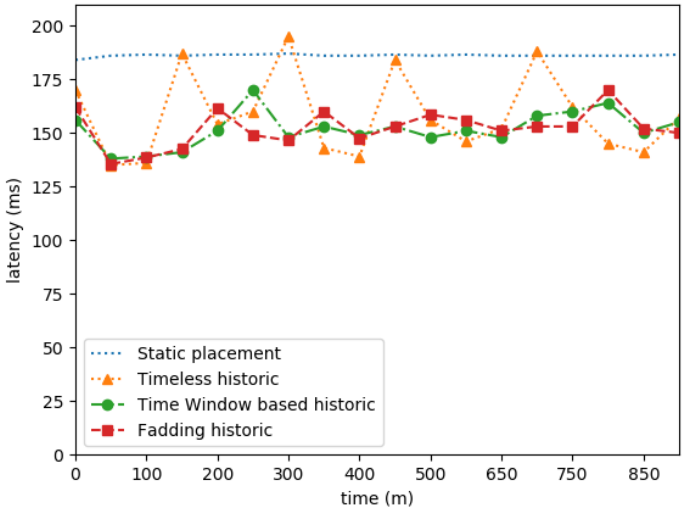


FIGURE 8.20 – Études des différentes stratégies d'utilisation de l'historique avec le scénario 3

Habitudes irrégulières

Dans ce scénario, les sources ne suivent pas de comportements particuliers : chaque utilisateur peut émettre ou non des requêtes de manière régulière ou non. Les résultats de cette simulation sont décrits dans la Figure 8.21. Nous pouvons y observer que les méthodes de placements dynamiques ne permettent pas de placement plus efficace que le placement statique. En particulier, $\mathcal{C}_{FT}$ et $\mathcal{C}_{EE}$ essayent de s'adapter aux événements récents, qui sont décorrélés des événements suivants, menant à de mauvaises performances. $\mathcal{C}_{IT}$ converge vers un placement équivalent à tous les utilisateurs, approchant les performances d'un placement statique ne prenant pas en compte le comportement des utilisateurs.

8.2.1 Étude de cas des méthodes d'actualisation de la période

De nombreuses stratégies d'utilisation de l'historique des événements utilisent une période τ, pour déterminer la taille de la fenêtre ou du palier selon les cas. Cette période peut être fixée par le concepteur ou évoluer dynamiquement lors de l'exécution. Une période fixe demande une étude *a priori* du service afin de déterminer

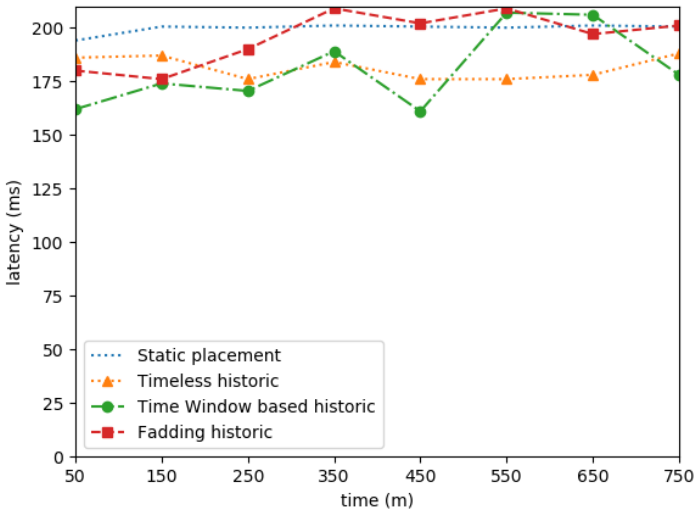


FIGURE 8.21 – Études des différentes stratégies d'utilisation de l'historique avec le scénario 4

la taille correspondant le plus. Si cette étude s'avère complexe, ou si une période unique n'est pas adaptée au service, il est alors nécessaire de mettre en place un algorithme de mise à jour de la période. Cet algorithme doit également répondre aux spécificités du service. Chacune de ces méthodes demande donc une bonne connaissance du service en place. Nous avons effectué une dernière étude de cas sur les différentes méthodes d'adaptation de cette période dans le cadre du placement dynamique des réplicas de données. Pour cela nous utilisons CAnDoR, le service que nous avons développé dans cette thèse afin de permettre un placement dynamique des données adaptées aux contraintes de cohérences. Ce service se base sur les requêtes des utilisateurs, ainsi que des contraintes liées aux garanties des données, afin de proposer un placement efficace. Les mécanismes de CAnDoR ont été détaillés dans le Chapitre 5 et le fonctionnement global du simulateur utilisé dans le Chapitre 7. L'historique des événements considéré est donc constitué de requêtes d'accès en lecture ou écriture de la part d'utilisateurs. Les adaptations de ce service sont des migrations de données. Le coût des adaptations est donc celui d'une migration. La mesure de performance de CAnDoR est simplifiée, dans cette section, à la latence médiane d'accès des utilisateurs.

Dans le cadre de cette évaluation, CAnDoR cherche un placement efficace parmi les différents clusters alors que les utilisateurs suivent le comportement en alternance détaillé dans la Section 6.2.1 : Au début de l'exécution, deux groupes d'utilisateurs sont tirés aléatoirement. Ces deux groupes alternent leur période d'activité

(i.e. les utilisateurs du groupe actif envoient de nombreuses requêtes d'accès à la donnée).

Les Figures 8.22 et 8.23 illustrent une étude comparative du temps de latence médiane des utilisateurs actifs entre un placement statique (en gris sur les graphiques), un placement dynamique en utilisant un stratégie se basant sur une fenêtre glissante (Figure 8.22) et une stratégie utilisant des événements évanescents (Figure 8.23), à chaque fois avec une période fixe de 500 secondes (8, 3 minutes, en bleu continu) puis de 1000 secondes (166 minutes, en vert pointillé).

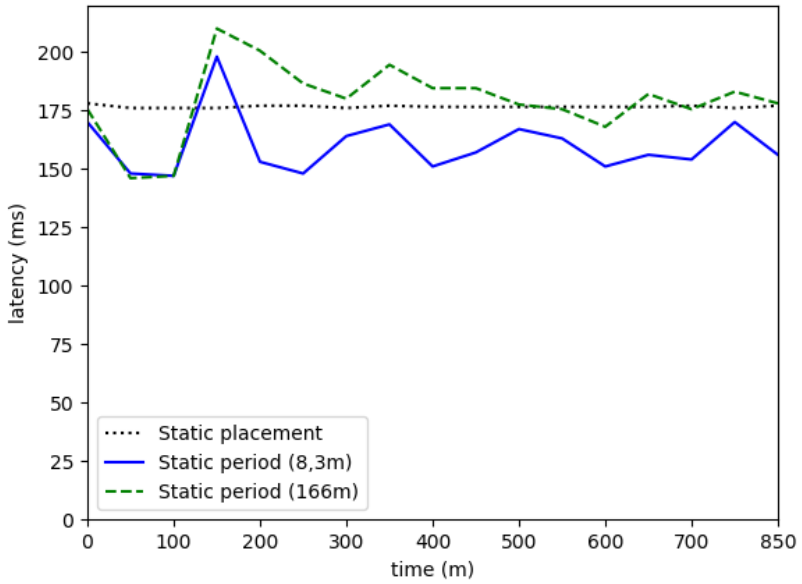


FIGURE 8.22 – Performance des méthodes avec une fenêtre glissante de période fixe

Nous pouvons noter que ces deux périodes fixes permettent d'obtenir des résultats similaires que ce soit en utilisant une fenêtre glissante ou des événements évanescents. Lors de l'utilisation d'une stratégie utilisant des événements évanescents, la période de 166 minutes s'avère trop longue et rapproche le service d'un placement statique. Une période de 8, 3 minutes permet, en revanche, d'obtenir de bons résultats et une adaptation rapide.

Les différentes méthodes d'adaptation de la période reposent sur l'observation suivante : si un changement de configuration entraîne une perte de performance, alors cette configuration est jugée inadaptée. Cette mauvaise adaptation peut survenir pour plusieurs raisons :

1. les sources ont changé de comportement de manière imprévisible ;

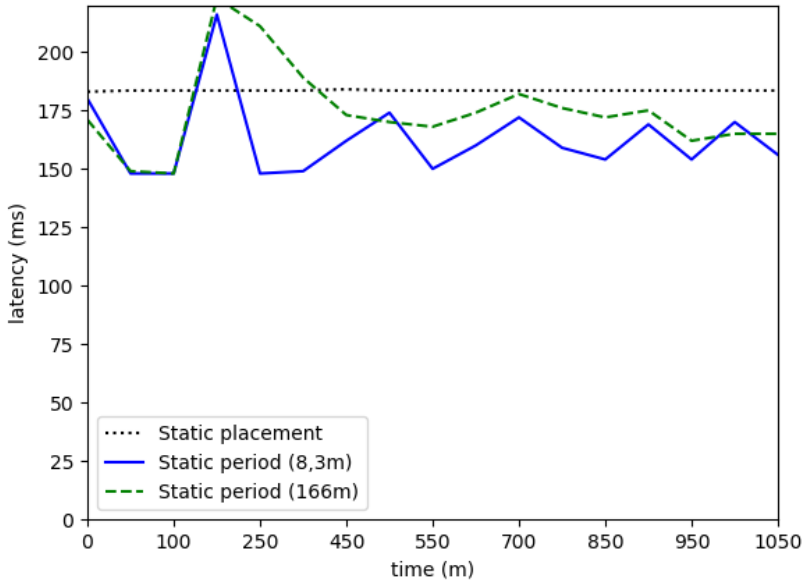


FIGURE 8.23 – Performance des méthodes avec des événements évanescents de période fixe

2. trop peu d'événements ont été utilisés avec un poids suffisant lors des estimations ;
3. trop d'événements ont été utilisés avec un poids trop important lors des estimations.

Dans le cas de changements de comportements imprévisibles, aucune méthode ne permettrait de mettre en place des configurations efficaces de manière régulière. Nous avons donc choisi de ne pas développer davantage cette situation. Nous considérons donc qu'une perte de performance est maintenant liée à une période trop longue ou trop courte. Comme précisé en 6.2.3, plus la période d'évanescence (ou la taille de la fenêtre) est courte, plus les événements passés auront rapidement un poids négligeable face aux événements récents. De ce fait, le service cherche à s'adapter à l'activité présente sans donner un poids important au passé, ce qui entraîne une augmentation du nombre de changements de configuration.

Situation de sur ou sous-adaptation

La première méthode d'adaptation de la période repose sur l'observation de la capacité d'adaptation du service. Le service est doté d'un compteur indiquant le nombre de fois où le service a changé de configuration par rapport au nombre de

calculs pour déterminer si une nouvelle configuration est nécessaire. Un compteur proche de 1 signifie que le service est en *sur-adaptation*. En effet, une valeur proche de 1 est obtenu lorsque le service lance une adaptation dès que cela est possible. Dans ce cas de figure, il est probable que le service considère une quantité insuffisante d'événements. Il convient alors d'augmenter la taille de la période afin de donner davantage de poids aux événements.

À l'inverse, un compteur proche de 0 indique une *sous-adaptation* de la part du service. Cette fois, le service ne change de configuration que rarement, ce qui peut être causé par l'affectation d'un poids trop important à des événements qui ne devraient avoir que peu d'impact dans les calculs du service. Dans ce cas, le service doit raccourcir la taille de la période afin de permettre aux événements les plus récents d'avoir plus d'impact dans les calculs et ainsi permettre au service de s'adapter.

Cette approche est simple à mettre en place, mais est cependant peu précise. En effet, il est possible que le service ne change plus de configuration si une configuration optimale a été trouvée et tant que les sources ne changent pas de comportement. Nous recommandons donc d'utiliser cette approche en complément d'une approche plus fine.

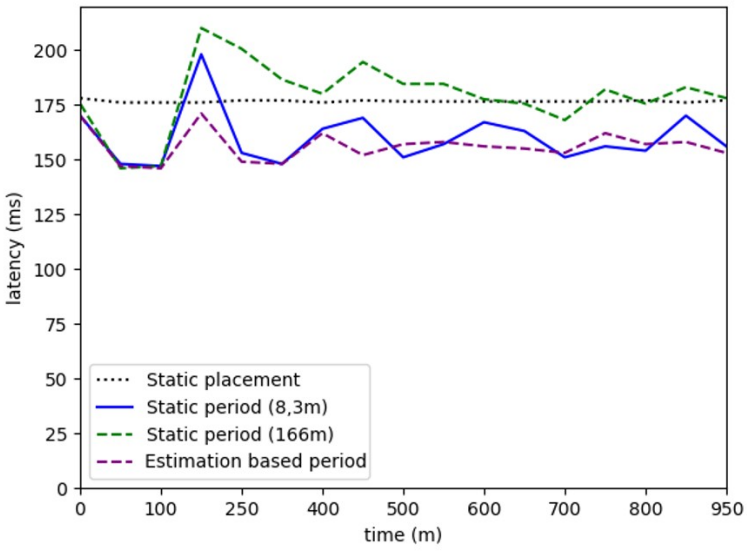


FIGURE 8.24 – Méthode d'actualisation d'une fenêtre glissante avec une période basée sur le nombre d'adaptations précédentes

Les Figure 8.24 et 8.25 illustrent les performances obtenues en utilisant cette méthode de mise à jour (en violet) et la compare avec l'utilisation de périodes

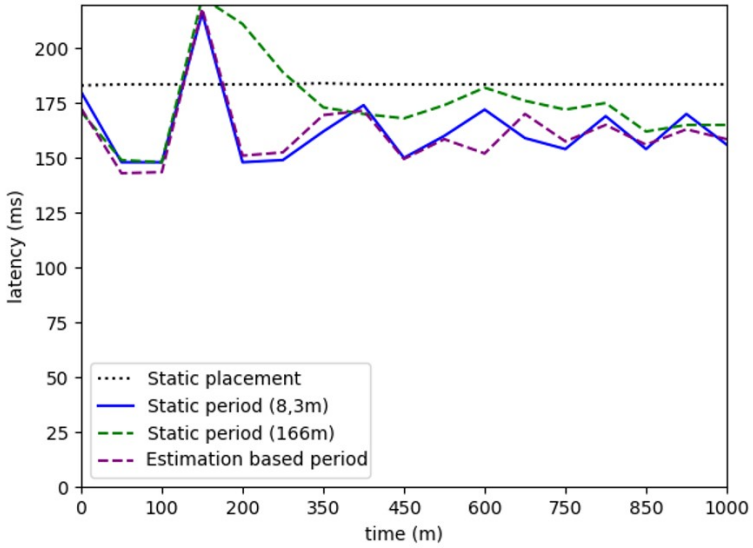


FIGURE 8.25 – Méthode d'actualisation d'événements évanescents avec une période basée sur le nombre d'adaptations précédentes

statiques d'une fenêtre glissante (Figure 8.24) et d'événements évanescents (Figure 8.25). Nous pouvons observer que cette méthode nous permet d'obtenir des résultats sensiblement plus efficaces en utilisant des événements évanescents mais globalement équivalents.

Inspiration du shadow page cache

La seconde méthode d'adaptation de la taille de la période est inspirée d'un mécanisme du noyau UNIX : les *shadow page cache* ([Joh], [Jon]). Sommairement, cette approche consiste à détecter l'accès à des pages récemment sorties du cache. Dans le cadre d'un service de placement de donnée, nous nous intéressons à conserver une configuration précédente en mémoire afin de déterminer si les utilisateurs auraient contacté un noeud de la configuration précédente.

Lorsque le service décide de changer de configuration, i.e. de changer quels nœuds du système stockent des réplicas des données, ce dernier garde donc en mémoire la configuration précédente. Lorsqu'un nœud reçoit une requête, celui-ci doit calculer sa performance pour y répondre et vérifier si un nœud de la configuration précédente y aurait répondu plus rapidement. Le service peut alors déterminer le nombre de requêtes qui aurait eu un meilleur traitement dans la configuration précédente.

Cette approche permet de détecter les périodes trop courtes. En effet, comme

rappelé précédemment, une période trop longue entraîne un immobilisme des adaptations et non pas à des changements de configurations non performantes. Si le service détermine que la configuration actuelle est significativement moins efficace que la précédente, ce dernier va alors augmenter la durée de la période.Toutefois, cette approche ne permet pas de déterminer si la période est trop longue et doit être réduite. En effet, dans ce cas le service aurait tendance à ne pas proposer de nouvelle configuration assez rapidement. Ce phénomène n'étant pas détectable par cette méthode, il est nécessaire de s'assurer que le la période n'augmente pas de manière trop sensible. Cette méthode se révèle cependant être très coûteuse en calcul durant l'exécution : pour chaque requête, il faut évaluer si un nœud de la configuration précédente aurait été plus efficace, ce calcul pouvant être fait de manière périodique ou au fil des requêtes.

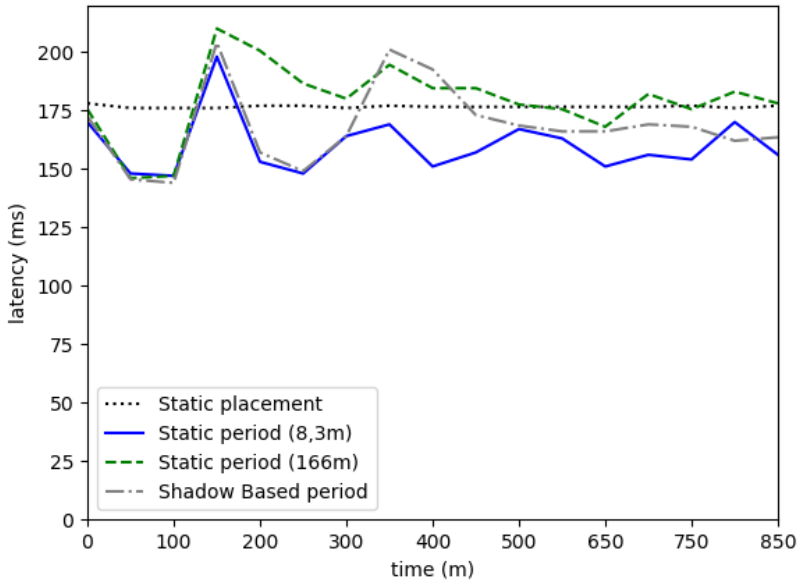


FIGURE 8.26 – Méthode d'actualisation d'une fenêtre glissante avec une période basée sur les résultats précédents

Les Figures 8.26 et 8.27 illustrent les performances obtenues en utilisant cette approche pour actualiser la période (en gris) et la compare avec l'utilisation de périodes statiques. Nous pouvons observer qu'une fois encore cette approche n'apporte pas de différence significative. Les résultats obtenus de cette manière sont même légèrement moins bon que lors de l'utilisation de période statiques.

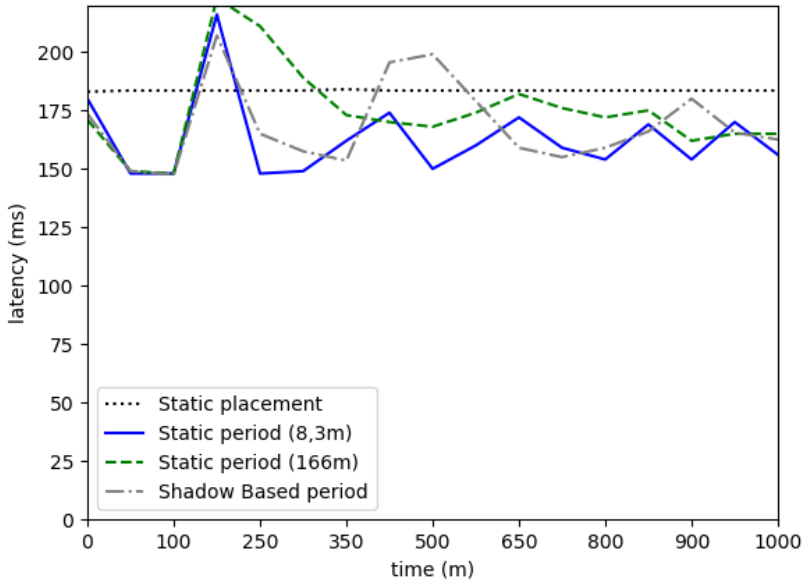


FIGURE 8.27 – Méthode d'actualisation d'événements évanescents avec une période basée sur les résultats précédents

Adaptions selon le nombre de requêtes

La troisième méthode que nous avons considérée ne se base pas sur le nombre d'adaptations du service, mais le nombre d'éléments utilisés pour les calculs du service. Dans un premier temps, nous avons déterminé le nombre k d'événements nécessaires à une estimation correcte de la part du service. Lors d'une nouvelle période, le service observe le nombre n d'événements ajouté dans l'historique depuis la dernière fenêtre de temps (i.e. le nombre d'événements dont le poids vaut 100). Si n n'est pas proche de k (i.e. $n \notin [k - \epsilon; k + \epsilon]$ avec ϵ le seuil de tolérance) alors le service augmente ou réduit la taille de la période pour permettre aux futures valeurs de n de correspondre au voisinage de k.

Cette approche permet une adaptation de la taille de la période selon l'activité des sources : lorsque les sources sont peu actives, le service augmentera la taille de la période. À l'inverse, dans les périodes de haute activité des sources, où une adaptation est plus profitable, la taille de période sera réduite, permettant ainsi une adaptation plus rapide et une meilleure qualité de service.

Les Figures 8.28 et 8.29 illustrent les performances obtenues en utilisant cette approche pour actualiser la période (en rouge) avec $k = 50$ et $\epsilon = 5$ et la compare avec l'utilisation de périodes statiques pour une fenêtre glissante (Figure 8.28) et d'événements évanescents (Figure 8.29). Les valeurs de k et ϵ ont été obtenues par

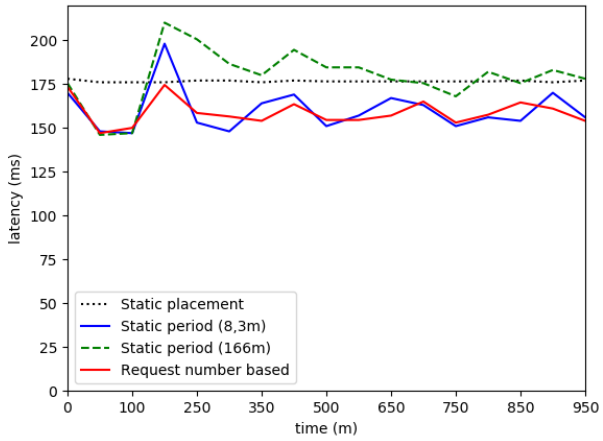


FIGURE 8.28 – Méthode d'actualisation de la période d'une fenêtre glissante basée sur le nombre de requête

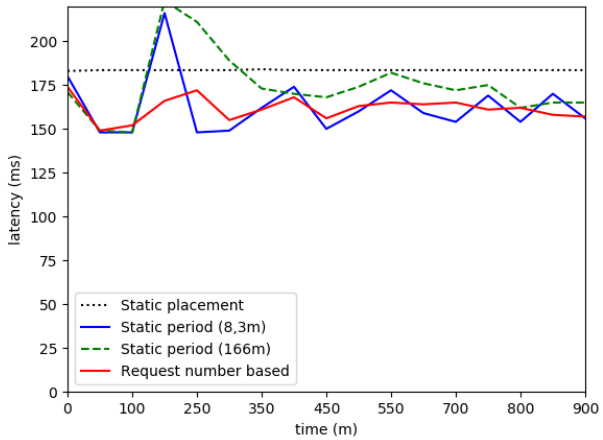


FIGURE 8.29 – Méthode d'actualisation de la période d'événements évanescents basée sur le nombre de requêtes

le biais d'une étude comparatives de différents couples de valeurs. Nous pouvons observer qu'une fois encore cette approche n'apporte pas de différence significative, bien que sensiblement plus efficace.

Conclusion sur les méthodes de réglages de la période

Afin de déterminer l'efficacité de ces méthodes, nous avons effectué une étude comparative de la latence médiane au cours d'une exécution, comme présenté ci-

dessus. Les Figures 8.30 et 8.31 présentent les résultats de cette étude : la Figure 8.30 illustre les différentes performances en fonction de la méthode utilisée avec une stratégie basée sur une fenêtre glissante et la Figure 8.31 en utilisant des événements évanescents.

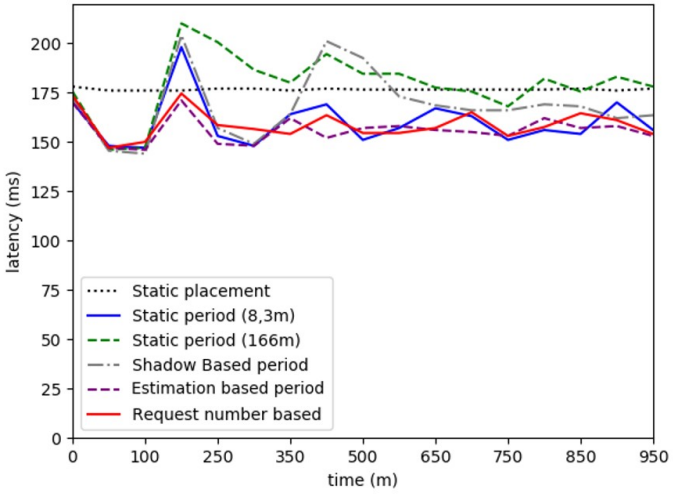


FIGURE 8.30 – Comparatifs des méthodes d'utilisation d'une fenêtre glissante

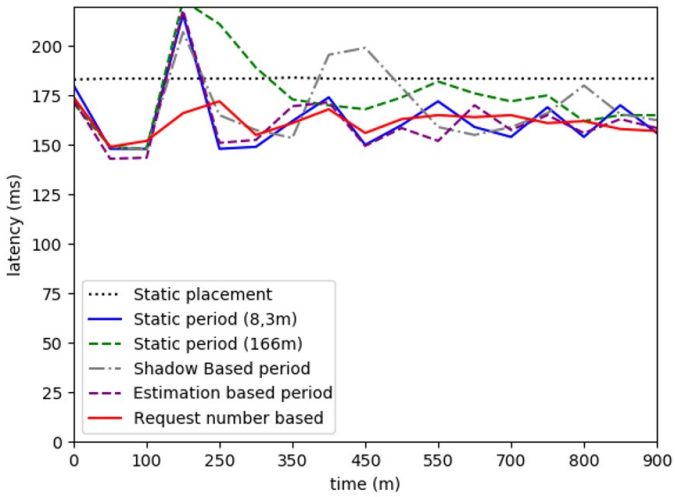


FIGURE 8.31 – Comparatifs des méthodes d'utilisation d'événements évanescents

Cette étude a révélé que, dans le cadre de notre problème de placement de

données, la méthode employée pour mettre à jour la période n'a pas d'impact significatif sur les performances de nos méthodes. L'approche consistant à évaluer les performances des configurations précédentes s'avère peu efficace et coûteuse en calcul. Les méthodes d'actualisation de la période se basant sur le nombre de requêtes ou d'adaptations obtenues permettant d'avoir des résultats similaires mais sensiblement plus efficace que les méthodes de période statique. De plus, l'utilisation d'une méthode statique demande d'être capable d'estimer correctement cette période, nous pouvons en effet observer qu'une mauvaise estimation de la période peut mener à une dégradation du service.

8.3 Conclusion

Nous avons présenté, dans ce chapitre, différentes études se reposant sur notre simulateur. L'étude des performances de CAnDoR expose des résultats prometteurs en se comparant à une méthode statique n'utilisant que les contraintes de cohérence et une méthode de placement aléatoire, utilisée notamment par BigTable [CDG+08] ou Cassandra [LM10]. Nous avons ensuite effectué une étude sur la façon d'utiliser l'historique des événements dans le cadre de CAnDoR afin de déterminer la meilleure stratégie à adopter.

Les résultats de nos expérimentations mettent en avant qu'il est effectivement important de modifier les paramètres de pondération en fonction du modèle de cohérence utilisé. En effet, lorsqu'un protocole de propagation bloquante est utilisé, pour implémenter un modèle de cohérence forte par exemple, il est préférable d'utiliser des valeurs de c_c et c_a équivalentes (au voisinage de 50%). L'utilisation de ces valeurs avec un protocole de propagation non bloquant, pour implémenter un modèle de cohérence à terme par exemple, entraîne une baisse de performances de l'ordre de 11% en comparaison de paramètre plus adapté ($c_a \geq 70\%$). La seconde étude met en avant que l'utilisation de ces paramètres permet un gain de performances en comparaison des méthodes classiques de placement statiques, utilisées par de nombreux systèmes de gestion de données distribués. En effet, l'utilisation de CAnDoR nous permet un gain de 20% sur les temps de latences au quatre-vingt-quinzième centile. Ces résultats soulignent l'importance de la prise en compte des différents critères pour placer les réplicas dans des systèmes à modèle de cohérence adaptable.

Nous avons ensuite mené deux études préliminaires en utilisant CAnDoR ; la méthode de placement dynamique des données présentée dans le Chapitre 5. La première porte sur les méthodes d'actualisation de la taille de fenêtre glissante ou de la période d'évanescence. Cette étude souligne l'importance de ce paramètre : une trop grande période dégrade les performances du service. Cependant les différentes méthodes d'actualisation apportent des résultats similaires à une période

fixe correctement choisie. Nous recommandons donc d'utiliser de ces méthodes dans un premier temps, puis de fixer ce paramètre grâce aux valeurs obtenues par lesdites méthodes.

La seconde étude de cas porte sur les conséquences des différentes stratégies d'utilisation de l'historique. Cette étude a mis en avant l'intérêt des stratégies $\mathcal{C}_{FT}$ et $\mathcal{C}_{EE}$. Dans les différents scénarios que nous avons explorés, ces approches permettent une adaptation rapide et efficace. En utilisant un bon paramétrage de ces stratégies, nous avons obtenu des résultats proches avec ces deux stratégies. En revanche, l'approche intemporelle s'est révélée peu efficace et inadaptée aux systèmes dynamiques. Ces études ont été menée spécifiquement dans le cadre du développement de CAnDoR. Nous prévoyons de mener des études plus approfondies dans de futurs travaux.

Comme nous l'avons évoqué dans la Section 6.2.4, il existe de nombreuses fonctions de pondérations et nous souhaitons dans la suite de nos travaux nous concentrer sur l'approfondissement d'autres fonctions de pondérations, en particulier dans le cadre des stratégies à événements évanescents ou des stratégies de résurgences d'événements évanescents.

IV

Partie IV - Conclusion

Sommaire

Chapitre 9

Conclusion et perspectives

9.1 Retour sur les travaux présentés

Comme nous avons pu le voir dans les différents chapitres de ce manuscrit, la gestion de données à large échelle est devenue un enjeu capital pour de nombreuses applications pouvant désormais proposer des services à échelle mondiale. Ces applications se basent sur des systèmes de gestion de données distribués qui permettent une gestion efficace des quantités de données manipulées par ces applications. En particulier, ces systèmes ont recours à la réplication des données : chaque donnée est répliquée un certain nombre de fois et les différents réplicas sont stockés sur des nœuds différents.

Cette thèse explore les différentes problématiques liées à l'utilisation de ces systèmes. Une attention particulière a été apportée sur les problèmes de cohérence entre les réplicas d'une donnée : Les réplicas étant stockés sur différents nœuds, il est possible d'appliquer des requêtes de manière concurrente, ce qui peut mener à des états non équivalents entre les réplicas. C'est ce qu'on appelle l'incohérence entre les réplicas. Nous avons cherché à définir et expliquer les nuances et conséquences de certains modèles de cohérence, parmi les plus courants. Nous nous sommes en particulier intéressés aux modèles de cohérence forte, causale et à terme. Cette analyse est nécessaire afin de bien comprendre les enjeux de ces modèles et leur relation avec la disponibilité et la résilience des données. En effet, le théorème du CAP Th. 2 stipule qu'il est impossible pour un système distribué

de garantir des propriétés de cohérence forte et une grande disponibilité, tout en étant tolérant aux partitions. Il apparaît donc que lors de la conception d'un système de gestion de données, il convient de déterminer quel est l'équilibre entre cohérence et disponibilité souhaitable. Cet équilibre peut varier d'une application à l'autre, par exemple des applications comme Amazon ou Facebook cherchent à fournir le service le plus rapide possible, sans avoir besoin de garanties fortes sur la cohérence entre les réplicas. En revanche, Kafka utilise un système de gestion de données distribué pour monitorer l'état des nœuds et une liste des sujets et messages. Ce système doit donc être capable de fournir des données fiables et donc d'imposer des garanties de cohérences fortes entre les réplicas.

Cependant, certaines applications modernes deviennent de plus en plus complexes et peuvent avoir besoin de différents types de garanties sur différents types de données. Naturellement, nous pouvons observer l'émergence de systèmes de gestion de données à modèle de cohérence adaptable, pouvant proposer des modèles différents pour chaque donnée. Le projet RainbowFS propose d'étudier les possibilités et les garanties de tels systèmes et travaille au développement d'Antidote, un système de gestion de données se basant sur les CRDTs et le modèle de cohérence Rouge/Bleu, permettant de modifier les garanties de cohérence selon les besoins. Dans le cadre du projet RainbowFS, nous avons travaillé sur la question du placement des réplicas dans de tels systèmes. L'étude menée sur les différents modèles de cohérence nous a permis de mettre en avant que différents modèles sont associés à différentes contraintes et qu'il convient alors d'adapter la stratégie de placement en fonction. Nous avons donc mis au point CAnDoR, une méthode permettant de calculer un placement efficace pour les réplicas, en traitant le problème donnée par donnée afin de limiter la charge de calcul liée à la difficulté du problème. Afin de pouvoir proposer une adaptation rapide et efficace, nous nous sommes également intéressés aux différentes méthodes d'utilisation des connaissances dans un système distribué, et plus précisément comment gérer l'impact d'événements passés sur les décisions futures et les changements de configuration du système. Nous avons évalué nos méthodes d'adaptation du poids des événements et notre approche de placement dynamique des réplicas à l'aide d'un simulateur développé durant cette thèse et paramétré à l'aide d'études comparatives de systèmes de gestion de données distribués existants. Les résultats de CAnDoR sont prometteurs et permettent une amélioration du délai de réponse pouvant aller jusqu'à 20% lorsque le comportement des utilisateurs est prévisible. De plus, la prise en compte du modèle de cohérence permet également une amélioration du délai de réponse de 11% en comparaison des méthodes ne se basant que sur le comportement des utilisateurs. Ces résultats mettent en avant le fait que lors de la mise en place de système de gestion de données distribués et à modèle de cohérence adaptable, il est important de considérer ces critères afin de fournir un service efficace. Nous

pensons cependant qu'il existe encore de nombreuses pistes à explorer afin d'affiner notre approche et obtenir de meilleurs résultats ou afin d'orienter les gains de performances pour d'autres métriques, telle que la consommation d'énergie.

9.2 Perspectives de travaux futurs

9.2.1 Un outil d'aide au placement des données

CandorSim permet aujourd'hui de simuler un système simplifié de gestion de données distribué afin d'étudier l'impact du placement des données dans le système. À ce stade du développement, nous considérons la latence comme unique métrique d'évaluation. Mais nous souhaiterions proposer d'autres métriques telles que les dépenses énergétiques ou les temps de convergence des réplicas. Avec ces fonctionnalités, ainsi que la possibilité de fournir des traces d'exécutions réelles, CandorSim pourrait évoluer vers un outil d'aide au placement des données.

Ainsi un utilisateur pourrait mettre en place son système de gestions de données et profiter du retour de notre outil en fournissant quelques propriétés du système (comme les liens de communications entre les noeuds de stockage par exemple) et les traces d'exécution du système. CandorSim estimerait alors les placements des données qui auraient été préférables selon les métriques envisagées par l'utilisateur et lui proposerait alors un placement plus pertinent (soit statique soit évolutif). Cet outil a l'avantage de ne pas être rattaché à un système de gestion préétabli ou à une configuration du système, ces paramètres étant configurables dans CandorSim.

Enfin, en affinant encore davantage l'efficacité du simulateur, il serait possible de l'utiliser comme base d'un système de placement automatisé des réplicas, qui déclencherait directement les migrations des réplicas.

9.2.2 Affinements des stratégies CAnDoR : plus de modèles, green computing, IA

À l'image de CandorSim, notre approche se focalise actuellement sur la réduction du délai de traitement des requêtes. Nous souhaiterions cependant être capables d'adapter notre méthode à différentes métriques, en particulier celle de la consommation d'énergie. En effet, les systèmes de gestions de données distribués consomment énormément d'énergie. Nous pensons qu'il serait possible d'établir des stratégies de placement permettant de réduire ses dépenses en prenant en compte les modèles de cohérences et les requêtes des utilisateurs. Il y a une forte relation entre modèle de cohérence et nombre de communications (ayant donc un impact direct sur les dépenses énergétiques) : un modèle de cohérence forte demandant d'effectuer des synchronisations entres les réplicas, augmentant ainsi le nombre de

communications entre eux. Ces paramètres doivent donc être pris en compte lors de l'évaluation du coût énergétique d'un placement des réplicas.

Une autre piste d'amélioration de Candor serait l'utilisation de méthodes issues de l'intelligence artificielle. À l'aide de méthodes adaptées, nous pourrions fournir aux réplicas des modèles d'apprentissage sur leur placement afin de déterminer l'efficacité des différents placements. Nous évaluons également la possibilité d'avoir recours à des modèles d'apprentissage distribués afin de détecter des motifs récurrents d'utilisation des données. Cet apprentissage permettrait aux données d'anticiper la façon dont les utilisateurs pourraient accéder aux réplicas et d'adapter le placement dans de meilleurs délais afin de fournir une meilleure qualité de service. Ces outils d'apprentissage pourraient également permettre de mieux déterminer le moment opportun pour lancer la migration des données considérant le coût de migration des données à l'encombrement réseau attendu, afin de ne pas dégrader les performances du système lors d'une migration.

9.2.3 Du point de vue de la donnée à la donnée autonome

CAnDoR est une méthode de placement dynamique des données qui se place du point de vue la donnée : le système va chercher, pour chaque donnée, comment répartir efficacement les réplicas dans le système. Il est possible d'approfondir cette démarche et de considérer des données autonomes. Ce nouveau paradigme de programmation vise à redéfinir le rapport des données aux systèmes de gestions de données distribués et propose d'utiliser les connaissances de différents domaines de l'apprentissage et des multi-agents afin d'enrichir les données de comportements autonomes.

Parmi ces comportements, la capacité de se répliquer et de migrer de manière efficace semble primordiale. Les données étant autonomes, il est nécessaire que celles-ci soient capables de maintenir d'elles-mêmes un nombre pertinent de réplicas, afin d'assurer la survie de la donnée sans compromettre la qualité de service potentielle. De même, les données autonomes doivent être capables de déterminer quelle stratégie de migration appliquer afin que les réplicas d'une même donnée se répartissent efficacement dans le système.

Bien que les travaux menés au cours de cette thèse se reposent sur la vision classique des systèmes de gestions de données, ils se concentrent autant que possible sur le caractère indépendant des données des systèmes non transactionnels. De ce fait, les différents résultats obtenus sont étroitement liés aux comportements que devront avoir les données autonomes. En effet, celles-ci devront apprendre quelles sont les meilleures stratégies de placement, et donc de déplacement, selon la place des autres réplicas, des différents comportements des utilisateurs dont elles ont connaissance et des protocoles de cohérences attachés. Il nous parait alors pertinent d'utiliser les résultats observés avec CAnDoR afin de paramétrer efficacement les

bases de l'apprentissage.

Nous envisageons donc de nous intéresser à la mise en place des systèmes de données autonomes en y intégrant les concepts de CAnDoR afin d'être capables de déterminer les bases d'apprentissage pour les méthodes de réplication et de déplacement les plus efficaces.

Annexes

Propriétés, définitions et théorèmes utilisés

Propriétés

Propriété 1 (Ordre global)**.** *Chaque requête est livrée à l'application selon un ordre unique et global.*

Propriété 2 (Ordre causal)**.** *Chaque requête est livrée à l'application selon un ordre causal, établi d'après la relation de causalité Def. 1.*

Propriété 3 (Temps réel)**.** *Si un nœud s_1 a fini de traiter une requête r_α avant qu'un autre nœud s_2 n'ait commencé à traiter une autre requête r_β, alors tous les nœuds doivent appliquer r_α avant r_β.*

Propriété 4 (PRAM (ou FIFO))**.** *Si un nœud s_1 a fini de traiter une requête r_α avant de commencer à traiter une autre requête r_β, alors tous les nœuds doivent appliquer r_α avant r_β.*

Propriété 5 (Convergence (à terme))**.** *Tous les nœuds corrects ayant appliqué les mêmes opérations de mise à jour de la donnée finiront par avoir un état équivalent.*

Propriété 6 (Convergence forte)**.** *Tous les réplicas corrects ayant appliqué les mêmes opérations de mise à jour de la donnée ont un état équivalent.*

Propriété 7 (Réception à terme)**.** *Toutes les opérations émises dans le système finissent par être appliquées sur tous les réplicas des données concernées.*

Propriété 8 (Monotonic Reads)**.** *Les requêtes de lectures successives doivent renvoyer une série d'ensembles croissants de requêtes.*

Propriété 9 (Reads your Writes)**.** *Les requêtes de lectures sont traitées par un nœud qui a appliqué toutes les requêtes d'écriture de la même source.*

Propriété 10 (Monotonic Writes)**.** *Les requêtes d'écritures sont traitées par un nœud qui a appliqué toutes les requêtes d'écriture de la même source.*

Propriété 11 (Writes follows Reads)**.** *Les requêtes d'écritures traitées effectuées sur un nœud ayant appliqué une requête de lecture sur une même donnée doivent être appliquées sur une version au moins aussi récente de la donnée.*

Propriété 12 (t-fraîcheur)**.** *Une requête de lecture est dite* t-fraîche *si elle retourne un état qui prend en compte une opération d'écriture ayant eu lieu depuis* $\tau(R,t)$ *unités de temps avec :*

- $\tau(R,t)$ *la date de la dernière écriture si celle-ci est plus vieille que* t *unités de temps ;*
- $\tau(R,t) = \tau(R) - t$ [1] *si au moins une écriture a été appliquée au cours des* t *unités de temps précédant la requête de lecture.*

Propriété 13 (t-latence)**.** *Une requête de lecture satisfait la* t-latence *si elle retourne en moins de* t *unités de temps.*

Propriété 14 (One-Way convergent)**.** *Un système est dit* One-Way convergent *si, pour tout nœud* s_1 *et* s_2*, si* s_1 *n'émet plus d'écriture et ne reçoit plus de message, alors* s_1 *finira par envoyer un ensemble de messages tels que, si* s_2 *reçoit ces messages alors* s_1 *et* s_2 *convergent tant que* s_2 *ne reçoit pas d'autres écritures.*

1. avec $\tau(R)$ la date de la requête de lecture

Définitions

Définition 1 (Causalité). *Un événement e_1 précède causalement un événement e_2, noté $e_1 \rightsquigarrow e_2$, si une des conditions suivantes est vérifiée :*

1. *e_1 et e_2 se produisent sur le même nœud et e_1 se termine avant le commencement de e_2 ;*
2. *e_1 est l'envoi d'un message par un nœud et e_2 la réception de ce message par un autre nœud ;*
3. *il existe un événement e_3 tel que $e_1 \rightsquigarrow e_3 \rightsquigarrow e_2$.*

Définition 2 (Cohérence linéarisable). *Un modèle de cohérence est dit linéarisable s'il respecte les propriétés d'ordre global et temps réel.*

Définition 3 (Cohérence séquentielle). *Un modèle de cohérence est dit séquentiel s'il respecte les propriétés d'ordre global et PRAM.*

Définition 4 (Cohérence causale). *Un modèle de cohérence est dit causal s'il respecte la propriété d'ordre causal.*

Définition 5 (Cohérence causale en temps réel). *Un modèle de cohérence est dit causal en temps réel s'il respecte les propriétés d'ordre causal et de temps réel.*

Définition 6 (Cohérence causale+). *Un modèle de cohérence est dit causal+ s'il respecte les propriétés d'ordre causale et de convergence forte.*

Définition 7 (Cohérence à terme). *Un modèle de cohérence est dit à terme s'il respecte la propriété de convergence et de réception à terme.*

Définition 8 (Cohérence forte à terme). *Un modèle de cohérence est dit fort à terme s'il respecte la propriété de convergence forte et de réception à terme.*

Définition 9 (Cohérence k-fraîche). *Un modèle de cohérence est dit k-frais, avec une probabilité $1 - p_{sk}$, si, pour tout quorum de lecture, au moins une valeur est issue d'une des k dernière requêtes d'écriture. Avec p_{sk} la probabilité calculée par :*

$$p_{sk} = \left(\frac{\binom{RF-qw}{qr}}{\binom{RF}{qr}} \right)^k$$

Définition 10 (Cohérence t-visible). *Un modèle de cohérence est dit t-visible, avec une probabilité $1 - p_{st}$, si, pour tout quorum de lecture, au moins une valeur est issue d'une requête d'écriture traitée il y a t unités de temps. Avec $P_w(c, t)$ la*

fonction de cumul des densités des c nœuds ayant reçu une mise à jour après t et p_{st} la probabilité calculée par :

$$p_{st} = 1 - \left(\frac{\binom{RF-qw}{qr}}{\binom{RF}{qr}} + \sum_{c \in [qr, RF)} \frac{\binom{RF-c}{qr}}{\binom{RF}{qr}} . [P_w(c+1,t) - P_w(c,t)] \right)$$

Définition 11 (Cohérence $\langle k,t \rangle$-*fraîcheur*)**.** *Un modèle de cohérence est dit* $\langle k,t \rangle$-frais*, avec une probabilité* $1 - p_{skt}$*, si, pour tout quorum de lecture, au moins une valeur est issue d'une des k dernière requêtes d'écriture à condition que la lecture commence t unités de temps après les k dernières requêtes d'écriture. Avec* p_{skt} *la probabilité calculée par :*

$$p_{skt} = 1 - \left(\frac{\binom{RF-qw}{qr}}{\binom{RF}{qr}} + \sum_{c \in [qr, RF)} \frac{\binom{RF-c}{qr}}{\binom{RF}{qr}} . [P_w(c+1,t) - P_w(c,t)] \right)^k$$

Définition 12 (Cohérence (t,p)-probabiliste)**.** *Un modèle de cohérence est dit* (t,p)*-probabiliste si le rapport de requête de lecture t-fraîche est d'au moins* $(1-p)$*.*

Définition 13 (Cohérence ([GL02]))**.** *Il existe un ordre total sur toutes les opérations de façon à ce que chaque opération puisse être exécutée en un instant.*

Définition 14 (Disponibilité ([GL02]))**.** *Un système est dit disponible s'il garantit que toute requête reçue par un nœud non fautif résulte en une réponse.*

Définition 15 (Partition ([GL02]))**.** *Un système est partitionné s'il existe deux groupes de nœuds tels que tous les messages d'un groupe vers l'autre sont perdus.*

Définition 16 (Latence (t,p)-probabiliste)**.** *Un système atteints une* (t,p)*-latence si la proportion de lectures satisfaisant la t-latence est d'au moins* $(1-p)$*.*

Définition 17 (Partition (t,p)-probabiliste)**.** *Un système souffre d'une* (t,p)*-partition si la proportion de chemin d'un réplica à un autre avec une latence supérieur à t est supérieure à p.*

Théorèmes

Théorème 1 (FLP85)**.** *Aucun protocole de consensus ne peut être correct en présence d'une faute dans un système asynchrone.* [2]

Théorème 2 (Théorème du CAP)**.** *Tout système distribué ne peut satisfaire que deux des trois propriétés désirables suivantes :*

2. No consensus protocol is totally correct in spite of one fault.

- *Cohérence ;*
- *Disponibilité ;*
- *Tolérance aux partitions.*

Théorème 3 (**Théorème du CAC**)**.** *Aucun modèle de cohérence plus fort que la cohérence causale naturelle ne peut être implémenté dans un système toujours disponible et convergent*[3]*.*

Théorème 4 (**Théorème du PACELC**)**.** *Tout système distribué doit, s'il y a une partition (P) dans le système, établir un compromis entre disponibilité (A) et cohérence (C) ; sinon (E), quand le système s'exécute normalement, le système doit établir un compromis entre latence (L) et cohérence (C)*[4]*.*

Théorème 5 (**Théorème du PCAP**)**.** *Si* $t_c + t_\alpha < t_p$ *et* $p_\alpha + p_c < p_p$*, il est impossible d'implémenter un objet en lecture/écriture en présence d'une* (t_α, p)*-partition tout en achevant la* (t_c, p)*-cohérence et la* (t_α, p)*-latence.*

Problème

Problème 1 (Placement des réplicas)**.** *Soit un ensemble* D *de* n_d *données répliquées* RF *fois et soit un ensemble* S *de* n_s *nœuds de stockage à capacité finie.*

Comment placer les $\sum_{i=0}^{n_d} RF_i$ *réplicas des* n_d *données sur les* n_s *nœuds de stockage de façon à minimiser les latences médianes pour les accès utilisateurs ?*

Problème 2 (Placement des réplicas à modèles de cohérence variables)**.** *Soit un ensemble* D *de* n_d *données répliquées* RF *fois et avec des modèles de cohérence différent. Soit un ensemble* S *de* n_s *nœuds de stockage à capacité finie.*

Comment placer les $\sum_{i=0}^{n_d} RF_i$ *réplicas des* n_d *données sur les* n_s *nœuds de stockage de façon à minimiser les latences médianes pour les accès utilisateurs ?*

3. "No consistency semantics stronger than natural causal consistency can be enforced by a one-way convergent and always available distributed storage implementation."

4. If there is a partition (P) how does the system tradeoff between availability and consistency (A and C) ; else (E) when the system is running as normal in the absence of partitions, how does the system tradeoff between latency (L) and consistency (C) ?

Tables de références des systèmes de gestion de données distribués

1. Antidote : [ATB+16]
2. AzureDB : https ://docs.microsoft.com /en-us/azure/cosmos-db/introduction, Accessed : November 2017 to January 2018
3. BigTable : https ://cloud.google.com/bigtable/docs/overview, Accessed : November 2017 to January 2018 ; [CDG+08]
4. Cassandra : https ://docs.microsoft.com/en-us/azure/cosmos-db/consistency-levels, Accessed : November 2017 to January 2018
5. CoackroachDB : https ://www.cockroachlabs.com/, Accessed : November 2017 to January 2018 ; [Gri16]
6. Depot : [MAD+11]
7. Dynamo : https ://aws.amazon.com/ dynamodb/, Accessed : November 2017 to January 2018 ; [DHJ+07]
8. FaunaDB : https ://fauna.com/, Accessed : November 2017 to January 2018 ; [Fre18]
9. HyperTable/HBase : http ://www.hypertable.org/, https ://hbase.apache.org/, Accessed : November 2017 to January 2018 ; [KG06]
10. Megastore : [BBC+11]
11. MongoDB : https ://www.mongodb.com/, Accessed : November 2017 to January 2018

12. OpenSwift : https ://docs.openstack.org/ swift/latest/admin/objectstorage-intro.html, Accessed : November 2017 to January 2018 ; [BPS15]
13. Redis : https ://redis.io/documentation, Accessed : November 2017 to January 2018
14. Scalaris : http ://scalaris.zib.de/, Accessed : November 2017 to January 2018 ; [SSR08]
15. Scatter : [GBKA11]
16. Spanner : https ://cloud.google.com/spanner/, Accessed : November 2017 to January 2018 ; [CDE+13]
17. Voldemort Project : http ://www.project-voldemort.com/voldemort, Accessed : November 2017 to January 2018 [Fei11]
18. Zookeeper : https ://zookeeper.apache.org/, Accessed : November 2017 to January 2018 ; [HKJR10]

Bibliographie

[Aba12] Daniel Abadi. Consistency tradeoffs in modern distributed database system design : Cap is only part of the story. *Computer*, 45(2) :37–42, 2012.

[AG96] Sarita V Adve and Kourosh Gharachorloo. Shared memory consistency models : A tutorial. *computer*, 29(12) :66–76, 1996.

[Aga18] Sachin Agarwal. Public cloud inter-region network latency as heatmaps, 2018.

[AGS19] Anshul Ahuja, Geetesh Gupta, and Subhajit Sidhanta. Edge applications : Just right consistency. In *2019 38th Symposium on Reliable Distributed Systems (SRDS)*, pages 351–3512. IEEE, 2019.

[Amaa] Amazon. Aws s3 documentation. https://docs.aws.amazon.com/AmazonS3/latest/userguide/Welcome.html. Accessed : September 2021.

[Amab] Amazon AWS. Data lakes and analytics on aws. Accessed : May 2020.

[Apa] Apache Cassandra. Cassandra. http://cassandra.apache.org/doc/latest/. Accessed : November 2017 to January 2018.

[AS14] Arif Ahmed and Abadhan Saumya Sabyasachi. Cloud computing simulators : A detailed survey and future direction. In *2014 IEEE international advance computing conference (IACC)*, pages 866–872. IEEE, 2014.

[ATB+16] Deepthi Devaki Akkoorath, Alejandro Z Tomsic, Manuel Bravo, Zhongmiao Li, Tyler Crain, Annette Bieniusa, Nuno Preguiça, and Marc Shapiro. Cure : Strong semantics meets high availability and low latency. In *Distributed Computing Systems (ICDCS), 2016 IEEE 36th International Conference on*, pages 405–414. IEEE, 2016.

[BBC+11] Jason Baker, Chris Bond, James C Corbett, JJ Furman, Andrey Khorlin, James Larson, Jean-Michel Leon, Yawei Li, Alexander Lloyd, and Vadim Yushprakh. Megastore : Providing scalable, highly available storage for interactive services. In *CIDR*, volume 11, pages 223–234, 2011.

[BCK+10] Chris Bunch, Navraj Chohan, Chandra Krintz, Jovan Chohan, Jonathan Kupferman, Puneet Lakhina, Yiming Li, and Yoshihide Nomura. An evaluation of distributed datastores using the appscale cloud platform. In *Cloud Computing (CLOUD), 2010 IEEE 3rd International Conference on*, pages 305–312. IEEE, 2010.

[BDF+13] Peter Bailis, Aaron Davidson, Alan Fekete, Ali Ghodsi, Joseph M Hellerstein, and Ion Stoica. Highly available transactions : Virtues and limitations. *Proceedings of the VLDB Endowment*, 7(3) :181–192, 2013.

[BG13] Peter Bailis and Ali Ghodsi. Eventual consistency today : Limitations, extensions, and beyond. *Queue*, 11(3) :20, 2013.

[BK14] Peter Bailis and Kyle Kingsbury. The network is reliable. *Queue*, 12(7) :20, 2014.

[BMS02] Marin Bertier, Olivier Marin, and Pierre Sens. Implementation and performance evaluation of an adaptable failure detector. In *Proceedings International Conference on Dependable Systems and Networks*, pages 354–363. IEEE, 2002.

[BPS15] Prosunjit Biswas, Farhan Patwa, and Ravi Sandhu. Content level access control for openstack swift storage. In *Proceedings of the 5th ACM Conference on Data and Application Security and Privacy*, CODASPY '15, pages 123–126, New York, NY, USA, 2015. ACM.

[BR01] Ivan D Baev and Rajmohan Rajaraman. Approximation algorithms for data placement in arbitrary networks. In *Proceedings of the twelfth annual ACM-SIAM symposium on Discrete algorithms*, pages 661–670, 2001.

[Bre00] Eric A Brewer. Towards robust distributed systems. In *PODC*, volume 7, 2000.

[Bre12] Eric Brewer. Cap twelve years later : How the" rules" have changed. *Computer*, 45(2) :23–29, 2012.

[BSW04] Jerzy Brzezinski, Cezary Sobaniec, and Dariusz Wawrzyniak. From session causality to causal consistency. In *PDP*, pages 152–158, 2004.

[BVF+12] Peter Bailis, Shivaram Venkataraman, Michael J Franklin, Joseph M Hellerstein, and Ion Stoica. Probabilistically bounded staleness for practical partial quorums. *Proceedings of the VLDB Endowment*, 5(8) :776–787, 2012.

[Cat11] Rick Cattell. Scalable sql and nosql data stores. *Acm Sigmod Record*, 39(4) :12–27, 2011.

[CDE+13] James C Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, Jeffrey John Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, et al. Spanner : Google's globally distributed database. *ACM Transactions on Computer Systems (TOCS)*, 31(3) :1–22, 2013.

[CDG+08] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C Hsieh, Deborah A Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E Gruber. Bigtable : A distributed storage system for structured data. *ACM Transactions on Computer Systems (TOCS)*, 26(2) :4, 2008.

[CFQS11] Arnaud Casteigts, Paola Flocchini, Walter Quattrociocchi, and Nicola Santoro. Time-varying graphs and dynamic networks. In *International Conference on Ad-Hoc Networks and Wireless*, pages 346–359. Springer, 2011.

[CGL+14] Henri Casanova, Arnaud Giersch, Arnaud Legrand, Martin Quinson, and Frédéric Suter. Versatile, scalable, and accurate simulation of distributed applications and platforms. *Journal of Parallel and Distributed Computing*, 74(10) :2899–2917, June 2014.

[CH76] K Mani Chandy and JE Hewes. File allocation in distributed systems. In *Proceedings of the 1976 ACM SIGMETRICS conference on Computer performance modeling measurement and evaluation*, pages 10–13, 1976.

[CL10] Arnaud Casteigts and Rémi Laplace. Jbotsim, a tool for fast prototyping of distributed algorithms in dynamic networks. *arXiv preprint arXiv :1001.1435*, 2010.

[CMM+13] Francisco Cruz, Francisco Maia, Miguel Matos, Rui Oliveira, Joao Paulo, José Pereira, and Ricardo Vilaça. Met : workload aware elasticity for nosql. In *Proceedings of the 8th ACM European Conference on Computer Systems*, pages 183–196, 2013.

[CRDRB09] Rodrigo N Calheiros, Rajiv Ranjan, César AF De Rose, and Rajkumar Buyya. Cloudsim : A novel framework for modeling and simulation of cloud computing infrastructures and services. *arXiv preprint arXiv :0903.2525*, 2009.

[CRS+08] Brian F Cooper, Raghu Ramakrishnan, Utkarsh Srivastava, Adam Silberstein, Philip Bohannon, Hans-Arno Jacobsen, Nick Puz, Daniel Weaver, and Ramana Yerneni. Pnuts : Yahoo !'s hosted data serving platform. *Proceedings of the VLDB Endowment*, 1(2) :1277–1288, 2008.

[CT96] Tushar Deepak Chandra and Sam Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM (JACM)*, 43(2) :225–267, 1996.

[Dan] Danny Sullivan. Google still world's most popular search engine by far, but share of unique searchers dips slightly. Accessed : May 2020.

[DDH10] Theodoros K Dikaliotis, Alexandros G Dimakis, and Tracey Ho. Security in distributed storage systems by communicating a logarithmic number of bits. In *2010 IEEE International Symposium on Information Theory*, pages 1948–1952. IEEE, 2010.

[DF82] Lawrence W Dowdy and Derrell V Foster. Comparative models of the file assignment problem. *ACM Computing Surveys (CSUR)*, 14(2) :287–313, 1982.

[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. In *ACM SIGOPS operating systems review*, volume 41, pages 205–220. ACM, 2007.

[DNS13] Balla Wade Diack, Samba Ndiaye, and Yahya Slimani. Cap theorem between claims and misunderstandings : What is to be sacrificed.

International Journal of Advanced Science and Technology, 56 :1–12, 2013.

[DS82] Dvora Dolev and HR Strong. Distributed commit with bounded waiting. In *Proceedings of the*, 1982.

[Fac] Facebook Research. Facebook's top open data problems. Accessed : May 2020.

[FB99] Armando Fox and Eric A Brewer. Harvest, yield, and scalable tolerant systems. In *Hot Topics in Operating Systems, 1999. Proceedings of the Seventh Workshop on*, pages 174–178. IEEE, 1999.

[Fei11] Alex Feinberg. Project voldemort : Reliable distributed storage. In *Proceedings of the 10th IEEE International Conference on Data Engineering*, 2011.

[Fis83] Michael J Fischer. The consensus problem in unreliable distributed systems (a brief survey). In *International conference on fundamentals of computation theory*, pages 127–140. Springer, 1983.

[FLP85] Michael J Fischer, Nancy A Lynch, and Michael S Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM (JACM)*, 32(2) :374–382, 1985.

[Fre18] Matt Freels. Faunadb : An architectural overview, 2018.

[FVC03] Roy Friedman, Roman Vitenberg, and Gregory Chockler. On the composability of consistency conditions. *Information Processing Letters*, 86(4) :169–176, 2003.

[GBKA11] Lisa Glendenning, Ivan Beschastnikh, Arvind Krishnamurthy, and Thomas Anderson. Scalable consistency in scatter. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, pages 15–28. ACM, 2011.

[GCG01] Indranil Gupta, Tushar D Chandra, and Germán S Goldszmidt. On scalable and efficient distributed failure detectors. In *Proceedings of the twentieth annual ACM symposium on Principles of distributed computing*, pages 170–179, 2001.

[GGL03] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The google file system. In *Proceedings of the nineteenth ACM symposium on Operating systems principles*, pages 29–43, 2003.

[GGSZ09] Robert L Grossman, Yunhong Gu, Michael Sabala, and Wanzhi Zhang. Compute and storage clouds using wide area high performance networks. *Future Generation Computer Systems*, 25(2) :179–183, 2009.

[GHM+00] Rachid Guerraoui, Michel Hurfinn, Achour Mostéfaoui, Riucarlos Oliveira, Michel Raynal, and André Schiper. Consensus in asynchronous distributed systems : A concise guided tour. In *Advances in Distributed Systems*, pages 33–47. Springer, 2000.

[GKK+09] Leana Golubchik, Sanjeev Khanna, Samir Khuller, Ramakrishna Thurimella, and An Zhu. Approximation algorithms for data placement on parallel disks. *ACM Transactions on Algorithms (TALG)*, 5(4) :1–26, 2009.

[GKM+19] Rachid Guerraoui, Petr Kuznetsov, Matteo Monti, Matej Pavlovič, and Dragos-Adrian Seredinschi. The consensus number of a cryptocurrency. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing*, pages 307–316, 2019.

[GL02] Seth Gilbert and Nancy Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *Acm Sigact News*, 33(2) :51–59, 2002.

[GM02] Sudipto Guha and Kamesh Munagala. Improved algorithms for the data placement problem. In *SODA*, volume 2, pages 106–107. Citeseer, 2002.

[Gre] Greg Linden. Make data useful. Accessed : May 2020.

[Gri16] Max Wouter Grim. Consistency analysis of cockroachdb under fault injection. *Science*, 2016.

[GYF+16] Alexey Gotsman, Hongseok Yang, Carla Ferreira, Mahsa Najafzadeh, and Marc Shapiro. 'cause i'm strong enough : Reasoning about consistency choices in distributed systems. In *ACM SIGPLAN Notices*, volume 51, pages 371–384. ACM, 2016.

[HDYK04] Naohiro Hayashibara, Xavier Defago, Rami Yared, and Takuya Katayama. The/spl phi/accrual failure detector. In *Proceedings of the 23rd IEEE International Symposium on Reliable Distributed Systems, 2004.*, pages 66–78. IEEE, 2004.

[HF89] Joseph Y Halpern and Ronald Fagin. Modelling knowledge and action in distributed systems. *Distributed computing*, 3(4) :159–177, 1989.

[HHLD11] Jing Han, E Haihong, Guan Le, and Jian Du. Survey on nosql database. In *Pervasive computing and applications (ICPCA), 2011 6th international conference on*, pages 363–366. IEEE, 2011.

[HKJR10] Patrick Hunt, Mahadev Konar, Flavio Paiva Junqueira, and Benjamin Reed. Zookeeper : Wait-free coordination for internet-scale systems. In *USENIX annual technical conference*, volume 8. Boston, MA, USA, 2010.

[HW90] Maurice P. Herlihy and Jeannette M. Wing. Linearizability : A correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3) :463–492, July 1990.

[JBP+13] Yu Jia, Ivan Brondino, Ricardo Jiménez Peris, Marta Patiño Martínez, and Dianfu Ma. A multi-resource load balancing algorithm for cloud cache systems. In *Proceedings of the 28th Annual ACM Symposium on Applied Computing*, pages 463–470, 2013.

[Joh] Johannes Weiner. thrash detection-based file cache sizing. Accessed : June 2020.

[Jon] Jonathan Corbet. Better active/inactive list balancing. Accessed : June 2020.

[JWF+11] Jiehui Ju, Jiyi Wu, Jianqing Fu, Zhijie Lin, and Jianlin Zhang. A survey on cloud storage. *J. Comput.*, 6(8) :1764–1771, 2011.

[KG06] Ankur Khetrapal and Vinay Ganesh. Hbase and hypertable for large scale distributed storage systems. *Dept. of Computer Science, Purdue University*, pages 22–28, 2006.

[KJBH+88] Leonard Kawell Jr, Steven Beckhardt, Timothy Halvorsen, Raymond Ozzie, and Irene Greif. Replicated document management in a group communication system. In *Proceedings of the 1988 ACM conference on Computer-supported cooperative work*, page 395. ACM, 1988.

[Kle15] Martin Kleppmann. A critique of the CAP theorem. *CoRR*, abs/1509.05393, 2015.

[KNR+11] Jay Kreps, Neha Narkhede, Jun Rao, et al. Kafka : A distributed messaging system for log processing. In *Proceedings of the NetDB*, volume 11, pages 1–7, 2011.

[KRS00] Paddy Krishnan, Danny Raz, and Yuval Shavitt. The cache location problem. *IEEE/ACM transactions on networking*, 8(5) :568–582, 2000.

[Lam78] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7) :559–565, 1978.

[LC01] Bo Leuf and Ward Cunningham. *The Wiki way : quick collaboration on the Web*. Addison-Wesley Longman Publishing Co., Inc., 2001.

[LFKA11] Wyatt Lloyd, Michael J Freedman, Michael Kaminsky, and David G Andersen. Don't settle for eventual : scalable causal consistency for wide-area storage with cops. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, pages 401–416. ACM, 2011.

[LM10] Avinash Lakshman and Prashant Malik. Cassandra : a decentralized structured storage system. *ACM SIGOPS Operating Systems Review*, 44(2) :35–40, 2010.

[LPC+12] Cheng Li, Daniel Porto, Allen Clement, Johannes Gehrke, Nuno M Preguiça, and Rodrigo Rodrigues. Making geo-replicated systems fast as possible, consistent when necessary. In *OSDI*, volume 12, pages 265–278, 2012.

[LWY93] Avraham Leff, Joel L Wolf, and Philip S. Yu. Replication algorithms in a remote caching architecture. *IEEE Transactions on Parallel and Distributed Systems*, 4(11) :1185–1204, 1993.

[Lyn96] Nancy A Lynch. *Distributed algorithms*. Elsevier, 1996. Distributed Consensus with Link/Process Failures - Chapitres 5,6.

[MAA05] Ahmed Metwally, Divyakant Agrawal, and Amr El Abbadi. Efficient computation of frequent and top-k elements in data streams. In *International conference on database theory*, pages 398–412. Springer, 2005.

[MAD+11] Prince Mahajan, Lorenzo Alvisi, Mike Dahlin, et al. Consistency, availability, and convergence. *University of Texas at Austin Tech Report*, 11, 2011.

[Mic] Microsoft AzureDB. Azure cosmosdb. https://docs.microsoft.com/en-us/azure/cosmos-db/consistency-levels. Accessed : November 2017 to January 2018.

[MJ09] Alberto Montresor and Márk Jelasity. Peersim : A scalable p2p simulator. In *2009 IEEE Ninth International Conference on Peer-to-Peer Computing*, pages 99–100. IEEE, 2009.

[MJAS19] Etienne Mauffret, Denis Jeanneau, Luciana Arantes, and Pierre Sens. The weakest failure detector to solve the mutual exclusion problem in an unknown dynamic environment. In *Proceedings of the 20th International Conference on Distributed Computing and Networking*, pages 11–20, 2019.

[MN16] Bahareh Alami Milani and Nima Jafari Navimipour. A comprehensive review of the data replication techniques in the cloud environments : Major trends and future directions. *Journal of Network and Computer Applications*, 64 :229–238, 2016.

[MSL+11] Prince Mahajan, Srinath Setty, Sangmin Lee, Allen Clement, Lorenzo Alvisi, Mike Dahlin, and Michael Walfish. Depot : Cloud storage with minimal trust. *ACM Trans. Comput. Syst.*, 29(4) :12 :1–12 :38, December 2011.

[MVM19] Etienne Mauffret, Flavien Vernier, and Sébastien Monnet. Candor : Consistency aware dynamic data replication. In *2019 IEEE 18th International Symposium on Network Computing and Applications (NCA)*, pages 1–5. IEEE, 2019.

[MVM20] Etienne Mauffret, Flavien Vernier, and Sébastien Monnet. How to use the past to face the future ? Research report, LISTIC, 2020.

[NBL+06] Stephen Naicken, Anirban Basu, Barnaby Livingston, Sethalat Rodhetbhai, and Ian Wakeman. Towards yet another peer-to-peer simulator. In *Proceedings of the 4th International Working Conference on Performance Modelling and Evaluation of Heterogeneous Networks (HET-NETs' 06)*. Citeseer, 2006.

[NVPC+12] Alberto Núñez, Jose L Vázquez-Poletti, Agustin C Caminero, Gabriel G Castañé, Jesus Carretero, and Ignacio M Llorente. icancloud : A flexible and scalable cloud infrastructure simulator. *Journal of Grid Computing*, 10(1) :185–209, 2012.

[Pop19] Diana Andreea Popescu. *Latency-driven performance in data centres*. PhD thesis, University of Cambridge, 2019.

[PRRR14] Joao Paiva, Pedro Ruivo, Paolo Romano, and Luís Rodrigues. Autoplacer : Scalable self-tuning data placement in distributed key-value

stores. *ACM Transactions on Autonomous and Adaptive Systems (TAAS)*, 9(4) :1–30, 2014.

[QKD13] Abdul Quamar, K Ashwin Kumar, and Amol Deshpande. Sword : scalable workload-aware data placement for transactional workloads. In *Proceedings of the 16th International Conference on Extending Database Technology*, pages 430–441, 2013.

[Rai] RainbowFS ANR project. Rainbowfs anr project. https://rainbowfs.lip6.fr//. Accessed : March 2022.

[Red] Redis. Redis. https://redis.io/documentation. Accessed : November 2017 to January 2018.

[RTN+17] Muntasir Raihan Rahman, Lewis Tseng, Son Nguyen, Indranil Gupta, and Nitin Vaidya. Characterizing and adapting the consistency-latency tradeoff in distributed key-value stores. *ACM Transactions on Autonomous and Adaptive Systems (TAAS)*, 11(4) :20, 2017.

[Rup00] Eric Ruppert. Determining consensus numbers. *SIAM Journal on Computing*, 30(4) :1156–1168, 2000.

[Sah20] Rahul Sahay. Cosmos db. In *Microsoft Azure Architect Technologies Study Companion*, pages 699–716. Springer, 2020.

[SBP+18] Marc Shapiro, Annette Bieniusa, Nuno Preguiça, Valter Balegas, and Christopher Meiklejohn. Just-right consistency : reconciling availability and safety. *arXiv preprint arXiv :1801.06340*, 2018.

[SKRC10] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The hadoop distributed file system. In *2010 IEEE 26th symposium on mass storage systems and technologies (MSST)*, pages 1–10. Ieee, 2010.

[SM03] Reza Olfati Saber and Richard M Murray. Consensus protocols for networks of dynamic agents. *Proceedings of the 2003 American Control Conference, 2003*, 2003.

[SN04] Robert C Steinke and Gary J Nutt. A unified theory of shared memory consistency. *Journal of the ACM (JACM)*, 51(5) :800–849, 2004.

[SPBZ11] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. Conflict-free replicated data types. In *Symposium on Self-Stabilizing Systems*, pages 386–400. Springer, 2011.

[SSAP16] Marc Shapiro, Masoud Saeida Ardekani, and Gustavo Petri. Consistency in 3D. Research Report RR-8932, Institut National de la Recherche en Informatique et Automatique (Inria), July 2016.

[SSR08] Thorsten Schütt, Florian Schintke, and Alexander Reinefeld. Scalaris : reliable transactional p2p key/value store. In *Proceedings of the 7th ACM SIGPLAN workshop on ERLANG*, pages 41–48. ACM, 2008.

[ST01] Hadas Shachnai and Tami Tamir. On two class-constrained versions of the multiple knapsack problem. *Algorithmica*, 29(3) :442–467, 2001.

[TDP+94] Douglas B Terry, Alan J Demers, Karin Petersen, Mike J Spreitzer, Marvin M Theimer, and Brent B Welch. Session guarantees for weakly consistent replicated data. In *Parallel and Distributed Information Systems, 1994., Proceedings of the Third International Conference on*, pages 140–149. IEEE, 1994.

[TSJ+10] Ashish Thusoo, Joydeep Sen Sarma, Namit Jain, Zheng Shao, Prasad Chakka, Ning Zhang, Suresh Antony, Hao Liu, and Raghotham Murthy. Hive-a petabyte scale data warehouse using hadoop. In *2010 IEEE 26th international conference on data engineering (ICDE 2010)*, pages 996–1005. IEEE, 2010.

[TVS07] Andrew S Tanenbaum and Maarten Van Steen. *Distributed systems : principles and paradigms*. Prentice-Hall, 2007. Consistency and Replication - Chapitre 7.

[Twi] Twitter Blog. Twitter data. Accessed : May 2020.

[TXC+15] Wenhong Tian, Minxian Xu, Aiguo Chen, Guozhong Li, Xinyang Wang, and Yu Chen. Open-source simulators for cloud computing : Comparative study and challenging issues. *Simulation Modelling Practice and Theory*, 58 :239–254, 2015.

[Var01] András Varga. The omnet++ discrete event simulation system. *Proc. ESM'2001*, 9, 01 2001.

[Vog09] Werner Vogels. Eventually consistent. *Communications of the ACM*, 52(1) :40–44, 2009.

[Vol] Voldemort Project. Voldemort project. http://www.project-voldemort.com/voldemort/. Accessed : November 2017 to January 2018.

[VV16] Paolo Viotti and Marko Vukolić. Consistency in non-transactional distributed storage systems. *ACM Comput. Surv.*, 49(1) :19 :1–19 :34, June 2016.

[YHJ13] Gae-Won You, Seung-Won Hwang, and Navendu Jain. Ursa : Scalable load and power management in cloud storage systems. *ACM Transactions on Storage (TOS)*, 9(1) :1–29, 2013.

[YYLC10] Dong Yuan, Yun Yang, Xiao Liu, and Jinjun Chen. A data placement strategy in scientific cloud workflows. *Future Generation Computer Systems*, 26(8) :1200–1214, 2010.

[ZG11] Sharrukh Zaman and Daniel Grosu. A distributed algorithm for the replica placement problem. *IEEE Transactions on Parallel and Distributed Systems*, 22(9) :1455–1468, 2011.

Etienne Mauffret
Thèse dirigée par Flavien Vernier et Sebastien Monnet

Placement des réplicas dans un système de gestion de données distribué à large échelle à protocole de cohérence adaptable

Résumé

A l'ère des applications et services à échelle mondiale, il est important d'assurer des services pouvant faire face à un grand nombre d'utilisateurs simultanés. De plus, les applications modernes gèrent d'énormes quantités de données. Facebook, par exemple, manipule des pétabytes de données par jour. Afin d'être capables de fournir un tel service, les systèmes de gestions de données distribués répliquent les données à travers le système. Bien que cette approche offre de bonnes garanties de résilience, il existe un équilibre à trouver entre la vitesse d'accès et la fiabilité des données. Certains services visant à devenir de plus en plus complets, il est de moins en moins rare de devoir gérer les différentes données avec différentes garanties. Il faut alors être capable d'adapter certaines stratégies selon des critères pertinents. Nous nous intéressons ici à une méthode de placement des réplicas selon le comportement des utilisateurs et des garanties d'accès et de cohérence.

Abstract

In the age of global applications and services, it is important to provide services that can handle a large number of concurrent users. In addition, modern applications handle huge amounts of data. Facebook, for example, handles petabytes of data daily. In order to be able to provide such a service, distributed data management systems replicate the data throughout the system. While this approach offers good resiliency, there is a balance to be struck between speed of access and reliability of data. As some services become more and more comprehensive, it become less and less rare to have to manage different data with different guarantees. It is then necessary to be able to adapt certain strategies according to relevant criteria. We propose here in a method for placing replicas according to user behavior and access and consistency guarantees.