Stockage de données sur systèmes de Distributed Hash Table

Pierre Buyle

Facultés Universitaires Notre Dame de la Paix,
Namur, Belgique
pbuyle@info.fundp.ac.be

Résumé En un peu plus de deux ans les systèmes Peer-to-Peer se sont imposés dans la communauté Internet, principalement via des applications de partage de fichiers telles que Napster,Gnutella ou eDonkey. Avec le temps, ces applications dominantes ont permis de mettre en evidence certaines lacunes des modèles de Peer-to-Peer sur lesquels elles reposent. Pour tenter de résoudre les problèmes de cette première génération de nouveaux modèles de Peer-to-Peer ont été mis au point. Parmi ceux-çi on retrouve principalement les systèmes dit de "Distributed Hash Table". Ces systèmes permettent, à partir de fonctions simples de mettre au point des applications complexes. Pour montrer leur potentiel, cet article présente des applications de stockage de données sur de tels systèmes.

1 Introduction

En un peu plus de deux ans les systèmes Peer-to-Peer se sont imposés dans la communauté Internet, principalement via des applications de partage de fichiers tels que Napster, Gnutella ou eDonkey. Mais le partage de fichiers n'est pas le seul service qui peut être réalisé en Peer-to-Peer. Il est possible de trouver bien d'autres applications basées sur les principes du Peer-to-Peer qui fournissent d'autres types de services.

Avec le temps, ces applications dominantes ont permis de mettre en evidence certaines lacunes des modèles de Peer-to-Peer sur lesquels elles reposent. Elles ont en effet tendance à voir leur performance diminuer avec leur popularité, au delà d'un certain nombre d'utilisateurs elles ne semblent pas résister à la charge qu'ils leur imposent. Pour tenter de pallier à ces problèmes, de nouveaux modèles de Peer-to-Peer ont été mis au point. Parmi ceux-ci on retrouve principalement les systèmes dit de "Distributed Hash Table" (DHT).

Des applications plus complètes que le simple partage de fichiers ont pu être proposées et spécifiées pour ces systèmes. Elles démontrent que le concept du Peer-to-Peer offre d'autres possibilités que le partage de fichiers, en le généralisant afin d'offrir le partage de différents types de services plus ou moins complexes. Parmi ces services on retrouve des services de stockage de données distribués qui permettent de stocker un grande quantité de données distribuée entre les participants d'un réseau Peer-to-Peer.

Pour soutenir ce propos, après la présentation des concepts de base des systèmes Peer-to-Peer et des systèmes de “Distributed Hash Table”, cet article présente et explique dans les grandes lignes cinq cas d'utilisation de ces systèmes pour la réalisation de systèmes de stockage de données distribués. Il existe, probablement, d'autres systèmes de ce type assurant des fonctionnalités similaires. Le choix de ces cinq systèmes plutôt que d'autres est fondé sur le seul critère d'utilisation de “Distributed Hash Table” et sur l'accès aisé aux articles, ou autres documents, les présentant.

2 Les systèmes Peer-to-Peer

2.1 Généralités

Les systèmes peer-to-peer sont aujourd'hui largement connus et diffusés parmi les utilisateurs de l'Internet. En effet, depuis près de deux ans de multiples applications reposant sur ce principe leur ont été présentées. Parmis ces applications les plus connues sont Gnutella[6], Napster[5], eDonkey[10] et Kazaa[7]. Il reste cependant important de définir un minimum le concept.

Un système peer-to-peer est un système distribué dans lequel les participants reposent l'un sur l'autre pour les services plutôt que de reposer sur une infrastructure dédiée, le plus souvent centralisée. L'idée de base du Peer-to-Peer est que l'application distribuée n'est plus conçue avec une séparation stricte entre ses clients, consommateurs de services, et ses serveurs, fournisseurs de services. A la place, un système Peer-to-Peer est composé de *Peers* égaux en fonctionnalités, qui décident séparement de se joindre au système et d'y proposer un ou plusieurs services . Un système Peer-to-Peer est ainsi un réseau dynamique de *Peers* interconnectés, consommateurs et fournisseurs de services.

Les premiers systèmes Peer-to-Peer tels que Napster ou Gnutella ont rapidement montré leurs limites avec l'accroissement de leur popularité. Soit parce que, comme Napster, ils reposent encore partiellement sur un serveur centralisé pour une partie de leurs services, conservant ainsi un point de 'failure' unique. Soit parce que, comme pour Gnutella, leur modèle de distribution ne peut supporter le nombre toujours croissant de nouveaux *Peers*, accusant en résultat une baisse dans leurs performances. Même si aujourd'hui d'autres applications, telles que Kazaa ou eDonkey, prétendent prendre la relève de ces pionniers, il est souvent admis qu'elles conservent leurs faiblesses, poussant à leurs limites les modèles qu'elles utilisent et compromettant ainsi grandement leur évolutivité. C'est dans une optique d'amélioration de ces modèles que sont proposés les systèmes Peer-to-Peer structurés dit “de seconde génération”, aussi souvent appelés “Distributed Hash Table”.

2.2 Distributed Hash Table

Les systèmes de Distributed Hash Table (DHT) tentent de résoudre le principal problème rencontré par la génération précédente de système Peer-to-Peer, le support d'un nombre important de participants. Alors que les anciens systèmes utilisent souvent un système de routage simpliste utilisant un “broadcast” massif des messages, avec d'éventuels raffinements telle que la découpe du réseau en sous réseaux avec concentrateurs, les systèmes DHT tentent de minimiser le nombre de messages transmis en cherchant à localiser rapidement le ou les destinataires d'un message. Il s'agit donc de systèmes de localisation de *Peers* et de routage de messages complets.

Dans un système DHT, les mécanismes de routage et de localisation sont melés, localiser un objet dans le système revient à y router un message, ou plutôt à transmettre ce message au *Peer* responsable de cet objet. Ce système de routage/localisation repose sur les principes suivants :

Identification : dans un système DHT tout est identifié dans un seul *espace de noms*. Les *Peers* formant le réseau et les objets qui s'y trouvent se voient chacun attribuer un identifiant unique issu de *espace de noms*. C'est cet identifiant qui est utilisé lors du routage d'un message à destination d'un *Peer* ou d'un objet.

Responsabilité des *Peers* : chaque *Peer* du réseau se voit confié, via un processus distribué, la responsabilité d'un sous-ensemble d'identifiants de l'*espace de noms* global. Un *Peer* est ainsi responsable des objets dont les identifiants appartiennent au sous-ensemble dont il a la responsabilité.

Routage de messages : pour transmettre un message à destination d'un identifiant, les *Peers* du réseau se le transmettent de proche en proche. Un *Peer* qui recoit un message le retourne aux N *Peers* qu'il connait et qui sont plus proche, dans l'*espace de noms*, de la destination (avec N un paramètre du système). Pour assurer le routage des messages, chaque *Peer* maintient des connaissances sur le ou les moyens de transmission physique qu'il possède pour communiquer avec d'autres *Peers*. L'apprentissage et le maintien de ces connaissances par un *Peer* sont les résultats d'un processus défini qui distribue l'information globale entres les *Peers* afin d'assurer. un routage efficace.

On appelle de tels systèmes des DHT car il est simple de les utiliser pour offrir des méthodes proches de celles associées aux structures de données de type table de hachage. On utilise donc le plus souvent ces systèmes pour publier des *objets* et les récupérer grâce un clef d'accès arbitraire, leur identifiant. On utilise alors deux primitives de base qui sont l'insertion d'un objet identifié par une clef et la récupération d'un objet via sa clef identifiante. On peut également avoir une méthode de suppression d'un objet grâce à sa clef identifiante.

Les divers systèmes de DHT existants, donnés ci-dessous, différent principalement par la nature et la taille de l'espace de nomage utilisé et par la métrique de distance utilisée lors du routage des messages. Certains de ces systèmes définissent également les mécanismes qui permettent de les utiliser en tant que table de hachage. Ces mécanismes ne sont pas présentés, il est sufisant de retenir que les fonctionnalités de tables de hachage sont toujours réalisables au dessus de ces systèmes.

Tapestry : Dans Tapestry[19], l'espace de nomage est un ensemble numérique linéaire en base 2 de taille limitée exprimé sur m bits, on a donc un maximum de $T = 2^m - 1$ *Peers*. Pour mesurer la distance et sélectionner, à chaque étape, le *Peer* le plus proche de la destination, Tapestry propose l'utilisation de la correspondance entre suffixes d'identifiant en base 2 similaire à ce qui est utilisé dans le routage IP. Le *Peer* le plus proche de la destination, parmi tous les *Peers* connus, est celui dont l'identifiant partage le plus grand suffixe avec l'identifiant de la destination. Dans Tapestry, chaque *Peer* maintient, pour chaque i entre 1 et $log_2(T)$ les connaissances pour contacter physiquement un *Peer* dont l'identifiant partage un suffixe de taille i avec le sien. Un *Peer* se considère comme responsable d'un identifiant lorsqu'il ne peut pas trouver de *Peer* plus proche.

Pastry : Dans Pastry[16], l'espace de nomage est un ensemble numérique linéaire en base 2^b (avec b paramètre du système), codé sur m bits, avec un maximum de $2^m - 1$ *Peers*. Il utilise le même système de routage que Tapestry, mais en travaillant sur les préfixes d'identifiant. Pastry propose un second mécanisme pour les messages dont l'identifiant de destination est sous la responsabilité des l *Peers* dont les identifiants sont numériquement les plus proches du *Peer* courant. Lorsqu'il reçoit un message à destination d'un identifiant sous la responsabilité d'un de ces l *Peers*, un *Peer* le transmets à celui dont l'identifiant est le plus proche numériquement de l'identifiant de destination. Dans Pastry un *Peer* d'identifiant I est responsable des identifiants N_i tel que $\forall i, J$ avec J l'identifiant d'un *Peer*, $|I - N_i| < |J - N_i|$.

Chord : Dans Chord[18], les identifiants sont des nombres entiers codés sur m bits, on a donc un maximum de $T = 2^m - 1$ *Peers*. Un *Peer* d'identifiant N connait les adresses physiques des *Peers* dont les identifiants font partie de l'ensemble $\{X = (N + 2^{i-1}) modT, 1 \leq i \leq log_2(T)\}$. Un *Peer* d'identifiant N est responsable des identifiants situés dans l'intervale $]M, N]$ avec $M < N$ tel que M est l'identifiant d'un *Peer* du réseau et $\forall O \neq M$, identifiant d'un *Peer* du réseau, $|O - N| > |M - N|$. La mesure de distance utilisée entre deux identifiants est la valeur absolue de leur différence numérique.

Kademlia : Kademlia[9] utilise un système de nomage numérique linaire sur 160 bits. Il utilise comme métrique de distance entre deux identifiants le résultat d'un opération XOR entre leurs bits interprété comme un entier. Pour chaque $i, 0 \leq i < 160$ un *Peer* d'identifiant N maintient entre 0 et k (paramètre du système) "connaissances" pour le contact physique des *Peers* d'identifiants M_j tel que les distance $|N - M_j|$ (pour $0 \leq j \leq k$) sont comprises entre 2^i et 2^{i+1}. Lorsqu'il reçoit un message, un *Peer* le transmet aux l *Peers* dont il a connaissance et dont l'identifiant est le plus proche de l'identifiant de destination. Dans Kademlia, un *Peer* se considère responsable d'un identifiant I lorsqu'il ne peut trouver d'autres *Peers* dont l'identifiant est plus proche de I que le sien.

CAN : Dans CAN[15] le réseau de *Peers* est vu comme un espace circulaire fini à n dimensions (un hypertore de dimension n). Les identifiants sont ainsi des vecteurs de n élements représentants les coordonnées de points dans cet espace. La distance géométrique entre deux *Peers* est utilisée comme métrique pour le routage. Un *Peer* est responsable des indentifiants correspondants aux points situés dans un sous-espace de l'espace global. Un *Peer* connait les informations de contacts physiques de *Peers* responsables de sous-ensembles de l'espace adjacent au sien.

2.3 Précision sur le stockage de données

Parmi les systèmes Peer-to-Peer les plus connus, on retrouve de nombreuses applications de partages de fichiers telles que Gnutella, Napster ou eDonkey. Les cas d'utilisation de systèmes Peer-to-Peer qui nous intéressent dans cet article ne sont pas de la même nature. Les systèmes de partage de fichiers permettent, comme leur nom l'indique, le partage de fichiers entre leurs utilisateurs. Il ne s'agit pas de systèmes de stockage de données, mais de systèmes d'index de fichiers distribués. Ces systèmes permettent à leurs participants de découvrir d'autres participants offrant un ou des fichiers déterminés et de les récupérer auprès d'eux. Rien dans ces systèmes n'assure une durée de vie pour une donnée dépassant la durée de la participation des *Peers* qui l'ont amenée.

Un système de stockage garantit la conservation, au moins pour un temps, des données au delà de la participation des *Peers* dont elles sont issues. En plus de l'indexation, ou d'une méthode de recherche, distribuée des données qui y sont stockées, ces systèmes fournissent un mécanisme pour assurer leur distribution, et parfois aussi leur réplication, parmi les *Peers* présents. Il s'agit donc de systèmes plus fiables et plus complets que les simples systèmes de partage de fichiers.

3 Présentation des cas d'utilisation

3.1 PAST

PAST[17][4] est un système de stockage en lecture seule basé sur Pastry. PAST stocke les fichiers en entier dans la DHT avec comme identifiant le hachage SHA-1[14] de la concatenation de leur nom, de la clef publique de leur propriétaire et d'une *valeur de sel*[1]. Un même fichier est stocké par k *Peer* enfin d'en assurer la réplication. Avec les fichiers sont stockés des *certificats*, pour chaque fichier un *certificat* contient son nom, sa taille, son facteur de replication k, la *valeur de sel* utilisée pour son insertion et le hachage SHA-1 de son contenu. Pour en assurer la validité, un certificat est signé avec la clef secrète de son émetteur.

Pour assurer la disponibilité de l'espace de stockage PAST utilise un système de quota par utilisateur reposant sur l'utilisation de smartcards. Chaque utilisateur qui veut utiliser le système doit possèder une smartcard, qui contient une paire de clefs asymétriques. Lors de la publication d'un fichier, son *certificat* est

[1] Valeur aléatoire introduite pour diminuer la probabilité que deux concaténation de noms et de clefs publiques différentes n'obtiennent un même hachage

produit par la smartcard. Le certificat est ainsi signé avec la clef secrète de l'utilisateur sans que celle-ci ne lui soit communiquée. La smartcard peut ainsi refuser de générer un nouveau *certificat* lorsque le quota d'espace de son utilisateur est dépassé.

Pour récupérer de l'espace, un utilisateur peut demander au système PAST de ne plus assurer le maintien d'un des ses fichiers dans la DHT. A partir de ce moment, PAST ne doit plus garantir la disponibilité du fichier, mais il ne garantit pas son indisponibilité. Pour limiter la *supression* d'un fichier à son seul propriétaire, à l'insertion, PAST ajoute au fichier un *owner-credential*. Seul le propriétaire d'un fichier, à l'aide de sa smartcard, peut (re)générer un même *owner-credential*. Lors de la *supression* du fichier, ce *owner-credential* doit être fourni pour que l'opération soit réellement effectuée.

3.2 Pasta

Pasta[11] propose un système plus complet que PAST en assurant plus de fonctionnalités le rapprochant d'un véritable système de fichier standard. On y retrouve ainsi la possibilité de modifier un fichier déjà présent dans le système. Comme PAST, Pasta utilise Pastry comme système de DHT pour assurer la distribution, la réplication et la localisation des données. Les fichiers qui y sont stockés sont découpés en blocs de tailles variables. Pour définir ces blocs, Pasta utilise l'algorithme proposé pour le "Low Bandwdith File System"[12] qui maximise la probabilité de créer des blocs communs à plusieurs fichiers différents et permet ainsi de réduire l'espace nécessaire pour stocker ces fichiers. Chaque bloc de données est stocké dans la DHT avec le hachage SHA-1 de son contenu comme identifiant. On assure ainsi la validité des blocs lors de leur transmission mais également leur caractère immuable après insertion.

En plus des blocs de données, Pasta stocke des blocs d'index. Un bloc d'index contient un fragment d'un espace de nomage utilisateur, la structure hiérarchique de ses fichiers et répertoires. Ce bloc d'index fournit pour chaque répertoire, les fichiers et sous-répertoires qu'il contient et pour chaque fichier la liste de blocs de données qui le composent. A un répertoire, peut aussi correspondre l'identifiant d'un autre bloc d'index qui est alors "monté" dans le système de fichier comme contenu du répertoire. Ce système permet de créer de larges systèmes de fichiers hiérarchiques. La figure 1 donne une réprésentation graphique de la structure de données utilisée dans Pasta.

Pour assurer l'intégrité des données, Pasta limite les modifications d'un bloc d'index à son créateur. Lors de sa création, un bloc d'index se voit assigner une paire de clefs asymétriques et est toujours stocké avec sa clef publique comme clef d'accès. A chaque modification, le contenu d'un bloc d'index doit être signé avec sa clef secrète. Sa clef publique servant de clef d'accès, tout *Peer* peut vérifier la validité d'une modification et refuser les modifications invalides. Pasta prévoit néanmoins un système permettant à plusieurs utilisateurs de modifier un même système de fichier, chaque utilisateur maintient alors un bloc d'index, qu'il synchronise volontairement, et de façon périodique, avec les blocs d'index des autres utilisateurs.

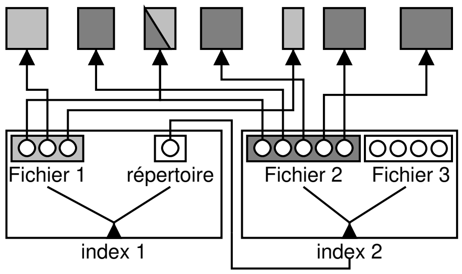

Fig. 1. Pasta stocke dans la DHT des blocs de tailles différentes qui composent les fichiers. Des blocs d'index sont utilisés pour créer une structure de répertoire héirarchique et donner les blocs qui composent un fichier.

3.3 CFS

CFS[2][3] est un système de fichiers distribué basé sur Chord, tout comme Pasta, il offre au *propriétaire* d'un fichier, ou toute personne connaissant sa clef privée, la possibilité de le modifier. Comme dans Pasta, CFS stocke les fichiers sous forme de blocs de données de tailles variables et utilise le hachage SHA-1 de leur contenu comme identifiant. Par contre, alors que Pasta utilise un système de bloc d'index, dans CFS les meta-données du système de fichiers, sa structure, ses fichiers et les blocs qui les composent, sont conservés dans des blocs semblables aux blocs des fichiers. Le système de fichier d'un utilisateur, est construit à partir d'un bloc spécial, servant de racine. La structure de donnée utilisée pour stocker le système de fichier dans CFS est représentée sur la figure 2.

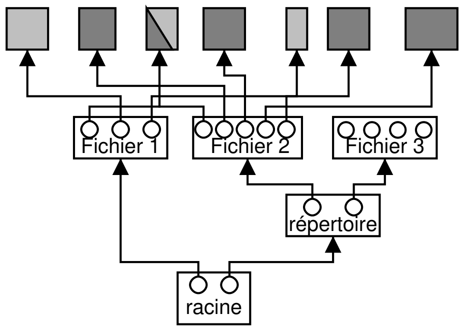

Fig. 2. CFS stocke dans la DHT des blocs de tailles différentes qui composent les fichiers mais aussi la structure des repertoires et les méta-données des fichiers. Un bloc *racine* permet de retrouver les fichiers et répertoires.

Un bloc racine est stocké en utilisant la clef publique d'une paire de clefs asymétriques et est signé en utilisant la clef secrète correspondante. Comme dans Pasta, ce système permet la modification d'un bloc racine par toute personne connaissant sa clef secrète, ce qui permet de modifier le contenu de tout le système de fichier associé. Dans un système de fichier CFS, les blocs de données sont stockés pour une durée déterminée, au dela de cette durée, les *Peers* responsables du stockage du bloc ne sont plus obligés de les maintenir.

3.4 Ivy

Ivy[13] est un système de fichiers en lecture et écriture multi-utilisateur, un même fichier peut y étre écrit ou modifié par plusieurs utilisateurs différents. Ivy diffère des autres systèmes de stockage de données présentés dans sa manière d'utiliser le stockage de l'information dans la table de hachage distribuée. Alors que les autres systèmes fonctionnent en y stockant les données "brutes" des fichiers ainsi que des meta-données de description du système de fichiers lui même, Ivy place dans sa DHT les audits, ou *logs*, d'événements, des actions sur le système de fichiers, de ses participants. Chaque évenement se voit stocké dans la DHT accompagné de la référence de son prédécesseur dans la suite logique des événements constituant le *log* d'un utilisateur comme indiqué sur la figure 3. Comme dans les autres systèmes, les données d'un événement sont stockées avec pour identifiant le hachage SHA-1 de leur contenu afin de garantir leur immuabilité.

Chaque utilisateur du système possède sa paire de clefs asymétriques ainsi qu'une entrée spéciale dans la DHT appelé *log-head* qu'il signe avec sa clef privée pour en garantir l'authenticité. Le *log-head* contient simplement la référence vers l'entrée de la DHT correspondante à l'événement le plus récent engendré par son utilisateur.

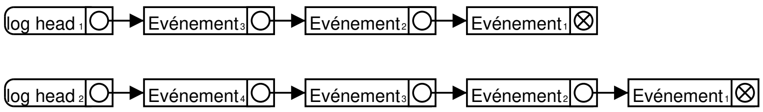

Fig. 3. Ivy stocke dans la DHT les *logs* de ses utilisateurs. Le *log-head* d'un utilisateur référence son événement le plus récent. Pour retrouver les données d'un fichier, Ivy parcourt les *logs* de utilisateurs y ayant accès pour retrouver les événements qui lui permettent de le regénérer.

Dans Ivy, un système de fichiers particulier est désigné par l'ensemble des utilisateurs, de leurs clefs publiques, qui y ont un accès en écriture. Les clefs permettant de constituer un système de fichier ne sont pas stockées dans le DHT mais seulement par les *Peers* de ses utilisateurs . Pour y récupérer un fichier, *Peer* parcourt en ordre anti-chronologique, les *logs* de tous les utilisateurs, en récupérant un à un les événements qui les constituent depuis la DHT, jusqu'a avoir suffisament d'informations pour (re)construire le fichier. Pour permettre d'ordonner des événements issus de plusieurs *logs*, Ivy utilise un système de numérotation de version de fichier complexe. Lors d'un événement sur un fichier, son numéro de version est calculer à partir de ceux définis dans les *logs* d'événements des des autres utilisateurs. Il peut cependant arriver que deux événements sur un même fichier issus de *logs* différents lui assignent le même numéro de version, dans ce cas Ivy ordonne les événements en fonction de la clef publique de leurs utilisateurs selon une méthode commune à tous les utilisateurs du système de fichier.

3.5 OceanStore

OceanStore [8][1] se base sur Tapestry et se veut un système d'archivage de données beaucoup plus complet et plus riche que les autres systèmes présentés. Si tous les *Peers* du système participent à la distribution des messages, OceanStore les divise en trois groupes. Un premier groupe agit simplement comme une DHT qui stocke pour une longue durée les données qu'on lui transmet. Un second groupe organise la distribution et le stockage de données par le premier groupe et maintient des copies *actives* des données qui s'y trouvent. Enfin on retrouve dans le troisième groupe les clients du système, fournisseurs et demandeurs de données. Ces groupes ne sont pas exclusifs, un même *Peer* peut participer à plusieurs groupes.

Dans OceanStore, les informations, fichiers ou méta-données, sont également gérées par blocs. Cependant, les blocs ne sont pas stockés comme dans les autres systèmes. Pour assurer la disponibilité tout en minimisant l'espace requis par la réplication des données, OceanStore utilise un système d'*erasus codes*. Un bloc est découpé en n fragments qui servent à recoder kn fragments avec $k > 1$. Ce sont ces kn fragments qui sont stockés dans la DHT. La fonction employée pour créer ces kn fragments assure que n'importe quelle combinaison de n fragments est suffisante pour reconstruire le bloc en entier. Afin de garantir de bonnes performances, l'ensemble des blocs qui forment un fichier sont les feuilles d'un "arbre B" (B-Tree) dont chaque noeud est stocké en tant qu'un bloc différent qui référence les blocs de ses noeuds enfants.

Pour le stockage des données Tapestry propose un modèle plus complexe sur lequel repose OceanStore et qui permet à des données différentes de partager un même identifiant, une même clef d'accès. Ainsi les fragments d'un même bloc sont tous insérés avec un même identifiant, calculé à partir du contenu du bloc et de ses fragments. Lors de la récupération d'un bloc via son identifiant , au moins n fragments identifiés par son identifiant peuvent être récupérés dans la DHT et utilisés pour reconstruire le bloc. La fonction de calcul de l'identifiant d'un bloc et de ses fragments est une fonction de hachage hiérarchique qui permet de vérifier la validé du bloc complet mais également de chacun de ses kn fragments.

Comme dans Pasta et CFS, les méta-données sur les fichiers et sur la structure d'un système de fichiers sont stockées dans des blocs. Comme CFS, OceanStore utilise des blocs de répertoires qui contiennent les références vers des fichiers

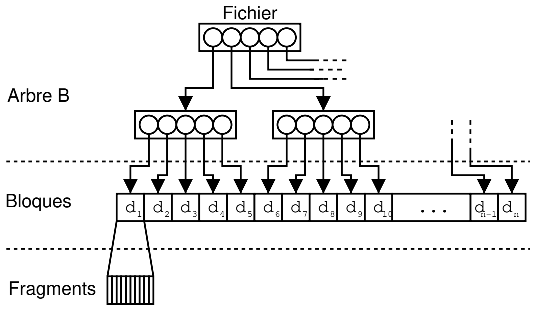

Fig. 4. Pour être stocké un fichier est décomposé en blocs eux même décomposés en frangments. La composition (en bloc) d'un fichier est donnée par les feuilles d'un arbre B. Pour être stockés dans la DHT, les noeuds des arbres B sont eux même des en bloc décomposés en fragments. Les fragments d'un même blocs sont tous accessibles à partir de sa référence.

accompagnées de leurs noms ainsi que des noms d'autres répertoires et des références de blocs qui définissent leurs contenus. Un système de fichier peut ainsi être présenté à l'utilisateur à partir d'un bloc de repertoire racine et de l'arbre de ses enfants.

En plus de cette structure de données de base, la modification d'un fichier est gérée via un système d'*updates* de fichier. Sans ce mécanisme la modification d'un fichier donnerait lieu à un nouvel "arbre B" et à des nouveaux blocs pour tout le fichier qui possèderait alors une nouvelle clef d'accès. OceanStore gère la modification d'un fichier via la création d'un *update* qui possède sa propre référence et par modification du bloc du répertoire qui le contient afin que le nom du fichier soit associé à la référence de l'update. Pour une nouvelle version d'un fichier, un *update* fournit un nouvel "arbre B" qui référence les nouveaux blocs de données, ceux qui contiennent de l'information qui ne se trouvait pas dans l'ancienne version du fichier, et qui re-référence, de manière indirecte, les blocs de l'ancienne version qui n'ont pas été modifiés. La gestion de la numérotation et de l'ordonnacement des *updates* de fichier emploie des mécanismes complexes qui permettent d'assurer les propriétés ACID lors de l'accès en lecture/écriture aux fichiers.

Pour chaque fichier stocké, OceanStore offre la possibilité de crypter son contenu, pour en restreindre l'accès en lecture et la possibilité d'y attacher une *liste de contrôle d'accès* qui permet de limiter les accès en écriture aux seuls utilisateurs définis dans cette liste. Le contôle d'accès en lecture se fait donc par le coté "client" du programme, qui est incapable de lire un bloc dont il ne connait pas la clef. Tandis que le système de protection en écriture est forcé par le coté "serveur" qui refuse la modification d'un fichier avec des données qui ne sont pas valides en regard de la *liste de contrôle d'accès*.

4 Comparaison

PAST se limite au stockage d'objets immuables, éventuellement temporaire fortement sécurisé via l'utilisation de smartcards. Dans PAST les fichiers sont stockés entiers par les *Peers*, ce qui n'assure qu'une faible distribution des données qui pourrait nuire à leur disponibilité. De plus PAST n'offre aucune organisation des fichiers, qui ne peuvent être extraits que grâce à leur identifiant direct. Cependant, PAST propose un système de gestion de l'espace disponible et de quota utilisateur qu'on ne retrouve pas dans les autres systèmes. D'un point de vue purement Peer-to-Peer ce système de sécurité peut être génant, puisque nécessitant une certaine centralisation, mais il permet d'offrir simplement des garanties difficiles à obtenir dans un système Peer-to-Peer.

A l'opposé de PAST, OceanStore est un système beaucoup plus complexe offrant nettement plus de fonctionnalités. Il utilise des *erasus codes* afin de maximiser la disponibilité de données et est capable d'assurer des transactions respectant les propriétés ACID via un mécanisme complexe de gestion de modifications de fichiers. OceanStore permet le stockage sécurisé de fichiers qui sont alors cryptés tout en maintenant les autres focntionnalités du système. Il est cependant beaucoup plus complexe que les autres systèmes et repose sur des spécifités précises de Tapestry ce qui le rend dificile à utiliser sur un autre système de DHT.

Les trois autres systèmes présentés sont assez similaires en fonctionnalités, ils représentent un bon compromis entre la pauvreté de PAST et la complexité d'OceanStore. En plus du stockage de fichiers immuables, il offrent tous les trois des mécanismes de modification des fichiers et de lecture sécurisés. Comme souvent de nos jours, leur sécurité repose sur l'utilisation de systèmes d'encryption/désencryption avec une paire de clefs asymétriques.

Pour stocker les fichiers et leur méta-données, Pasta et CFS utilisent des systèmes fort proches. Ils proposent un système de stockage de fichiers modifiables sécurisé. Ils offrent également un structuration des fichiers stockés permettant la création de systèmes de fichiers hiérarchiques. Cette structure peut être présentée à l'utilisateur pour fournir une méthode d'identification de données plus efficace que l'accès par identifiant de PAST.

Pour leur stockage, Pasta et CFS séparent les fichiers en blocs, améliorant ainsi leur distribution et leur disponibilité par rapport à PAST. Pasta autorise le propiétaire d'un objet à les modifier via la modificiation de ses méta-données, son identité est vérifiée selon le principe de signature par clefs asymétriques. CFS n'offre à un utilisateur que la possibilité de remplacer un des ses fichiers existant par un autre, en réutilisant éventuellement une partie de ses blocs.

Pasta propose un stockage permanent de fichiers, tout y est mis en oeuvre pour les conserver mais avec une politique de *best effort*, il peut arriver que des blocs soient perdus, rendant ainsi les fichiers auxquels ils appartiennent irrécupérables. CFS offre une meilleure garantie de conservation mais sur une durée déterminée et finie.

Ivy propose une approche différente des autres systèmes de structuration de données pour offrir un système de fichiers en lecture et écriture multi-utilisateurs. Pour chaque utilisateur Ivy mémorise dans un DHT les évenements qu'il provoque au sein du système. Lorsque l'on y récupére un fichier, Ivy utilise les évenements mémorisés qui le concerne pour le regénérer. Dans Ivy l'écriture dans un fichier ou répertoire précis n'est pas restreinte à un seul utilisateur. Les modifications faites par chaque utilisateur sont stockées séparement, lors de la récupération de données, le consommateur définit lui même les utilisateurs dont les événements sont à prendre en compte.

5 Conclusion

Si ce sont les applications de partage de fichiers qui se sont imposées comme modèle d'application en Peer-to-Peer c'est probablement parce qu'il s'agit d'un type d'application à la fois populaire pour lequel les premiers modèles de Peer-to-Peer semblaient bien adaptés. Même si l'avenir de telles applications est encore assuré, les nouveau modèles de Peer-to-Peer tels les DHT offrent, en corrigeant le principal problème des premiers modèles, de nouvelles oportunités d'applications.

Des applications reposant sur ces modèles existent déjà, ainsi plusieurs systèmes de stockage de données ont été présentés dans cet article. Les systèmes PAST, Pasta, CFS, Ivy et OceanStore offrent des mécanismes de stockage de données aux propriétés variées. Grâce aux modèles de DHT, tous supportent, du moins en théorie, une importante montée en charge sans pour autant sacrifier les fonctionnalités qu'ils proposent.

Ces systèmes montrent bien la possibilité de créer des applications complexes sur base des fonctionnalités simples offertes par les systèmes de DHT. Elles montrent également l'existence de systèmes Peer-to-Peer offrant d'autres services que le trop connu partage de fichier entre utilisateurs de l'Internet. Les sytèmes présentés ont au moins déjà été implémentés en prototypes et subi des tests de mesures de performances. Les résultats sont des plus encourageants et une utilisation réelle de ces systèmes est donc parfaitement envisageable. On peut conclure, sur base de l'existence des ces applications, que le Peer-to-Peer, s'il ne les révolutionne pas, apporte de nouvelles possibilités aux systèmes distribués.

Références

1. D. Bindel, Y. Chen, P. Eaton, D. Geels, R. Gummadi, S. Rhea, H. Weatherspoon, W. Weimer, C. Wells, B. Zhao, and J. Kubiatowicz. Oceanstore : An extremely wide-area storage system, 2000.
2. Frank Dabek. A cooperative file system.
3. Frank Dabek, M. Frans Kaashoek, David Karger, Robert Morris, and Ion Stoica. Wide-area cooperative storage with CFS. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP '01)*, Chateau Lake Louise, Banff, Canada, October 2001.
4. P. Druschel and A. Rowstron. PAST : A large-scale, persistent peer-to-peer storage utility. In *HotOS VIII*, pages 75–80, Schloss Elmau, Germany, May 2001.
5. Shawn Fanning. Napster. `http://www.napster.com/`.

6. Justin Frankel. Gnutella. `http://www.gnutellanews.com/`.
7. Kazaa. `http://www.kazaa.com`.
8. John Kubiatowicz, David Bindel, Yan Chen, Patrick Eaton, Dennis Geels, Ramakrishna Gummadi, Sean Rhea, Hakim Weatherspoon, Westly Weimer, Christopher Wells, and Ben Zhao. Oceanstore : An architecture for global-scale persistent storage. In *Proceedings of ACM ASPLOS*. ACM, November 2000.
9. P. Maymounkov and D. Mazieres. Kademlia : A peerto -peer information system based on the xor metric, 2002.
10. Jed McCaleb. eDonkey2000. `http://www.eDonkey2000.com`.
11. Tim D. Moreton, Ian A. Pratt, and Timothy L. Harris. Storage, mutability and naming in pasta.
12. Athicha Muthitacharoen, Benjie Chen, and David Mazieres. A low-bandwidth network file system. In *Symposium on Operating Systems Principles*, pages 174–187, 2001.
13. Athicha Muthitacharoen, Robert Morris, Thomer M. Gil, and Benjie Chen. Ivy : A read/write peer-to-peer file system. In *Proceedings of 5th Symposium on Operating Systems Design and Implementation*, 2002.
14. Nat. Institute of Standards and Technology (NIST). Announcement of weakness in the secure hash standard, 1994.
15. Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard Karp, and Scott Shenker. A scalable content addressable network. In *Proceedings of ACM SIGCOMM 2001*, 2001.
16. 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– ? ?, 2001.
17. Antony I. T. Rowstron and Peter Druschel. Storage management and caching in PAST, a large-scale, persistent peer-to-peer storage utility. In *Symposium on Operating Systems Principles*, pages 188–201, 2001.
18. Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. Chord : A scalable peer-to-peer lookup service for internet applications. In *Proceedings of the 2001 conference on applications, technologies, architectures, and protocols for computer communications*, pages 149–160. ACM Press, 2001.
19. B. Y. Zhao, J. D. Kubiatowicz, and A. D. Joseph. Tapestry : An infrastructure for fault-tolerant wide-area location and routing. Technical Report UCB/CSD-01-1141, UC Berkeley, April 2001.