Contributions à la réplication de données dans les systèmes distribués à grande échelle

Sébastien Monnet

THÈSE D'HABILITATION À DIRIGER LES RECHERCHES

présentée à

L'UNIVERSITÉ PIERRE ET MARIE CURIE

Spécialité

Informatique

soutenue par

Sébastien MONNET

Le 3 Novembre 2015

CONTRIBUTIONS À LA RÉPLICATION DE DONNÉES DANS LES SYSTÈMES DISTRIBUÉS À GRANDE ÉCHELLE

JURY

Rapporteurs		
Roberto	BALDONI	Professeur, Université de Rome, La Sapienza
Pascal	FELBER	Professeur, Université de Neuchâtel
Anne-Marie	KERMARREC	Directrice de recherche, Inria Rennes
Examinateurs		
Frédéric	DESPREZ	Directeur de Recherche, Inria Rhône-Alpes
Esther	PACITTI	Professeur, Université de Montpellier 2
Pierre	SENS	Professeur, Université Pierre et Marie

À Caroline, Mathilde, Paul, Gauthier et Eléonore

REMERCIEMENTS

Je remercie chaleureusement Roberto Baldoni, Pascal Felber et Anne-Marie Kermarrec d'avoir accepté de rapporter mon travail, ainsi que Frédéric Desprez et Esther Pacitti pour avoir accepté d'être examinateurs. Leur présence dans mon jury est un grand honneur.

Mes remerciements vont également à Pierre Sens et Marc Shapiro pour toutes les discussions que nous avons eues et tous les conseils qu'ils m'ont prodigués dans le cadre de mes premiers encadrements de thèses. Lors de la rédaction de ce document, leurs retours m'ont été d'une aide précieuse.

Un merci particulier aux doctorants que j'ai encadrés, Sergey Legtchenko, Guthemberg Da Silva Silvestre, Pierpaolo Cincilla, Karine Pires et Maxime Véron ; et à ceux que j'encadre actuellement, Maxime Lorrillere, Lyes Hamidouche, Damien Carver et Guillaume Turchini. Je n'oublie pas Véronique Simon, ingénieur, avec qui j'ai également eu un grand plaisir à travailler.

Je souhaite remercier tous les membres de l'équipe Regal, en particulier mes jeunes amis et collègues Olivier Marin, Gaël Thomas et Julien Sopena.

J'ai également une pensée pour Gabriel Antoniu, Luc Bougé, et Pierre Sens pour tous leurs précieux conseils.

Enfin, un immense merci à ma famille. Merci à Caroline, ma femme, merci à Claude, mon père, pour leur soutien sans faille...

TABLE DES MATIÈRES

1. INTRODUCTION

La gestion des données est une des principales raisons d'être de l'informatique. Avec la démocratisation de l'Internet, la popularisation des périphériques tels les *smartphones*, les appareils photos et les caméscopes numériques, le nombre de sources de données numériques est très grand. De plus, de nombreuses applications, dans le domaine scientifique ou dans le monde de la finance notamment, génèrent de grandes masses de données. La croissance exponentielle des données générées avec l'avènement de l'ère du *Big Data* crée de nouveaux défis, à la fois pour leur stockage et leur traitement. Les systèmes de gestion de données se doivent d'offrir un stockage fiable, performant et garantissant un certain niveau de cohérence. Pour cela, la réplication est une technique clé.

Fiabilité. Afin d'éviter de perdre définitivement une donnée en cas de crash disque (pérennité) ou de permettre l'accès même lorsque certaines machines sont déconnectées (disponibilité), il est nécessaire d'ajouter de la redondance. Il existe pour cela deux grandes familles de solutions : *(i)* la *réplication* et *(ii)* les codes correcteurs. Dans mon travail, je me suis concentré sur la réplication. Le principe de base est simple : disposer de plusieurs copies d'une même donnée de manière à ce que celle-ci soit toujours disponible. Cependant, si l'on prend en compte la disposition des copies (affinités de données/fautes corrélées), l'efficacité des mécanismes de réparation en cas de faute, la répartition de la charge sur les nœuds et sur le réseau, la conception de mécanismes de réplication s'avère complexe.

Performance. La réplication des données est également utile pour offrir de bonnes performances d'accès. En effet, elle permet de rapprocher des copies des utilisateurs pour diminuer la latence, ou de créer de nombreuses copies pour des données extrêmement populaires afin de répartir la charge sur différents serveurs. En revanche, la maintenance des copies d'une donnée peut s'avérer coûteuse et dégrader les performances, surtout lorsque la donnée est fréquemment mise à jour. Il est donc important d'adapter la réplication des données en fonction des schémas d'accès, et de localiser efficacement les copies des données.

Cohérence. Maintenir plusieurs copies d'une même donnée peut poser des problèmes de cohérence lors des mises à jour. Il existe de nombreux modèles et protocoles de cohérence offrant différents niveaux de garanties/performances. Cependant, il n'existe pas de solution universelle. Le théorème CAP stipule qu'il est même impossible d'offrir à la fois cohérence, disponibilité et support des partitions [1, 2]. Il est donc nécessaire de faire des compromis sur l'une des trois propriétés en fonction des contraintes des applications. Offrir un mécanisme de maintien de la cohérence de données répliquées à grande échelle est donc toujours un problème ouvert pour de nombreuses applications.

1.1. CONTRIBUTIONS

Au cours de ces huit dernières années, ma recherche a été principalement centrée autour de la gestion des données dans les grands systèmes distribués. Elle s'est faite en collaboration avec huit doctorants, dont quatre ont soutenu (Sergey Legtchenko [3], Guthemberg Da Silva Silvestre [4], Pierpaolo Cincilla [5] et Karine Pires [6]) et quatre sont encore en cours de thèse (Maxime Véron, Maxime Lorrillere, Lyes Hamidouche et Damien Carver), ainsi qu'avec plusieurs chercheurs (notamment Pierre Sens, Marc Shapiro, Olivier Marin, Julien Sopena, Gaël Thomas et Gilles Muller). Nous nous sommes intéressés au stockage fiable des données, aux performances des accès, mais également aux problèmes de cohérence dans les bases de données répliquées. Enfin, nous nous sommes intéressés à la gestion des données dans un cadre applicatif particulièrement exigeant : les jeux massivement multijoueurs en ligne (MMOG pour *Massively Multiplayer Online Games*).

Stockage de données fiable. Nous nous sommes tout d'abord intéressés à la tolérance aux défaillances, au stockage fiable des données. Les tables de hachage distribuées (DHT pour *Distributed Hash Tables*) offrent une interface d'utilisation très pratique de type *put*/*get*, passent bien à l'échelle et permettent une bonne distribution des données. Cependant, il a été montré qu'elles supportaient mal les connexions/déconnexions fréquentes (*churn* en anglais) [7]. En effet, le *churn* peut induire des pertes de données y compris en présence de mécanismes de réparation. Dans un premier temps, nous avons proposé *RelaxDHT* [LMSM09, LMSM12] qui permet de relâcher les contraintes de placement des copies dans les DHTs grâce à un mécanisme de gestion de métadonnées.

Dans un cadre plus large que celui des DHTs, nous nous sommes ensuite concentrés sur un problème moins étudié, la pérennité des données à long terme. Les résultats obtenus lors de la conception de RelaxDHT nous ont incités à étudier plus en détail l'impact du placement des données sur les mécanismes de réparation et sur les pertes. Nous avons donc proposé *SPLAD* pour *Scattering and placing data replicas to enhance long-term durability* [SMF+15]. SPLAD permet d'observer l'impact de l'éparpillement et du placement des copies des données sur le taux de perte à long terme et sur la répartition des données sur les nœuds.

Enfin, il existe un autre paramètre qui impacte de manière importante les temps de réparation : les temps de détection. C'est même un facteur déterminant pour la fiabilité des systèmes [8]. Nous avons donc proposé un détecteur de défaillances basé sur un système de réputation pour les réseaux dynamiques *RepFD* [VMMS15b], ce dernier n'est pas détaillé dans ce document.

Amélioration des performances d'accès. Nous avons identifié deux grands facteurs limitant les performances d'accès : *(i)* les temps de transferts qui peuvent être améliorés par la réplication et *(ii)* la localisation des données qui dépend des performances du routage sous-jacent.

Dans le cadre d'une collaboration avec Orange Labs, nous avons étudié l'opportunité d'utiliser les périphériques de bordure de réseaux tels que les *box* d'accès Internet pour le stockage des données (*Caju* [SMKS12b]). Les données n'étant pas toutes également populaires (certaines ne seront jamais accédées alors que d'autres le seront des milliers de fois) nous avons montré l'importance de prendre en compte la popularité pour déterminer le facteur de réplication. Nous avons proposé *AREN* [SMKS12a, SMBS13], un mécanisme de réplication de données qui adapte le nombre de copies d'une donnée en fonction de sa popularité. Dans la continuité de ce travail, nous avons également conçu *POPS* [PMS14b], un service de diffusion de flux vidéo en direct prenant en compte la popularité des flux.

Dans un autre contexte, les caches étant une bonne solution pour conserver des copies proches du lieu d'utilisation, nous avons conçu *Puma* [LSMS15b]. Puma est un mécanisme noyau permettant de mutualiser la mémoire des machines virtuelles dans un *cloud* pour leur permettre d'avoir de plus grands caches. Puma n'est pas détaillé dans ce document.

Dans les systèmes à grande échelle, la localisation d'une donnée peut également impacter les performances. Si l'on utilise un réseau logique (*overlay*) de type DHT, il est aisé de créer des liens longs permettant un routage efficace [9,10], la fonction de hachage offrant une distribution uniforme des identifiants. Si l'on souhaite placer les données en fonction de leurs affinités ou de la localité des accès, la distribution des identifiants est en général très hétérogène. Pour offrir un routage efficace dans les systèmes distribués à grande échelle lorsque la distribution n'est pas uniforme, nous avons proposé *DONUT* [LMS11]. DONUT permet de créer une carte locale donnant une approximation de la distribution des densités afin de permettre aux nœuds de mettre en place des liens longs offrant un routage performant.

Gestion de la cohérence. Lors de ma thèse [PhD06] je m'étais intéressé à la gestion de la cohérence de données répliquées au sein d'un système de gestion de données pour les grilles de calcul. Nous nous intéressons désormais à la gestion de la cohérence dans le cadre de bases de données répliquées. Il ne s'agit plus d'assurer la cohérence entre les copies d'une même donnée, mais également de prendre en compte les transactions, les dépendances inter-données. Les transactions parallèles sur les bases de données répliquées induisent des surcoûts en contrôle de concurrence et en abandons de transactions (*aborts*). Nous avons proposé *Gargamel* [CMS12] qui utilise un classificateur afin de séquentialiser les transactions possiblement conflictuelles et de paralléliser celles qui sont indépendantes.

Gestion de données pour les MMOGs. Du point de vue du système de gestion de données, les MMOGs apportent des défis très intéressants : ils sont par nature à très grande échelle, les accès aux données (en lecture, mais également en écriture) sont extrêmement fréquents, et les besoins de performance sont critiques. De plus, ces applications sont très dynamiques, en nombre de joueurs, mais également au niveau de la répartition de ces joueurs et donc, de la répartition des accès aux données. Nous avons étudié les mouvements des joueurs et proposé *Blue Banana* [LMT10], permettant de mieux supporter leur mobilité dans le cadre des architectures pair-à-pair.

De manière indépendante de la gestion des données, nous avons également, dans le domaine des jeux, eu des contributions portant sur la détection décentralisée de la triche [VMMG12, VMMG14], et sur la mise en relation de joueurs (*matchmaking*) [VMM14a]. Ces travaux ne sont pas décrits ici.

Ma recherche actuelle se focalise principalement sur la gestion des données pour ce type d'application. De nombreuses architectures, plus ou moins distribuées sont possibles, du client/serveur au pur pair-à-pair en passant par des solutions intermédiaires, hybrides. Dans tous les cas, la distribution dynamique des données, le maintien de la cohérence et des garanties de performances restent des défis importants [TMM15].

Organisation du document. Le reste de ce document est organisé comme suit. Le chapitre 2 présente RelaxDHT et SPLAD, nos travaux sur la pérennité des données. Le chapitre 3 décrit Caju, AREN, POPS et DONUT, nos contributions liées à la performance des accès aux données. Enfin, Gargamel est présenté dans le chapitre 4. Le chapitre 5 conclut ce document, et décrit nos perspectives.

2. RÉPLICATION POUR LA DISPONIBILITÉ/PÉRENNITÉ

Les systèmes de stockage distribués doivent assurer la disponibilité et la pérennité des données, malgré la présence de fautes. À cette fin, de nombreux systèmes reposent sur la réplication. Dans ce chapitre, nous donnons un aperçu des grandes familles de mécanismes de réplication puis nous présentons *RelaxDHT*, notre solution, conçue au début de la thèse de Sergey Legtchenko [3], qui permet de relâcher les contraintes de placement des tables de hachage distribuées afin d'améliorer la résistance au *churn*. Nous analysons ensuite l'impact de la répartition des données sur l'efficacité des mécanismes de réparation, et par conséquent, sur la fiabilité du système de stockage.

Dans ce chapitre, nous n'abordons pas les problèmes liés à la gestion de la cohérence des données. Parmi les travaux récents de ce domaine, on peut notamment citer Scatter [11]. Nous nous concentrons ici sur la mise en œuvre d'un système de stockage fiable, les mécanismes de maintien de la cohérence pouvant se greffer au-dessus.

2.1. STRATÉGIES DE RÉPLICATION

Les systèmes de stockage distribués comme *Hadoop Distributed File System* (HDFS) [12], *Google File System* (GFS) [13], et Windows Azure [14] assurent la disponibilité et la pérennité des données grâce à la réplication. Chaque donnée est copiée sur différents nœuds et des mécanismes de maintenance tentent de maintenir un nombre minimum de copies disponibles dans le système. Ainsi, lorsque des copies sont perdues à la suite d'une panne, le système les régénère à partir des copies restantes.

Une première famille de travaux consiste à choisir les nœuds de stockage en fonction de prédictions de leur présence en ligne [15, 16] ou de l'estimation de leur fiabilité. Van Renesse [17] propose un algorithme de placement de copies de données pour les DHTs en considérant la fiabilité des nœuds et en plaçant des copies jusqu'à ce que le niveau de fiabilité désiré soit atteint. Dans FARSITE [18], il est proposé une stratégie de placement dynamique pour améliorer la disponibilité des fichiers. Les fichiers changent de serveurs en fonction de leur fiabilité courante. Ces approches permettent de réduire le nombre de copies nécessaires pour atteindre un certain niveau de fiabilité. Cependant, elles peuvent conduire à un déséquilibre important de la répartition de la charge de stockage sur les nœuds. En effet, les nœuds fiables risquent d'avoir à stocker de très nombreuses copies et de se retrouver surchargés. Dans nos travaux, nous n'avons pas pris en considération d'estimation de la fiabilité des nœuds de stockage, mais la souplesse apportée par notre approche RelaxDHT présentée dans la section 2.2 offre la possibilité de choisir plus librement les nœuds de stockage. Les critères guidant le choix final du nœud peuvent alors être variés, et notamment prendre en compte la fiabilité des nœuds.

Dans le contexte des systèmes de stockage de données pair-à-pair, on distingue deux principales familles de placement des copies des données : contiguës ou éparpillées.

Les stratégies de placement contiguës consistent à stocker les copies d'une même donnée sur des nœuds adjacents dans un réseau logique (les nœuds sont ordonnés en fonction d'un identifiant). Les tables de hachage distribuées pair-à-pair (DHTs) telles que DHash [19] et PAST [20]

fonctionnent selon cette stratégie. L'un des principaux intérêts est qu'un tel placement fournit une localisation implicite des différentes copies d'une donnée, y compris en cas de faute, les copies se trouvant dans le voisinage immédiat du nœud "racine" responsable de la donnée. En revanche, cela implique de nombreux mouvements de données inutiles afin de maintenir l'invariant de placement : en cas d'insertion d'un nouveau nœud, de nombreuses données devront être déplacées pour conserver un placement contigu. De plus, cela implique que des nœuds adjacents ont des contenus similaires, la section 2.3 détaille en quoi ceci est nuisible à la fiabilité du système.

À l'opposé, les stratégies éparpillant les données, permettent de les répartir sur l'ensemble des nœuds du système. C'est le cas par exemple de la réplication à base de clés multiples : pour chaque donnée, le système génère autant de clés de stockage que de copies. Avec une telle stratégie, les contenus des nœuds sont très différents. Cette famille de solutions a été mise en œuvre dans CAN [21] et Tapestry [22]. Google file system (GFS) [13] utilise une variante basée sur un placement aléatoire afin d'améliorer les performances des réparations. Il existe d'autres variantes de cette famille [23, 24] mais elles présentent toutes le même inconvénient : comme chaque nœud stocke des données dont les copies sont potentiellement éparpillées sur l'ensemble des autres nœuds, le protocole de maintenance peut alors s'avérer complexe et coûteux.

Il existe des mécanismes de réplication hybrides, Lian *et al.* par exemple, proposent une approche où les petits objets sont groupés en blocs avant d'être placés aléatoirement [25]. Cidon *et al.* [26] proposent le concept de "jeu de copies" (*copyset*) dans lesquels les copies d'une donnée doivent être stockées. Dans ce dernier travail, le but est de réduire la fréquence des pertes lors de fautes corrélées et non le nombre de données perdues.

2.2. RELAXDHT : RELÂCHER LES CONTRAINTES DE PLACEMENT DES DHTS

Les stratégies de placement éparpillant les données supportent bien le *churn* mais sont coûteuses en terme de maintenance. Les stratégies de placement contigu, utilisées par les DHT pair-à-pair, permettent une maintenance légère mais sont sensibles au *churn*. RelaxDHT a pour but de relâcher les contraintes de ces dernières pour améliorer leur résistance au *churn*.

Les DHTs pair-à-pair sont des systèmes de stockage distribués basés sur une couche de routage à base de clés (KBR pour *key-based routing*) comme Pastry [9], Chord [10], Tapestry [22] ou Kademlia [27]. Elles permettent de masquer la complexité du routage, de la réplication et de la tolérance aux fautes. Elles sont utilisées pour des applications de sauvegarde [28], mais également comme base pour des systèmes de fichiers distribués [29, 30] ou de distribution de contenus [31]. Malheureusement, Rodrigues et Blake ont montré que l'utilisation de DHTs classiques pour stocker des grandes masses de données n'est viable que si les nœuds restent en ligne pour des périodes assez longues, de l'ordre d'au moins quelques jours [7]. Le problème de la tolérance au *churn* a essentiellement été étudié au niveau de la couche de routage pair-à-pair afin d'assurer que les nœuds demeurent accessibles en maintenant leur voisinage logique [9, 32]. Cela ne règle en rien les problèmes au niveau de la couche de stockage où des migrations de données doivent être effectuées lors des insertions/déconnexions de nœuds.

Dans cette section, nous présentons RelaxDHT [LMSM09,LMSM12], une variante des mécanismes de réplication à placement contigu, basée sur PAST [20]. RelaxDHT permet de tolérer des fréquences de *churn* élevées. Notre but est de limiter les transferts de données lorsque le nombre de copies est suffisant. Nous relâchons donc la contrainte de contiguïté pour le placement des copies, permettant à un nouveau nœud de rejoindre le système sans forcer de migrations. Ensuite, pour localiser efficacement les copies de chaque donnée, nous ajoutons des métadonnées dont la gestion est intégrée au mécanisme de maintien du voisinage pair-à-pair. Cela nous permet de diminuer considérablement le nombre de transferts de données et d'offrir

une bien meilleure tolérance au *churn* qu'une solution contiguë comme PAST.

2.2.1. Les DHTs face au *churn*

La réplication dans les DHTs. Au sein d'une DHT, un identifiant (une "clé") est affecté à chaque nœud de stockage et à chaque bloc de données. La clé d'un bloc de données est en général le résultat de l'application d'une fonction de hachage sur la donnée. Le nœud ayant l'identifiant le plus proche de celui de la donnée est appelé nœud *racine* ou *root* pour cette donnée. Les identifiants forment une structure logique comme un anneau (Chord [10], Pastry [9]) ou un tore à d dimensions (CAN [21], Tapestry [22]).

Un mécanisme de réplication classique repose sur l'utilisation du *leafset* du nœud racine. Le *leafset* est la liste des $\mathtt{L}$ plus proches voisins dans la structure logique. Chaque nœud possède alors une vision du réseau logique restreinte à son *leafset* plus quelques liens longs utiles pour le routage. Il surveille régulièrement l'ensemble de son *leafset*, supprimant les nœuds qui se sont déconnectés et ajoutant les nouveaux qui rejoignent l'anneau.

Afin de pouvoir supporter les fautes, chaque bloc de données est répliqué sur $\mathtt{k}$ nœuds qui composent le *replica-set* de la donnée. Nous retrouvons les deux grandes familles de placement vues dans la section 2.1 : contigu (à base de *Leafsets*) et éparpillé (à base de clés multiples).

Réplication basée sur les *Leafsets*. Le nœud racine de la donnée stocke une copie du bloc. Le bloc est également répliqué sur les voisins immédiats du nœud racine. Dans ce cas le *replica-set* est un sous ensemble contigu du *leafset*. Selon les implémentations, les voisins stockant une copie de la donnée peuvent être les successeurs de la racine, les prédécesseurs, ou les deux. Dans tous les cas, les différentes copies d'un même bloc sont stockées de manière contiguë. Cette stratégie a été mise en œuvre notamment dans PAST [20] et DHash [19].

Réplication basée sur des clés multiples. Cette approche consiste à calculer $\mathtt{k}$ clés pour chaque donnée. Chaque clé va correspondre à un nœud "racine" distinct en charge de stocker une copie de la donnée. Le *replica-set* est donc éclaté, constitué de nœuds éloignés logiquement. Cette solution a été mise en œuvre par CAN [21] et Tapestry [22].

Dans le cas des stratégies de réplication à base de clés multiples, la maintenance doit être faite "par donnée", c'est-à-dire que pour chaque bloc il faut surveiller la disponibilité de chacune de ses racines. En revanche, dans le cas des stratégies de réplication contiguës, il suffit que chaque nœud ait une connaissance de son voisinage local, toutes les copies des données que stocke un nœud s'y trouvant rassemblées. C'est pourquoi, nous nous concentrons sur ce type de solution où la localité apportée par la notion de *leafset* permet de concevoir des protocoles de maintenance passant à l'échelle.

Le protocole de maintenance doit maintenir $\mathtt{k}$ copies de chaque bloc de données et conserver les contraintes de placement. Dans le cadre d'une réplication basée sur les *leafsets*, les $\mathtt{k}$ copies de chaque bloc de données doivent toujours être stockées par le nœud racine et ses voisins immédiats. La maintenance est généralement basée sur des échanges périodiques au sein des *leafset*. Par exemple, dans le protocole de maintenance de PAST, complètement distribué [20], chaque nœud envoie à l'ensemble de son *leafset* un filtre de bloom [1] des blocs qu'il stocke. À la réception d'un tel message, un nœud examine les données qu'il stocke, détermine celles que l'émetteur du filtre devrait également stocker et utilise le filtre reçu pour déterminer s'il les stocke effectivement. S'il détecte des blocs manquants, il répond avec la liste des clés correspondantes. Donc, après avoir diffusé son filtre de bloom à l'ensemble de son *leafset*, un nœud reçoit en réponse un ensemble de listes contenant les clés des blocs qu'il devrait stocker et qui

1. Le filtre de bloom envoyé est une vue compacte et approximative de la liste des blocs stockés par l'émetteur.

lui manquent. Il peut alors envoyer des requêtes afin de les récupérer. En l'absence de *churn*, une fois que le système a convergé, les diffusions de filtres n'induisent aucune réponse.

Impact du *churn*. Les connexions et déconnexions fréquentes induisent de nombreux changements du réseau logique pair-à-pair. Le protocole de maintenance doit fréquemment copier/migrer des blocs de données afin d'adapter le stockage à la nouvelle configuration. Si certains transferts sont inévitables car ils permettent de restaurer les `k` copies d'une donnée, d'autres ne sont utiles que pour le respect des invariants de placement.

Un premier exemple apparaît avec la notion de nœud racine. Si un nouveau nœud avec un identifiant plus proche rejoint le système, la racine d'une donnée change (par définition). Dans cette situation, la donnée est migrée sur le nouveau nœud. De plus, avec une stratégie de réplication contiguë, un nouveau nœud s'insérant dans le système s'intercale nécessairement entre des nœuds appartenant à des *replica-sets*, brisant leur contiguïté.

Le problème vient du fait que lors du départ d'un nœud, les copies qu'il stockait sont perdues. Le système doit donc les régénérées à partir des copies restantes, or cette réparation peut être longue. Si d'autres départs surviennent avant que toutes les copies n'aient été restaurées, des données peuvent être définitivement perdues. Les transferts de données "inutiles" ne servant qu'à respecter les contraintes de contiguïté congestionnent le réseau et ralentissent les réparations.

2.2.2. Fonctionnement de RelaxDHT

Notre but est de concevoir une DHT qui tolère de forts taux de *churn* sans dégrader les performances. Le principe est de limiter les transferts de données lorsque cela n'est pas nécessaire pour restaurer une copie manquante. Nous proposons RelaxDHT, un mécanisme de réplication basé sur la notion de *leafset* mais qui relâche les contraintes de contiguïté des *replica-sets*.

RelaxDHT est conçu au-dessus d'une couche de routage à base de clé (KBR) comme Pastry ou Chord. Nous conservons la notion de nœud racine. Cependant, le nœud racine d'une donnée n'en stocke plus nécessairement une copie. Son rôle est de maintenir des métadonnées décrivant le *replica-set* de la donnée et d'envoyer des messages réguliers aux nœuds du *replica-sets* afin qu'ils continuent de stocker leur copie. L'utilisation des métadonnées permet à une copie d'un bloc de données d'être n'importe où au sein du *leafset*. Un nœud peut donc rejoindre un *leafset* sans nécessairement induire de migration de blocs de données.

Nous avons choisi de contraindre le placement des copies à l'intérieur du *leafset* du nœud racine pour deux raisons. Premièrement, pour le passage à l'échelle : le nombre de messages de notre protocole ne dépend pas du nombre de blocs de données stockés, mais uniquement de la taille des *leafsets*. Deuxièmement, la couche de routage induit déjà de nombreux échanges au sein des *leafsets* et maintient une vue locale qui peut être utilisée par la couche de stockage comme un détecteur de fautes. Nous détaillons maintenant le fonctionnement de RelaxDHT.

Insertion d'un nouveau bloc de données. Afin d'être stocké dans le système, un bloc de données est inséré en utilisant l'opération `put(k,b)`. Cette opération génère un "*insert message*" qui est routé jusqu'au nœud racine. Le nœud racine choisi alors un *replica-set* de `k` nœuds, non forcément contigus, vers le centre [2] de son *leafset*. Enfin, le nœud racine envoie aux nœuds choisis un "*store message*" contenant :

1. la donnée à stocker,

2. Cela réduit la probabilité que le nœud choisi quitte rapidement le *leafset* suite à des arrivées de nouveaux nœuds, la notion de "centre" est paramètrable.

2. l'identifiant des nœuds choisis (le *replica-set*),
3. son identifiant (la racine).

Comme un nœud peut à la fois être racine pour plusieurs blocs de données et faire partie du *replica-set* de (c'est-à-dire stocker) plusieurs autres données [3], les métadonnées sur chaque nœud sont composées de deux listes.

1. Une liste `rootOfList` dont chaque élément correspond à une donnée pour laquelle le nœud est racine. Un élément est composé de l'identifiant (clé) du bloc de données et de la liste des identifiants des nœuds composant le *replica-set*.
2. Une liste `replicaOfList` dont chaque élément correspond à une donnée stockée. Un élément de cette liste est composé de l'identifiant du bloc de données, de la liste des identifiants des nœuds composant le *replica-set* et de l'identifiant du nœud racine.

Un *lease counter* est associé à chaque donnée stockée. Ce compteur est initialisé à une valeur "`Lease`" qui est un paramètre de notre système, il est décrémenté à chaque maintenance de la couche de routage et est utilisé pour déterminer si une copie est obsolète.

Protocole de maintenance. Le but de ce protocole périodique est d'assurer que : *(i)* pour chaque donnée, le nœud racine est toujours disponible et n'a pas changé ; *(ii)* chaque donnée est toujours répliquée sur `k` nœuds présents au sein du *leafset* du nœud racine.

À chaque période T, un nœud `n` exécute l'algorithme 1 qui crée et envoie des messages de maintenance aux autres nœuds du *leafset*. Il est important de noter que cet algorithme s'appuie sur la connaissance maintenue par la couche de routage (KBR) dont le délai inter-maintenance est bien plus court que celui de la couche de stockage (T).

Algorithm 1: Construction des messages de maintenance de RelaxDHT.

```
   Result: msgs, les message envoyés.
 1 for data ∈ rootOfList do
 2     for replica ∈ data.replicaSet do
 3         if NOT isInCenter (replica,leafset) then
 4             newPeer =choosePeer (replica,leafset);
 5             replace (data.replicaSet, replica,newPeer);
 6         end
 7     end
 8     for replica ∈ data.replicaSet do
 9         add (msgs [replica ],<STORE, data.blockID, data.replicaSet >) ;
10     end
11 end
12 for data in replicaOfList do
13     if NOT checkRoot (data.rootPeer,leafset) then
14         newRoot =getRoot (data.blockID,leafset);
15         add (msgs [newRoot ],<NEW ROOT, data.blockID, data.replicaSet >) :
16     end
17 end
18 for p ∈ leafset do
19     if NOT empty (msgs [p ]) then
20         send (msgs [p ],p) ;
21     end
22 end
```

Les messages construits par l'algorithme 1 contiennent des éléments de deux types différents :

STORE pour demander à un nœud de continuer à stocker un bloc de données ;

3. Il est possible, bien que non obligatoire, qu'un nœud soit à la fois racine pour une donnée et en stocke une copie.

NEW ROOT pour signaler à un nœud qu'il est devenu racine pour un bloc de données.

Ces éléments contiennent à la fois l'identifiant du bloc de données concerné et la liste des nœuds composant le *replica-set*. Pour passer à l'échelle en terme de nombre de blocs de données, l'algorithme 1 n'envoie pas plus d'un message à chaque membre du *leafset*.

Il est composé de trois phases : la première génère les éléments de type $STORE$ en utilisant la liste `rootOfList` et le *leafset* (lignes 1 à 11), la seconde génère les éléments $NEWROOT$ en se basant sur la liste `replicaOfList` et le *leafset* (lignes 12 à 17), la dernière envoie les messages construits aux nœuds du *leafset* (lignes 18 jusqu'à la fin). Les éléments générés par les deux premières phases sont ajoutés dans un tableau *msgs[]*. Chaque entrée de ce tableau est un ensemble d'éléments composant le message pour l'un des nœuds du *leafset*.

Lors de la première phase, l'algorithme vérifie que chaque copie de chaque bloc de données est toujours dans le centre du *leafset* (ligne 3). Si une copie se retrouve en dehors, un nouveau nœud est choisi aléatoirement au centre du *leafset* afin de stocker le bloc de données, le *replica-set* de la donnée est ensuite mis à jour (lignes 4 et 5). Enfin, un élément de type $STORE$ est ajouté dans les messages à destination de chacun des nœuds composant le *replica-set* (lignes 8, 9 et 10).

Lors de la seconde phase, pour chaque bloc stocké localement, la racine est vérifiée. Cette vérification est effectuée en comparant la liste `replicaOfList` et la vision locale courante du *leafset* (ligne 13). Si la racine a changé, un élément de type $NEWROOT$ est ajouté afin de prévenir le nouveau nœud racine qu'il est responsable d'un bloc de données [4]. Enfin, les messages construits sont envoyés à chaque membre du *leafset* (lignes 18 à 22).

Algorithm 2: Réception d'un message de maintenance par RelaxDHT.

```
   Data: message, le message reçu.
1  elt ∈ message  switch elt.type do
2      case STORE
3          if elt.data ∈ replicaOfList then
4              newLease(replicaOfList,elt.data);
5              updateRepSet(replicaOfList,elt.data);
6          end
7          else
8              requestBlock(elt.data);
9          end
10     end
11     case NEW ROOT
12         rootOfList = rootOfList ∪ elt.data ;
13     end
14 endsw
```

Traitement des messages de maintenance.

Pour chaque élément de type $STORE$ (ligne 2, algorithme 2), si le nœud stocke déjà une copie de la donnée, il réinitialise juste le *lease counter* et met à jour le *replica-set* si nécessaire (lignes 4 et 5). Si le nœud ne stocke pas la donnée, il la récupère à partir d'un des nœuds du *replica-set* (ligne 8).

Pour chaque élément de type $NEWROOT$ le nœud ajoute l'identifiant du bloc de données et le *replica-set* correspondant dans sa liste `rootOfList` (ligne 12).

4. Il est possible (bien que rare) d'avoir temporairement deux nœuds agissant comme racine pour une même donnée, cela ne fait qu'engendrer quelques messages supplémentaires.

Traitement des fins de bail. Si le *lease counter* d'un bloc de données atteint 0, cela signifie qu'aucun élément de type $STORE$ n'a été reçu durant les dernières maintenances. Cela peut être dû à de nombreuses insertions qui ont poussé le nœud en dehors du *leafset* de la racine de cette donnée. Le nœud envoie alors un message à la racine de la donnée demandant l'autorisation de supprimer la copie. La réponse autorisera cette suppression ou demandera au nœud de réinsérer la donnée dans le cas où elle a été perdue.

2.2.3. Effets de bord et limitations

Notre stratégie de réplication pour les DHT pair-à-pair, en relâchant les contraintes de placement, permet de réduire significativement le nombre de transferts de blocs de données lorsque des nœuds rejoignent ou quittent le système. Ceci permet de mieux tolérer le *churn*, mais cela implique d'autres effets. Notamment, cela impacte la répartition des données sur les nœuds et les performances d'accès aux données.

Répartition des données. Nous avons vu que les DHTs existantes distribuent les blocs de données sur les nœuds en fonction du résultat d'une fonction de hachage. Avec un nombre de blocs suffisamment grand, les données sont donc réparties uniformément sur l'ensemble des nœuds. Avec RelaxDHT, cela reste vrai s'il n'y a pas de connexion/déconnexion de nœuds. En revanche, en présence de *churn*, comme notre mécanisme de maintenance ne transfert pas les blocs de données à chaque insertion de nœud, les nouveaux nœuds stockeront bien moins de données que ceux présents depuis longtemps dans le système. Cependant cet effet de bord peut avoir des aspects positifs : plus un nœud est stable, plus il stockera de données. De plus, il est aisé de contrer cet effet en prenant en compte la charge de stockage des nœuds lorsque l'on en choisit un pour stocker une donnée.

Performance d'accès. Avec RelaxDHT, il est possible d'avoir temporairement des racines incohérentes, impliquant un surcoût réseau pour retrouver une donnée. Cela peut se produire par exemple lorsqu'un nœud racine pour une donnée quitte le réseau, les copies de la donnée sont toujours présentes mais la nouvelle racine (un des nœuds adjacents) peut ne pas connaître le bloc de données. Dans ce cas, le protocole de routage standard ne la trouvera pas. Cette situation va perdurer jusqu'à ce que la panne soit détectée par un des nœuds du *replica-set* et réparée par le protocole de maintenance. Il serait toutefois possible, dans les rares cas où un message de routage aboutit sur un nœud ne connaissant pas la donnée recherchée, de diffuser la requête dans l'ensemble du *leafset* ou d'effectuer une marche aléatoire bornée. Cela permettrait de pouvoir retrouver un bloc de données en l'absence de nœud racine.

2.2.4. Résultats

Nous avons implémenté à la fois PAST et RelaxDHT sur le moteur de simulation PeerSim [33]. Les principaux résultats de nos évaluations montrent que :

— RelaxDHT offre une meilleure disponibilité des données en présence de *churn* que le mécanisme de réplication de PAST. Pour des connexions/déconnexions survenant chaque minute notre stratégie perd jusqu'à deux fois moins de données.

— RelaxDHT implique en moyenne deux fois moins de transferts que PAST.

Des résultats détaillés sont disponibles dans la référence [LMSM12].

2.3. SPLAD : ÉPARPILLER ET PLACER LES DONNÉES

Lors de nos travaux autour de RelaxDHT, nous avons remarqué qu'une part importante des gains provenait d'une meilleure parallélisation des transferts de données. Nous avons étudié l'impact de la disposition des données sur les nœuds de stockage sur la pérennité à long terme. Nous avons proposé *SPLAD*, qui fournit la possibilité de régler la manière dont les données sont éparpillées et placées sur un ensemble de nœuds de stockage.

La façon dont sont disposées les copies des données a un très fort impact : plus les réparations sont rapides, plus les pertes de données seront limitées. Chaque nœud hébergeant de nombreuses copies de différents blocs de données, nous contrôlons la parallélisation des transferts de données en éparpillant plus ou moins les données sur les nœuds. Nous étudions également précisément la manière dont sont éparpillées les données et la répartition de la charge de stockage sur les nœuds.

Cette étude dépasse le contexte du domaine pair-à-pair. Nous supposons simplement que les nœuds peuvent communiquer entre eux. Notre approche est indépendante de la manière dont les nœuds et les blocs de données sont indexés et localisés.

2.3.1. Modèle de système

Le système est composé de n nœuds stockant m blocs de données répliqués k fois. Il y a donc $m*k$ copies de blocs de données réparties sur les n nœuds. Évidemment, deux copies d'une même donnée ne sont jamais placées sur un même nœud. Nous considérons que les nœuds ont des identifiants uniques et forment un anneau logique (à la manière des DHT, comme Past [20] ou DHash [19]).

Nous ne prenons pas en compte l'arrivée de nouveaux blocs de données : toutes les données sont ajoutées au système au début, puis nous observons son comportement en présence de fautes. Nous considérons que les nœuds sont homogènes ; ils ont les mêmes caractéristiques réseau, aussi bien en termes de latence que de bande passante. Nous étudions les cas des réseaux avec bandes passantes symétriques et asymétriques.

Les fautes des nœuds suivent une distribution de Poisson, et chaque nœud à la même probabilité de tomber en panne. Les fautes sont des crashs : un nœud en panne ne revient jamais, les copies des données qu'il stockait sont perdues. Chaque nœud fautif est immédiatement remplacé par un nouveau nœud, vide, ayant un nouvel identifiant. Le nombre de nœuds au sein du système est donc constant.

2.3.2. Mécanisme de réplication et parallélisation des transferts

Lorsqu'une faute survient, des copies de blocs de données sont perdues. Le système doit être réparé en recréant les copies manquantes tant que d'autres copies restent disponibles dans le système, sinon, les données risquent d'être perdues définitivement.

Comme chaque nœud stocke de nombreux blocs de données, en cas de faute d'un nœud, de nombreux blocs sont perdus. Les copies de ses blocs perdus sont localisées sur un sous-ensemble des nœuds restants. Plus ce sous-ensemble est grand, plus il y a de sources potentielles pour régénérer les blocs perdus. Notre approche repose sur la notion "d'espace de sélection" ou *selection range* dont la taille est réglable.

Nous utilisons toujours la notion de nœud racine : chaque bloc de données est confié à un nœud racine. Le nœud racine d'une donnée choisit des nœuds de stockage parmi ces voisins proches, comme illustré par la figure 2.1. Cet ensemble de voisins, centré par le nœud racine, est appelé le *selection range*. Chaque nœud a son propre *selection range*, centré par lui-même. Tous les *selection ranges* ont la même taille, c'est un paramètre du système.

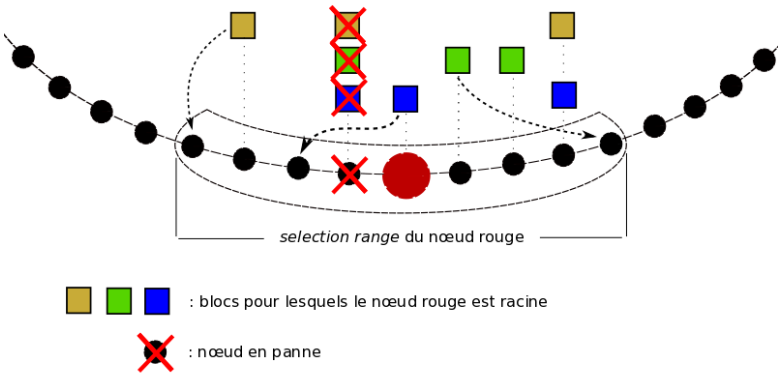


FIGURE 2.1 – Le nœud rouge est racine pour les blocs de données bleus, verts et or. Nous ne représentons que les copies de ces blocs. Les copies sont représentées juste au-dessus des nœuds qui les hébergent. Lorsqu'une faute survient, le nœud rouge choisit au sein de son *selection range* des nœuds pour stocker une nouvelle copie pour chaque donnée pour laquelle il est racine.

En réglant la taille du *selection range*, on peut contrôler le degré d'éparpillement des données sur les nœuds et donc, modifier le nombre de sources disponibles lors de la réparation d'une faute. Chaque nœud ayant son propre *selection range*, comme une fenêtre glissante, un nœud peut héberger une donnée pour laquelle le nœud racine est à la limite de son *selection range*. Il existe une probabilité non nulle que le nœud racine ait choisi l'autre extrémité de son *selection range* pour héberger une autre copie de cette même donnée. Il est donc possible qu'un nœud ait des blocs de données en commun avec des nœuds situés à l'extérieur de son *selection range*. Un nœud peut avoir des données communes avec approximativement [5] deux fois la taille du *selection range*, comme illustré par la figure 2.2.

2.3.3. Choix du nœud cible

Notre approche ne permet pas seulement de régler le degré d'éparpillement des données via la taille du *selection range*. Lorsqu'une faute survient, pour chaque bloc de données dont une copies est perdue, le nœud racine correspondant doit choisir un nouveau nœud au sein de son *selection range*. Tout nœud du *selection range* n'hébergeant pas déjà une copie du bloc de données peut être choisi. Ce choix peut ce faire suivant différentes politiques. Notre étude montre que ce choix a une influence importante non seulement sur le taux de pertes de données, mais également sur la répartition de la charge de stockage.

Nous nous sommes focalisés sur trois politiques : *aléatoire*, *moins chargé* et *power of choice*.

Aléatoire. C'est la politique la plus simple. Le nœud racine choisit un nouveau nœud de stockage uniformément, aléatoirement parmi les nœuds composant son *selection range* n'héber-

5. Même si un nœud de stockage est originellement choisit au sein du *selection range* du nœud racine, avec les insertions de nouveaux nœuds, il est possible qu'il quitte le *selection range*. Cependant, en pratique, les nœuds ne s'éloignent pas très loin de leur position d'origine.

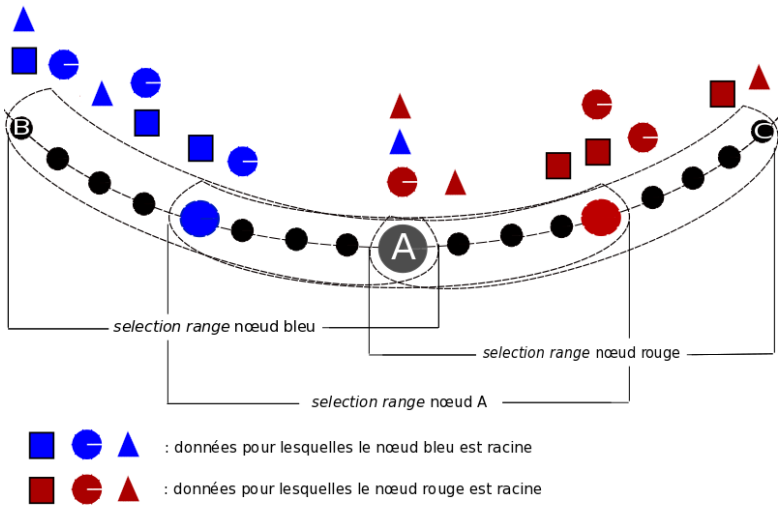


FIGURE 2.2 – Les copies des blocs de données qu'un nœud stocke peuvent être éparpillées sur $2 * |selectionrange|$ nœuds. Ici, nous représentons seulement six blocs de données, répliqués trois fois, pour lesquels les racines sont soit le nœud rouge, soit le nœud bleu. Le nœud A est à la fois dans le *selection range* du nœud bleu et celui du nœud rouge, il héberge des données communes avec les nœuds B et C localisés en dehors de son propre *selection range*. En cas de faute du nœud A, les nœuds B et C peuvent être impliqués dans la réparation.

geant pas déjà une copie de la donnée. Cette solution est facile à mettre en œuvre, elle ne nécessite pas de connaissance particulière sur les nœuds. Cependant, cette stratégie induit une mauvaise répartition de la charge de stockage : les nœuds plus anciens vont stocker de nombreux blocs. En effet, avec la stratégie aléatoire, un nœud est toujours candidat pour de nouveaux blocs de données. Donc, plus un nœud est vieux, plus il a de chance de stocker une grande quantité de données.

Le moins chargé. Afin de contrer la mauvaise répartition de la charge de stockage induite par la stratégie aléatoire, une solution consiste à choisir le nœud le moins chargé du *selection range*. Cette politique est plus coûteuse à mettre en œuvre : elle nécessite de maintenir à jour une connaissance de la charge des différents nœuds du *selection range*. Elle offre une excellente répartition de la charge de stockage. Cependant, nos évaluations ont montré que cette stratégie présente de mauvais effets de bords, impactant le taux de pertes de données lorsque le *selection range* est grand. En effet, lors de l'occurrence de fautes, un nœud qui est le moins chargé de son entourage va recevoir de très nombreuses données, limitant la parallélisation des transferts. Plus le *selection range* est grand, plus cet effet sera important.

Power of choice. Cette dernière politique a été étudiée en détail. Mitzenmacher*te al.* [34] présentent une étude détaillée de nombreuses applications du *"power of two random choices paradigm"* que nous appelons ici simplement *power of choice*. Dans notre cas, pour un bloc de données, cette stratégie consiste à choisir aléatoirement deux nœuds dans le *selection range* de la racine, puis à ne retenir que le moins chargé des deux. Cela ne nécessite pas nécessairement de maintenir une connaissance de la charge des nœuds : le nœud racine peut interroger les deux nœuds tirés aléatoirement le moment venu. Nos évaluations ont montré que cette stratégie se

comportait très bien : la probabilité pour un nœud d'être choisi décroît avec sa charge de stockage, évitant une grande surcharge pour les nœuds anciens ; et un nœud très peu chargé ne sera tout de même pas choisi de manière trop fréquente, évitant les écueils que la politique "le moins chargé".

2.3.4. Résultats

Nous avons utilisé un simulateur basé sur le moteur de simulation PeerSim [33] pour évaluer l'impact de l'éparpillement et du placement des données, voici les principaux résultats.

Premièrement, un critère important est la charge du système. Pour une capacité réseau et un taux de fautes donnés, un système de stockage distribué ne peut héberger qu'un nombre limité de blocs de données sans trop de pertes. Si l'on surcharge le système, les pertes de données vont croître exponentiellement avec la charge. Enfin, plus un système est surchargé, plus la stratégie de réplication a un impact important. Cela explique en partie pourquoi de nombreuses études se placent volontairement dans des configurations surchargées.

Deuxièmement, il est intéressant d'éparpiller les blocs de données afin de permettre au mécanisme de réparation de paralléliser les transferts. Cependant, on remarque que le gain ne croît pas linéairement avec la taille du *selection range* : lorsque l'on augmente la taille d'un petit *selection range*, les gains sont très importants, si l'on continu d'augmenter la taille, on réduit encore un peu le taux de pertes, mais cette réduction est de plus en plus petite. Comme il peut être complexe et coûteux de maintenir de grands *selection range*, un compromis entre le coût de la maintenance et les bénéfices de l'éparpillement des données doit être trouvé en fonction de l'architecture.

Enfin, la manière d'éparpiller les données, c'est-à-dire le choix des nœuds de stockage au sein des *selection ranges* est également important. Pour des petits *selection ranges*, la stratégie "le moins chargé" se comporte bien : elle offre de bons résultats à la fois en terme de pérennité et de répartition de charge. Pour des *selection ranges* plus grands, la stratégie "*power of choice*" est un bon choix. Elle permet des réparations rapides en distribuant les nouvelles copies sur de nombreuses destinations tout en conservant une répartition de la charge de stockage raisonnable, et ce, sans nécessiter un important surcoût de maintenance. Comme déjà montré par Cidon *et al.* [26], un placement aléatoire des copies des blocs de données ne donne en général pas de bons résultats.

Pour plus de détails, se référer à la référence [SMF+15].

2.4. CONCLUSIONS

La réplication permet d'améliorer la pérennité des données, mais nous avons montré que tous les mécanismes de réplication ne se valent pas. Notamment, la manière dont sont placées les copies des données a un impact important sur les temps de réparation et par conséquent, sur les taux de perte.

Nous avons proposé RelaxDHT qui permet de relâcher les contraintes de placement des DHTs, et ainsi d'offrir un plus grand degré de liberté sur le choix des nœuds de stockage. RelaxDHT améliore la tolérance au *churn* en limitant les transferts de données inutiles. Nous avons également conçu SPLAD qui permet de simuler différents degrés d'éparpillement des données et différentes stratégies de placement. SPLAD nous a permis d'observer l'impact de la manière dont sont réparties les copies des données sur leur pérennité.

Lors de la conception d'un mécanisme de réplication, la tolérance aux défaillances n'est qu'un aspect. Il est également nécessaire de prendre en compte les performances des accès aux données. C'est l'objet du chapitre suivant.

3. RÉPLICATION ET PERFORMANCES

La réplication des données permet d'améliorer les performances. D'une part, en rapprochant les copies d'une donnée des utilisateurs, il est possible de réduire les latences des accès. D'autre part, lorsqu'une donnée est extrêmement populaire, l'augmentation du nombre de copies permet de répartir la charge des requêtes. Dans ce chapitre, nous présentons nos travaux sur les mécanismes de réplications adaptés à l'utilisation des données. Ces recherches ont été réalisées dans le cadre d'une collaboration avec Orange Labs, lors des thèses de Guthemberg Da Silva Silvestre [4] et de Karine Pires [6].

Je me suis également intéressé à l'efficacité des systèmes de gestion de données dans le cadre de grands réseaux dynamiques. Un des facteurs limitant la performance est la localisation des données. Nous avons donc proposé *DONUT*, qui construit sur chaque nœud une vision globale approximative permettant un routage efficace. Ce travail a été effectué dans le cadre de la thèse de Sergey Legtchenko [3].

3.1. RÉPLICATION ADAPTÉE À L'UTILISATION

Tous les contenus ne sont pas accédés de la même manière. Certains ne seront jamais accédés (des copies de sauvegarde par exemple) alors que d'autres le seront par des dizaines de millier d'utilisateurs répartis géographiquement (un fichier vidéo populaire par exemple). Il est important d'identifier ces derniers et d'utiliser des architectures et mécanismes de distribution de contenus appropriés.

3.1.1. Distribution de contenus

Nous commençons par donner un aperçu des architectures distribuées utilisées pour distribuer les contenus. Puis nous présentons les principales manières de faire varier le nombre de copies.

Architectures distribuées de distribution de contenus. La façon de distribuer les contenus multimédia a grandement changé ces dernières années. Les réseaux de distribution de contenus (CDNs pour l'anglais *Content Delivery Networks*) permettent de fournir du contenu à grande échelle. Les CDNs sont des systèmes composés de serveurs de contenus géographiquement distribués dont le but est de passer à l'échelle et d'améliorer les temps d'accès pour les utilisateurs. Il y a en général deux types de serveur, appelés *origin* et *replica* [35]. Il est possible de classer les CDNs en fonction de la manière dont les *replicas* sont placés. On distingue deux grandes familles : cœur et bordure. L'architecture de type cœur repose sur des centres de données privés, au cœur du réseau. Cette approche a été mise en œuvre avec succès par les pionniers comme Akamai et les principaux fournisseurs de contenus et services. La plateforme Akamai [36] a été construite au-dessus d'un grand nombre de petits *clusters* de serveurs éparpillés dans de nombreux pays. De telles architectures nécessitent des algorithmes complexes pour localiser et distribuer le contenu. Certains fournisseurs de contenus, dont Amazon et Google [37], ainsi que

des fournisseurs de service, comme Limelight, ont opté pour le déploiement de gros et coûteux centres de données disposés à quelques endroits stratégiques. Ce type d'architecture présente un inconvénient majeur : il n'y a pas de contrôle sur le débit de bout en bout (jusqu'à l'utilisateur final). Les CDNs de réseaux de bordure ont émergé pour répondre à ce problème. Une des approches mise en œuvre par ce type d'architecture est le pair-à-pair. Cela consiste à déployer des serveurs de contenus directement sur les périphériques des utilisateurs, ces derniers coopérant afin de partager et distribuer le contenu.

Variation du nombre de copies. Il est possible de classer les mécanismes de réplication en trois grandes catégories en fonction de la manière dont elles déterminent le nombre de copies des contenus : fixe, proportionnel (comme les mécanismes de cache) ou adaptatif.

Google File System (GFS) [13] et Ceph [38] adoptent une approche pragmatique : le nombre de copies d'une donnée est fixe et identique pour chaque donnée. Cette approche a eu un succès considérable dans l'industrie, en particulier au sein des centres de données, car elle est facile à mettre en œuvre. Cependant, elle nécessite souvent un surdimensionnement afin d'avoir suffisamment de ressources pour les contenus populaires.

Cohen *et al.* [39] ont proposé de prendre en compte la capacité de stockage et la bande passante pour améliorer la réplication proportionnelle. Cependant, leur but était de limiter l'espace de recherche dans les réseaux pair-à-pair non structurés, sans prendre en compte la popularité des contenus. Adya *et al.* [18] et On *et al.* [40] proposent un mécanisme de réplication proportionnel intéressant en fonction de la disponibilité des nœuds de stockage, mais ils ne s'intéressent pas au temps de réponse, ni à la capacité de stockage ou à la bande passante.

Carbonite [41] étend ces concepts et propose un mécanisme de réplication adaptatif qui prend en compte à la fois la disponibilité et la durabilité. Leurs travaux, basés sur un modèle mathématique, montrent l'importance de la prise en compte de l'utilisation de la bande passante pour les mécanismes de réplication, en revanche, ils ne prennent pas en compte la popularité des données. EAD [42] et Skute [43] s'attaquent à ce problème en utilisant une approche de type coût/bénéfices pour des réseaux pair-à-pair structurés. EAD crée et supprime les copies le long du chemin de requêtes en fonction de la fréquence d'accès. Skute fournit un mécanisme de gestion de la réplication qui évalue le coût et les bénéfices de chaque copie. Plus récemment, Zhou *et al.* ont proposé DAR [44], un algorithme de réplication adaptatif pour les plateformes de vidéo à la demande (VOD). DAR permet de distribuer du contenu à grande échelle en répartissant la bande passante utilisée. Cependant nos évaluations ont montré que DAR se comporte moins bien qu'une approche basée sur du cache pour respecter des métriques de qualité de service.

Limites. Peu de travaux ou systèmes existants pour la distribution de contenus prennent en compte les réseaux de bordure. Les opérateurs, fournisseurs d'accès Internet, envisagent pourtant l'opportunité d'utiliser les *box* placées chez leurs clients et des petits centres de données régionaux, proches des utilisateurs. Evans *et al.* [45] montrent que l'utilisation des réseaux de bordure est un élément clé pour offrir des garanties de qualités de service. Par exemple, cela permet d'offrir des garanties de bande passante, ce qui peut être important pour des contenus multimédia ; alors que les métriques habituelles sont les temps de réponse ou le taux de requête moyen [46].

Il est important de concevoir des mécanismes de réplication s'adaptant à la popularité des contenus, et s'appuyant sur l'utilisation des réseaux de bordure, afin d'offrir un système de distribution de contenus garantissant une bonne qualité de service.

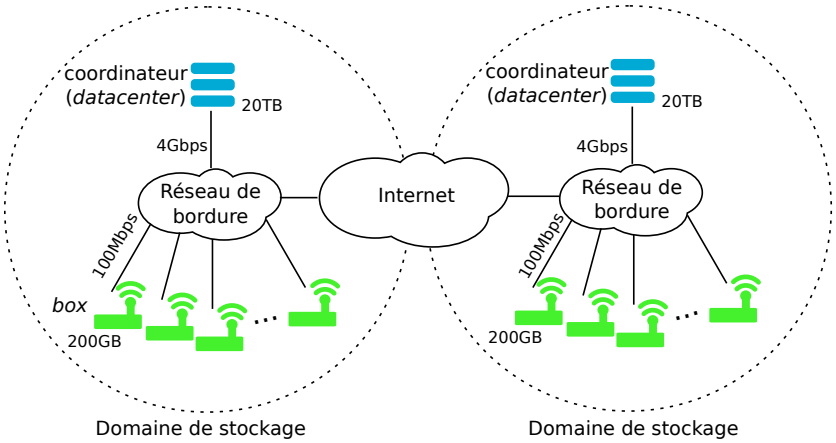


FIGURE 3.1 – *Caju* : les éléments de stockage, *box* et *datacenters*, sont rassemblés dans des domaines de stockage.

3.1.2. Distribution de contenus pour réseaux de bordure

Architecture. L'utilisation de ressources en bordure de réseaux permet d'offrir de très faibles latences et d'allouer les bandes passantes de manière précise. De plus, en réduisant la longueur du trajet entre la source du contenu et la destination, on diminue la dépense énergétique.

Les capacités en termes de bande passante et de stockage des nœuds en bordure de réseaux ont progressé de manière impressionnante ces dernières années. Les fournisseurs de réseau ont l'opportunité de fournir un stockage de type *cloud* au niveau des réseaux de bordure. Un tel type de stockage permet d'offrir des services présentant des garanties de distribution de contenus exceptionnelles, tirant parti notamment des faibles latences.

Caju [SMKS12b] est notre modèle d'architecture de systèmes de distribution de contenus, pour les réseaux de bordure, conçu en collaboration avec Orange Labs. Dans Caju, l'infrastructure du fournisseur de services est organisée en domaines de stockage fédérés, comme illustré par la figure 3.1. Un domaine de stockage est une entité logique qui rassemble un ensemble d'éléments de stockage géographiquement proches les uns des autres. Par exemple, un domaine de stockage peut être composé d'éléments de stockage d'un fournisseur d'accès (ISP pour *Internet Service Provider*) qui sont tous connectés au même DSLAM (*digital subscriber line access multiplexer*). Les éléments de stockage sont partitionnés en deux classes : *(i)* opérateur-*edge*, fournis par les opérateurs de stockage, par exemple de petits centres de données régionaux (*mini-datacenters*), et *(ii) client-edge*, fournis par les utilisateurs, comme les *box* d'accès à Internet. Les périphériques de bordure contribuent avec leur ressources au stockage *cloud*. Chaque domaine de stockage possède un nœud central (en général du type opérateur-*edge*) qui joue le rôle particulier de coordinateur. Il est responsable de la réplication au sein du domaine de stockage. Les coordinateurs de différents domaines de stockage coopèrent afin de partager les informations sur la disponibilité et la localisation des contenus et des ressources.

Les utilisateurs, effectuent des requêtes via leur *box* afin d'accéder aux contenus. Les requêtes sont traitées par le coordinateur du domaine de stockage qui, connaissant les transferts en cours, choisit une source pour la distribution du contenu demandé.

AREN : mécanisme de réplication adaptatif pour réseaux de bordure. Au-dessus de Caju, nous avons proposé *AREN*, un mécanisme de réplication adaptable pour les réseaux de bordure qui permet de respecter des contraintes de qualité de service strictes, avec une allocation de ressources efficace. La qualité de service est définie précisément par des accords entre four-

nisseurs et utilisateurs (SLA pour *Service Level Agreement*). AREN fait varier dynamiquement le nombre de copies des données en fonction de leur popularité, les deux principaux objectifs sont : *(i)* réduire le nombre de violations de SLA et *(ii)* minimiser l'espace de stockage et le réseau utilisés.

Le mécanisme de réplication d'AREN s'appuie sur la réservation de bande passante et sur un cache collaboratif. Au sein d'un domaine de stockage, AREN repose sur un coordinateur centralisé afin de traquer les réservations de bande passante et sélectionner les nœuds source des transferts. Un nœud source est sélectionné pour répondre à une requête s'il lui reste suffisamment de bande passante non réservée.

Afin d'améliorer l'allocation de ressources, AREN met en œuvre deux politiques.

— **Divide-and-conquer.** Les requêtes de type "GET" sont servies soit par des nœuds *client-edge* (les *box*) soit par les *mini-datacenters*. La politique *divide-and-conquer* donne la priorité aux nœuds *client-edge* et n'utilise les *mini-datacenters* que s'il ne reste plus de bande passante disponible à la réservation dans l'ensemble des nœuds *client-edge* hébergeant la donnée requise. Cela permet de sauvegarder la bande passante des *mini-datacenters* pour créer des copies des contenus populaires.

— **Source la plus proche.** Nous faisons l'hypothèse que les transferts intra-domaine sont plus performants. Cette politique favorise les sources présentes dans le domaine de stockage du nœud effectuant la requête lorsque cela est possible. Cela permet à AREN de diminuer le trafic inter-domaine.

À chaque une requête, le coordinateur met à jour sa vue de la demande de bande passante cumulée et décide s'il est nécessaire de créer une nouvelle copie ou non. Si une nouvelle copie doit être créée, il demande simplement au nœud ayant émit la requête de conserver la donnée transmise.

La détermination de l'utilité ou non d'une copie se fait donc de manière centralisée, par le coordinateur, en fonction de franchissement de seuils de réservation de bande passante et du nombre de copies existantes.

Résultats. Nous avons évalué AREN avec un simulateur. Nous avons mesuré le nombre de violations de SLA, la consommation de bande passante et d'espace de stockage. En comparaison avec des mécanismes de réplication simples (degré de réplication fixe, cache) AREN permet d'éviter la grande majorité des violations de SLA lorsque le système est chargé, utilise moins d'espace de stockage (jusqu'à sept fois moins dans nos évaluations) et permet d'offrir un meilleur débit.

Plus de détails sont donnés dans la référence [SMKS12a].

3.1.3. POPS : diffusion de flux vidéos en direct prenant en compte la popularité

Nous nous intéressons maintenant au cas particulier des flux vidéos diffusés en direct (live), devenus très populaires. De nombreux systèmes, comme Twitch ou YouTube, collectent des flux vidéos live d'*uploaders* et les diffusent à des utilisateurs spectateurs en utilisant un ensemble de serveurs. Cependant, le nombre de spectateurs varie considérablement dans le temps et la distribution de la popularité parmi les flux est extrêmement hétérogène. Cela amène là encore les fournisseurs de service à utiliser des plateformes surdimensionnées afin d'absorber les pics de spectateurs.

Nous nous appuyons sur des traces de justin.tv et de YouTube que nous avons collectées pour étudier le compromis entre le nombre de serveurs provisionnés et l'utilisation de la bande passante entre ces serveurs. Nous avons conçu *POPS*, un service de diffusion en direct de flux

vidéos se basant sur des prédictions de popularité pour le provisionnement des ressources. Nous commençons par présenter brièvement les leçons que nous avons apprises grâce à la collecte de traces utilisateurs sur les systèmes justin.tv [47] (devenu Twitch [48]) et YouTube [49] avant d'offrir un aperçu de POPS.

Étude des traces utilisateurs de justin.tv et YouTube. Justin.tv est une plateforme de diffusion en direct de flux vidéos générés par des utilisateurs (*uploaders*). Cette plateforme est très populaire : sur trois semaines, il y a plus de 600 millions de flux, générés par plus de 350000 *uploaders*. Ces flux, collectés par la plateforme, ont été transmis, chaque jour à plusieurs centaines de milliers de spectateurs.

Les principaux enseignements que nous en avons tirés sont : *(i)* la popularité parmi les flux est très hétérogène, les flux les plus populaires générant l'essentiel du trafic ; *(ii)* la charge globale du système est très variable dans le temps ; *(iii)* il y a en permanence un très grand nombre de flux, des ordres de grandeur au-dessus du nombre de chaînes d'une télévision câblée traditionnelle ; *(iv)* le comportement des utilisateurs est assez prévisible : il est possible d'observer une oscillation jour/nuit, mais surtout, la popularité d'un flux est très fortement liée à l'*uploader* qui la diffuse et elle a tendance à croître avec la durée du flux. C'est-à-dire que la popularité parmi les *uploaders* est très hétérogène, mais que les différents flux vidéos d'un même *uploader* auront une popularité assez semblable.

Compromis nombre de serveurs/surcoût en bande passante. Nous avons conçu un outil qui modélise une plateforme de diffusion de flux vidéos en direct. Cette plateforme reçoit des flux en direct d'*uploaders* et les retransmet aux spectateurs. Dans notre modèle, un serveur ne peut servir qu'un nombre limité, fixe, de spectateurs, noté MAX_SERV. Une approche simple consiste à attribuer des flux à un serveur tant que celui-ci n'est pas complètement chargé. Si un serveur déjà chargé voit un des flux qu'il sert acquérir de nouveaux spectateurs, il risque de dépasser sa capacité. Si l'on souhaite pouvoir continuer la diffusion de l'ensemble des flux de manière convenable, il est alors nécessaire d'allouer un nouveau serveur, ou d'en utiliser un servant déjà d'autres flux mais n'ayant pas atteint sa capacité maximale. Le serveur d'origine, continue à diffuser le flux aux spectateurs qu'il servait, il envoie également ce flux au nouveau serveur afin qu'il puisse le diffuser aux nouveaux clients, ce mécanisme de "réplication de flux" est illustré par la figure 3.2. Ce mécanisme, s'il permet de servir tous les spectateurs, a un coût en terme de bande passante. Nous nous concentrons ici sur les transferts inter-serveurs sur lesquels le placement des flux à un impact important.

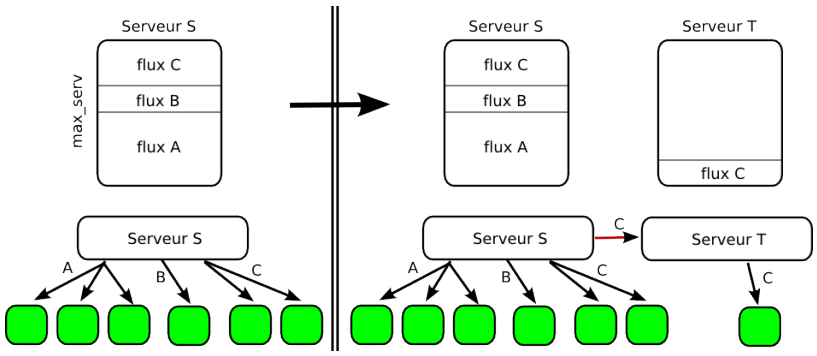


FIGURE 3.2 – Le serveur S a atteint sa capacité maximale, l'arrivée d'un nouveau client implique une réplication du flux demandé (C) sur un nouveau serveur (T).

Le nombre de spectateurs pour un flux variant au cours du temps, il est possible que le phénomène décrit ci-dessus mène à des configurations pathologiques au sein desquelles de nombreux flux sont transférés entre les serveurs.

Une solution simple permettant de limiter la nécessité de transférer des flux entre serveurs est de surprovisionner, et de prévoir une marge de progression du nombre de spectateurs pour les flux diffusés par un serveur. La valeur à laquelle la marge de progression est fixée permet de régler le compromis nombre de serveurs alloués/bande passante inter-serveurs utilisée.

Pour déterminer la marge, nous prenons en compte une estimation de la popularité à venir d'un flux. POPS s'appuie sur la constatation que la popularité des flux vidéos d'un même *uploader* varie peu. Aussi, lorsqu'un *uploader* connu (c'est-à-dire dont au moins un flux vidéo est contenu dans notre historique) diffuse un nouveau flux, POPS provisionne suffisamment de ressources (serveurs) pour servir son pic estimé de spectateurs. La prédiction étant imparfaite, et certains *uploaders* n'ayant pas d'historique, il est également possible d'ajouter une petite marge fixe. Dans nos évaluations nous avons expérimenté : *(i)* une estimation simple, un flux aura la même popularité que le précédent flux émis par le même *uploader*, et *(ii)* une estimation prenant également en compte *l'évolution* de popularité entre les derniers flux émis par le même *uploader* lorsque l'historique le permet.

Dans nos évaluations, nous avons comparé POPS à une stratégie utilisant des marges fixes (ne prenant pas en compte la popularité) et un oracle (un prédicteur parfait lisant en amont dans les traces injectées). Les résultats montrent : *(i)* l'importance de prendre en considération une estimation de la popularité, et *(ii)* la pertinence d'utiliser les popularités passées d'un *uploader*.

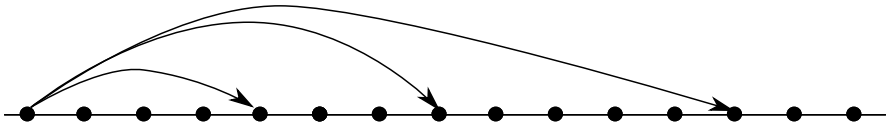
Plus de détails sont disponibles dans la référence [PMS14b].

3.2. DONUT : LOCALISATION EFFICACE DES DONNÉES

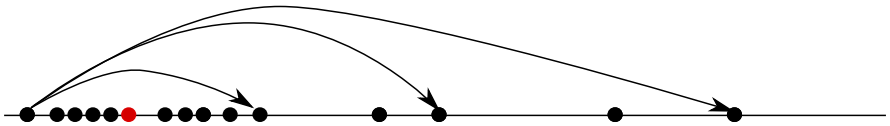
Avoir un nombre de copies adapté à la popularité des données n'est pas suffisant. Il est également nécessaire de pouvoir localiser efficacement les données. Les systèmes distribués à grande échelle rassemblent des milliers de nœuds répartis à travers le monde [50–52]. Les données sont réparties sur un réseau logique en fonction de leur identifiant. Pour offrir une localisation efficace, les systèmes doivent maintenir des raccourcis dans leur graphe logique (*overlay*). Cependant, pour créer des raccourcis efficaces, les nœuds doivent avoir des informations sur la topologie de l'*overlay*. En cas de distribution *hétérogène* des nœuds, obtenir une telle information n'est pas aisé. De plus, à cause de la présence de *churn*, la topologie peut évoluer rapidement, rendant les informations collectées obsolètes.

De nombreux systèmes reposant sur des tables de hachage distribuées (DHTs [53–55]). Construisent leur raccourcis en forçant une distribution uniforme. Malheureusement, cette distribution uniforme arbitraire fait souvent perdre les relations sémantiques entre les ressources. Pourtant, de telles relations peuvent s'avérer utiles, par exemple pour offrir un support pour les requêtes d'intervalles (*ranges queries*) [56–58].

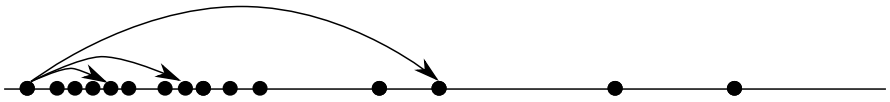
Pour faire face à ce problème *DONUT* construit une carte locale qui donne une approximation de la distribution des nœuds, permettant ainsi une estimation de la distance entre les nœuds dans le graphe. Les évaluations que nous avons réalisées avec des latences réelles et des traces de *churn* ont montré que notre carte permet d'accroître l'efficacité du routage de plus de 20% par rapport aux techniques de l'état de l'art. Notre carte est légère et peut efficacement être propagée à travers le réseau et consommant moins de $10bps$ sur chaque nœud.



a) Distribution uniforme - liens longs "classiques"



b) Distribution hétérogène - liens longs "classiques"



c) Distribution hétérogène - liens longs prenant en compte la distribution

FIGURE 3.3 – Exemples de liens longs avec des distributions uniformes et hétérogènes.

3.2.1. Localisation dans les systèmes distribués

Lorsque l'on recherche une ressource dans un *overlay*, les requêtes sont retransmises, par plusieurs nœuds avant d'atteindre leur destination. Un routage efficace doit minimiser la distance dans le graphe. On appelle "distance dans le graphe" entre deux nœuds le plus court chemin dans le graphe (en nombre d'arêtes ou *hops*) entre ces deux nœuds. Il existe des topologies ayant un petit chemin caractéristique (moyenne des plus courts chemins, *characteristic path length*), c'est-à-dire qu'elles ont naturellement des "raccourcis" ou liens longs dans leur graphe [59]. De telles topologies sont dites "petit monde", par analogie au phénomène petit monde [60].

Malheureusement, une grande partie des topologies n'ont pas cette propriété. C'est le cas par exemple des anneaux, des tores ou des topologies basées sur des triangulations de Delaunay ou des pavages de Voronoï [61, 62]. Pour offrir un routage efficace pour de telles topologies, il faut ajouter des "liens longs" au graphe de base. Kleinberg a montré qu'un routage optimal peut être atteint si la probabilité pour un nœud p de choisir un nœud q pour construire un lien long dépend de la distance dans le graphe entre p et q [63]. Avec la distribution de raccourcis de Kleinberg, appelée *d-harmonique*, la probabilité d'un lien long entre deux nœuds décroît avec la distance. Pour atteindre cette distribution, un nœud doit trouver des nœuds qui sont à une distance dans le graphe donnée. Cela implique qu'un nœud ait un minimum d'information sur la topologie du graphe (de l'*overlay*).

Impact de la distribution des identifiants. Lorsque la distribution des identifiants est uniforme, la topologie est connue, les liens longs peuvent être créés facilement. Cela correspond au cas a de la figure 3.3, où aucun des nœuds n'est à plus de 3 *hops* du nœud situé à l'extrême gauche. Malheureusement, de nombreuses études ont montré que les distributions de coordonnées sémantiques sont en général hautement non uniformes. C'est le cas par exemple dans le cadre des jeux massivement multijoueurs en ligne au sein desquels on peut observer des points chauds,

populaires, rassemblant presque l'intégralité des joueurs, alors que de grandes portions de l'espace sont désertes. Cette non uniformité est également observable dans les systèmes de partage de fichiers pair-à-pair comme Gnutella [64] : la popularité des fichiers tend à suivre une distribution Zipf [65, 66]. Si les données sont réparties de manière hétérogène, les mécanismes d'équilibrage de charge induiront une répartition hétérogène des nœuds également.

Si la distribution hétérogène des identifiants n'est pas prise en compte pour la création des liens longs, ils ne seront pas efficaces. C'est ce que l'on peut observer dans le cas b de la figure 3.3, où le nœud rouge est à 5 *hops* du nœud situé à l'extrême gauche (le nombre de liens et de nœuds sont inchangés par rapport au cas a, seule la distribution des nœuds change).

Notre but est de prendre en compte la distribution des nœuds pour créer de liens longs efficaces. Il a été prouvé que si la fonction de distribution est connue, l'approche de Kleinberg pouvait être utilisée, y compris lorsque les coordonnées sont distribuées de manière non uniforme [67]. C'est ce qui se passe dans le cas c de la figure 3.3 où l'on retrouve les mêmes distances dans le graphe que dans le cas a.

Prise en compte de la distribution des identifiants. Dans les systèmes distribués à grande échelle, obtenir et maintenir une vue globale précise est généralement impossible. Les nœuds doivent donc utiliser un mécanisme leur permettant d'approximer la distribution des identifiants, et donc la distance dans le graphe, sans pour autant avoir à maintenir la topologie exacte du graphe dynamique.

Il existe des systèmes qui prennent implicitement en compte une connaissance sur la distribution de densité pour construire des liens longs. GosSkip permet un routage efficace au-dessus d'un anneau formé par des nœuds répartis de manière hétérogène [68]. Un mécanisme de *gossip* parcours l'anneau et un système de compteurs astucieux permet de compter le nombre de nœuds "sautés" (*skip*) par les liens longs, prenant ainsi en compte la distribution. Mercury [69] et Oscar [65], quant à eux, sondent l'espace de nommage pour obtenir une approximation de sa densité.

Ces travaux sont les plus proches du notre. Nous nous concentrons sur Oscar car son mécanisme de sondage est plus précis que celui de Mercury [65].

Oscar [65] utilise un algorithme local, appelé *log-partition*, qui partitionne la population des nœuds en $log_2(n)$ ensembles de nœuds. Cependant, comme les nœuds ne connaîssent pas les distances dans le graphe, Oscar approxime la population des ensembles via des marches aléatoires, sans construire de vision globale de la distribution. Les évaluations d'Oscar [65] montrent qu'il se comporte mieux que les autres systèmes pour des *overlays* avec des distributions d'identifiants non uniformes. Pourtant, en cas de *churn*, Oscar ne détecte pas les modifications de distribution de densité tant qu'il n'y a pas de dégradation de performance. Il n'y a en effet pas de surveillance mise en œuvre. De plus les approches comme Oscar et Mercury, basées sur des marches aléatoires, sont limitées aux graphes de type "expander" [69].

3.2.2. Fonctionnement de DONUT

DONUT (*Density-aware Overlay for Non-Uniform Topologies*) est un mécanisme qui fournit sur chaque nœud, une carte globale qui permet d'estimer localement la distance avec les autres nœuds dans le graphe. La principale idée est de construire, sur chaque nœud, une vue floue de la distribution des nœuds sur l'ensemble du système. Cette carte est ensuite utilisée pour construire des liens longs efficaces. Avec DONUT, la création des liens longs est peu coûteuse, car elle est faite localement en utilisant la carte. En cas de modification due au *churn*, la carte locale s'adapte progressivement afin de refléter les nouvelles distributions de densité, permettant aux nœuds de remplacer les liens longs devenus obsolètes par de nouveaux.

À notre connaissance, il n'existe pas d'algorithme pour construire des liens longs basé sur l'utilisation d'une carte qui approxime la distribution globale des nœuds dans le système. De plus, notre carte peut également être utile à d'autres mécanismes, comme un équilibrage de charge globale, la surveillance du système, l'estimation de la taille du système, etc.

Les contributions apportées par DONUT sont : *(i)* un algorithme distribué procurant à chaque nœud une carte globale de la distribution des nœuds dans le système, et *(ii)* un algorithme qui construit des liens longs efficaces basé sur l'utilisation de cette carte.

Construction de la carte globale. Chaque nœud stocke en local un arbre qui indexe chaque région de la carte. Nous nous concentrons sur un espace de nommage à deux dimensions (x, y), nous prenons donc un quadtree. En effet, ce travail a originellement été mené dans le cadre des environnements virtuels pair-à-pair. Pour un espace à une dimension, nous aurions choisi un arbre binaire, pour un espace à trois dimensions, un octree. Chaque nœud du quadtree est responsable d'une région carrée sur la carte. Une région peut être découpée en quatre sous-régions de même taille, chaque sous-région sera confiée à un fils du nœud du quadtree. L'union des feuilles de l'arbre représentent une partition de l'espace de nommage, comme illustré par la figure 3.4. Les feuilles de l'arbre contiennent l'information sur la densité du carré qu'elles représentent. Quand une feuille est coupée en quatre, les fils héritent de la valeur de densité. L'ensemble des feuilles stocke donc une approximation de la fonction de distribution de la densité des clés.

Au démarrage, un nœud ne possède aucune information sur la distribution des densités. La seule information disponible est l'information locale. Quand le nœud rejoint l'*overlay*, on lui assigne des coordonnées dans l'espace de nommage puis il est routé à la position dans le graphe logique en fonction de ses coordonnées. Durant ce processus, il obtient les coordonnées de ses voisins dans l'*overlay*. Grâce à cela, il est capable de déterminer la densité autour de lui et de ses voisins. La densité est obtenue en calculant la surface d'un cercle de l'espace des coordonnées centré par le nœud avec un rayon rad égal à la distance du voisin le plus éloigné. Cette surface est ensuite divisée par le nombre de voisins pour obtenir la densité d. L'information disponible localement est donc le triplet ($coord$,rad,d).

Après l'insertion de l'information locale, chaque nœud construit sa carte locale avec l'estimation de la densité qui l'entoure (à gauche sur la figure 3.4). L'union de toutes les cartes forme une vue précise de la distribution de la densité des clés (à droite sur la figure 3.4). Afin de compléter leur carte, les nœuds n'ont donc plus qu'à échanger leurs informations.

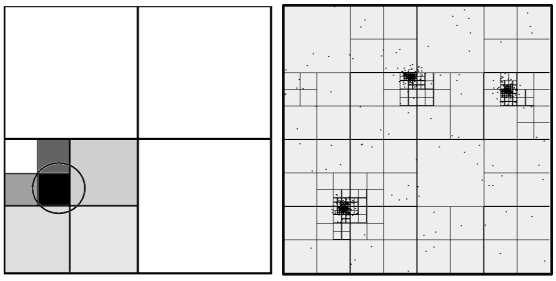


FIGURE 3.4 – **À gauche :** résultat de l'exécution de l'algorithme d'insertion sur une carte vierge. (l'échelle de gris représente le niveau de densité, noir pour la plus forte). **À droite :** distribution de la densité approximée par trois zones de densité (ici, chaque nœud est représenté par un point, il n'y a pas de niveaux de gris).

Échange d'informations entre nœuds. Sur chaque nœud, l'information sur la densité d'une région donnée est représentée par un sous-arbre. Un nœud a souhaitant partager son information concernant cette région avec un nœud b lui envoie un message contenant le sous-arbre. À la réception d'un tel message, b fusionne le sous-arbre reçu avec son sous-arbre local. Lorsque b reçoit une nouvelle version d'une région de la carte, il est capable de localiser le nœud de l'arbre correspondant. Cependant, l'arbre local peut ne pas avoir de nœud correspondant à la région reçue. Dans ce cas, l'algorithme de fusion coupe les régions du quadtree jusqu'à ce que le nœud correspondant soit créé. L'algorithme de fusion est conçu pour effacer les informations obsolètes, ce qui est important si l'on considère que la densité évolue dynamiquement.

Moyen de propagation. Nous avons implémenté deux mécanismes distribués pour propager les mises à jour des sous-arbres entre les nœuds. Le premier repose sur un algorithme de gossip. Une propriété importante de notre approche est que les nœuds sémantiquement proches ont des cartes très similaires. L'envoi de mises à jour entre eux est donc superflu. Pour cette raison, dans notre protocole de gossip chaque nœud choisit les destinataires avec une priorité inversement proportionnelle à la distance sémantique. Cela signifi qu'un nœud a de grandes chances de ne propager ses mises à jour que via les liens longs. De plus, comme la distribution de densité évolue au cours du temps, afin de limiter la retransmission de données obsolètes, notre protocole propage prioritairement les données les plus récentes.

Le deuxième mécanisme de propagation s'appuie sur l'existence de messages de type "*join*" à l'arrivée de nouveaux nœuds dans le système. Quand un nœud j rejoint l'*overlay*, il se connecte d'abord sur un nœud servant de point d'entrée e. j doit ensuite rejoindre sa position sémantique dans l'*overlay*. Une requête de type "join" est alors routée par e jusqu'aux coordonnées sémantiques. À chaque *hop* de la route, il est possible d'ajouter un peu d'information sur la densité entourant le nœud en train d'être traversé. Chaque nœud situé sur la route peut également bénéficier de la connaissance accumulée dans la requête jusqu'à lui. Cette méthode est efficace pour propager la connaissance sur la distribution de densité. Cependant, contrairement à celle basée sur le protocole de gossip, elle est tributaire du nombre de messages transitant entre les nœuds.

Construction de liens longs prenant en compte la densité. L'utilisation des mécanismes décrits ci-dessus permet aux nœuds d'acquérir progressivement une carte approximative de la distribution des densités. Nous voyons maintenant comment cette connaissance peut servir à construire des liens longs afin d'améliorer le routage dans l'*overlay*.

Estimation de la distance dans le graphe. La carte permet à chaque nœud d'estimer la distance dans le graphe qui le sépare de chaque coordonnées. Considérons src, les coordonnées du nœud, $dest$ les coordonnées de la destination dont on cherche à estimer la distance, $d_{[src,dest]}$ est la distance sémantique et $GD_{[src,dest]}$ la distance dans le graphe. Si la distribution des clés est uniforme, $GD_{[src,dest]}$ est à peu près proportionnel à $d_{[src,dest]}$. La distance sémantique est alors égale au nombre de *hops* dans le graphe multiplié par la distance euclidienne moyenne d'un *hop* multiplié par une constante k, *C'est-à-dire,* $d_{[src,dest]} = GD_{[src,dest]} \times meanHopDist \times k$.

La constante k sert à compenser le fait que la distance euclidienne est plus courte que la somme des distances de chaque lien dans le graphe. Sa valeur dépend de la topologie du graphe. Par exemple, pour un graphe de Delaunay, utilisé dans nos simulations, nous avons $k = 0.5$. Il reste à calculer $meanHopDist$, la distance euclidienne moyenne d'un *hop*.

Si n points sont uniformément distribués dans un carré SQ de taille $S = side \times side$. Il est possible d'approximer la distance euclidienne moyenne entre deux points voisins dans le

carré. Soit $meanDist_x$ et $meanDist_y$ les projections de la distance moyenne respectivement sur les axes x et y. Comme les nœuds sont distribués uniformément dans SQ, $meanDist_x = meanDist_y = \frac{side}{n}$. Cela nous permet de calculer la distance moyenne entre deux voisins $meanHopDist_{SQ} = \sqrt{meanDist_x^2 + meanDist_y^2}$.

En cas de distributions hétérogènes, les régions traversées par le segment $[src, dest]$ ont des densités différentes, la distance dans le graphe n'est donc plus proportionnelle à la distance euclidienne. Le nœud doit alors rechercher dans la carte l'information sur les densités des zones traversées par le segment. Le nœud liste l'ensemble des carrés traversés par le segment $S = \{sq_1, sq_2, ..., sq_n\}$ de sa carte approximative locale. Considérons alors $L = \{l_1, l_2, ..., l_n\}$ l'ensemble des bouts de segments de $[src, dest]$ formés des intersections avec S. Il est important de noter que :

— L est une partition de $[src, dest]$. Nous faisons donc l'hypothèse que $GD_{[src,dest]} = \sum_{i=1}^{n} GD_{l_i}$.

— Le quadtree contient le niveau de densité de chaque carré de telle manière que la densité à l'intérieur de chaque carré est considérée comme uniforme. Donc, $\forall i$, GD_{l_i} est calculable.

Le nœud est donc en mesure d'estimer localement la distance dans le graphe, en nombre de *hops*, de deux ressources en ayant leurs coordonnées dans l'espace de nommage.

La construction des liens longs. Nous utilisons l'algorithme de partitionnement d'Oscar. Cependant, nous ne faisons pas de marches aléatoires pour prendre en compte la distribution des densité, à la place, nous utilisons l'estimation de distance, basée sur la carte, décrite ci-dessus.

Le processus de création de liens longs est très léger : toutes les estimations sont faites localement. Il n'y a donc besoin que d'un routage de message par lien long créé (la requête de création de lien long). Comme le nombre de liens à ajouter est en $log(n)$ et que le processus de routage est efficace, le coût global en nombre de messages est en $log^2(n)$.

3.2.3. Résultats

Nos évaluations ont montré que : *(i)* les raccourcis construits offraient un routage efficace ; *(ii)* la performance de routage est améliorée de plus de 20% si on compare à une solution appliquant Kleinberg sans prendre en compte les différences de densité ; *(iii)* les cartes de densité sont très légères (la taille moyenne est de 2.2 Ko seulement pour 2500 nœuds) et leur propagation consomme moins de 10 bps de bande passante sur chaque nœud.

Des résultats plus détaillés sont disponibles dans la référence [LMS11].

3.3. CONCLUSIONS

Lors de la conception d'un mécanisme de réplication, les aspects "performances" sont importants. En effet, la réplication permet d'améliorer *(i)* la localité des accès en plaçant des copies des données proches des utilisateurs finaux, et *(ii)* la répartition de la charge en adaptant le nombre de copies d'une donnée à sa popularité.

Nous avons proposé Caju qui étudie l'opportunité d'exploiter les périphériques de bordure de réseaux, proches des utilisateurs. Au-dessus de Caju, nous avons construit AREN, un mécanisme de réplication adaptable qui permet de faire correspondre le nombre de copies d'une donnée à la demande. Nous avons également étudié le cas particulier des flux vidéos en direct dans le cadre de POPS.

Enfin, afin de permettre une localisation efficace des données tout en préservant leurs relations sémantiques, nous avons conçu DONUT. DONUT construit sur chaque nœud une carte floue globale de distribution des densités qui permet de mettre en place un routage efficace. De plus, cette carte peut être utile à d'autres mécanismes importants pour les systèmes distribués, comme l'équilibrage de charge, l'estimation de la taille du système, la surveillance du système ou des mécanismes de routage particuliers.

La performance est un critère important des mécanismes de réplication de données. Mais il faut également prendre en compte la cohérence des données. Souvent, un compromis cohérence/ performance est nécessaire.

4. GESTION DE LA COHÉRENCE DES DONNÉES

La réplication permet d'améliorer la tolérance aux défaillances et les performaces des systèmes de stockage. Mais l'existence de plusieurs copies d'une même donnée pose des problèmes de cohérence lors des mises à jour. Par exemple, les transactions parallèles sur les bases de données répliquées ont un surcoût important, dû à la nécessité du contrôle de la concurrence et à l'occurence de conflits pouvant mener à l'abandon de transactions [70]. Nous nous sommes intéressés à ces problèmes lors de la thèse de Pierpaolo Cincilla [5].

4.1. GARGAMEL : AMÉLIORATION DES PERFORMANCES DES BASES DE DONNÉES RÉPLIQUÉES

Nous proposons une approche, *Gargamel*, qui consiste à sérialiser en amont les transactions pouvant mener à des conflits, et à paralléliser celles qui sont indépendantes. Notre système offre des garanties transactionnelles fortes selon le modèle PSI. Chaque réplique de la base de données s'exécute de manière séquentielle, et les synchronisations entre répliques restent minimales. Nos simulations et nos expérimentations sur Amazon EC2 ont montré que Gargamel améliore d'un facteur 10 le temps de réponse, lorsque le système est très chargé, et que dans les autres cas, son surcoût est négligeable.

4.1.1. Les bases de données répliquées

La réplication des bases de données a pour objectif d'améliorer à la fois les performances et la disponibilité, en permettant à plusieurs transactions de s'exécuter en même temps, sur des copies distinctes. Cela fonctionne bien pour les transactions en lecture seule, mais reste un défi en présence d'écritures. En effet, le contrôle de concurrence est un mécanisme coûteux, de plus il est inefficace d'exécuter concurremment des transactions qui sont en conflit, car l'une des deux devra de toutes façons être abandonnée et redémarrée. Ces problèmes bien connus, limitent l'utilisation des architectures modernes telles que les multi-cœur, les *clusters*, les grilles de calcul et les *clouds*. Par exemple, Salomie *et al.* ont montré que PostgreSQL et MySQL avaient de mauvaises performances sur des architectures multi-cœur [71]. Mais c'est également le cas pour les approches réparties comme Tashkent+ [72] : comme Tashkent+ vise à maximiser l'utilisation des ressources, il souffre d'un taux d'abandon élevé.

Une réponse possible consiste à ne paralléliser que les transactions en lecture seule, et de sérialiser les transactions effectuant des écritures : c'est le cas de Ganymed [73] et de Multimed [71]. Cependant, les solutions consistant à centraliser les écritures ne fonctionnent bien que lorsqu'il y a une écrasante majorité de lectures. Gargamel peut paralléliser des transactions effectuant des écritures, lorsqu'elles ne sont pas en conflit.

4.1.2. Aperçu de Gargamel

Notre approche consiste à *classer* les transactions en fonction d'une prédiction de leurs conflits *en amont*, c'est-à-dire au niveau de l'ordonnanceur, avant toute exécution. Deux transactions qui ne sont pas en conflit l'une avec l'autre sont exécutées en parallèle sur des copies distinctes de la base de données, ce qui maximise le débit. Les transactions qui pourraient avoir des conflits entre elles sont soumises séquentiellement sur une même réplique, ce qui assure qu'elles ne seront pas abandonnées (*abort*). Cela optimise l'utilisation des ressources. Gargamel n'impose pas une synchronisation globale, forcément coûteuse. Tout ceci permet d'améliorer le débit, le temp de réponse et l'utilisation des ressources : nos simulations montrent une amélioration considérable des performances lors des pics de charge et extrêmement peu de pertes dans les autres cas.

Pour classer une transaction, notre classificateur se base sur une analyse statique de son code, ce qui nous restreint aux procédures stockées (*stored procedures* en anglais). Cela n'est pas trop restrictif car de nombreuses applications, comme les e-commerces, n'utilisent qu'un petit ensemble fixe de transactions paramétrées, sans transaction ad hoc [74]. L'analyse statique est complète : si un conflit existe, il est détecté (pas de faux négatifs). Cependant, il peut y avoir des faux positifs, c'est-à-dire que le classificateur peut signaler un conflit possible, qui n'aura pas lieu à l'exécution.

Nos contributions dans le cadre de Gargamel sont les suivantes. *(i)* Nous montrons que les transactions indépendantes peuvent être parallélisées sur des répliques différentes grâce à un classificateur situé en amont de la base de données. Chaque réplique s'exécute séquentiellement, donc sans contention. *(ii)* Nous proposons un premier classificateur faisant une analyse statique simple des procédures stockées. *(iii)* Nous avons évalué notre approche en la comparant avec un simple tourniquet, avec l'état de l'art Tashkent+ [72], et avec une solution basée sur la centralisation des écritures. Cette évaluation a été faite par simulation dans un premier temps, puis avec notre prototype dans le *cloud* Amazon EC2.

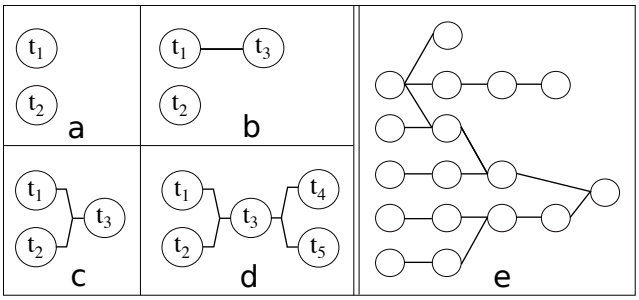
4.1.3. Fonctionnement de l'ordonnanceur

Le but de Gargamel est de partitionner dynamiquement la charge de travail dans des files d'attente distinctes de manière à ce qu'il n'existe pas de conflit entre deux transactions situées dans deux files différentes. Chaque file peut alors être exécutée de manière indépendante. En revanche, les transactions d'une même file sont exécutées en séquence, afin d'éviter les abandons.

Gargamel s'interpose entre les clients et les copies de la base de données. Lorsqu'il reçoit une nouvelle transaction, Gargamel vérifie les conflits et la place dans une file.

Nous appelons *classificateur de transactions* le composant qui prédit les possibles conflits. Le classificateur vérifie, pour chaque nouvelle transaction T, si elle est en conflit avec une transaction validée récemment ou avec une trasaction déjà ordonnancée (c'est-à-dire, déjà placée dans une file d'attente). T est placée dans la file à la suite de toutes les transactions avec lesquelles elle pourrait être en conflit ; si aucun conflit n'est anticipé, T amorce une nouvelle file.

Notre classificateur se conforme à la sémantique PSI (pour l'anglais *Parallel Snapshot Isolation*) qui est similaire à SI (pour l'anglais *Snapshot Isolation*) mais qui permet au système de répliquer les transactions de manière asynchrone [75]. Les transactions sont donc en conflit si et seulement si elles écrivent les mêmes données (l'intersection de leur ensemble d'écriture est non vide).



Un système vide reçoit t_1, t_2, etc. : *(a)* t_1 et t_2 sont indépendantes. *(b)* t_1 et t_2 sont indépendantes et t_3 est en conflit avec t_1 mais pas avec t_2. *(c)* t_3 est en conflit avec t_1 et t_2. *(d) merge* et *split*. *(e)* chaînes générales.

FIGURE 4.1 – Exemples de scénarios d'ordonnancement.

Algorithme d'ordonnancement. Les transactions apparaissant et disparaissant du système, les conflits apparaissent et disparaissent de manière imprévisible. L'une des principales tâches de Gargamel est de tenir à jour une représentation de l'ensemble des conflits. Nous détaillons ici l'utilisation des files d'attentes généralisées de Gargamel, que nous appelons *chaînes*.

Soit une transaction t entrant dans le système. Si le classificateur l'identifie comme étant potentiellement en conflit avec t', alors elle est mise en file derrière t'. Si t présente des conflits potentiels à la fois avec t' et t'', situées dans deux files distinctes (c'est-à-dire que t' et t'' sont indépendantes) alors la transaction est placée dans une chaîne *réunissant* les deux files (*merge*). De manière symétrique, deux nouvelles transactions t' et t'' qui sont toutes deux en conflit potentiel avec t déjà ordonnancée, mais qui sont indépendantes entre elles, elles seront placées dans deux files distinctes derrière t. Nous appelons ce cas un *split*.

La figure 4.1 présente quelques exemples. Les utilisateurs soumettent les transactions t_1, t_2, etc. Initialement, le système est vide. La transaction t_1 n'a évidemment pas de conflit à son arrivée ; l'ordonnanceur crée donc une première chaîne contenant uniquement t_1. Lorsque la transaction t_2 arrive, elle est comparée à t_1. Si elle est en conflit potentiel, l'ordonnanceur ajoute t_2 à la fin de la file contenant t_1 ; sinon, l'ordonnanceur affecte t_2 à une nouvelle file (figure 4.1(*a*)).

Plaçons nous dans ce deuxième cas. À l'arrivée de la transaction t_3, l'ordonnanceur la compare aux transactions t_1 et t_2. Si t_3 est indépendante des deux, elle est placée dans une nouvelle file. Si elle est possiblement en conflit avec l'une des deux, elle est placée à la fin de la file contenant la transaction conflictuelle (Figure 4.1(*b*)). Si elle est en conflit potentiel avec les deux transactions t_1 et t_2 les deux files sont alors réunies (*merge*) (Figure 4.1(*c*)). Si maintenant les transactions t_4 et t_5 sont toutes deux en conflit potentiel avec t_3 mais pas entre elles, elles seront placées sur des files parallèles démarrant derrière t_3 (la file "*split*" (Figure 4.1(*d*))). En répétant l'algorithme, Gargamel génère des chaînes qui se prolongent, se réunissent (*merge*) ou se séparent (*split*) en fonction des conflits potentiels entre transactions (Figure 4.1(*e*)).

Une fois qu'une réplique a fini d'exécuter une transaction, elle propage les écritures (le *write-set*) à toutes les autres répliques de la base de données, en utilisant une diffusion respectant l'ordre causal [76]. La diffusion du *write-set* est faite en tâche de fond. Les répliques doivent attendre la propagation des écritures des transactions en conflit avant d'exécuter une transaction. Par exemple, pour exécuter une transaction après un *merge*, la réplique attend de recevoir les *write-sets* des transactions dans les files avant le point de réunion.

4.1.4. Le caractère correct du protocole

Gargamel assure le niveau d'isolation PSI [75] grâce aux propriétés suivantes :

(i) Gargamel assure que deux transactions en conflit ne seront jamais exécutées de manière parallèle. Une transaction n'est donc validée que si ses écritures n'entrent pas en conflit avec une transaction parallèle.

(ii) Chaque réplique de la base de données assure SI localement. La propagation des écritures dans l'ordre causal assure que les écritures d'une transaction t ne seront jamais visibles après celles de t' si t' a vu les écritures de t.

(iii) Une réplique ne commence pas une transaction tant qu'elle n'a pas reçu les écritures des transactions précédentes causalement.

Pour préciser la propriété *(iii)*, lorsque l'ordonnanceur envoie une nouvelle transaction à une réplique de la base de données, il ajoute à son message la liste des transactions conflictuelles récentes exécutées sur une autre réplique (cela peut arriver, par exemple lorsque que deux chaînes se réunissent). À sa réception, la réplique vérifie que toutes les écritures correspondantes ont été reçues. Si ce n'est pas le cas, la nouvelle transaction est mise en attente.

4.1.5. Travaux connexes

Nous avons vu qu'une première famille de solutions consiste à centraliser toutes les écritures sur une réplique et à ne paralléliser que les transactions en lecture. C'est l'approche adoptée par Ganymed [73] au sein d'un *cluster* et par Multimed [71] pour les machines multi-cœur. Cependant, cette approche très prudente donne de mauvais temps de réponse, elle ne prend pas en compte que certaines transactions en écriture peuvent être exécutées en parallèle, si ces dernières sont indépendantes. Notons, que Gargamel, doté un classificateur considèrant simplement que toutes les transactions en écriture sont en conflit entre elles, serait équivalent à cette approche.

L'approche de Tashkent+ [72], que nous confrontons à Gargamel dans nos évaluations, vise à maximiser le débit des transactions. Cette approche optimise l'affectation des transactions aux répliques en fonction de la mémoire disponible sur les nœuds. Tashkent+ surveille également les CPUs et l'utilisation disque de chaque réplique afin de bien équilibrer la charge. Pour estimer la capacité nécessaire à une transaction Tashkent+ examine le plan d'exécution de la base de données.

Une autre approche consiste à relâcher les contraintes sur la cohérence, et d'abandonner la cohérence forte [77]. Ainsi, de nombreux systèmes comme Facebook, Zynga ou Twitter reposent sur *memcached*, un système de stockage clé-valeur présentant des garanties faibles de cohérence [78]. Cette approche est prometteuse pour certaines applications, mais elle est source de complexité pour les développeurs et ne peut s'appliquer à toutes les applications.

H-Store [74] s'appuie sur une connaissance préalable pour partitionner et paralléliser la charge entre les répliques. Il nécessite que *l'ensemble* de la charge de travail soit spécifiée en amont, définie statiquement dans des procédures stockées. Cette Grâce à cette connaissance, H-Store améliore les performances de plusieurs ordres de grandeur par rapport à d'autres systèmes de gestion de bases de données commerciaux.

Le système de Pacitti *et al.* [79] est un système de bases de données répliquée multi-maître. Il impose un ordre total sur les transactions grâce à une diffusion fiable et un mécanisme d'ordonnancement des messages à base d'estampilles temporelles.

Un ordonnanceur prenant en compte les conflits est proposé par Amza et al. [80, 81]. Leur système assure le niveau d'isolation SI car il exécute toutes les écritures sur toutes les répliques

dans un ordre total. Gargamel quant à lui n'exécute les transactions qu'une seule fois, ce qui permet notamment d'éviter des divergences en cas de non-déterminisme des transactions. Gargamel parallélise les transactions indépendantes et diffuse leur *write-sets* en dehors du chemin critique.

Similairement à notre approche, Middle-R [82], sérialise les transactions en conflit en les classant en amont de l'exécution. Middle-R associe chaque transaction en écriture à une classe de conflit. À chaque classe correspond une copie principale, sur laquelle sont exécutées les transactions de cette classe. Après exécution, les écritures sont propagées aux autres copies. La principale différence avec notre travail est que Middle-R associe statiquement des copies de la bases à des classes de transactions et n'évite pas l'occurrence d'abandons. Gargamel partitionne la base dynamiquement en fonction des transactions reçues et ne force jamais une transaction à abandonner.

4.1.6. Résultats

Nos simulations utilisant le *benchmark* TPC-C ont montré que : *(i)* lorsque la charge est importante, Gargamel améliore les temps de réponse et le débit des transactions en écriture de 25% et 75% respectivement par rapport à un ordonnanceur de type tourniquet et à Tashkent+. Lorsque la charge est faible, notre approche n'apporte aucun bénéfice, mais le surcoût reste négligeable. *(ii)* Pour une même charge de travail, Gargamel nécessite moins de ressources, permettant de réduire les coûts financiers et écologiques.

Des résultats plus détaillés sont disponibles dans les références [CMS12] et [CMS14].

4.2. GARGAMEL MULTI-SITES : SYNCHRONISATION OPTIMISTE POUR BASES DE DONNÉES GÉO-DISTRIBUÉES.

Nous avons étendu Gargamel aux configurations géo-distribuées. Ainsi, si un centre de données (*datacenter*) entier est indisponible, la base de données, elle, reste disponible à une autre localisation. Gargamel multi-sites présente un faible coût de synchronisation, les copies étant synchronisées de manière optimiste.

4.2.1. Architecture

Chaque site, ou *datacenter* possède un nœud Gargamel en charge d'ordonnancer les requêtes et de maintenir les chaînes de transactions en conflit potentiel. Les utilisateurs envoient leurs requêtes au nœud Gargamel de leur site, lequel vérifie localement les conflits et ordonnance une exécution sur une réplique, exactement comme le Gargamel mono-site présenté ci-dessus. Les nœuds ordonnanceurs Gargamel de chaque site s'echangent les transactions reçues de manière asynchrone. Pour éviter une divergence, ils doivent se synchroniser. Cette synchronisation est optimiste : le nœud Gargamel n'attend pas de se synchroniser avant d'envoyer la requête à une réplique pour exécution.

4.2.2. Résolution des collisions

Un nœud Gargamel peut recevoir des requêtes de ses utilisateurs (transactions *locales*), ou d'un autre nœud Gargamel (transactions *distantes*). Lorsqu'il reçoit une transaction locale, il l'ordonnance, l'envoie pour exécution, puis la transmet (avec ses dépendances dans la chaîne) de manière asynchrone et fiable aux autres nœuds Gargamel. Quand la transaction est exécutée et validée, ses écritures sont propagées à toutes les répliques (locales et distantes) afin qu'elles appliquent les écritures de la transaction sans avoir à la ré-exécuter.

Les mêmes transactions ne parviennent donc pas nécessairement aux différents nœuds Gargamel dans le même ordre, et il se peut que deux sites ordonnancent les mêmes transactions dans un ordre différent, nous appelons cela une *collision*. Deux sites ne doivent pas valider des transactions qui s'exécutent dans des ordres différents, car sinon, leurs états pourraient alors diverger.

Détection Une collision peut être détectée à différents moments. Si elle est détectée après l'insertion d'une transaction dans une chaîne mais avant son exécution, la transaction est alors *annulée* et déplacée. Si en revanche l'exécution de la transaction a commencé, Gargamel force son abandon [1]. On dit alors que la transaction est *tuée*. Enfin, une transaction ne doit pas être validée (*commit*) tant que les informations sur la présence ou non d'une collision ne sont pas disponibles. Dans ce cas la validation doit attendre et la transaction est dite *bloquée*.

Conséquences. Une transaction annulée ne coûte rien. Une transaction tuée, que ce soit pendant son exécution, ou après avoir été bloquée, dégrade les performances et gaspille les ressources (temps de calcul perdu inutilement) ; enfin, le blocage d'une transaction qui finit par être validée ne gaspille pas de ressources, mais ralentit l'exécution.

Résolution. En cas de collision, Gargamel exécute un protocole de consensus entre nœuds Gargamel. En présence de plusieurs branches, le protocole favorise la plus longue afin de minimiser le travail perdu. Les transactions se trouvant dans d'autres branches sont alors annulées ou tuées.

4.2.3. Résultats

Les évaluations réalisées avec notre prototype dans le *cloud* Amazon EC2 ont montré que Gargamel divisait les temps de réponse par 10 quand la contention est élevée dans le cas mono-site. Dans le cas géo-distribué, qui offre une bonne garantie de tolérance au défaillances et la possibilité d'accès en lecture plus rapides grâce à une meilleure localisation, la dégradation de performances du *benchmark* TPC-C est négligeable.

Nous concluons de nos expériences que : *(i)* l'approche optimiste pour l'ordonnancement des requêtes atténue l'impact de la latence réseau inter-site ; *(ii)* il y a des collisions, mais leur impact reste limité.

Pour plus de détails, se référer à la référence [CMS14].

4.3. CONCLUSIONS

Gargamel permet d'exécuter des transactions parallèles sur une base de données répliquée. En configuration mono-site, toutes les transactions sont ordonnancées de manière centralisée et deux transactions qui risquent d'être en conflit sont toujours exécutées séquentiellement. Il n'y a donc aucun travail perdu. Les transactions dont l'indépendance est détectée peuvent être exécutées en parallèle sur des répliques différentes. Plus le classificateur est précis dans ses anticipations de conflits, plus il est possible de paralléliser les transactions indépendantes.

En configuration multi-sites, la base de données et les métadonnées (les chaînes généralisées représentant le graphe de dépendances des transactions) sont géo-répliquées. Dans cette configuration Gargamel offre de bonnes garanties de tolérance aux fautes tout en conservant de bonnes performances grâce à son mécanisme de synchronisation optimiste.

1. Contrairement à Gargamel mono-site, Gargamel multi-sites peut forcer des transactions à abandonner.

5. Conclusions et perspectives

5.1. Conclusions

Nous avons vu que les systèmes de gestion de données distribués à grande échelle doivent offrir un stockage fiable, performant et cohérent. La réplication de données est au cœur de ces problématiques.

Au cours de ces huit dernières années, ma recherche, axée sur la gestion des données, a visé à répondre à ces besoins. Elle s'est effectuée en collaboration avec huit doctorants dont quatre ont soutenu et quatre sont encore en cours de thèse. Notre recherche s'est articulée autour de la tolérance aux fautes, de la performance des accès, et de la gestion de la cohérence.

Tolérance aux fautes. Avec RelaxDHT et SPLAD, nous avons étudié l'impact de la répartition des données sur leur pérennité. Nous avons proposé un mécanisme de gestion de métadonnées ainsi qu'un modèle permettant de définir efficacement et précisément la façon dont les copies des données sont placées. Notre approche permet de concevoir des systèmes de stockage de données fiables, avec une maintenance peu coûteuse.

Performances. Dans le cadre de collaborations avec Orange Labs (une thèse CIFRE et un projet collaboratif), nous avons proposé des approches pour améliorer les performances d'accès aux données, avec notamment Caju pour exploiter les périphériques de bordure de réseau comme les *box* d'accès Internet, AREN pour adapter le degré de réplication de chaque donnée en fonction de la demande, et POPS pour une diffusion efficace de flux vidéos en direct.

Nous avons également conçu DONUT, qui permet d'offrir un routage (et donc une localisation des données) efficace dans les systèmes distribués à grande échelle lorsque la distribution des identifiants n'est pas uniforme. DONUT, en apportant une vue globale approximative de la distribution sur chaque nœud, offre de nombreuses perspectives intéressantes (équilibrage de charge, routage intelligent, etc.).

Cohérence. Dans le contexte des bases de données répliquées, Gargamel propose une approche permettant de détecter, en amont, des conflits potentiels entre transactions. Les transactions dont on prévoit qu'elles seront éventuellement en conflit sont exécutées de manière séquentielle. Gargamel améliore les performances à la fois en terme de débit de transactions, et en quantité de ressources utilisées, car il garantit qu'aucune transaction n'est abandonnée pour cause de conflit.

Vers une gestion conjointe de la tolérance aux fautes, la performance des accès et la cohérence des données. Les systèmes distribués à grande échelle sont complexes : les capacités et la fiabilité des nœuds et des réseaux sont très hétérogènes ; la répartition des accès aux données dans le temps et dans l'espace est loin d'être uniforme ; enfin les besoins applicatifs, notamment en terme de performances et de niveau de cohérence des données, sont également très variables.

Lors de la conception d'un système de gestion de données distribué, toutes ces contraintes doivent être prise en considération. Par exemple, le choix du nombre de copies d'une donnée doit prendre en compte le niveau de tolérance aux fautes souhaité, la popularité de la donnée et le niveau de cohérence requis par l'application. Lors du placement des copies, là encore on considérera la localisation des accès, la localisation des autres copies pour les aspects liés à la tolérance aux fautes (degré d'éparpillement, fautes corrélées) et les contraintes de cohérence. C'est pourquoi nous nous proposons d'étudier la gestion de données en considérant conjointement tolérance aux fautes, performance et cohérence.

5.2. PERSPECTIVES

Nous nous concentrerons sur deux contextes particuliers pour l'étude de la tolérance aux fautes, de la performance des accès et de la cohérence des données : *(i)* les jeux massivement multijoueurs en ligne (MMOG pour l'anglais *Massively Multiplayer Online Games*), un type d'application particulièrement exigeant ; et *(ii)* le placement dynamique de données visant à économiser l'énergie.

Les MMOGs : exigeants vis-à-vis du système de gestion de données. Les MMOGs sont des applications à grande échelle [83], faisant des accès fréquents aux données, et présentant de fortes contraintes de performances [84, 85]. De surcroît, ces applications présentent une grande variabilité des accès aux données, à la fois dans le temps et dans l'espace. Il existe des périodes dans la journée où très peu de joueurs sont connectés, et d'autres où le système doit faire face à des pics de charge [86]. Les joueurs connectés ne sont pas placés uniformément dans l'environnement virtuel : ils sont répartis entre différentes zones denses, et leurs mouvements font évoluer rapidement la distribution des densités [87, 88]. Cela implique des accès aux données qui sont répartis de manière très hétérogène et qui varient fortement au cours du temps. Les MMOGs nécessitent donc un système de stockage performant et passant à l'échelle tout en supportant une utilisation extrêmement hétérogène et dynamique.

En revanche, les contraintes apportées par l'environnement virtuel peuvent faciliter la gestion de la cohérence. Par exemple, un joueur ne peut généralement voir et agir que sur ce qui l'entoure, et ses mouvements peuvent souvent être anticipés [LMT10].

Les solutions commerciales actuelles simplifient le problème en divisant l'espace de jeu (l'environnement virtuel) entre instances distinctes [89, 90]. Les joueurs se retrouvent donc partitionnés, répartis dans des zones cloisonnées. Ces barrières artificielles éliminent les contraintes de passage à l'échelle, mais empêchent les joueurs de véritablement jouer ensemble, de tous interagir.

Il existe de nombreux travaux visant à offrir aux utilisateurs un monde unifié. Certains reposent sur le modèle pair-à-pair où chaque nœud gère les données qui l'entourent dans l'environnement virtuel [58, 62, 91, 92]. De telles solutions, complètement décentralisées, présentent souvent de mauvaises performances, notamment en présence de joueurs très mobiles [LMT10]. Une autre famille de solutions met en œuvre de multiples serveurs se partageant l'environnement, découpé en micro-zones dynamiques [93] ; c'est le cas par exemple de *BigWorld* [94].

Nous visons une approche hybride, s'appuyant sur un nombre variable de serveurs, alloués dans un *cloud* et sur des périphériques de bordure de réseau (comme les *box* d'accès Internet des joueurs). Nous nous intéresserons particulièrement à la gestion de la cohérence des données et à leur distribution dynamique sur les serveurs et les *box*.

Distribution et redistribution dynamique de données pour l'économie d'énergie. À l'instar des MMOGs, de nombreuses applications ont une charge de travail qui varie beaucoup au cours du temps. Ces dernières années, cela a fait le succès des *clouds*, qui permettent une utilisation *élastique*, à la demande, des ressources. Si l'utilisation d'un nombre variable de nœuds est aisée pour certaines applications, ce n'est pas le cas pour celles manipulant de grandes masses de données complexes. Dans ce cas, l'ensemble des données ne peut pas être simplement répliqué sur chaque nœud, car d'une part, elles sont souvent trop volumineuses, et d'autre et part, une réplication totale sur une grand nombre de nœuds rendrait la gestion de la cohérence trop coûteuse. Chaque nœud n'héberge donc qu'une partie des données qui sont réparties et répliquées sur les nœuds en fonction de leurs affinités, afin d'offrir de bonnes performances. Lorsque des nœuds sont ajoutés ou retirés pour faire face au début ou à la fin d'un pic de charge, les données doivent être redistribuées sur le nouvel ensemble de nœuds.

Cette redistribution, à chaud, doit préserver les affinités des données et répartir convenablement la charge. Car, en plus des pics d'activité, la distribution des accès aux données est souvent hautement non-uniforme et peut varier au cours du temps. Pour maintenir de bonnes performances, la solution consiste généralement à surdimensionner l'infrastructure, d'où un gaspillage de ressources. Nous souhaitons concevoir des mécanismes de gestion dynamique de données qui redistribuent automatiquement les données. Le but étant de n'utiliser que le nombre de nœuds strictement nécessaire au traitement de la charge courante. De plus, puisque la redistribution vise à réduire la consommation énergétique, il faut prendre en compte l'énergie nécessaire au redéploiement des données.

Les *datacenters* consommant une quantité importante et continuellement croissante d'énergie, cette recherche vise à réduire l'empreinte énergétique de la gestion des données.

BIBLIOGRAPHIE

BIBILIOGRAPHIE PRINCIPALE

[1] Eric A. Brewer. Towards robust distributed systems (abstract). In *Proceedings of the Nineteenth Annual ACM Symposium on Principles of Distributed Computing*, PODC '00, pages 7–, New York, NY, USA, 2000. ACM.

[2] Seth Gilbert and Nancy Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *SIGACT News*, 33(2) :51–59, 2002.

[3] Sergey Legtchenko. *Dynamic adaptation of distributed architectures for massively multiplayer online games*. PhD thesis, Université Pierre et Marie Curie, October 2012.

[4] Guthemberg Da Silva Silvestre. *Distributed data storage leveraging edge devices*. PhD thesis, Université Pierre et Marie Curie, October 2013.

[5] Pierpaolo Cincilla. *Adaptive consistency for large-scale data replication*. PhD thesis, Université Pierre et Marie Curie, September 2014.

[6] Karine Pires. *Delivery and Transcoding for Large Scale Live Streaming Systems*. PhD thesis, Université Pierre et Marie Curie, March 2015.

[7] Rodrigo Rodrigues and Charles Blake. When multi-hop peer-to-peer lookup matters. In *IPTPS '04 : Proceedings of the 3rd International Workshop on Peer-to-Peer Systems*, pages 112–122, San Diego, CA, USA, February 2004.

[8] Joshua B. Leners, Hao Wu, Wei-Lun Hung, Marcos K. Aguilera, and Michael Walfish. Detecting failures in distributed systems with the Falcon spy network. In *Twenty-Third ACM Symposium on Operating Systems Principles*, pages 279–294, New York, NY, USA, 2011.

[9] Antony Rowstron and Peter Druschel. Pastry : Scalable, decentralized object location and routing for large-scale peer-to-peer systems. *Lecture Notes in Computer Science*, 2218 :329–350, 2001.

[10] Ion Stoica, Robert Morris, David Liben-Nowell, David R. Karger, Frans F. Kaashoek, Frank Dabek, and Hari Balakrishnan. Chord : a scalable peer-to-peer lookup protocol for Internet applications. *IEEE/ACM Trans. Netw.*, 11(1) :17–32, February 2003.

[11] 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*, SOSP '11, pages 15–28, New York, NY, USA, 2011. ACM.

[12] Dhruba Borthakur. HDFS architecture guide. *HADOOP APACHE PROJECT http ://hadoop. apache. org/common/docs/current/hdfs design. pdf*, 2008.

[13] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google File System. In *SOSP '03 : Proceedings of the 9th ACM symposium on Operating systems principles*, pages 29–43, New York, NY, USA, October 2003. ACM Press.

[14] Brad Calder, Ju Wang, Aaron Ogus, Niranjan Nilakantan, Arild Skjolsvold, Sam McKelvie, Yikang Xu, Shashwat Srivastav, Jiesheng Wu, Huseyin Simitci, et al. Windows Azure storage : a highly available cloud storage service with strong consistency. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, pages 143–157. ACM, 2011.

[15] Stevens le Blond, Fabrice le Fessant, and Erwan le Merrer. Finding good partners in availability-aware P2P networks. In *International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS'09)*, Nov 2009.

[16] Alessio Pace, Vivien Quema, and Valerio Schiavoni. Exploiting node connection regularity for DHT replication. *Reliable Distributed Systems, IEEE Symposium on*, 0 :111–120, 2011.

[17] Robbert van Renesse. Efficient reliable internet storage. In *WDDDM '04 : Proceedings of the 2nd Workshop on Dependable Distributed Data Management*, Glasgow, Scotland, October 2004.

[18] A. Adya, W. Bolosky, M. Castro, R. Chaiken, G. Cermak, J. Douceur, J. Howell, J. Lorch, M. Theimer, and R. Wattenhofer. Farsite : Federated, available, and reliable storage for an incompletely trusted environment. In *OSDI '02 : Proceedings of the 5th Symposium on Operating Systems Design and Implementation*, Boston, MA, USA, December 2002.

[19] Frank Dabek, Jinyang Li, Emil Sit, James Robertson, Frans F. Kaashoek, and Robert Morris. Designing a DHT for low latency and high throughput. In *NSDI '04 : Proceedings of the 1st Symposium on Networked Systems Design and Implementation*, San Francisco, CA, USA, March 2004.

[20] Antony I. T. Rowstron and Peter Druschel. Storage management and caching in PAST, a large-scale, persistent peer-to-peer storage utility. In *SOSP '01 : Proceedings of the 8th ACM symposium on Operating Systems Principles*, pages 188–201, December 2001.

[21] Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard Karp, and Scott Schenker. A scalable content-addressable network. In *SIGCOMM*, volume 31, pages 161–172. ACM Press, October 2001.

[22] Ben Y. Zhao, Ling Huang, Jeremy Stribling, Sean C. Rhea, Anthony D. Joseph, and John D. Kubiatowicz. Tapestry : A global-scale overlay for rapid service deployment. *IEEE Journal on Selected Areas in Communications*, 2003.

[23] Ali Ghodsi, Luc Onana Alima, and Seif Haridi. Symmetric replication for structured peer-to-peer systems. In *DBISP2P '05 : Proceedings of the 3rd International Workshop on Databases, Information Systems and Peer-to-Peer Computing*, page 12, Trondheim, Norway, August 2005.

[24] Salma Ktari, Mathieu Zoubert, Artur Hecker, and Houda Labiod. Performance evaluation of replication strategies in DHTs under churn. In *MUM '07 : Proceedings of the 6th international conference on Mobile and ubiquitous multimedia*, pages 90–97, New York, NY, USA, December 2007. ACM Press.

[25] Qiao Lian, Wei Chen, and Zheng Zhang. On the impact of replica placement to the reliability of distributed brick storage systems. In *ICDCS '05 : Proceedings of the 25th IEEE International Conference on Distributed Computing Systems*, pages 187–196, Washington, DC, USA, June 2005. IEEE Computer Society.

[26] Asaf Cidon, Stephen M. Rumble, Ryan Stutsman, Sachin Katti, John Ousterhout, and Mendel Rosenblum. Copysets : Reducing the frequency of data loss in cloud storage. In

Proceedings of the 2013 USENIX Conference on Annual Technical Conference, USENIX ATC'13, pages 37–48, Berkeley, CA, USA, 2013. USENIX Association.

[27] Petar Maymounkov and David Mazieres. Kademlia : A peer-to-peer information system based on the xor metric. In *IPTPS '02 : Proceedings of the 1st International Workshop on Peer-to-Peer Systems*, pages 53–65, Cambridge, MA, USA, March 2002.

[28] Martin Landers, Han Zhang, and Kian-Lee Tan. Peerstore : Better performance by relaxing in peer-to-peer backup. In *P2P '04 : Proceedings of the 4th International Conference on Peer-to-Peer Computing*, pages 72–79, Washington, DC, USA, August 2004. IEEE Computer Society.

[29] Jean-Michel Busca, Fabio Picconi, and Pierre Sens. Pastis : A highly-scalable multi-user peer-to-peer file system. In *Euro-Par '05 : Proceedings of European Conference on Parallel Computing*, pages 1173–1182, August 2005.

[30] Frank Dabek, Frans M. Kaashoek, David Karger, Robert Morris, and Ion Stoica. Wide-area cooperative storage with CFS. In *SOSP '01 : Proceedings of the 8th ACM symposium on Operating Systems Principles*, volume 35, pages 202–215, New York, NY, USA, December 2001. ACM Press.

[31] Jimmy Jernberg, Vladimir Vlassov, Ali Ghodsi, and Seif Haridi. Doh : A content delivery peer-to-peer network. In *Euro-Par '06 : Proceedings of European Conference on Parallel Computing*, page 13, Dresden, Germany, September 2006.

[32] Sean Rhea, Dennis Geels, Timothy Roscoe, and John Kubiatowicz. Handling churn in a DHT. In *Proceedings of the 2004 USENIX Technical Conference, Boston, MA, USA*, June 2004.

[33] Márk Jelasity, Alberto Montresor, Gian Paolo Jesi, and Spyros Voulgaris. The Peersim simulator. http://peersim.sourceforge.net/.

[34] Michael Mitzenmacher, Andréa W. Richa, and Ramesh Sitaraman. The power of two random choices : A survey of techniques and results. In *in Handbook of Randomized Computing*, pages 255–312. Kluwer, 2000.

[35] M. Pathan and R. Buyya. A taxonomy of cdns. In *Content Delivery Networks*. Springer Berlin Heidelberg, 2008.

[36] E. Nygren, R. K. Sitaraman, and J. Sun. The akamai network : a platform for high-performance internet applications. *SIGOPS*, 2010.

[37] P. Gill, M. Arlitt, Z. Li, and A. Mahanti. The flattening internet topology : Natural evolution, unsightly barnacles or contrived collapse ? In *PAM*. 2008.

[38] S. Weil, S. A. Brandt, E. L. Miller, D. D. E. Long, and C. Maltzahn. Ceph : A scalable, high-performance distributed file system. In *OSDI*, 2006.

[39] Edith Cohen and Scott Shenker. Replication strategies in unstructured peer-to-peer networks. In *SIGCOMM*, 2002.

[40] G. On, J. Schmitt, and R. Steinmetz. Quality of availability : Replica placement for widely distributed systems. In *IWQoS*, 2003.

[41] B.-G. Chun, F. Dabek, A. Haeberlen, E. Sit, H. Weatherspoon, F. Kaashoek, J. Kubiatowicz, and R. Morris. Efficient replica maintenance for distributed storage systems. In *NSDI*, 2006.

[42] H. Shen. An efficient and adaptive decentralized file replication algorithm in P2P file sharing systems. *IEEE Transactions on Parallel and Distributed Systems*, 2010.

[43] N. Bonvin, T. G. Papaioannou, and K. Aberer. A self-organized, fault-tolerant and scalable replication scheme for cloud storage. In *SOCC*, 2010.

[44] Y. Zhou, T. Z. J. Fu, and D. M. Chiu. A unifying model and analysis of P2P vod replication and scheduling. In *INFOCOM*, 2012.

[45] J. Evans and C. Filsfils. Deploying diffserv at the network edge for tight slas, part 1. *Internet Computing, IEEE*, 2004.

[46] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo : amazon's highly available key-value store. In *SIGOPS*, 2007.

[47] Justin.tv, http://www.justin.tv/, January 2014.

[48] Twiitch, http://www.twitch.tv/, January 2014.

[49] Youtube, http://www.youtube.com/live/, January 2014.

[50] Saikat Guha, Neil Daswani, and Ravi Jain. An experimental study of the skype peer-to-peer voip system. In *IPTPS*, February 2006.

[51] Petar Maymounkov and David Mazières. Kademlia : A peer-to-peer information system based on the xor metric. In Peter Druschel, M. Frans Kaashoek, and Antony I. T. Rowstron, editors, *IPTPS*, volume 2429 of *LNCS*, pages 53–65. Springer, 2002.

[52] Johan A. Pouwelse, Pawel Garbacki, Dick H. J. Epema, and Henk J. Sips. The bittorrent P2P file-sharing system : Measurements and analysis. In Miguel Castro and Robbert van Renesse, editors, *IPTPS*, volume 3640 of *LNCS*, pages 205–216. Springer, 2005.

[53] Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard M. Karp, and Scott Shenker. A scalable content-addressable network. In *SIGCOMM*, pages 161–172, 2001.

[54] Peter Druschel and Antony I. T. Rowstron. PAST : A large-scale, persistent peer-to-peer storage utility. In *HotOS*, pages 75–80. IEEE Computer Society, 2001.

[55] Frank Dabek, M. Frans Kaashoek, David R. Karger, Robert Morris, and Ion Stoica. Wide-area cooperative storage with cfs. In *SOSP*, pages 202–215, 2001.

[56] S. Voulgaris, A. M. Kermarrec, L. Massoulie, and M. van Steen. Exploiting semantic proximity in peer-to-peer content searching. In *10th International Workshop on Future Trends in Distributed Computing Systems (FTDCS 2004)*, Suzhou, China, May 2004.

[57] Ozgur D. Sahin, Abhishek Gupta, Divyakant Agrawal, and Amr El Abbadi. A peer-to-peer framework for caching range queries. In *ICDE*, pages 165–176. IEEE Computer Society, 2004.

[58] Ashwin R. Bharambe, Jeffrey Pang, and Srinivasan Seshan. Colyseus : A distributed architecture for online multiplayer games. In *NSDI*. USENIX, 2006.

[59] D. J. Watts and S. H. Strogatz. Collective dynamics of 'small-world' networks. *Nature*, 393(6684) :440–442, June 1998.

[60] Stanley Milgram. The small world problem. *Psychology Today*, 1 :61, 1967.

[61] Olivier Beaumont, Anne-Marie Kermarrec, and Etienne Riviere. Peer to peer multidimensional overlays : Approximating complex structures. In *OPODIS*, volume 4878 of *LNCS*, pages 315–328. Springer, 2007.

[62] Shun-Yun Hu, Jui-Fa Chen, and Tsu-Han Chen. Von : a scalable peer-to-peer network for virtual environments. *IEEE Network*, 20(4) :22–31, 2006.

[63] Jon M. Kleinberg. The small-world phenomenon : an algorithmic perspective. In *STOC*, pages 163–170, 2000.

[64] Matei Ripeanu, Ian T. Foster, and Adriana Iamnitchi. Mapping the gnutella network : Properties of large-scale peer-to-peer systems and implications for system design. *CoRR*, cs.DC/0209028, 2002.

[65] Sarunas Girdzijauskas, Anwitaman Datta, and Karl Aberer. Structured overlay for heterogeneous environments : Design and evaluation of oscar. *TAAS*, 5(1), 2010.

[66] Lee Breslau, Pei Cao, Li Fan, Graham Phillips, and Scott Shenker. Web caching and zipf-like distributions : Evidence and implications. In *INFOCOM*, pages 126–134, 1999.

[67] Sarunas Girdzijauskas, Anwitaman Datta, and Karl Aberer. On small world graphs in non-uniformly distributed key spaces. In *ICDE Workshops*, page 1187, 2005.

[68] Rachid Guerraoui, Sidath B. Handurukande, Kevin Huguenin, Anne-Marie Kermarrec, Fabrice Le Fessant, and Etienne Riviere. Gosskip, an efficient, fault-tolerant and self organizing overlay using gossip-based construction and skip-lists principles. In *International Conference on Peer-to-Peer Computing (P2P'06)*, 2006.

[69] Ashwin R. Bharambe, Mukesh Agrawal, and Srinivasan Seshan. Mercury : supporting scalable multi-attribute range queries. In Raj Yavatkar, Ellen W. Zegura, and Jennifer Rexford, editors, *SIGCOMM*, pages 353–366. ACM, 2004.

[70] M Tamer Özsu and Patrick Valduriez. *Principles of distributed database systems*. Springer Science & Business Media, 2011.

[71] Tudor-Ioan Salomie, Ionut Emanuel Subasu, Jana Giceva, and Gustavo Alonso. Database engines on multicores, why parallelize when you can distribute ? In *Euro. Conf. on Comp. Sys. (EuroSys)*, pages 17–30, Salzburg, Austria, 2011. ACM.

[72] Sameh Elnikety, Steven Dropsho, and Willy Zwaenepoel. Tashkent+ : memory-aware load balancing and update filtering in replicated databases. In *Euro. Conf. on Comp. Sys. (EuroSys)*, volume 41, pages 399–412, Lisbon, Portugal, March 2007. ACM SIG on Op. Sys. (SIGOPS), Assoc. for Computing Machinery.

[73] Christian Plattner and Gustavo Alonso. Ganymed : Scalable replication for transactional web applications. In *Int. Conf. on Middleware (MIDDLEWARE)*, pages 155–174, 2004.

[74] Michael Stonebraker, Samuel R. Madden, Daniel J. Abadi, Stavros Harizopoulos, Nabil Hachem, and Pat Helland. The end of an architectural era (it's time for a complete rewrite). In *VLDB*, Vienna, Austria, 2007.

[75] Yair Sovran, Russell Power, Marcos K. Aguilera, and Jinyang Li. Transactional storage for geo-replicated systems. In *Symp. on Op. Sys. Principles (SOSP)*, pages 385–400, Cascais, Portugal, October 2011. Assoc. for Computing Machinery.

[76] Rachid Guerraoui and Luís Rodrigues. *Introduction to Reliable Distributed Programming*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006.

[77] Werner Vogels. Eventually consistent. *ACM Queue*, 6(6) :14–19, October 2008.

[78] Jure Petrovic. Using memcached for data distribution in industrial environment. In *ICONS*, pages 368–372. IEEE Computer Society, 2008.

[79] Esther Pacitti, M. Tamer Özsu, and Cédric Coulon. Preventive multi-master replication in a cluster of autonomous databases. In *Euro-Par'03*, pages 318–327, 2003.

[80] Cristiana Amza, Alan L. Cox, and Willy Zwaenepoel. Distributed versioning : Consistent replication for scaling back-end databases of dynamic content web sites. In Markus Endler and Douglas C. Schmidt, editors, *Middleware*, volume 2672 of *Lecture Notes in Computer Science*, pages 282–304. Springer, 2003.

[81] Cristiana Amza, Alan L. Cox, and Willy Zwaenepoel. Conflict-aware scheduling for dynamic content applications. In *USENIX Symposium on Internet Technologies and Systems*, 2003.

[82] Marta Patiño Martinez, Ricardo Jiménez-Peris, Bettina Kemme, and Gustavo Alonso. MIDDLE-R : Consistent database replication at the middleware level. *Trans. on Computer Systems*, 23(4) :375–423, November 2005.

[83] Number of subscriber for world of warcraft by quarters from 2005 to 2014 by statista, from blizzard entertainment. http://www.statista.com/statistics/276601/number-of-world-of-warcraft-subscribers-by-quarter/. Accédé le : 16/03/2015.

[84] Tom Beigbeder, Rory Coughlan, Corey Lusher, John Plunkett, Emmanuel Agu, and Mark Claypool. The effects of loss and latency on user performance in unreal tournament 2003. In Wu chang Feng, editor, *NETGAMES*, pages 144–151. ACM, 2004.

[85] Mark Claypool and Kajal Claypool. Latency can kill : Precision and deadline in online games. In *Proceedings of the First Annual ACM SIGMM Conference on Multimedia Systems*, MMSys '10, pages 215–222, New York, NY, USA, 2010. ACM.

[86] Yeng-Ting Lee and Kuan-Ta Chen. Is server consolidation beneficial to mmorpg ? a case study of world of warcraft. In *Cloud Computing (CLOUD), 2010 IEEE 3rd International Conference on*, pages 435–442, July 2010.

[87] Daniel Pittman and Chris GauthierDickey. A measurement study of virtual populations in massively multiplayer online games. In *NetGames '07 : Proc. of the 6th ACM SIGCOMM workshop on Network and system support for games*, pages 25–30, New York, NY, USA, 2007. ACM.

[88] Chi-Anh La and Pietro Michiardi. Characterizing user mobility in Second Life. In *SIGCOMM 2008, ACM Workshop on Online Social Networks, August 18-22, 2008, Seattle, USA*, August 2008.

[89] Second Life. http://secondlife.com/.

[90] World of Warcraft. http://www.worldofwarcraft.com/.

[91] Joaquín Keller and Gwendal Simon. Solipsis : A massively multi-participant virtual world. In Hamid R. Arabnia and Youngsong Mun, editors, *PDPTA*, pages 262–268. CSREA Press, 2003.

[92] Matteo Varvello, Christophe Diot, and Ernst W Biersack. P2P Second Life : experimental validation using Kad. In *Infocom 2009, 28th IEEE Conference on Computer Communications*, pages 19–25, Rio de Janeiro, Brazil, April 2009.

[93] Alexandre Denault, César Cañas, Jörg Kienzle, and Bettina Kemme. Triangle-based obstacle-aware load balancing for massively multiplayer games. In *Proceedings of the 10th Annual Workshop on Network and Systems Support for Games*, NetGames '11, pages 4 :1–4 :6, Piscataway, NJ, USA, 2011. IEEE Press.

[94] Bigworld server. http://bigworldtech.com/en/technology/bigworld-server.

BIBILIOGRAPHIE PERSONNELLE

JOURNAUX INTERNATIONAUX (5)

[SBP+15] Guthemberg Silvestre, David Buffoni, Karine Pires, Sébastien Monnet, and Pierre Sens. Boosting streaming video delivery with wisereplica. *Transactions on Large-Scale Data- and Knowledge-Centered Systems, TLDKS*, 9070 :34–58, 2015.

[VMMG14] Maxime Véron, Olivier Marin, Sébastien Monnet, and Zahia Guessoum. Towards a scalable refereeing system for online gaming. *Multimedia Systems*, 20(5) :579–593, 2014.

[LMSM12] Sergey Legtchenko, Sébastien Monnet, Pierre Sens, and Gilles Muller. RelaxDHT : A Churn Resilient Replication Strategy for Peer-to-Peer Distributed Hash-Tables. *ACM Transactions on Autonomous and Adaptive Systems, TAAS*, 7(2), July 2012.

[MMAG07] Ramsés Morales, Sébastien Monnet, Gabriel Antoniu, and Indranil Gupta. Move : Design and evaluation of a malleable overlay for group-based applications. *IEEE Transactions on Network and Service Management, TNSM*, 4(2) :107–116, September 2007.

[ADM06] Gabriel Antoniu, Jean-François Deverge, and Sébastien Monnet. How to bring together fault tolerance and data consistency to enable grid data sharing. *Concurrency and Computation : Practice and Experience, CCPE*, 18(13) :1705–1723, November 2006. Extended and revised version of [ADM04].

CONFÉRENCES INTERNATIONALES (14)

[VMMS15b] Maxime Véron, Olivier Marin, Sébastien Monnet, and Pierre Sens. RepFD - Using reputation systems to detect failures in large dynamic networks. In *the 44th International Conference on Parallel Processing (ICPP 2015)*, Beijing, China, September 2015.

[LSMS15b] Maxime Lorrillere, Julien Sopena, Sébastien Monnet, and Pierre Sens. Puma : Pooling unused memory in virtual machines for I/O intensive applications. In *the 8th ACM International Systems and Storage Conference (SYSTOR 2015)*, Haifa, Israel, May 2015.

[SMF+15] Véronique Simon, Sébastien Monnet, Mathieu Feuillet, Philippe Robert, and Pierre Sens. Scattering and placing data replicas to enhance long-term durability. In *the 14th IEEE International Symposium on Network Computing and Applications (NCA 2015)*, Cambridge, USA, September 2015. Short paper.

[PMS14b] Karine Pires, Sébastien Monnet, and Pierre Sens. POPS : a popularity-aware live streaming service. In *the 20th IEEE International Conference on Parallel and Distributed Systems (ICPADS 2014)*, Hsinchu, Taiwan, December 2014.

[SMBS13] Guthemberg Silvestre, Sébastien Monnet, David Buffoni, and Pierre Sens. Predicting popularity and adapting replication of Internet videos for high-quality delivery. In *the 19th IEEE International Conference on Parallel and Distributed Systems (ICPADS 2013)*, Seoul, South Korea, December 2013.

[SMKS12a] Guthemberg Silvestre, Sébastien Monnet, Ruby Krishnaswamy, and Pierre Sens. AREN : a popularity aware replication scheme for cloud storage. In *the 19th IEEE International Conference on Parallel and Distributed Systems (ICPADS 2012)*, Singapore, December 2012.

[CMS12] Pierpaolo Cincilla, Sébastien Monnet, and Marc Shapiro. Gargamel : boosting DBMS performance by parallelising write transactions. In *the 18th IEEE International Conference on Parallel and Distributed Systems (ICPADS 2012)*, Singapore, December 2012.

[LMS11] Sergey Legtchenko, Sébastien Monnet, and Pierre Sens. DONUT : Building Shortcuts in Large-Scale Decentralized Systems with Heterogeneous Peer Distributions. In *the 30th IEEE Symposium on Reliable Distributed Systems (SRDS 2011)*, Madrid, Spain, October 2011.

[LMT10] Sergey Legtchenko, Sébastien Monnet, and Gaël Thomas. Blue banana : resilience to avatar mobility in distributed MMOGs. In *the 40th IEEE/IFIP International Conference on Dependable Systems and Networks (DSN 2010)*, Chicago, USA, July 2010.

[LMSM09] Sergey Legtchenko, Sébastien Monnet, Pierre Sens, and Gilles Muller. Churn-resilient replication strategy for peer-to-peer distributed hash-tables. In *the 11th International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS 2009)*, volume 5873 of *Lecture Notes in Computer Science*, pages 485–499, Lyon, France, November 2009.

[MMAG06] Sébastien Monnet, Ramsés Morales, Gabriel Antoniu, and Indranil Gupta. MOve : Design of An Application-Malleable Overlay. In *the 25th IEEE Symposium on Reliable Distributed Systems 2006 (SRDS 2006)*, pages 355–364, Leeds, UK, October 2006.

[ACM06a] Gabriel Antoniu, Loïc Cudennec, and Sébastien Monnet. Extending the entry consistency model to enable efficient visualization for code-coupling grid applications. In *6th IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGrid 2006)*, pages 552–555, Singapore, May 2006.

[ABJM04] Gabriel Antoniu, Luc Bougé, Mathieu Jan, and Sébastien Monnet. Large-scale deployment in P2P experiments using the JXTA distributed framework. In *the European Conference on Parallel Computing (Euro-Par 2004)*, volume 3149 of *Lecture Notes in Computer Science*, pages 1038–1047, Pisa, Italy, August 2004.

[MMB04b] Sébastien Monnet, Christine Morin, and Ramamurthy Badrinath. Hybrid checkpointing for parallel applications in cluster federations. In *the 4th IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGrid 2004)*, Chicago, IL, USA, April 2004. IEEE. Poster, electronic version.

WORKSHOPS INTERNATIONAUX (11)

[VMM14a] Maxime Véron, Olivier Marin, and Sébastien Monnet. Matchmaking in multiplayer on-line games : studying user traces to improve the user experience. In *the 24th International Workshop on Network and OS Support for Digital A/V (NOSDAV 2014)*, pages 579–593, Singapore, March 2014.

[CMS14] Pierpaolo Cincilla, Sébastien Monnet, and Marc Shapiro. Multi-site gargamel : Optimistic synchronization for reliable geo-replicated databases. In *the 1st International Workshop on Middleware for Dependable Systems and Networks (MW4NG 2014)*, pages 4 :1–4 :6, Bordeaux, France, December 2014.

[SM13] Guthemberg Silvestre and Sébastien Monnet. Performing accurate simulations for deadline-aware applications. In *the 4th International Workshop on Modeling*

and Simulation of Peer-to-Peer and Autonomic Systems (MOSPAS 2013), held in conjunction with HPCS, Helsinki, Finland, July 2013.

[VMMG12] Maxime Véron, Olivier Marin, Sébastien Monnet, and Zahia Guessoum. Towards a scalable refereeing system for online gaming. In *the 11th International Workshop on Network and Systems Support for Games (NetGames 2012) (Poster)*, Venice, Italy, November 2012.

[MAM+12] Felipe Maia, Rafael Araujo, Luiz Carlos Muniz, Rayrone Zirtany, Luciano Coutinho, Samyr Vale, Francisco Silva, Sébastien Monnet, Luciana Bezerra Arantes, Pierpaolo Cincilla, Ikram Chabbouh, and Marc Shapiro. A Grid Based Distributed Cooperative Environment for Health Care Research. In *the 2nd International Symposium on Foundations of Health Information Engineering and Systems (FHIES 2012)*, Paris, France, August 2012.

[SMKS12b] Guthemberg Silvestre, Sébastien Monnet, Ruby Krishnaswamy, and Pierre Sens. Caju : a content distribution system for edge networks. In *the 1st International Workshop on Big Data Management in Clouds (Big Data Cloud 2012), held in conjunction with Euro-Par*, Rhodes Island, Greece, August 2012.

[ACM+07] Gabriel Antoniu, Antonio Congiusta, Sébastien Monnet, Domenico Talia, and Paolo Trunfio. Peer-to-peer metadata management for knowledge discovery applications in grids. In *the CoreGRID Workshop on Grid Middleware, held in conjunction with the Internationnal Supercomputing Conference (ISC 2007)*, Dresden, Germany, June 2007.

[MB06] Sébastien Monnet and Marin Bertier. Using failure injection mechanisms to experiment and evaluate a grid failure detector. In *the International Workshop on Computational Grids and Clusters (WCGC 2006), Held in conjunction with VECPAR*, volume 4395 of *Lecture Notes in Computer Science*, pages 610–621, Rio de Janeiro, Brazil, July 2006. Selected for publication in the post-conference book.

[ACM06b] Gabriel Antoniu, Loïc Cudennec, and Sébastien Monnet. A practical evaluation of a data consistency protocol for efficient visualization in grid applications. In *the International Workshop on High-Performance Data Management in Grid Environment (HPDGrid 2006), Held in conjunction with VECPAR*, volume 4395 of *Lecture Notes in Computer Science*, pages 692–706, Rio de Janeiro, Brazil, July 2006. Selected for publication in the post-conference book.

[MMB04a] Sébastien Monnet, Christine Morin, and Ramamurthy Badrinath. A hierarchical checkpointing protocol for parallel applications in cluster federations. In *the 9th IEEE Workshop on Fault-Tolerant Parallel Distributed and Network-Centric Systems (FTPDS 2004), held in conjunction with IPDPS*, page 211, Santa Fe, New Mexico, April 2004.

[ADM04] Gabriel Antoniu, Jean-François Deverge, and Sébastien Monnet. Building fault-tolerant consistency protocols for an adaptive grid data-sharing service. In *the International ACM Workshop on Adaptive Grid Middleware (AGridM 2004)*, Antibes Juan-les-Pins, France, September 2004.

CHAPITRES DE LIVRE (3)

[MT11] Sébastien Monnet and Gaël Thomas. *Large-Scale peer-to-peer game applications*, pages 81–103. John Wiley & Sons, Ltd., 2011.

[MTM11] Olivier Marin, Gaël Thomas, and Sébastien Monnet. *Peer-to-Peer storage*, pages 59–80. John Wiley & Sons, Ltd., 2011.

[ABC+06] Gabriel Antoniu, Marin Bertier, Eddy Caron, Frédéric Desprez, Luc Bougé, Mathieu Jan, Sébastien Monnet, and Pierre Sens. Gds : An architecture proposal for a grid data-sharing service. In *Future Generation Grids*, CoreGRID series, pages 133–152. Springer, 2006.

CONFÉRENCES ET JOURNAUX NATIONAUX (10)

[LSMS15a] Maxime Lorrillere, Julien Sopena, Sébastien Monnet, and Pierre Sens. Conception et évaluation d'un système de cache réparti adapté aux environnements virtualisés. *Technique et Science Informatiques (TSI)*, 2015. To appear.

[VMMS15a] Maxime Véron, Olivier Marin, Sébastien Monnet, and Pierre Sens. Etude des services de matchmaking dans les jeux mutlijoueurs en ligne : récupérer les traces utilisateur afin d'améliorer l'expérience de jeu. *Technique et Science Informatiques (TSI)*, 2015. To appear.

[LSM15] Maxime Lorrillere, Julien Sopena, and Sébastien Monnet. Gestion dynamique du cache dans les machines virtuelles. In *Conférence d'informatique en Parallélisme, Architecture et Système (ComPAS 2015)*, pages 1–10, Villeneuve d'Ascq, France, June 2015.

[LSMS14] Maxime Lorrillere, Julien Sopena, Sébastien Monnet, and Pierre Sens. PUMA : un cache distant pour mutualiser la mémoire inutilisée des machines virtuelles. In *Conférence d'informatique en Parallélisme, Architecture et Système (ComPAS 2014)*, pages 1–12, Neuchâtel, Suisse, April 2014.

[PMS14a] Karine Pires, Sébastien Monnet, and Pierre Sens. Pops : service de diffusion de flux vidéos live prenant en compte la popularité. In *Conférence d'informatique en Parallélisme, Architecture et Système (ComPAS 2014)*, pages 1–12, Neuchâtel, Suisse, April 2014.

[VMM14b] Maxime Véron, Sébastien Monnet, and Olivier Marin. Matchmaking dans les jeux multijoueurs en ligne : étudier les traces utilisateurs pour améliorer l'expérience de jeu. In *Conférence d'informatique en Parallélisme, Architecture et Système (ComPAS 2014)*, pages 1–12, Neuchâtel, Suisse, April 2014.

[LSMS13] Maxime Lorrillere, Julien Sopena, Sébastien Monnet, and Pierre Sens. Vers un cache réparti adapté au cloud computing. In *Conférence d'informatique en Parallélisme, Architecture et Système (ComPAS 2013)*, Grenoble, France, January 2013.

[VMMG13] Maxime Véron, Olivier Marin, Sébastien Monnet, and Zahia Guessoum. Vers un système d'arbitrage décentralisé pour les jeux en ligne. In *Conférence d'informatique en Parallélisme, Architecture et Système (ComPAS 2013)*, Grenoble, France, January 2013.

[CM05] Loïc Cudennec and Sébastien Monnet. Extension du modèle de cohérence à l'entrée pour la visualisation dans les applications de couplage de code sur grilles. In *Actes des Journées francophones sur la Cohérence des Données en Univers Réparti (CDUR 2005)*, Paris, France, November 2005.

[DM05] Jean-François Deverge and Sébastien Monnet. Cohérence et volatilité dans un service de partage de donnés dans les grilles de calcul. In *Actes des Rencontres francophones du parallélisme (RenPar 2005)*, pages 47–55, Le Croisic, France, April 2005.

AUTRES (3)

[TMM15] Guillaume Turchini, Sébastien Monnet, and Olivier Marin. Scalability and availability for massively multiplayer online games. In *the 11th European Dependable Computing Conference (EDCC 2015)*, Paris, France, September 2015. Fast abstract.

[PhD06] Sébastien Monnet. Gestion des données dans les grilles de calcul : support pour la tolérance aux fautes et la cohérence des données. Thèse de doctorat, Université de Rennes 1, novembre 2006.

[M03] Sébastien Monnet. Conception et évaluation d'un protocole de reprise d'applications parallèles dans une fédération de grappes de calculateurs. Rapport de stage de DEA, DEA d'informatique de l'IFSIC, Université de Rennes 1, juin 2003.

Abstract. Data replication is a key mechanism for building a reliable and efficient data management system. Indeed, by keeping several replicas for each piece of data, it is possible to improve durability. Furthermore, well-placed copies reduce data access time. However, having multiple copies for a single piece of data creates consistency problems when the data is updated. Over the last years, I made contributions related to these three aspects : data durability, data access performance and data consistency. RelaxDHT and SPLAD enhance data durability by placing data copies smartly. Caju, AREN and POPS reduce access time by improving data locality and by taking popularity into account. To enhance data lookup performance, DONUT creates efficient shortcuts taking data distribution into account. Finally, in the replicated database context, Gargamel parallelizes independent transactions only, improving database performance and avoiding aborting transactions. My research has been carried out in collaboration with height PhD students, four of which have defended. In my future work, I plan to extend these contributions by *(i)* designing a storage system tailored for MMOGs, which are very demanding, and *(ii)* designing a data management system that is able to re-distribute data automatically in order to scale the number of servers up and down according to the changing workload, leading to a greener data management.

Résumé. La réplication de données est une technique clé pour permettre aux systèmes de gestion de données distribués à grande échelle d'offrir un stockage fiable et performant. Comme il gère un nombre suffisant de copies de chaque donnée, le système peut améliorer la pérennité. De plus, la présence de copies bien placées réduit les temps d'accès. Cependant, cette même existence de plusieurs copies pose des problèmes de cohérence en cas de modification. Ces dernières années, mes contributions ont porté sur ces trois aspects liés à la réplication de données : la pérennité des données, la performance des accès et la gestion de la cohérence. RelaxDHT et SPLAD permettent d'améliorer la pérennité des données en jouant sur le placement des copies. Caju, AREN et POPS permettent de réduire les temps d'accès aux données en améliorant la localité et en prenant en compte la popularité. Pour accélérer la localisation des copies, DONUT crée des raccourcis efficaces prenant en compte la distribution des données. Enfin, dans le contexte des bases de données répliquées, Gargamel permet de ne paralléliser que les transactions qui sont indépendantes, améliorant ainsi les performances et évitant tout abandon de transaction pour cause de conflit. Ces travaux ont été réalisés avec huit étudiants en thèse dont quatre ont soutenu. Pour l'avenir, je me propose d'étendre ces travaux, d'une part en concevant un système de gestion de données pour les MMOGs, une classe d'application particulièrement exigeante ; et, d'autre part, en concevant des mécanismes de gestion de données permettant de n'utiliser que la quantité strictement nécessaire de ressources, en redistribuant dynamiquement les données en fonction des besoins, un pas vers une gestion plus écologique des données.