Stratégie de réplication résistante au churn pour les tables de hachage distribuées en pair-à-pair

Sergey Legtchenko
LIP6/Université Pierre et Marie Curie/CNRS/INRIA
sergey.legtchenko@systeme.lip6.fr

Résumé

Les systèmes pair-à-pair basés sur les tables de hachage distribuées (*Distributed Hash Table*, DHT) fournissent un moyen passant à l'échelle et tolérant aux fautes pour stocker des blocs de données de manière totalement distribuée. Malheureusement, des études récentes ont montré que si la fréquence de connexion/déconnexion (*churn*) est trop élevée, des blocs de données peuvent être perdus. Pour adresser ce problème, nous avons concu RelaxDHT. Il s'agit d'une table de hachage distribuée avec une résistance au churn accrue grâce à la mise en place de mécanismes de maintenance et de réplication adaptés. Dans nos travaux, nous étudions l'effet de connexions/ déconnexions intempestives sur un système pair-à-pair existant : Pastry. Notre proposition pour améliorer la tolérance au *churn* est ensuite présentée, et son évaluation, à l'aide de simulations événementielles discrètes, est menée.

1 Introduction

Les tables de hachage distribuées (*Distributed Hash Table*, ou DHT) sont des systèmes de stockage répartis qui utilisent une infrastructure s'appuyant sur des protocoles de routage par clés (*Key-Based Routing*, ou KBR) [1, 2]. Les DHT fournissent au développeur un haut niveau d'abstraction pour l'implémentation de systèmes de stockage persistant à large échelle, masquant ainsi la complexité des protocoles de tolérance aux fautes, de la réplication et du routage réseau. C'est pourquoi les DHT sont de plus en plus utilisées pour des applications ayant un fort besoin de fiabilité, telles que les systèmes de sauvegarde [3], de gestion distribuée de fichiers [4, 5], ou les systèmes de distribution de données [6].

En pratique, la variation de la structure du réseau pair-à-pair due à l'arrivée et au départ non anticipé de pairs constitue une limite pour la performance et la disponibilité de systèmes à base de DHT. De telles variations, appelées *churn*, induisent au pire la perte de certains blocs de données, et au mieux une dégradation de la performance du système. Cette perte d'efficacité est due à la réorganisation de l'ensemble des réplicats de la donnée affectée. En effet, Rodrigues et Blake ont montré que l'utilisation de DHT classiques pour le stockage de larges quantités de données n'est pertinente que si le temps de session des pairs est de l'ordre de plusieurs jours [7]. Les travaux sur la résistance au *churn* réalisés jusqu'à présent traitaient pour la plupart du problème du routage entre pairs, c'est-à-dire du maintien de la cohérence du voisinage logique [8, 9]. La gestion pertinente de la migration de données dans la couche de stockage continue donc d'être un problème lorsqu'une reconfiguration des pairs doit avoir lieu.

Dans une DHT, chaque bloc de données se voit associer un nœud *racine*. L'identifiant de cette racine est celui qui est numériquement le plus proche de la clé de stockage de ce bloc dans l'espace d'adressage. En outre, chaque nœud de la DHT dispose d'un ensemble de voisins logiques appelé *leafset*. Le schéma de réplication traditionnel consiste à utiliser un sous-ensemble du leafset de la racine pour stocker les copies d'un bloc de données [1]. Ainsi, si un pair rejoint ou quitte le leafset,

la DHT fait respecter cette contrainte, ce qui a pour conséquence possible la migration de nombreux blocs de données. En effet, il a été demontré que le coût de ces migrations peut être élevé en terme de bande passante [3]. Une solution à ce problème repose sur la création de clés multiples pour un même bloc de données [10, 11] ; par conséquent, seul le pair responsable d'une clé peut être affecté par une reconfiguration. Cependant, un pair en charge d'un bloc de données doit de façon périodique vérifier l'état de tous les pairs en possession d'un réplicat. Étant donné que ces copies sont dispersées de manière aléatoire dans l'infrastructure logique, le nombre de pairs à tester peut être très grand.

Notre travail propose une variante de la stratégie de réplication à base de leafset pouvant tolérer un fort taux de *churn*. Notre objectif est d'éviter la migration de blocs de données tant qu'un nombre suffisant de réplicats est présent dans la DHT. Nous relâchons donc les contraintes de placement pour les copies de blocs, autorisant un pair à s'insérer dans le leafset sans forcément entraîner la migration de blocs. Pour localiser de manière fiable les copies des blocs dans un tel système, le pair racine d'un bloc maintient des métadonnées décrivant l'emplacement des réplicats. En pratique, la gestion des métadonnées est intégrée au protocole existant de maintenance du leafset et n'induit donc pas de réel surcoût.

Nous avons implementé PAST et notre stratégie de réplication au dessus de PeerSim [12]. Les principaux résultats de notre évaluation sont :

- Nous montrons que notre approche permet une meilleure disponibilité en présence de *churn* que la stratégie de réplication de PAST. Pour une fréquence d'une connexion/déconnexion par minute, notre stratégie permet de perdre deux fois moins de blocs que celle de PAST.
- Nous montrons que notre stratégie de réplication permet en moyenne de réduire de moitié les transferts de blocs par rapport à la stratégie native de PAST.

Le reste du papier s'organise comme suit. La section 2 présente une vue d'ensemble des principaux schémas de réplication et algorithmes de maintenance communément utilisés dans les systèmes pair-à-pair basés sur des DHT et met en évidence leurs limites. La section 3 introduit un schéma de réplication amélioré dans lequel les contraintes de placement sont relâchées afin d'obtenir une meilleure résistance au *churn*. Les résultats de simulation de cet algorithme sont présentés dans la section 4. La section 5 conclut avec une vue d'ensemble de nos résultats.

2 Contexte et motivation

Les systèmes pair-à-pair basés sur des DHT sont habituellement structurés en trois couches : 1) une couche de routage 2) la DHT en elle même, 3) l'application qui utilise la DHT. La couche de routage est communément appelée KBR (*Key-Based Routing*) car elle utilise des clés pour identifier les pairs. La couche KBR permet de masquer la complexité de l'infrastructure, du routage et des algorithmes de tolérance aux fautes pour les couches supérieures. Récemment, de nombreuses recherches ont été menées pour améliorer la résistance de la couche KBR à de forts taux de *churn* [8]. Les principaux exemples de couches KBR sont Pastry [13], Chord [2], Tapestry [14].

La couche DHT est résponsable du stockage des blocs de données. Elle implémente un service de stockage distribué, persistant et tolérant aux fautes, et peut être extensible à un grand nombre de pairs. Les DHT fournissent les primitives `get` et `put`, simplifiant grandement la conception d'applications à large échelle. PAST [1] et DHash [15] sont des DHT implémentées respectivement au dessus de Pastry [13] et Chord [2]. Enfin, la couche applicative représente toute application distribuée pouvant nécessiter l'utilisation d'une DHT. Quelques exemples représentatifs sont le système de fichiers distribué CFS [5] et le système de sauvegarde PeerStore [3].

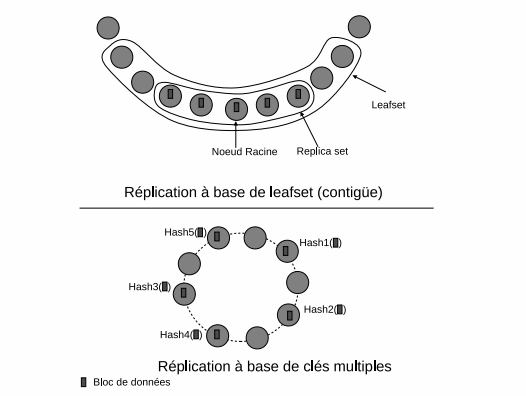

FIG. 1 – Réplication à base de leafset et à base de clés multiples ($\mathtt{k} = 5$).

2.1 La Réplication dans les DHT

Dans une DHT, un identifiant (i.e., une clé) est associé à chaque pair et à chaque bloc de données. La clé d'un bloc de données est généralement obtenue par l'application d'une fonction de hachage sur le contenu du bloc. Le pair dont l'identifiant est le plus proche de la clé du bloc est appelé *racine* du bloc. L'ensemble des identifiants forment une structure logique : un anneau dans les systèmes Chord [2] et Pastry [13] ou bien un tore d-dimensionnel dans CAN [10].

Un pair possède une connaissance locale du réseau pair-à-pair, i.e., son leafset, constitué d'une liste de ses voisins dans l'anneau. Par exemple, dans Pastry le leafset contient les adresses des L/2 voisins les plus proches dans le sens positif de l'anneau, et celles des L/2 voisins dans le sens négatif. Chaque pair entretient son leafset, supprimant les pairs déconnectés de l'infrastructure et ajoutant les nouveaux arrivants.

Afin de tolérer les défaillances, chaque bloc de données est repliqué sur `k` pairs qui composent alors le *replica-set* de ce bloc. Deux protocoles sont en charge de la gestion des réplicats : le protocole de placement initial et le protocole de maintenance.

Les protocoles de placement de réplicats

Il existe deux principales stratégies de placement des réplicats : à base de leafset, et à base de clés multiples :

- **Réplication à base de leafset.** La racine du bloc de données est responsable du stockage d'une copie du bloc. Le bloc est également répliqué sur un sous-ensemble du leafset de la racine formé par ses voisins les plus proches. Les voisins qui stockent une copie du bloc de données peuvent être soit successeurs de la racine dans l'anneau, soit prédessesseurs, soit les deux. Par conséquent, les differentes copies d'un bloc sont stockées de manière *contigüe* dans l'anneau, comme cela est illustré par la Figure 1. Cette stratégie a été implémentée dans PAST [1] et DHash [15]. La *réplication par successeur* est une variante de la stratégie précédente, dans laquelle les données sont répliquées sur les successeurs immédiats de la racine au lieu de l'être sur les pairs les plus proches [16].
- **Réplication à clés multiples.** Cette approche repose sur le calcul de `k` clés de stockage distinctes pour tout bloc de données, chaque clé correspondant à une racine différente du bloc. Les blocs de données sont ensuite répliqués sur ces `k` pairs racines. Cette solution a été implémentée dans CAN [10] et Tapestry [11]. GFS [17] utilise une variante basée sur le placement aléatoire pour améliorer la réparation de données. La *réplication symetrique* et par *chemins* sont des variantes de la réplication par clés multiples [18, 16].

Protocoles de maintenance

Les protocoles de maintenance se chargent de conserver `k` copies de chaque bloc de données tout en respectant la stratégie de placement initiale. Cela signifie que les `k` copies de chaque bloc de données doivent toujours être stockées sur les voisins attenant à sa racine dans le cas d'une stratégie de réplication à base de leafset, et sur les pairs racines pour le schéma de réplication par clés multiples.

Le mécanisme de maintenance à base de leafset s'appuie sur l'échange périodique d'information au sein des leafsets. Par exemple, dans l'algorithme de maintenance de PAST [1], chaque pair envoie un filtre de bloom[1] des blocs qu'il stocke dans son leafset. Lorsqu'un pair du leafset reçoit une telle requête, il utilise le filtre de bloom pour détérminer si il stocke un ou plusieurs blocs que l'envoyeur doit également stocker. Il répond ensuite avec la liste des clés de ces blocs. Le pair à l'origine de la requête peut ensuite récupérer les blocs manquants listés dans l'ensemble des réponses à ses requêtes.

Dans le cas des stratégies de réplication à base de clés multiples, la maintenance doit être effectuée séparément pour chaque bloc de données. Pour chaque bloc stocké dans le système, il est nécessaire de vérifier périodiquement si ses différents pairs racines sont toujours opérationnels et stockent encore une copie du bloc de données.

2.2 Impact du *churn* sur les performances des DHT

Un taux de *churn* élevé induit beaucoup de changements dans le réseau pair-à-pair, et le protocole de maintenance doit fréquemment migrer des blocs de données. Alors que certaines migrations sont obligatoires pour restaurer `k` copies, d'autres ne servent qu'a conserver les invariants de placement.

On peut considérer un premier cas de figure dans lequel un pair se connecte au système avec un identifiant suffisamment proche de celui d'un bloc de données pour en devenir la nouvelle racine. Dans cette situation, le bloc de données sera migré sur le pair arrivant. Une deuxième illustration de ce phénomène se produit dans la réplication à base de leafset lorsqu'un pair posséde un identifiant se situant à l'intérieur du *replica-set* d'un certain nombre de blocs de données. Par conséquent, ces blocs doivent être déplacés par la DHT pour assurer la conservation de la propriété de contiguïté du *réplica-set*. Il est nécessaire de remarquer que l'extension de la taille du *replica-set* augmente la probabilité pour un nouvel arrivant de provoquer des migrations de blocs. Kim et Park tentent de limiter ce problème en autorisant l'espacement des blocs de données dans les leafsets [19]. Cependant, ils doivent alors maintenir une connaissance de la totalité du leafset : chaque pair doit connaître le contenu de tous les pairs dans son leafset. Malheureusement, l'algorithme de maintenance n'est pas décrit en détail et son coût réel est inconnu.

Dans le cas de la stratégie à base de clés multiples, un nouvel arrivant peut être inséré entre deux réplicats sans que des blocs de données ne soient à migrer, à condition de ne pas devenir nouvelle racine d'un bloc. Cependant, il existe un incovénient : la maintenance doit être effectuée separément pour chaque bloc ; elle ne passe par conséquent pas à bien à l'échelle en terme de blocs par pair. Cela représente une serieuse limitation pour des systèmes de gestion et de sauvegarde de fichiers devant stocker jusqu'à plusieurs milliers de blocs de données par pair.

3 Relâcher les contraintes de placement de la DHT pour tolérer le *churn*

Le but de notre recherche est de concevoir une DHT pouvant tolérer un fort taux de *churn* sans déterioration de ses performances. Pour cela, nous évitons de copier des blocs de données lorsque

[1] Pour résumer, le filtre de bloom envoyé contient une vue compacte mais approximative de la liste des blocs stockés par un pair.

ce n'est pas nécessaire à la restauration d'un réplicat manquant. Nous introduisons une réplication à base de leafset qui relâche les contraintes de placement dans le leafset. Notre solution, appelée RelaxDHT, est présentée dans la suite.

3.1 Présentation de RelaxDHT

RelaxDHT est construite au dessus d'une couche KBR telle que Pastry ou Chord. Nous proposons d'utiliser des métadonnées pour la localisation des réplicats. Nous gardons la notion de pair racine pour chaque bloc de données. Cependant, un pair ne stocke pas forcément une copie des blocs dont il est racine. Au lieu de cela, il maintient seulement un ensemble de métadonnées décrivant son *replica-set*. Tous les pairs qui y sont contenus reçoivent alors périodiquement un message de la part de la racine, leur indiquant qu'il doivent continuer à stocker les réplicats associés. L'utilisation de métadonnées de localisation permet une totale liberté de placement pour un bloc de données au sein du leafset. Un pair arrivant peut rejoindre un leafset sans nécessairement déclencher la migration de blocs de données.

Deux raisons nous poussent à restreindre le placement des réplicats au leafset de la racine. Premièrement, par souci de passage à l'échelle, le nombre messages de notre protocole ne dépend alors pas du nombre de blocs de données gérés par un pair, mais seulement de la taille du leafset. Deuxièmement, étant donné que la couche de routage induit beaucoup d'échanges dans les leafsets, on considère que la vue locale de ces derniers au niveau DHT peut être utilisée comme détécteur de fautes.

Insertion d'un nouveau bloc de données

Pour être stocké dans le système, un bloc de données est inséré à l'aide de la primitive `put(k,b)`. Cette opération produit un message d'insertion envoyé au pair racine. Ensuite, la racine choisit aléatoirement un *replica-set* de `k` pairs autour du centre du leafset. Le placement autour du centre réduit la probabilité que le pair choisi sorte rapidement du leafset à cause de l'arrivée de nouveaux pairs. Enfin, la racine envoie aux pairs du *replica-set* un message "STORE" contenant :

1. le bloc de données lui-même,
2. l'identité des pairs du *replica-set* (i.e., les métadonnées),
3. l'identité de la racine.

Rappelons qu'un pair peut à la fois être la racine de plusieurs blocs de données et faire partie du *replica-set* d'autres blocs[2]. Il stocke donc :

1. une liste `rootOfList` d'identifiants de blocs de données avec la liste associée de pairs du *replica-set* pour tous les blocs dont il est racine.
2. une liste `replicaOfList` de blocs de données dans le *replica-set* desquels il est inclus. En plus de cela, cette liste contient également : l'identifiant du bloc de données, la liste des pairs du *replica-set* associé et l'identité du pair racine.

Un *compteur de bail* est associé à chaque bloc de données stocké. Ce compteur, initialisé à une valeur `L`, est décrémenté à chaque maintenance de la couche KBR. Le protocole de maintenance décrit ci-dessous est responsable de la réinitialisation périodique de ce compteur à `L`.

Protocole de Maintenance

Le but de ce protocole périodique est d'assurer que :

[2]Il est tout à fait possible, quoiqu'absolument non obligatoire, qu'un pair soit racine d'un bloc de données et fasse en même temps partie du *replica-set* associé.

– Un pair racine existe pour chaque bloc de données. Il s'agit du pair dont l'identifiant est le plus proche de celui du bloc dans l'espace de nommage.
– Chaque bloc de données est répliqué sur `k` pairs situés dans le leafset de la racine du bloc.

A chaque période T, un pair `p` exécute l'algorithme 1, de manière à envoyer les messages de maintenance à tous les pairs de son leafset. Il est important de remarquer que l'algorithme 1 utilise sa connaissance du leafset acquise via la couche KBR. Cette connaissance est relativement fine car la période de maintenance de la couche KBR est nettement plus petite que celle de la couche DHT.

Algorithm 1: Construction d'un message de maintenance de RelaxDHT.

```
   Result: msgs, les messages construits.
 1 for d ∈ rootOfList do
 2 |   for replica ∈ d.replicaSet do
 3 |   |   if NOT estAuCentre (replica,leafset) then
 4 |   |   |   newPeer =choisirPair (replica,leafset);
 5 |   |   |   remplacer (d.replicaSet, replica,newPeer);
   |   |   
 6 |   for replica ∈ d.replicaSet do
 7 |   |   ajouter(msgs [replica ],<STORE, d.blockID, d.replicaSet >);
   |   
 8 for d in replicaOfList do
 9 |   if NOT vérifierRacine (d.rootPeer,leafset) then
10 |   |   newRoot =getRacine (d.blockID,leafset);
11 |   |   ajouter (msgs [newRoot ],<NEW ROOT, d.blockID, d.replicaSet >) :
   |   
12 for p ∈ leafset do
13 |   if NOT vide (msgs [p ]) then
14 |   |   envoyer(msgs [p ],p);
   |   
```

Les messages construits par l'algorithme 1 contiennent deux types d'éléments distincts :
– Un élément **STORE** lorsqu'il s'agit de demander à un nœud de continuer à stocker le réplicat d'un bloc.
– Un élément **NEW ROOT** pour notifier un nœud qu'il est devenu la nouvelle racine d'un bloc de données.

Quelque soit son type, un élément contient un identifiant de bloc de données et la liste des pairs du *replica-set* associé. L'algorithme 1 envoie un unique message à chaque membre de son leafset.

L'algorithme 1 se décompose en trois phases : la première calcule les éléments STORE en utilisant la structure `rootOfList` -lignes 1 à 7-, la seconde calcule les éléments NEW ROOT en utilisant la structure `replicaOfList` -lignes 8 à 11- et la dernière distribue dans le leafset les messages aux pairs destinataires -à partir de la ligne 12 et jusqu'à la fin-. Les éléments calculés lors des deux premières phases sont ajoutés dans *msgs[]*. *msgs[q]* est un message contenant tous les éléments à envoyer à un nœud q lors de la dernière phase. Par conséquent, dans le pire cas, chaque pair envoie périodiquement `#leafset` messages de maintenance à ses voisins.

Dans la première phase, pour chaque bloc dont le pair est racine, ce dernier vérifie que tous les réplicats se situent encore dans le centre de son leafset[3] (ligne 3) en utilisant la vue locale de son leafset fournie par la couche KBR. Si un nœud réplicat se situe en dehors, le pair racine le remplace avec un pair aléatoirement choisi au centre du leafset, puis met à jour le *replica-set* du bloc (lignes 4 et 5). Enfin, le pair racine ajoute un élément STORE dans chaque message destiné aux pairs du *replica-set* (lignes 6 et 7).

Dans la seconde phase, pour chaque bloc stocké par le pair (i.e., le pair fait partie du *replica-set* du bloc), celui-ci vérifie si le nœud racine n'a pas changé. Cette vérification est faite en comparant les métadonnées `replicaOfList` avec l'état courant du leafset (ligne 9). Si la racine a changé, le

[3]Soit Un nœud `n` et `c` $= \frac{2}{3} \times \#leafset_n$. On considère comme centre du leafset de `n` les `c` nœuds dans le sens positif, et les `c` nœuds dans le sens négatif.

pair ajoute un élément NEW ROOT au message pour prévenir la future racine de son nouveau rôle vis à vis du bloc de données.

Pour finir, de la ligne 12 à la ligne 14, une boucle est chargée de l'envoi des messages obtenus à tous les membres du leafset.

Traitement des messages de maintenance

Algorithm 2: Recéption d'un message de maintenance RelaxDHT.

```
   Data: message, le message reçu.
 1 for elt ∈ message do
 2     switch elt.type do
 3         case STORE
 4             if elt.donnée ∈ replicaOfList then
 5                 nouveauBail(replicaOfList,elt.donnée);
 6                 majReplicaSet(replicaOfList,elt.donnée);
 7             else
 8                 demanderBloc(elt.donnée);
 9         case NEW ROOT
10             rootOfList = rootOfList ∪ elt.donnée ;
```

- **Pour un élément STORE** (ligne 3), si le pair stocke déjà une copie du bloc de données correspondant, il réinitialise le compteur de bail associé et met à jour le *replica-set* correspondant si nécessaire (lignes 4, 5 et 6). Si le pair ne stocke pas le bloc de données associé (i.e., c'est le premier élément STORE pour ce bloc de données reçu par ce pair), il récupère ce bloc depuis l'un des pairs mentionnés dans le *replica-set* reçu (ligne 8).
- **Pour un élément NEW ROOT** un pair ajoute l'identifiant du bloc de données et le *replica-set* correspondant dans la structure `rootOfList` (ligne 10).

3.2 Limitations et effets de bord

En relâchant les contraintes de placement des copies de blocs de données dans le leafset, notre stratégie de réplication pour les DHT pair-à-pair réduit significativement[4] le nombre de transferts de blocs de données lorsque des pairs rejoignent ou quittent le système. Nous montrons dans la section suivante que grâce à cela, notre mécanisme de maintenance nous permet de mieux tolérer le *churn*. Il introduit cependant quelques effets de bord.

Ainsi, la distribution des blocs de données qui en résulte est biaisée. En présence de *churn*, les nouveaux pairs stockeront probablement bien moins de blocs de données que les pairs faisant partie de la DHT depuis un plus long moment car notre mécanisme de maintenance n'effectue de transferts que si cela est nécessaire. Il est important de remarquer que cet effet secondaire est plutôt positif : plus un pair est stable, plus il stockera de blocs. En outre, il est possible de facilement contrer cet effet en prenant en compte, lors du choix des nouveaux pairs du *replica-set*, la quantité de blocs de données stockés.

Par ailleurs, la performance des requêtes de recherche peut être dégradée par les changements dans la distribution des blocs de données. Nous avons concentré nos efforts sur la perte de données. Cependant, avec RelaxDHT, il est possible que certaines racines de blocs soient temporairement incohérentes, induisant un surcoût de communication lors de la recherche de la donnée. Par exemple, lorsqu'un pair racine d'au moins un bloc tombe en panne, les copies du bloc correspondant sont toujours dans le système, mais une requête de recherche standard peut ne pas les trouver : le nouveau pair dont l'identifiant est le plus proche peut ne pas connaître le bloc. Cela reste vrai jusqu'à ce

[4] cf. Fig.3

que la panne soit détéctée par un des pairs du *replica-set* et réparée en utilisant un message "NEW ROOT" (c.f., les algorithmes ci-dessus). Il serait possible d'inonder le leafset ou bien effectuer un *"limited range walk"* lorsqu'une requête de recherche échoue, pour lui permettre de retrouver des blocs de données même en absence de racine à jour, mais cette solution peut ralentir les recherches et engendrer un surcoût en terme de communications. Cependant, il faut remarquer que 1) les requêtes de recherches qui ont lieu entre une panne du pair racine et sa réparation sont rares, 2) ceci pourrait être effectué en association avec le protocole de maintenance du leafset (qui utilise déjà l'inondation pour entretenir le leafset).

4 Évaluation

Afin d'évaluer RelaxDHT, nous avons réalisé un simulateur à événements discrets en utilisant comme base le simulateur PeerSim [12]. Plus précisement, nous avons basé notre simulateur sur un module existant de PeerSim simulant la couche KBR de Pastry. Nous avons implémenté la stratégie de PAST et celle de RelaxDHT au dessus de ce module. Il est important de remarquer que les differentes couches et tous les échanges de messages sont simulés. Notre simulateur prend également en compte la congestion réseau : dans notre cas, les liens du réseau peuvent souvent être congestionnés.

Paramètres de la simulation

Pour tous les résultats de simulation présentés dans la section, nous avons utilisé un réseau de 100 pairs avec les paramètres suivants (à la fois pour PAST et RelaxDHT) :

- une taille de leafset de 24 ;
- une période de maintenance de 10 minutes au niveau DHT ;
- une période de maintenance de 1 minute au niveau couche KBR ;
- 10 000 blocs de données de 10 000 KB repliqués 3 fois ;
- des liens réseau de 1 Mbits/s en flux montant et 10 Mbits/s en flux descendant avec un délai uniformément choisi entre 80 et 120 ms.

Un réseau de 100 pairs peut sembler relativement petit. Cependant, pour les deux stratégies de réplication, PAST et RelaxDHT, le comportement étudié est local, contenu à l'intérieur d'un leafset (dont la taille est bornée). Il est cependant nécessaire de simuler un anneau entier pour prendre en compte les effets de bords induits par les leafsets des voisins. De plus, un compromis doit être fait entre la précision du système et sa taille. Avec RelaxDHT, il est important de simuler très précisement toutes les communications entre pairs. Nous avons réalisé plusieurs simulations à une plus large échelle (1000 pairs et 100 000 blocs de données) et avons observé des phénomènes similaires.

Nous avons injecté du *churn* selon deux scénarios differents :

- **Période de churn d'une heure**. Cette phase est suivie par une autre sans connexions/ déconnexions. Pour l'étude de ce cas, pendant la phase de *churn* à chaque *période de perturbation*, nous choisissons aléatoirement soit une connexion d'un nouveau pair, soit une déconnexion. Cette perturbation peut survenir n'importe où dans l'anneau (aléatoirement généré). Nous avons effectué de nombreuses simulations en variant la période de perturbation.
- **Churn continu**. Pour cet ensemble de simulations, nous nous focalisons sur la première phase du cas précédent. Nous étudions la réaction du système aux changement du délai fixé entre deux perturbations. Dans ce cas, une "perturbation" signifie une connexion de nouveau pair, ou bien une déconnexion.

Le premier ensemble d'expériences nous permet d'étudier 1) combien de blocs de données sont perdus après une période de perturbation et 2) quel délai est nécessaire au système pour retourner

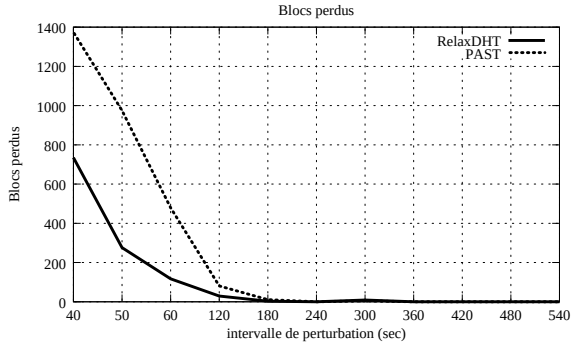

FIG. 2 – Nombre de blocs de données perdus (ie. toutes les copies ont été perdues).

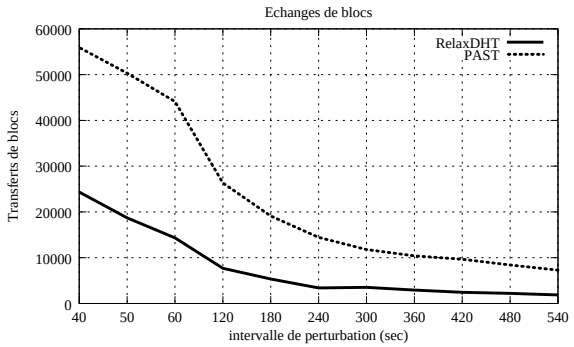

FIG. 3 – Nombre de blocs de données échangés pour rétablir un état stable.

dans un état pour lequel tous les blocs de données restants/non perdus sont de nouveau répliqués `k` fois. Lors de l'exploitation réelle de ces systèmes, certaines périodes seront sans *churn*, et les protocoles doivent en tirer avantage pour converger vers un état plus stable.

Le second ensemble d'expériences se concentre sur la période de perturbation, ce qui permet d'étudier la capacité du système à récupérer en présence de *churn* les réplicats perdus.

4.1 Pertes et temps de stabilisation après une heure de *churn*

Nous étudions tout d'abord le nombre de blocs de données perdus (blocs de données pour lesquels les 3 copies sont perdues) dans PAST et dans RelaxDHT avec les mêmes conditions de *churn*. La Figure 2 montre le nombre de blocs de données perdus après une période d'une heure de *churn*. Le délai entre deux perturbations est representé sur l'axe des abscisses. Le nombre de blocs de données perdus avec RelaxDHT est nettement plus faible qu'avec PAST : pour des périodes de perturbation inférieures à 50 *secondes*, le gain est de 50%.

La raison principale en est que les pairs utilisant la stratégie de réplication de PAST doivent télécharger plus de blocs de données. Ceci implique que le temps de téléchargement moyen pour un bloc de données est plus élevé avec la stratégie de réplication de PAST. En effet, la maintenance des contraintes de placement du schéma de réplication génère un traffic réseau continu qui ralentit le traffic critique dont le but est d'effectivement rétablir les copies perdues.

La Figure 3 montre le nombre total de blocs échangés dans les deux cas. Encore une fois, l'axe des abscisses représente le délai entre deux perturbations. La figure montre que le nombre de blocs échangés avec RelaxDHT est toujours pratiquement 2 fois plus faible que dans PAST. Ceci est principalement dû au fait que dans le cas de PAST, beaucoup de transferts (quasiment la moitié) sont faits uniquement dans le but de préserver les contraintes du schéma de réplication.Ce phenomène est particulièrement flagrant lors de la *connexion* d'un nouveau pair. Ainsi, à chaque fois qu'un nouveau pair rejoint la DHT, il devient racine de certains blocs de données (car son identifiant est plus proche que celui de la racine courante de l'identifiant du bloc), où bien s'il est inséré à l'intérieur de *replica-sets* dont les membres doivent rester contigüs.

Utilisant la stratégie de réplication de PAST, un pair nouvellement inséré peut avoir besoin de télécharger des blocs de données pendant de nombreuses heures, et ce même si aucune panne/ déconnexion ne s'est produite. Pendant tout ce temps, ses voisins doivent lui envoyer les blocs de données demandés, utilisant ainsi une grande partie de leur bande passante en sortie.

Dans RelaxDHT, lorsque de nouveaux pairs *rejoignent* le système, *aucun* ou *peu* de transferts de blocs de données sont réellement nécessaires. Les transferts sont inévitables uniquement si certaines copies se retrouvent trop éloignées de leur pair racine dans l'anneau logique. Dans ce cas, ils doivent être transférés plus près de la racine avant que leur pair hôte quitte le leafset du pair racine. Avec un degré de réplication de 3 et une taille de leafset de 24, de nombreux pairs peuvent rejoindre le

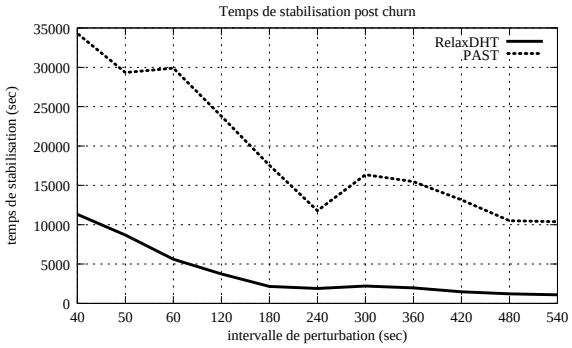

FIG. 4 – Temps de récupération : temps nécessaire pour récupérer toutes les copies de chaque bloc de données restant.

leafset avant qu'un transfert de blocs de données ne soit requis.

Enfin, nous avons mesuré le temps nécessaire au système pour retrouver un état normal, dans lequel chaque bloc de données restant[5] est repliqué `k` fois. La Figure 4 montre les résultats obtenus en faisant varier le délai entre les perturbations. On peut voir que le temps de récupération est deux fois plus long comparé à RelaxDHT lorsque PAST est utilisé. Ce résultat s'explique principalement par le faible nombre de blocs à transférer dans le cas de RelaxDHT : comparé à PAST, notre protocole de maintenance ne transfère que très peu de blocs pour faire respecter des contraintes de localisation.

Ce dernier résultat montre que la DHT utilisant RelaxDHT répare plus vite les blocs de données endommagés (blocs pour lesquels des copies ont été perdues) que PAST. Ceci implique que la récupération est très rapide, permettant au système de bien mieux gérer la phase de *churn* suivante. La prochaine section décrit nos simulations avec *churn* continu.

4.2 Churn continu

Avant de présenter les résultats de simulations de *churn* continu, il est important de mesurer l'impact d'une unique panne/déconnexion de pair.

Lorsqu'un pair tombe en panne de manière isolée, les blocs de données qu'il stockait doivent être répliqués sur un autre pair. Ces blocs sont alors transférés vers ce pair afin de rétablir le degré de réplication original `k`. Dans nos simulations, avec les paramètres donnés ci-dessus, il faut en moyenne 4609 secondes à PAST pour se remettre de la panne : i.e., créer un nouveau réplicat pour chaque bloc de données stocké sur le pair déficient. La même procédure ne prend que 1889 secondes avec RelaxDHT. Le nombre de pairs impliqués dans la réparation est en effet beaucoup plus important. Ce gain est dû à la parallélisation des transferts de blocs de données :

- dans PAST, le contenu de pairs contigüs est corrélé. Avec un degré de réplication de 3, seuls les pairs situés à un ou deux hops du pair en erreur peuvent être utilisés comme sources ou comme destination pour les transferts de données. En fait, seulement `k+1` (avec `k` le facteur de réplication) pairs sont impliqués dans la récupération d'un pair en erreur.
- dans RelaxDHT, la plupart des pairs contenus dans le leafset du pair en erreur peuvent participer aux transferts (le leafset contient 24 pairs dans nos simulations).

Les résultats de la simulation montrent que RelaxDHT : 1) induit moins de transferts de données et 2) les transferts de données restants sont mieux parallélisés. Grâce à ces deux points, RelaxDHT fournit une meilleure tolérance au *churn* et ce même avec un *churn* continu.

Ces résultats sont illustrés dans la Figure 5. On observe qu'avec les paramètres décrits au début de la section, PAST commence à perdre des blocs de données lorsque le délai entre les perturbations

[5]Les blocs pour lesquels toutes les copies ont été perdues ne retrouveront jamais un état normal et ne sont donc pas pris en compte.

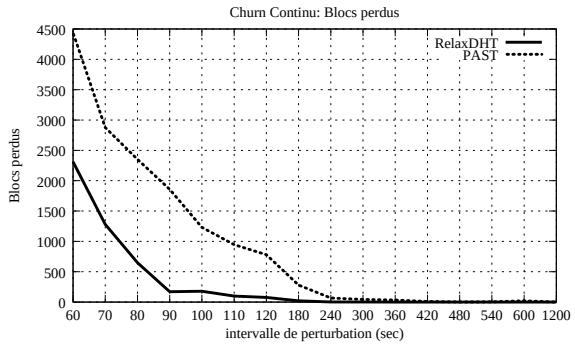

FIG. 5 – Nombre de pertes de blocs de données (les `k` copies sont perdues) avec *churn* continu en fonction de la période de perturbation.

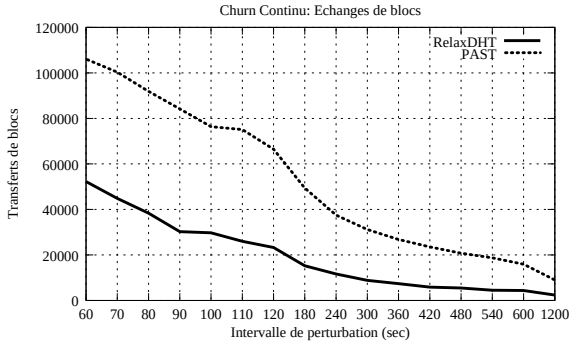

FIG. 6 – Nombre de transferts de blocs de données nécessaires lorsque le système se trouve sous *churn* continu en fonction du délai entre perturbations.

est proche de 7 minutes. Ce délai doit être inférieur à 4 minutes pour que des blocs soient perdus avec RelaxDHT. Lorsqu'on diminue encore la période des perturbations, le nombre de blocs de données perdus en utilisant RelaxDHT reste à peu près deux fois plus faible qu'en utilisant la stratégie de PAST.

Enfin, la Figure 6 confirme que même pour un schéma de *churn* continu, avec 5 heures de simulation, le nombre de transferts de données requis par la stratégie de RelaxDHT est bien plus petit (environ de moitié) que le nombre de transferts induits par la stratégie de réplication de PAST.

5 Conclusion

Les tables de hachage distribuées pair-à-pair fournissent un système de stockage passant à l'échelle, efficace et simple à utiliser. Cependant, les solutions existantes tolérent mal un fort taux de *churn* ou ne passent pas vraiment à l'échelle en terme de nombre de blocs à stocker. Nous avons identifié les raisons pour lesquelles ils tolérent mal un taux de *churn* élevé : elles imposent des contrantes de placement strictes, ce qui entraine des transferts de données non nécessaires.

Dans ce papier, nous proposons une nouvelle stratégie de réplication, RelaxDHT, qui relâche les contraintes de placement : elle repose sur l'utilisation de métadonnées (identifiants de données/ pairs de réplication) permettant un placement plus flexible des blocs de données dans les leafsets. Grâce à ce design, RelaxDHT génère moins de transferts de données que les mécanismes classiques de réplication à base de leafsets. De plus, comme les copies des blocs de données sont dispersées parmi un ensemble plus grand de pairs, le contenu des pairs est moins correlé. Cela signifie qu'en cas de panne, un nombre plus important de sources est disponible pour la récupération, rendant la procédure plus efficace et donc le système plus résistant au *churn*. Nos simulations, comparant la stratégie de réplication de PAST au notre, confirment que RelaxDHT 1) induit moins de transferts de blocs de données, 2) récupère plus vite les copies perdues et 3) perd moins de blocs de données.

Références

[1] A. I. T. Rowstron and P. 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*, December 2001, pp. 188–201.

[2] I. Stoica, R. Morris, D. Liben-Nowell, D. R. Karger, F. F. Kaashoek, F. Dabek, and H. Balakrishnan, "Chord : a scalable peer-to-peer lookup protocol for internet applications," *IEEE/ACM Trans. Netw.*, vol. 11, no. 1, pp. 17–32, February 2003.

[3] M. Landers, H. Zhang, and K.-L. 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*. Washington, DC, USA : IEEE Computer Society, August 2004, pp. 72–79.

[4] J.-M. Busca, F. Picconi, and P. Sens, "Pastis : A highly-scalable multi-user peer-to-peer file system," in *Euro-Par '05 : Proceedings of European Conference on Parallel Computing*, August 2005, pp. 1173–1182.

[5] F. Dabek, F. M. Kaashoek, D. Karger, R. Morris, and I. Stoica, "Wide-area cooperative storage with CFS," in *SOSP '01 : Proceedings of the 8th ACM symposium on Operating Systems Principles*, vol. 35, no. 5. New York, NY, USA : ACM Press, December 2001, pp. 202–215.

[6] J. Jernberg, V. Vlassov, A. Ghodsi, and S. Haridi, "Doh : A content delivery peer-to-peer network," in *Euro-Par '06 : Proceedings of European Conference on Parallel Computing*, Dresden, Germany, September 2006, p. 13.

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

[8] S. Rhea, D. Geels, T. Roscoe, and J. Kubiatowicz, "Handling churn in a DHT," in *Proceedings of the 2004 USENIX Technical Conference, Boston, MA, USA*, June 2004.

[9] M. Castro, M. Costa, and A. Rowstron, "Performance and dependability of structured peer-to-peer overlays," in *DSN '04 : Proceedings of the 2004 International Conference on Dependable Systems and Networks*. Washington, DC, USA : IEEE Computer Society, June 2004, p. 9.

[10] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Schenker, "A scalable content-addressable network," in *SIGCOMM*, vol. 31, no. 4. ACM Press, October 2001, pp. 161–172.

[11] B. Y. Zhao, L. Huang, J. Stribling, S. C. Rhea, A. D. Joseph, and J. D. Kubiatowicz, "Tapestry : A global-scale overlay for rapid service deployment," *IEEE Journal on Selected Areas in Communications*, 2003.

[12] M. Jelasity, A. Montresor, G. P. Jesi, and S. Voulgaris, "The Peersim simulator," `http ://peersim.sf.net`.

[13] A. Rowstron and P. Druschel, "Pastry : Scalable, decentralized object location, and routing for large-scale peer-to-peer systems," *Lecture Notes in Computer Science*, vol. 2218, pp. 329–350, 2001.

[14] B. Y. Zhao, L. Huang, J. Stribling, S. C. Rhea, A. D. Joseph, and J. D. Kubiatowicz, "Tapestry : A resilient global-scale overlay for service deployment," *IEEE Journal on Selected Areas in Communications*, vol. 22, pp. 41–53, 2004.

[15] F. Dabek, J. Li, E. Sit, J. Robertson, F. F. Kaashoek, and R. 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.

[16] S. Ktari, M. Zoubert, A. Hecker, and H. Labiod, "Performance evaluation of replication strategies in DHTs under churn," in *MUM '07 : Proceedings of the 6th international conference on Mobile and ubiquitous multimedia*. New York, NY, USA : ACM Press, December 2007, pp. 90–97.

[17] S. Ghemawat, H. Gobioff, and S.-T. Leung, "The google file system," in *SOSP '03 : Proceedings of the 9th ACM symposium on Operating systems principles*. New York, NY, USA : ACM Press, October 2003, pp. 29–43.

[18] A. Ghodsi, L. O. Alima, and S. 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*, Trondheim, Norway, August 2005, p. 12.

[19] K. Kim and D. Park, "Reducing data replication overhead in DHT based peer-to-peer system," in *HPCC '06 : Proceedings of the 2nd International Conference on High Performance Computing and Communications*, Munich, Germany, September 2006, pp. 915–924.