Stockage dans les systèmes pair à pair

Olivier Soyez

Numéro d'ordre : ...

THÈSE

présentée devant

l'Université de Picardie Jules Verne d'Amiens

pour obtenir

le Titre de Docteur
Spécialité : Informatique

au titre de la formation doctorale informatique d'Amiens

par Olivier Soyez

Stockage dans les systèmes pair à pair

Soutenue le 29 novembre 2005

Après avis des rapporteurs : Monsieur Franck CAPPELLO
Monsieur Pierre SENS

Devant la commission d'examen formée de :
Monsieur Alain BUI
Monsieur Franck CAPPELLO
Monsieur Cyril RANDRIAMARO
Monsieur Pierre SENS
Monsieur Gil UTARD
Monsieur Vincent VILLAIN

Thèse préparée au Laboratoire de Recherche en Informatique d'Amiens
Université de Picardie Jules Verne,
CNRS-FRE2733.

A Jeanne

A ma famille

A mes amis

Remerciements

Le présent manuscrit est le résumé de mes travaux réalisés en équipe durant les trois années de thèse. Je tiens à remercier chaleureusement chaque membre qui compose l'équipe du projet Us. Aussi, figurent des remerciements les concernant afin de ne jamais oublier ceux qui ont participé activement au bon déroulement de ces années de thèse.

Je souhaite adresser mes plus vifs remerciements aux membres du jury de ma soutenance de thèse :

- *Franck CAPPELLO* (Directeur de Recherche INRIA, INRIA Futurs) pour avoir accepté d'être un des rapporteurs de ma thèse, pour ses remarques et conseils concernant mon manuscrit, et pour m'avoir permis d'assister et participer aux réunions d'ACI GRID CGP2P. J'ai pu par le biai de ces réunions appréhender plus facilement divers problèmes scientifiques et découvrir avec grand intérêt les travaux et projets d'autres équipes et laboratoires.
- *Pierre SENS* (Professeur de l'université Paris VI, LIP6) pour avoir accepté d'être un des rapporteurs de ma thèse, pour son intérêt porté sur le projet Us, ses recommandations et avis qui ont permis une nette amélioration de mon manuscrit.
- *Vincent VILLAIN* (Professeur de l'université de Picardie Jules Verne d'Amiens, LaRIA) mon directeur de thèse, pour sa confiance. Je tiens également à le remercier d'avoir accepté d'être mon directeur de thèse. Merci tout particulièrement pour ses conseils dans le domaine des systèmes distribués et ses questions très pertinentes qui m'ont amené à de nombreuses réflexions.
- *Cyril RANDRIAMARO* (Maître de Conférence à l'université de Picardie Jules Verne d'Amiens, LaRIA) qui m'a orienté vers le monde passionnant de la recherche, qui a su me guider et m'encadrer durant toutes ces années en me faisant découvrir au fur et à mesure la communauté de notre domaine de recherche, en participant à des conférences et aux réunions d'ACI GRID CGP2P. Cyril m'a également permis d'encadrer des stagiaires de DEA. Je ne pourrai jamais assez le remercier pour sa gentillesse et sa compréhension. Je tiens aussi à souligner sa grande disponibilité.
- *Gil UTARD* (Maître de Conférence à l'université de Picardie Jules Verne d'Amiens, LaRIA) pour ses nombreux conseils avisés, pour sa patience, pour le temps passé à converser et trouver des idées novatrices. Je tiens aussi à le remercier pour m'avoir fait bénéficier de sa

grande expérience et connaissance dans le domaine de la recherche.
– *Alain BUI* (Professeur de l'université de Reims Champagne Ardenne, CReSTIC) pour avoir accepté d'être un membre du jury de ma thèse, et son intêret à mes travaux de recherche.

Je tiens également à remercier les personnes avec qui j'ai eu le plaisir de collaborer tout au long de ma thèse : *Olivier C.* qui a consacré beaucoup de temps à résoudre de nombreux problèmes techniques et divers survenus lors de cette thèse. *Francis* pour son indéniable maîtrise des mathématiques. *Ghislain* pour nos investigations sur diverses stratégies de reconstruction au sein de Us. *Sébastien* pour sa grande maîtrise du Latex et son aide dans l'écriture de mon manuscrit. *Stéphane* pour son soutien et nos discussions diverses sur Us. *Loys* pour son aide dans la compréhension de la géométrie projective et pour m'avoir prêté de nombreux ouvrages sur le sujet. *Philippe* et *Loïc* pour m'avoir supporté durant leurs stages de DEA. *Corinne* et *Claude* pour leur soutien et l'aide qu'ils m'ont apporté tout au long de ces trois années. *Florence* et *Bernard* pour l'ambiance du bureau 309. *Cyril* et *Sylvain* pour l'accueil chaleureux lors de mon arrivée à Reims. *Sandra* pour sa bonne humeur. Ainsi que l'ensemble des thésards et membres du laboratoire LaRIA et de l'équipe CRESTIC pour leur disponibilité et leur sympathie.

Je remercie également toute ma famille, non seulement pour les bons moments qu'ils m'ont fait passer, mais aussi pour les nombreuses corrections qu'ils ont apportées au rapport... souvent plus sur la forme que sur le fond ! A ce titre, je remercie particulièrement mes parents, ma mère *Martine* qui a relu la thèse en entier avec courage... et crayon rouge, ainsi que tous ceux qui m'ont donné de précieuses indications et corrections ...

Je remercie ma tante *Nicole* et mon grand-père *Henri* qui ont su m'encourager et m'aider pendant toutes ces années, ainsi que mes amies et amis (*Hélène*, *Audrey*, *Ophélie*, *Clémence*, *Julie*, *Fabrice*, *Cédric*, *Sébastien L.*, *Sébastien T.*, *Didier*, *Yann*, *Christophe*), mes camarades de tai chi, mes frères (*David* et *Emmanuel*) et ma soeur *Emilie* avec qui j'ai passé de très bonnes soirées et d'agréables moments.

Je remercie tout particulièrement *Jeanne* qui m'a grandement aidé à traverser l'épreuve de la rédaction de la thèse, et ma mère *Martine* pour son aide précieuse durant toutes ces années. Et surtout, je remercie du fond du coeur mon père *Daniel* qui m'a offert mon premier ordinateur à l'âge de 18 ans. Cet achat a été déterminant pour la suite de mes études. Encore MERCI !

Table des matières

Chapitre 1

Introduction

A l'heure actuelle, les systèmes pair à pair constituent une alternative à l'approche commune client/serveur d'Internet. L'idée de base du pair à pair est que l'application distribuée n'est plus conçue avec une séparation stricte entre les clients, consommateurs de services, et les serveurs, fournisseurs de services. A la place un système pair à pair est composé de pairs égaux en fonctionnalités, qui décident séparément de se joindre au système et d'y proposer un ou plusieurs services. Un système pair à pair est ainsi un réseau dynamique de pairs interconnectés, consommateurs et fournisseurs de services.

Contrairement aux idées reçues, le concept même du "pair à pair" est loin d'être récent. Bien au contraire, ce concept est à l'origine même d'Internet. Le protocole de communication d'Internet a été conçu dès le début comme un système symétrique pair à pair. Ainsi dès 1969, année de naissance d'Internet, par le biais du réseau précurseur ARPANET, projet militaire créé par l'ARPA (Advanced Research Projects Agency dépendant du DOD, Department of Defense USA), Internet fonctionnait déjà de manière autonome. Les premières applications et protocoles d'Internet, FTP pour l'échange de fichiers et Telnet pour utiliser une machine à distance, étaient déjà des applications pair à pair dans le sens où chaque pair pouvait être à la fois client et serveur. A partir de l'explosion d'Internet due à sa démocratisation au sein du grand public, le navigateur web et d'autres applications ont été construites sur le modèle simpliste du client/serveur : on pose une question et on attend la réponse. Pire, les fournisseurs d'accès à Internet ont attribué des adresses IP temporaires aux utilisateurs qui se connectaient par téléphone et les firewalls ont fini d'achever et de transformer le réseau Internet en un système essentiellement client/serveur.

C'est le logiciel Napster [45], créé par Shawn Fanning, qui a considérablement popularisé le concept du "pair à pair". Napster est une

application pair à pair de partage et d'échange de fichiers musicaux mp3. Pourtant, Napster est construit autour d'un serveur central, ce qui pourrait induire que Napster n'est pas un système pair à pair. Mais la question primordiale à se poser pour savoir si un système est un système pair à pair ou non est la suivante :

Qui possède les ressources qui alimentent le système ?

Si la réponse est l'ensemble des pairs, dans ce cas le système est un système pair à pair. Par exemple pour Napster : qui stocke les fichiers musicaux ? La réponse est bien l'ensemble des pairs, donc Napster est un système pair à pair.

Par définition, les systèmes pair à pair ou P2P (peer to peer) se caractérisent par des sur-réseaux *ad hoc* basés sur des connexions point-à-point formant un graphe faiblement connexe. L'objectif de ces systèmes est de fournir un mécanisme tolérant aux pannes de routage entre les pairs. Ce mécanisme peut avoir de plus une fonction de localisation des données.

Les avantages des systèmes pair à pair où systèmes dits à grande échelle, tel que Napster, sont nombreux. Ils donnent accès à un grand nombre de ressources, dont le nombre de pairs est supérieur à 10^x ($x > 2$). Ils disposent d'une administration transparente, un système pair à pair évolue sans machine(s) dédiée(s) pour son administration. Ces systèmes sont conçus de telle sorte qu'aucun des pairs ne soit indispensable au fonctionnement général : le système n'est pas paralysé par la défaillance d'un ou plusieurs pairs. En contrepartie, de par la volatilité des pairs, le réseau est non fiable. De plus, des problèmes de sécurité peuvent survenir dus à certains pairs malveillants non écartés du réseau. En résumé, un système pair à pair impose de par sa nature une gestion complexe, mais dispose d'une administration transparente et un accès à un grand nombre de ressources.

Les systèmes pair à pair sont très largement utilisés pour partager des ressources à travers l'Internet. Certains ont pour but de partager du temps CPU (Seti@Home [68], XtremWeb [20], Entropia [18]), d'autres de mettre des données à disposition d'autres utilisateurs (Napster [45],Gnutella [27], Freenet [23], KaZaa [33], eMule [17]). Parallèlement, des systèmes pair à pair dédiés au partage d'espace disque ont vu le jour tel Intermemory [28] ou Oceanstore [3]. Leur but premier est d'offrir un accès transparent à un service de stockage distribué. Cette dernière catégorie de systèmes partage de nombreuses bases communes avec les autres, comme les mécanismes de découverte de réseau ou de localisation de ressource.

Cependant, si les systèmes de partage de CPU peuvent assez facilement faire face à la disparition d'un pair, en relançant un calcul sur un autre

pair par exemple, pour un système de stockage, la défaillance d'un pair implique la perte définitive des données qu'il stockait. Ainsi, il est indispensable de mettre en place des méthodes de redondance dont le rôle est d'assurer la pérennité des données.

Dans un premier temps, une description de quelques systèmes de stockage pair à pair et leurs principes respectifs de fonctionnement seront abordés.

Ensuite, une première partie sera dédiée à définir un nouveau système de stockage pair à pair, nommé Us [1] ("Ubiquitous storage").

Ainsi, nous étudierons quelques techniques de redondance garantissant un certain degré de pérennité des données stockées.

Ensuite, l'architecture et les fonctionnalités du système de stockage pair à pair Us, seront dévoilées. Us est un prototype de stockage pair à pair pérenne en cours de développement au Laria, UPJV Amiens, par le projet Us dont je suis membre. Pour le passage à l'échelle, le système Us dissémine les données sur les pairs utilisateurs. L'objectif principal de Us est la pérennité des données.

Puis, nous présenterons l'interface Us, nommée UsFS. Cette interface couple un système de fichier avec notre système de stockage Us. Son utilisation est intuitive et classique. De surcroît, elle offre des fonctionnalités supplémentaires, telles que gestion de cache pour le gain de performance d'accès aux données et le versioning pour accéder aux différentes versions d'un document.

Finalement, dans une seconde partie, trois chapitres seront dédiés aux distributions des données pouvant être utilisées par un système de stockage pair à pair pérenne, tel que Us. Dans Us, quand un pair tombe en panne un processus de reconstruction régénère les données perdues avec l'aide des autres pairs. Dans nos précédents travaux [75], nous avons montré que pour assurer la pérénité des données, un tel système doit faire face à un grand nombre de reconstructions. Nous verrons donc des stratégies de distribution visant à minimiser un nouveau critère : le coût de perturbation maximal d'un pair, défini par son nombre de données envoyées lors de la reconstruction.

Enfin nous concluons par une synthèse de mes principales contributions et la présentation des perspectives de recherche dans les domaines que nous avons abordés. A noter que cette thèse s'intègre dans une action de recherche nationale ACI Grid CGP2P du MENRT-CNRS-INRIA regroupant plusieurs laboratoires, le LRI, IMAG, LIFL, LIP, LaRIA.

[1] Adresse Internet : "http ://www.ustorage.net"

Chapitre 2

Systèmes de stockage Pair à Pair

On observe aujourd'hui un engouement croissant pour les systèmes des fichiers pair à pair. Les motivations originales de nombre de ces systèmes étaient essentiellement le partage de fichiers musicaux ou vidéos à grande échelle. L'objectif avoué est essentiellement celui de l'anonymat des sources d'informations, ce qui impose un sur-coût non négligeable dans l'accès aux données. En général, il n'y a aucune garantie en ce qui concerne la fiabilité et la disponibilité des données.

Dans le même temps, d'autres systèmes moins populaires, car non dédiés exclusivement à la diffusion de fichiers musicaux ou vidéos, ont été développés. On peut citer InterMemory [10] ou OceanStore [34]. Ils se présentent comme des systèmes de partage d'espace de stockage. À la différence des systèmes précédemment cités, ces systèmes intègrent des mécanismes qui assurent la confidentialité et la pérennité des données.

Cette section propose une description de quelques systèmes pair à pair dédiés au stockage. Après avoir présenté les systèmes pionniers qui ont inspiré les systèmes de stockages, nous décrirons les plus connus. Puis, nous présenterons l'architecture type des systèmes actuels et les techniques mises en œuvre pour traiter les objets mutables.

2.1 Pionniers : publication de fichier

Depuis Napster [45], un grand nombre de systèmes de diffusion de fichiers avec stockage simple ont vu le jour, tels que eMule [17], KaZaa [33], Gnutella [27], Freenet [23], MojoNation [43], BitTorrent [51, 32].

Le principe de fonctionnement de **Napster** est le suivant : à chaque connexion de l'utilisateur de l'application Napster, l'application envoie

au serveur l'adresse IP assignée à la machine de l'utilisateur, ainsi que la liste des titres mp3 qu'il désire partager. Ceux-ci sont alors stockés dans un répertoire (centralisé sur un serveur). Si un autre utilisateur Napster cherche un titre particulier, Napster fournit la liste des IP des utilisateurs en ligne qui le possèdent et l'utilisateur demandeur peut ainsi directement télécharger le titre sur un autre utilisateur en ligne.

eMule [17], version open source de eDonkey, est basé sur le même principe que Napster : l'index des fichiers est centralisé. Cet index est en fait partagé par plusieurs serveurs. Il est donc nécessaire d'avoir une liste des serveurs à jour, afin d'accélérer et d'aboutir dans les recherches de fichiers. Les systèmes comme Napster ou eMule n'offrent aucun mécanisme garantissant une certaine pérennité des données.

Gnutella [62] est un des premiers systèmes pair à pair avec Napster destiné au partage de fichiers. Il se définit essentiellement comme un protocole de recherche et d'échange entre les pairs. Contrairement aux systèmes vus précédemment, Gnutella est un système totalement distribué, il n'utilise pas d'index centralisé. La recherche de fichier s'effectue par un parcours en largeur borné du graphe défini par l'ensemble des connexions. Les données sont accédées directement chez le propriétaire. La durée de vie des données est celle du pair propriétaire, il n'y a pas de mécanisme garantisant une certaine pérennité.

KaZaa [33] est un système de diffusion de fichier qui se différencie par une approche plus "pratique" de l'environnement dans lequel il évolue. Le constat est qu'il existe de nombreuses disparités au sein des pairs, dues à l'évolution rapide des technologies informatiques et en particulier celles des communications (nombreuses offres des FAI : RTC, ADSL, cable, satellite). Pour profiter pleinement et au mieux de l'ensemble des ressources des pairs, sans pénaliser le système par des pairs moins bien dotés que d'autres, deux niveaux de pairs ont été créés : ceux qui ont une connexion haut débit et ceux qui ont une connexion bas débit (exemple : ligne RTC). Les ordinateurs disposant d'une connexion bas débit se relient à un ordinateur ayant une connexion haut débit. Ce dernier devient dès lors un « superpair ». Chaque superpair indexe alors les fichiers des pairs bas débit qui lui sont rattachés, comme le faisait autrefois le serveur central de Napster entre deux superpairs en revanche, la connexion est directe. Donc la propagation des données est plus rapide, puisqu'elle n'utilise plus que les connexions haut débit. Une fois l'adresse IP communiquée à l'ordinateur d'origine, une connexion directe s'établit entre les deux pairs, quel que soit leur niveau. Il s'agit donc d'une solution hybride Napter/Gnutella. KaZaa n'offre pas de garantie supplémentaire sur la pérennité des données.

Freenet [11] est un projet dont l'ambition est d'offrir à ses utilisateurs une totale liberté d'expression en garantissant l'anonymat aussi bien des

diffuseurs d'informations que des lecteurs. La principale originalité de Freenet est que les fichiers ne sont pas stockés à la source, mais sur les pairs du système. Freenet propose un mécanisme de routage auto-adaptatif : la recherche de fichiers s'effectue à travers des pairs qui se spécialisent petit à petit. Les fichiers sont dupliqués le long des chemins d'accès afin d'améliorer leur durée de vie et réduire les temps d'accès. L'espace disponible est géré par un mécanisme de type LRU : les fichiers peu référencés disparaîtrons en premier. Ce système assure une certaine pérennité aux données les plus populaires. Il intègre des mécanismes de cryptage.

MojoNation [43] est, lui aussi, dirigé vers le partage distribué de fichiers. Les concepteurs ont ajouté un système de micro-paiement pour toutes actions entreprises dans le système. Chaque requête implique un échange d'une monnaie virtuelle, le *Mojo*. Ainsi pour récupérer des informations, il est nécessaire de commencer par constituer un capital en offrant au partage des documents que d'autres seront prêts à payer, en Mojo, pour les récupérer ou prêter une partie de son espace disque. Stocker des données ailleurs coûte, mais stocker les données rapporte. Télécharger des fichiers coûte, mais en partager rapporte. C'est ainsi que fonctionne l'économie de MojoNation. L'originalité de MojoNation est que les fichiers sont découpés en plusieurs fragments qui sont dispersés sur des pairs différents. Pour assurer une meilleure disponibilité des documents, ces fragments sont répliqués en fonction de la demande des utilisateurs vis à vis du document. Il n'existe cependant aucune garantie quant à l'existence de l'ensemble des fragments composant un fichier.

BitTorrent [51, 32] est un protocole récent d'échange de fichiers, créé par Bram Cohen, dont le principal atout est la rapidité. Il se concentre sur le téléchargement d'un fichier donné. L'indexation des fichiers est prise en charge par un autre protocole. On doit donc se servir d'un autre logiciel, tel que eMule, pour retrouver les fichiers issus de BitTorrent. Il reprend le principe du téléchargement multiple (comme dans eMule), mais il permet également de développer la coopération en utilisant le principe d'un prêté pour un rendu. Pour cela, un algorithme d'incitation à la coopération inspiré de la théorie des jeux est utilisé. A l'instar d'eMule, il faut redonner des morceaux du fichier pour pouvoir en acquérir de nouveaux plus rapidement. Cette manière de procéder évite la recrudescence des "leechers" (personnes nuisibles à un réseau P2P : elles téléchargent tout en bloquant leurs uploads, elles ne participent donc pas à l'effort commun et n'apportent rien de positif). Des fichiers de taille importante peuvent donc être ainsi distribués par de petits serveurs. L'inconvénient majeur de BitTorrent est que les fichiers ne restent pas partagés longtemps. BitTorrent est donc un système d'échange de fichiers rapide, mais non pérenne.

2.2 Systèmes dédiés au stockage

L'intérêt de la communauté scientifique concernant le domaine du stockage et de nouvelles infrastructures distribués n'a cessé de croître ces dernières années. Les raisons sont nombreuses. Tout d'abord, la nécessité de trouver des infrastructures sécurisées passant correctement à l'échelle, sans dégradation de performances et autres désagréments (déni de service). Puis, le besoin grandissant de stocker de plus en plus de données (issus d'expérimentation physique, biologique : décryptage et cartographie du génome humain, etc ...).

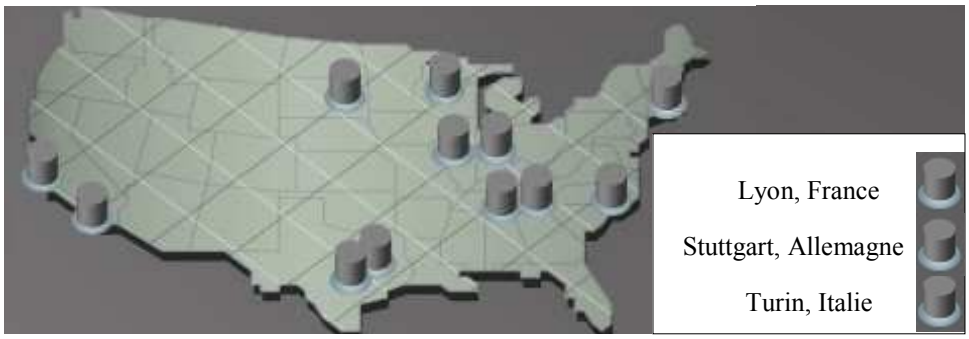
Ce vif intérêt a donc généré de nombreux projets, mais sans réelle coordination entre eux et sans standardisation des nouveaux principes émergeants. C'est dans ce but que le projet américain IRIS (Infrastructure for Resilient Internet Systems) a été créé. Il regroupe les universités de Cambridge (MIT), Californie (Berkeley), New York (ICS) et Rice. Ce projet a entraîné un investissement colossal de 12 millions de dollars de la part de la fondation NSF (National Science Foundation). A noter, le projet récent, nommé DELIS (Dynamic Evoluting Large-scale Information Systems), qui est le pendant européen de IRIS. Ce projet intègre des universités d'Allemagne, Espagne, France, Grèce, Italie, Norvège, Suisse, Tchécoslovaquie.

Parmi les systèmes pionniers précédemment cités, seul Freenet se définit aussi comme un système de stockage, bien que la pérennité soit limitée seulement aux documents les plus populaires. Par la suite de nombreux autres systèmes principalement dédiés au stockage et non plus à la diffusion sont apparus. Ils visent le stockage sûr grâce à des mécanismes garantissant la survie des données, ainsi que leur confidentialité. Désormais, nous allons étudier quelques systèmes dédiés au stockage.

IBP[49] (The Internet Backplane Protocol : Storage in the Network) est un middleware, créé à l'université du Tennessee, pour gérer et utiliser des unités de stockage distantes par Internet. Le point fort d'IBP vient de sa simplicité d'utilisation. Il fournit un client API qui est une suite de primitives très simples à utiliser pour gérer ses données. IBP utilise des serveurs dédiés disséminés dans le monde. Exemple : voir la carte 2.1, où les dépôts correspondent à des serveurs. Ainsi, la pérennité des données est forte, mais le contexte pair à pair est peu sollicité.

L'un des précurseurs dans le stockage pair à pair est **InterMemory** [10, 28]. InterMemory se veut être un système de partage dans lequel la pérennité des données est assurée. C'est un des pionniers dans le domaine. Il se caractérise par un mécanisme de protection des données extrêmement poussé. Dans InterMemory, les données initiales sont découpées en fragments de taille fixe qui sont dispersés et auxquels sont rajoutés des fragments de redondance pour pallier à la disparition de pairs. Dans des

FIG. 2.1 Exemple de carte des dépots internationaux de données IBP



schémas plus élaborés, les fragments sont eux-mêmes protégés par le même mécanisme de fragmentation/dispersion/redondance. La détection des pannes et la mise en oeuvre de la reconstruction s'effectuent par un mécanisme d'auto-surveillance. Ce système se caractérise donc par une très grande pérennité des données stockées.

OceanStore [34] est un vaste projet développé à l'université de Berkeley sur le stockage pérenne de données. C'est probablement le projet le plus abouti concernant le stockage de données pair à pair. Les données sont stockées sur des serveurs dédiés qui possèdent une forte connectivité et une large bande passante. Ces serveurs collaborent afin de fournir un service de stockage ayant la propriété d'ubiquité, c'est à dire que l'accès aux données se fait de façon transparente à partir de n'importe quel point d'accès. Ces serveurs sont par exemple situés chez les fournisseurs d'accès à Internet.

Plus précisément, les données sont organisées suivant deux niveaux de stockage. Le premier niveau est celui qui va assurer la disponibilité des données même lors de la défaillance d'un ou plusieurs serveurs de stockage. Pour cela un mécanisme de redondance et d'auto-surveillance des serveurs est mis en place, comme dans InterMemory. Le deuxième niveau de stockage est une réplication qui n'a pas pour objectif d'assurer la pérennité des données mais plutôt la proximité de celle ci. Il s'agit en fait de réplicats placés sur des serveurs proches de l'utilisateur afin de lui garantir les meilleurs temps d'accès à ses données. [76]

Le système de routage tolérant aux pannes d'OceanStore est Tapestry [79, 80]. Tapestry permet de localiser un pair par une approche [61] probabiliste, basée sur les filtres de Bloom, conjointe à une méthode déterministe, l'algorithme de Plaxton[50], en cas d'échec de la première. La méthode probabiliste permet un gain de temps notable par rapport à une méthode

déterministe, mais n'est pas certaine d'aboutir. D'où l'intérêt de lui adjoindre une méthode déterministe qui trouvera le pair cherché à coup sûr. Lors de l'insertion d'un nouveau fichier, il est placé sur un pair, et un lien est créé vers ce pair de stockage sur le pair dont l'identifiant est le plus proche de l'identifiant du fichier. Une requête est dirigée vers le pair qui a l'identifiant le plus proche de celui du fichier jusqu'à atteindre un pair sachant où se trouve réellement la donnée désirée. De la même façon que dans Freenet, lors du retour de cette information vers l'origine de la requête les pairs apprennent la provenance du fichier et pourront répondre directement la prochaine fois. Mais à la différence de Freenet, ici les pairs ne font qu'ajouter le lien , il n'y a pas de recopie de la donnée.

Bien que dédié à la publication anonyme de fichiers, **freenet** de par la duplication de ces derniers peut être aussi considéré comme un système de stockage. Freneet est basé sur des pairs plus volatiles qui n'ont pas d'identifiant. Par le jeu des insertions et des recherches, certaines parties du réseau se spécialisent dans certains types de fichiers par l'adaptation dynamique de leur table de routage. Il n'y a cependant pas la connaissance globale de cette spécialisation et certains pairs peuvent ne pas trouver un fichier demandé. La gestion de l'espace de Freenet est simple, puisque lorsqu'un pair n'a plus de place, les fichiers les plus anciennement accédés sont supprimés : Freenet se comporte comme un système de cache. Dans OceanStore et InterMemory, le mécanisme d'indirection permet de placer les données sur n'importe quels pairs. Dans OceanStore ou Freenet, le routage des requêtes est effectué dynamiquement, i.e si un pair reçoit deux requêtes portant sur le même objet, elles suivront un chemin différent. En effet, dans Freenet, lorsqu'une réponse à une requête arrive à un pair, celui-ci conserve une copie du fichier, une deuxième requête trouvera donc plus rapidement une réponse. Dans OceanStore, ce sont des liens vers l'actuel possesseur du fichier qui est créé tout le long du chemin de la réponse. La encore, les pairs apprennent au fur et à mesure de leur utilisation.

En ce qui concerne la pérennité, Freenet n'offre qu'une faible garantie : seuls les fichiers les plus demandés ont une forte probabilité de survie. Pour InterMemory et OceanStore, les garanties concernant la pérennité des données sont basées sur des systèmes de redondance sophistiqués à base de fragmentation redondante et de dispersion que nous présentons plus en détail par la suite dans le chapitre 3.

En résumé, la technique de routage des requêtes et la méthode de redondance sont deux éléments qui caractérisent les systèmes de stockage pair à pair. Pour ce qui est du routage des requêtes, une forme générale se dégage : les requêtes sont dirigées suivant un identifiant unique, ou considéré comme tel, qui caractérise un fichier. Chaque pair est spécialisé pour une certaine classe d'identifiants de fichier.

2.3 Tables de hachage distribuées : DHT

Les premiers systèmes de stockage avaient une architecture monolithique, i.e. les différentes fonctionnalités du système étaient interdépendantes. La tendance actuelle pour la conception de systèmes de fichiers pair à pair est de se baser sur une architecture à trois niveaux (Fig 2.2). Cette architecture a été introduite par CFS (Chord File System [14]).

FIG. 2.2 Schéma d'une application P2P basée sur une DHT

	Dénomination	Objectif
Niveau 2	Application	Utilisation des données
Niveau 1	DHT	Pérenniser les données
Niveau 0	Overlay	Localiser les données

Le but de cette architecture est de fournir une abstraction de table de hashage distribuée DHT (Distributed Hash Table).

Le premier niveau propose un mécanisme de routage entre les pairs par un sur-réseau tolérant aux pannes, nommé overlay. Quelques exemples d'overlays : Chord [73, 13], Pastry [64], Tapestry [79] (overlay utilisé par OceanStore), CAN [58].

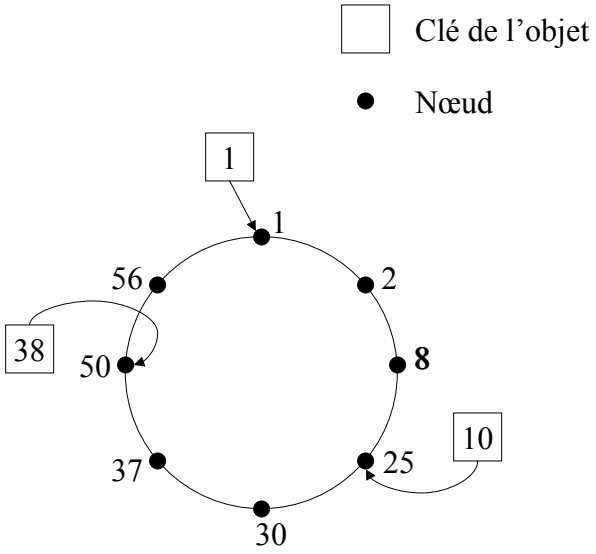
En général, le principe des overlays est que chaque pair est identifié au sein d'un grand espace de noms. La génération des identificateurs et des clés est assurée par une fonction de hachage sur respectivement les fichiers identificateurs et les fichiers à stocker identifiés par les clés[1]. Cette fonction assure avec une grande probabilité que les identifiants et clés sont uniques et uniformément répartis dans l'espace de noms. Pour router les messages, chaque pair maintient une table de routage avec les identificateurs d'autres pairs et leurs adresses IP.

Le deuxième niveau implémente un dictionnaire distribué et redondant sur l'overlay. Ce dictionnaire est nommé DHT : chaque entrée de ce dictionnaire est composée d'une clef et d'un objet associé (e.g. le fichier). L'objet est inséré dans la DHT qui le réplique afin d'assurer un certain niveau de tolérance aux pannes.

Au sein de la DHT, il existe une fonction qui projette l'espace des clefs du dictionnaire dans l'espace de noms des pairs, par exemple l'identité si les deux espaces de noms (celui de l'overlay et la DHT) sont identiques. Ainsi, chaque pair est responsable d'un ensemble de clés. Lors de l'insertion d'un nouveau objet dans la DHT, le pair qui le stockera est celui qui sera atteint par l'algorithme de routage en fonction de la clef projetée de

[1] La fonction de hachage utilisée par la plupart des overlays est SHA-1

FIG. 2.3 Overlay Chord avec insertion de 3 clés



l'objet. En général, l'algorithme de routage désigne le pair qui à l'identificateur le plus proche de la projection de la clef (exemple voir figure 2.3).

L'objet inséré est répliqué par la DHT pour faire face à la disparition de pairs. Typiquement, chaque pair maintient une liste des pairs qui ont un identifiant "voisin" (au sens de l'algorithme de routage) dans l'espace de noms, l'objet est alors dupliqué sur k pairs parmi ses pairs voisins (k paramètre du système).

Pour interagir avec la DHT, l'utilisateur dispose d'un ensemble simple de primitives : search, put, get, parfois free. Le rôle de la DHT est donc de gérer les données stockées dans le réseau. Pour cela, elle s'appuie sur un overlay pour connaître et localiser les pairs du réseau.

Finalement, le dernier niveau s'appuie sur les deux précédents (overlay et DHT) et est responsable de l'organisation logique des données.

Exemple d'overlay : Chord

Chord est un protocole qui a pour fonction de localiser une donnée dans un réseau constitué d'un ensemble de pairs, et plus précisément de localiser le pair qui possède la clé associée à la donnée. Dans un réseau Chord, les pairs sont organisés en un anneau virtuel de N pairs et possèdent un identifiant unique $\in [0, M[$, $M >> N$. De même pour les données stockées dans le réseau, qui sont identifiées de manière non équivoque.

Chord effectue un mapping entre la clé de la donnée et la clé du pair qui doit posséder cette donnée. En effet, une donnée d, représentée par une clé $H(d)$, est assignée au pair n tel que l'identifiant de n, $H(n)$ soit égal à $H(d)$ ou que $H(n)$ suit directement la valeur de $H(d)$ dans l'anneau (Fig. 2.3). On dit alors que n est le successeur de $H(d)$ (noté $successeur(H(d))$).

Chaque pair n a une vision locale du système. Il connaît :

1. son prédécesseur p : pair dont l'identifiant précède celui de n sur l'anneau.
2. son successeur s : pair dont l'identifiant suit celui de n sur l'anneau.
3. une table de routage de $log(N)$ entrées : l'entrée i correspond au pair p de l'anneau successeur de la clé $Id(n) + 2^{i-1}$, autrement dit tel que $Id(p)$ appartienne à $[2^i, 2^{i+1} - 1]$ avec $0 \leq i < log(N)$.
4. une liste de successeurs de taille $r = O(log(N))$: correspond aux r pairs qui succèdent immédiatement n.

Chord dispose d'une fonction de recherche lookup, dont le but est de trouver le pair n, détenteur d'une clé $H(d)$, (successeur($H(d)$)). Pour cela, il suffit de trouver le pair p prédécesseur de n, où n est successeur($H(d)$) : si un pair u contacte un pair v, tel que $H(d)$ appartienne à $[H(v), \text{successeur de } H(v)i]$, alors le successeur de v est successeur $(H(d))$.Dans le cas contraire, la requête lookup est envoyée au pair w, tel que $H(d) - H(w)$ soit minimal.

Avec une grande probabilité, le nombre de pairs contactés pour trouver le successeur d'un identifiant dans un réseau de N pairs est en $O(log(N))$.

La fonction de recherche n'est pas seulement une fonctionnalité offerte à la couche supérieure, mais aussi utile au fonctionnement intrinsèque de Chord. En effet, chaque pair n du réseau met à jour sa table de routage en exécutant un lookup. Pour chaque voisin i de sa table de routage, il effectue un lookup de $H(n)+2^{i-1}$, pour connaître le pair responsable de $H(n)+2^{i-1}$.

Dans un réseau dynamique Chord, les règles suivantes doivent toujours être respectées : chaque pair successeur est correctement maintenu et pour chaque clé k, le $successeur(k)$ est le pair responsable de k.

Ces deux invariants assurent la validité du lookup. En effet, le lookup s'arrête lorsque le pair p prédécesseur de successeur(k) est contacté. Si le lien successeur n'est pas maintenu, Le lookup peut ne pas s'arrêter sur p, et la validité du résultat retourné n'est plus garanti.

Dans la première phase, en supposant que le pair n connaisse un pair m déjà inséré dans le système, n initialise :

– son prédécesseur : à nil
– son successeur : par recherche de la clé $(H(n)$. En effectuant cette procédure, lookup($H(n)$) retourne le pair p responsable de la clé

$H(n)$. p est le premier pair dont l'identifiant $H(p)$ est supérieur ou égal à $H(n)$, autrement dit le pair du système dont l'identifiant suit directement celui de $H(n)$. p est donc le successeur de n.

– sa table de routage : par la procédure de lookup sur chaque entrée.

Dans une deuxième phase, chaque pair exécute de manière périodique un processus de stabilisation qui a pour but de rétablir la consistance des liens successeurs et prédécesseurs.

Exemple de DHT : PAST

PAST est un dispositif de stockage de données DHT développé conjointement par l'université de Rice et Microsoft. Le principe est que les pairs et les fichiers possèdent des identifiants uniques. Les fichiers sont stockés sur les pairs dont l'identifiant est le plus proche de l'identifiant du fichier. Le mécanisme de routage tolérant aux pannes est pastry [64].

PAST met à disposition de l'utilisateur les trois primitives de stockage, suivantes :

1. H(d) = insertion(k, file, Owner) qui stocke l'objet $file$ sur k pairs distincts ayant des identifiants les plus proches de $H(d)$ (la clé générée par SHA du fichier $file$).
2. d = lookup(H(d)) qui retourne une copie de l'objet d ayant comme identifiant $H(d)$.
3. reclaim($H(d)$, owner) qui récupère l'espace stocké par les k répliques de l'objet d. A l'instar d'un delete, reclaim ne garantit pas que le fichier ne soit pas disponible.

Il est de la responsabilité de PAST de maintenir les k répliques de chaque objet dans les k pairs les plus proches numériquement (en fonction des identifiants) et d'équilibrer l'espace de stockage disponible à travers le réseau. La pérennité des données est donc assurée par la réplication.

Dans PAST, le placement des objets et par conséquent le routage est passif dans le sens où les pairs n'apprennent pas où se trouvent les fichiers. Si un pair reçoit deux requêtes portant sur le même objet, elles suivront le même chemin.

Le système statique de PAST complique l'équilibrage de charge car les fichiers doivent être stockés sur le pair dont l'identifiant est le plus proche. Lorsqu'un pair n'a plus de place, les auteurs de PAST proposent plusieurs solutions pour y remédier [65] : dans la première, le pair qui n'a plus de place demande à ses voisins (des pairs dont l'identifiant est proche du sien) de stocker les fichiers pour lui, le pair d'origine conservant un lien vers les pairs qui stockent réellement le fichier ; une seconde méthode, plus radicale, consiste à changer l'identifiant du fichier.

2.4 Données mutables et écrivains multiples

Il existe des systèmes de stockage pair à pair disposant de mécanismes de modification des fichiers. Les premiers systèmes de stockage pair à pair étaient essentiellement des systèmes à écriture et rédacteur unique : les données stockées ne sont pas modifiables dans le temps. Les systèmes OceanStore, Ivy [44] ou Pastis [46, 47] sont des systèmes répartis multi-écrivains. Les systèmes, tels que Ivy ou OceanStore, mettent en œuvre des mécanismes permettant de modifier les fichiers tout en conservant les versions précédentes (versioning) : les modifications sont effectuées au niveau logique, celui de l'organisation des fichiers.

OceanStore utilise une technique de type "copy-on-write" dans laquelle les fichiers sont découpés en blocs et où une version d'un fichier est une liste des blocs qui le composent. Lorsque la dernière version du fichier est modifiée seul les blocs modifiés sont dupliqués et mis à jour : la nouvelle version correspond à une nouvelle liste qui intègre des liens sur ces nouveaux blocs.

Dans OceanStore les modifications d'un fichier par un écrivain passent par un "réplicat primaire" (primary replica) qui est associé à chaque fichier. Le rôle de ce réplicat primaire est de contrôler les droits d'accès de l'écrivain et de valider puis de propager les modifications sur les différents réplicats du fichier, il sérialise ainsi les opérations effectuées. Les droits d'accès sont représentés par une liste (ACL, Access Control List). Le réplicat primaire est implémenté par un ensemble de pairs qui mettent en œuvre un protocole Byzantin pour éviter qu'un pair malveillant n'interdise l'accès aux données ou modifie celles ci.

Ivy est un système qui propose une approche différente basée sur des mécanismes de journalisation qui ont été proposés par le système de fichiers distribués à écrivains multiples Zebra [31]. Au lieu de stocker les données dans une structure ad hoc, ce sont les opérations de mise à jour des fichiers (créations, écritures, suppressions, etc...) qui sont stockées de manière chaînée dans la DHT (les logs). Les fichiers étant obtenus en "rejouant" toutes les opérations effectuées. On peut obtenir une version précédente en omettant les dernières opérations.

Dans le cas d'écrivains multiples, chaque écrivain gère sa liste de modifications. Chaque opération est estampillée par un vecteur d'horloges représentant l'état courant des opérations des différents écrivains. Ces vecteurs d'horloges permettent de fusionner toutes les opérations de mise-à-jours en assurant la cohérence séquentielle de celles-ci par un algorithme de type "horloge de Lamport". Parce que ce mécanisme est coûteux, Ivy propose de générer l'ensemble des fichiers à intervalles réguliers.

Pastis est un système de fichiers réparti multi-écrivain. Il utilise le mécanisme de réplication des données pour assurer la persistance des données. Afin de garantir l'authenticité et l'intégrité des données, Pastis et OceanStore utilisent des techniques cryptographiques. Dans Pastis, pour qu'une écriture soit valide, elle doit être accompagnée d'un certificat émis par le propriétaire du fichier, autorisant l'écrivain à écrire sur le fichier.

Pastis utilise le protocole de routage Pastry et la DHT PAST pour le stockage des données. L'emploi de Pastry/PAST permet, grâce à leurs bonnes propriétés de localité, de minimiser la latence des accès réseau. Afin d'optimiser les performances du système, Pastis utilise un modèle de cohérence relâché, nommé "read-your-writes". Ce modèle de cohérence est plus souple en ce qui concerne la récupération d'une version d'un fichier en vue d'une modification.

Contrairement à d'autres systèmes de fichiers P2P multi-écrivain, tel qu'OceanStore, Pastis n'utilise pas de protocole lourd comme BFT [8] (Byzantine Fault Tolerance) pour valider les écritures. Cela facilite donc le passage à l'échelle du nombre d'utilisateurs.

A l'opposé de ces systèmes, Ivy évite la validation des écritures concurrentes par le mécanisme décrit précedemment. Cependant, la lecture d'un fichier entraîne la recherche des dernières modifications apportées au fichier à travers le parcours de l'ensemble des logs.Donc, Ivy ne passe pas à l'échelle sur le nombre d'écrivains.

A noter qu'il existe un prototype de Pastis codé en Java, utilisant une implémentation open source de Pastry/PAST. Les premières mesures de performance suggèrent que Pastis n'est qu'à peine deux fois plus lent que NFS [66], à configuration comparable. Tandis que'Yvy ou OceanStore sont de deux à trois fois plus lents que NFS.

2.5 Conclusion

Le stockage pair à pair est actuellement un domaine de recherche très actif qui arrive à maturité : il existe aujourd'hui quelques implémentations disponibles. Cependant les expérimentations qui ont été menées l'ont été à petite échelle en générale [59, 44]. Il faudra valider les solutions par des simulations et des expérimentations à grande échelle. D'autre part de nombreux problèmes restent encore à résoudre. En particulier ceux concernant le contrôle de l'usage des ressources et la sécurité.

L'ensemble des pairs coopèrent en échangeant leur espace de stockage. Il faut introduire des mécanismes pour garantir une équité dans l'usage de l'espace fourni. Typiquement l'espace externe consommé par un pair cor-

respond à l'espace qu'il fournit à l'ensemble de la communauté. De nouveaux mécanismes de transactions entre les pairs basés sur les théories économiques ont été introduits pour assurer une gestion efficace de l'espace [12, 24]. Un exemple d'un tel contrôle économique est celui du système de diffusion pair à pair MojoNation.

Dans le cadre du stockage, les questions de sécurité restent primordiales. Si la plupart des systèmes proposés introduisent des mécanismes de cryptographie et de signature électronique pour garantir la confidentialité et l'origine des données, la plupart font l'hypothèse que les pairs sont dignes de confiance. Seul OceanStore ne fait pas confiance aux pairs et intègre donc des mécanisme de consensus distribué et propose d'intégrer des mécanismes d'auto-surveillance des différents pairs. En effet, si on introduit des pairs malveillants qui ne respectent pas totalement le protocole, ou si l'ordinateur hôte corrompt les données gérées par le pair, alors l'intégrité du système de stockage peut être compromise. Différents articles proposent une étude des différents types d'attaque, en particulier les attaques de l'intérieur sur le sur-réseau de routage [9, 71].

Chapitre 3

Pérennisation des données : techniques de redondance

Dans les systèmes de stockage pair à pair, la pérénité des données est un enjeu important. En effet, lorsque la durée de vie des pairs est inconnue, il est nécessaire d'être fortement tolérant aux pannes.

La technique classique pour assurer une certaine pérennité est de faire appel à la redondance des données comme dans les DHT. En effet, stocker une donnée sur un seul pair est relativement risqué puisque ce pair peut disparaître à tout moment. Pour cela, il faut que les informations soient redondantes pour qu'il soit toujours possible de récupérer les données.

La technique la plus simple est la réplication des données sur différents pairs, c'est le principe utilisé par Freenet [11]. Pour faire face à r pannes, il sera nécessaire de dupliquer r fois les données. On note r le facteur de réplication. En d'autres termes r est le nombre de machines stockant une donnée qui peuvent être défaillantes simultanément sans que l'information soit perdue. Si $r+1$ pairs venaient à tomber en panne alors la donnée serait perdue. Pour $r = 1$, un seul réplicat, la tolérance est la défaillance d'un seul pair, pour $r = 10$, elle sera de 10 pairs.

D'autres techniques de redondance qui minimisent l'espace de stockage ont été développées, elles sont basées sur les mécanismes de codes correcteurs [7], [77]. Ces mécanismes sont souvent nommés "mécanisme de redondance avec fragmentation". Le principe générique à ces mécanismes est de fragmenter les blocs de données en s fragments de même taille. Ensuite, à partir de ces fragments, r fragments de redondance sont calculés. Les $s + r$ fragments d'un bloc de donnée sont tels qu'à partir de s fragments quelconques il est possible de reconstituer le bloc d'origine. L'ensemble des fragments d'un même bloc sont disséminés sur des pairs

distincts selon la technique de dispersion de Rabin [52]. Le système tolère donc r défaillances. La méthode usuelle pour déterminer les fragments redondants est basée sur le codage de Reed Solomon [48]. A noter que la réplication n'est en fait qu'un cas particulier où $s = 1$ et r est le nombre de réplicats.

Le ratio $\frac{s}{s+r}$ détermine l'espace utile, c'est à dire le rapport entre la taille de l'information et l'espace de stockage. Par exemple, dans le cas d'une seule réplication ($s = 1$, $r = 1$), l'espace utile est de $50\% = \frac{1}{1+1}$. Pour 9 réplicats ($s = 1$, $r = 9$), il n'est plus que de 10%. Dans le cas où $s = 9$ avec le même facteur de tolérance $r = 9$, l'espace utile est alors de 50%. La fragmentation redondance des blocs permet, pour une tolérance équivalente à la réplication, de gagner en espace utile et donc de stocker plus d'informations dans le système.

Par la suite, nous verrons la redondance dans les technologies RAID [38, 63], puis nous étudierons différents types de codes correcteurs existants, tel que Shamir [70] qui introduit aussi des mécanismes de cryptographie, Reed Solomon le code correcteur le plus couramment utilisé, et enfin Tornado [5] code conçu à l'origine pour la diffusion de données satellitaires.

3.1 Redondance dans les technologies RAID

La technologie RAID (Redundant Array of Inexpensive Disks, parfois Redundant Array of Independent Disks) est principalement utilisée pour optimiser les capacités, la vitesse d'accès et la tolérance aux pannes des disques durs avec la répartition des données sur plusieurs disques. Les disques assemblés selon la technologie RAID peuvent être utilisés selon différents niveaux de RAID variant de 0 à 7. Certaines implémentations du RAID 7 utilisent des techniques de type Reed Solomon présentées après une description des niveaux RAID les plus utilisés.

RAID niveau 0 : zéro tolérance

En mode RAID 0 les données sont écrites par bandes (stripes) distribuées de façon cyclique sur les disques pour offrir une vitesse de transfert plus élevée. Ainsi, un disque de capacité supérieure aux autres ne pourrait pas être entièrement rempli. Ce niveau n'intègre pas de tolérances aux pannes. La capacité de stockage est définie par la taille de l'unité la plus petite multipliée par le nombre d'unités.

RAID niveau 1 : la réplication

Le RAID 1 a pour but de dupliquer l'information à stocker sur plusieurs disques (nommé aussi mirroring, shadowing ou duplexing). La tolérance aux pannes est la plus grande possible, il suffit d'un seul disque en fonctionnement pour accéder à l'intégralité des informations stockées. Les améliorations sont faites uniquement pour la lecture répartie sur les disques.

RAID niveau 3

Le niveau 3 (disk array with bit-interleaved data) propose de stocker les données sous forme d'octets sur chaque disque et de dédier un des disques au stockage d'un bit de parité. La figure 3.1 présente un exemple de découpage en RAID 3. D_i représente le i^{eme} disque et o_i le i^{eme} octet de données.

FIG. 3.1 Exemple de découpage en RAID 3

D_1	D_2	D_3	D_4
o_1	o_2	o_3	$o_1 \oplus o_2 \oplus o_3$
o_4	o_5	o_6	$o_4 \oplus o_5 \oplus o_6$
o_7	o_8	o_9	$o_7 \oplus o_8 \oplus o_9$

La tolérance aux pannes est ici d'un seul disque, mais avec de bonnes performances et une capacité de stockage importante.

RAID niveau 5

Le niveau 5 (disk array with block-interleaved distributed parity) est similaire au niveau 4 : la parité est calculée au niveau d'un secteur. En revanche, elle est répartie sur l'ensemble des disques, pour pallier le déséquilibre d'écritures. La figure 3.2 présente un exemple de découpage en RAID 5. D_i représente le i^{eme} disque et b_i le i^{eme} bloc de données. Les blocs de parités sont $p_{1-2-3} = b_1 \oplus b_2 \oplus b_3$, $p_{4-5-6} = b_4 \oplus b_5 \oplus b_6$ et $p_{7-8-9} = b_7 \oplus b_8 \oplus b_9$.

De cette façon, RAID 5 améliore grandement l'accès aux données (aussi bien en lecture qu'en écriture) car l'accès aux bits de parités est réparti sur les différents disques de la grappe.

Le mode RAID 5 permet d'obtenir des performances très proches de celles obtenues en RAID 0, tout en assurant une tolérance aux pannes

FIG. 3.2 Exemple de découpage en RAID 5

D_1	D_2	D_3	D_4
b_1	b_2	b_3	p_{1-2-3}
b_4	b_5	p_{4-5-6}	b_6
b_7	p_{7-8-9}	b_8	b_9

élevées, c'est la raison pour laquelle c'est un des modes RAID les plus intéressants en terme de performance et de fiabilité. L'espace disque utile est le même qu'en RAID 3.

3.2 Shamir

Pour assurer la confidentialité des données, il est indispensable d'introduire des méthodes de cryptage. Nous allons voir comment inclure cette sécurité au sein du mécanisme de redondance.

L'algorithme proposé par Adi Shamir dans [69] est un algorithme de cryptographie permettant, à partir d'une donnée D, de créer n données cryptées telles qu'il en faut au moins $k < n$ pour reconstruire la donnée d'origine. La confidentialité est alors assurée tant qu'un utilisateur ne peut pas récupérer k fragments. Le code de Shamir fait intervenir des polynômes d'interpolation et se base sur le théorème d'interpolation de Lagrange suivant :

THÉORÈME 1
Soit k points (x_i, y_i) dans un plan tel que tous les x_i sont différents, alors il n'existe qu'un seul et unique polynôme $q(x)$ de degré $k-1$ tel que $q(x_i) = y_i$ pour tout i.

Considérant que la donnée D est stockée sous forme de bits, il est possible de s'abstraire du format de son contenu et de la considérer comme un nombre. Soit q un polynôme de degré $k-1$ dont les coefficients sont choisis aléatoirement $q(x) = a_0 + a_1 * x^1 + ... + a_{s-1} * x^{k-1}$ et tel que $a_0 = D$.

A partir de ce polynôme, on détermine n valeurs telles que pour toutes valeurs $i \in [1..n] : A_i = q(i)$.

Ainsi, à partir de n'importe quel sous ensemble d'au moins k A_i (avec leurs indices), il sera possible par interpolation de Lagrange de retrouver les coefficients du polynôme q (théorème 1). Il ne reste plus qu'à calculer $q(0) = a_0 = D$ pour retrouver la donnée originelle.

Cet algorithme est utile dans le cas où l'on souhaite rajouter une couche

cryptographique au principe de redondance. Cependant, il faut noter que la taille des éléments D_i est du même ordre que celle de D, donc au point de vue espace utile nous nous retrouvons dans la même situation que la réplication.

3.3 Reed Solomon

Dans Reed Solomon, les blocs de données de taille T sont découpés en s fragments de taille $\frac{T}{s}$. A partir de ces fragments sont calculés r fragments de redondance, eux aussi de taille $\frac{T}{s}$. Les r fragments sont tels qu'à partir de s fragments quelconques parmi les $s + r$ il est possible de reconstituer le bloc initial. Les $s + r$ fragments seront bien évidemment placés sur $s + r$ nœuds différents selon la technique de dispersion de Rabin [52]. Autrement dit, tant qu'il reste s fragments dans le système, le bloc de données pourra toujours être récupéré. Un bloc peut donc supporter jusqu'à r défaillances parmi les $s + r$ nœuds qui l'hébergent.

Considérons le bloc de données B vu comme un vecteur $B = (b_j)_{j\in[1..s]}$ de dimension s où les b_j sont les fragments. A partir de B, on crée un vecteur $E = (e_i)_{i\in[1..(s+r)]}$ de dimension $s + r$ dont les éléments seront les fragments à disséminer (incluant les fragments nécessaires à la redondance). Ceci correspond à l'étape d'encodage décrit ci dessous.

Le vecteur E est déterminé par une matrice A de dimension $(s + r) \times s$ telle que s vecteurs lignes quelconques de A sont linéairement indépendants.

Soit $A = (a_{(i,j)})_{i\in[1..(s+r)],j\in[1..s]}$ une matrice $(s + r) \times s$ telle que quels que soient s vecteurs lignes de A, ces s vecteurs lignes sont linéairement indépendants. Alors, le produit AB donne un vecteur de dimension $s + r$.

$$AB = E \tag{3.1}$$

$$\begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1s} \\ a_{21} & a_{22} & \cdots & a_{2s} \\ \vdots & \vdots & \ddots & \vdots \\ a_{s1} & a_{s2} & \cdots & a_{ss} \\ \vdots & \vdots & \ddots & \vdots \\ a_{(s+r)1} & a_{(s+r)2} & \cdots & a_{(s+r)s} \end{pmatrix} \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_s \end{pmatrix} = \begin{pmatrix} e_1 \\ e_2 \\ \vdots \\ e_s \\ \vdots \\ e_{s+r} \end{pmatrix}$$

Le vecteur B peut être reconstruit à partir de s fragments quelconques de E. Soit $(i_k)_{k\in[1..s]}$ les indices des s fragments de E que l'on va utiliser pour la reconstruction. Appelons E' le vecteur formé par ces fragments : $E' = (e_{i_k})_{k\in[1..s]}$.

On construit ensuite la matrice A' à partir des lignes de A correspondantes aux fragments de E'. A' est une matrice carrée $s \times s$. Par construction de E [3.1], on trouve que A', B et E' sont liés par la relation :

$$A'B = E' \tag{3.2}$$

Ce qui nous intéresse est de retrouver B à partir de A' et E'. Par hypothèse, s vecteurs lignes de A sont linéairement indépendants, il en est donc de même pour A'. Ainsi, par construction, A' est inversible et nous avons donc :

$$A'^{-1}E' = B \tag{3.3}$$

$$\begin{pmatrix} a_{i_1 1} & a_{i_1 2} & \cdots & a_{i_1 s} \\ a_{i_2 1} & a_{i_2 2} & \cdots & a_{i_2 s} \\ \vdots & \vdots & \ddots & \vdots \\ a_{i_s 1} & a_{i_s 2} & \cdots & a_{i_s s} \end{pmatrix}^{-1} \begin{pmatrix} e_{i_1} \\ e_{i_2} \\ \vdots \\ e_{i_s} \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_s \end{pmatrix}$$

Une fois la matrice inverse calculée, le décodage n'est qu'une question de multiplication matrice vecteur, tout comme l'encodage.

Si l'on cherche à déterminer la matrice A, en évitant des calculs, on suit le cheminement suivant : généralement, si l'on pose $A = \begin{pmatrix} I_s \\ V \end{pmatrix}$ où I_s est la matrice carrée identité de dimension s, et V est une matrice de dimension $r \times s$ telle que A possède les qualités requises (par exemple, V est une matrice de Vandermonde ou de Cauchy). Les s premiers éléments de E seront les éléments de B.

$$AD = E \tag{3.4}$$

$$\begin{pmatrix} I_s \\ V \end{pmatrix} B = \begin{pmatrix} B \\ F \end{pmatrix} \tag{3.5}$$

Ainsi, tant qu'il n'y a pas de défaillance des supports stockant les s premiers éléments de E, il n'y a pas besoin de reconstruction pour accéder à la donnée B, une concaténation suffit. La reconstruction n'a alors lieu que dans le cas de défaillance sur au moins un des s premiers éléments.

En ce qui concerne la confidentialité des données, nous avons vu que les s premiers fragments de l'encodage sont exactement les s fragments initiaux du bloc de données. Cela nuit à la sécurité puisque posséder un de ces fragments permettrait à une personne mal intentionnée de connaître une partie du contenu de la donnée initiale. Cela vient du fait que la matrice d'encodage A (section 3.3) est composée en partie de la matrice identité. Mais si l'on choisit A telle que la matrice identité n'apparaît plus tout en conservant les propriétés nécessaires, par exemple une matrice de Vandermonde de taille $s \times (s + r)$, l'encodage du bloc de données donne $s + r$ fragments E_i tous distincts des fragments initiaux B_i. Ainsi, plus de risque de laisser des informations en clair. Par contre, il est nécessaire de procéder systématiquement au décodage. Cela implique donc un surcoût de calcul.

Finalement Reed Solomon est une technique de redondance qui s'avère très pratique pour l'économie de l'espace utile. De plus, c'est un code largement utilisé et éprouvé. L'inconvénient majeur de Reed Solomon reste dans le temps de calcul global moyen d'encodage et décodage. Ce temps reste toujours relativement élevé.

3.4 Tornado

Les codes de type Tornado ont été conçus pour la diffusion satellitaire. Comme pour Reed Solomon, les codes de type Tornado [38, 39, 40] protègent un groupe de s fragments par r fragments de redondance, le nombre de fragment suffisant pour reconstruire l'information est alors de $s + x$, avec $x = \epsilon * s$. La valeur de $1 > \epsilon > 0$ dépend du codage Tornado utilisé. En général, ϵ est proche de 0.

Le principe est de protéger les s fragments d'origine avec h fragments de redondance par des "ou exclusifs". Notons $B = \{b_1, ..., b_s\}$ les fragments d'origine, on peut alors générer un ensemble de fragments de redondance $A = \{a_1, ..., a_r\}$ avec $a_i = b_1^i \oplus ... \oplus b_{z_i}^i$, avec $z_i < r$. Les a_i sont définis de manière aléatoire tout en respectant certaines contraintes sur les z_i.

Ce codage, expliqué dans [39], est schématisé sous la forme d'un graphe biparti. Les sommets de B et de A représentent chacun une partie. Ce graphe est caractérisé par son nombre total d'arrêtes E et la distribution des degrés des arrêtes. Ainsi, on note λ_i le ratio du nombre d'arrêtes sortantes de degré i issues de B sur E. ρ_i le ratio du nombre d'arrêtes entrantes de degré i de A sur E. Un exemple de caractérisation d'un graphe est donné par la figure 3.3.

Les fragments de A peuvent ensuite eux-même être protégés de la même façon par d'autres fragments de redondance et ainsi de suite. On note β ($0 < \beta < 1$) le facteur réducteur de l'information pour chaque ni-

FIG. 3.3 Caractérisation des graphes dans Tornado

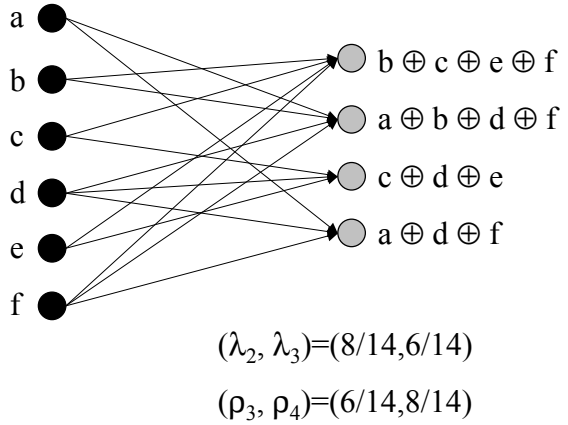
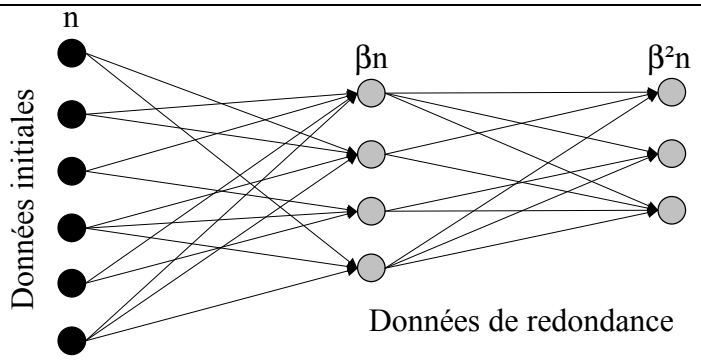


FIG. 3.4 Les niveaux dans Tornado



veau (graphe biparti), voir figure 3.4. Enfin, le dernier niveau est protégé par un code correcteur classique de type Reed Solomon par exemple.

Pour mesurer l'efficacité du codage, les auteurs ont créé une fonction qui calcule δ le taux de pertes acceptables avec une très forte probabilité. Cette fonction a pour paramètres d'entrées λ, ρ, β. Ainsi, les auteurs ont pu comparer l'efficacité de Tornado en fonction de la caractérisation et propriétés des graphes. Lorsque les graphes bipartis sont réguliers de degré d (λ_d=1,$\rho_{d/\beta}$=1), δ est petit. Lorsque les graphes bipartis sont irréguliers de degré moyen d, obtenu par heuristique, l'efficacité de ce codage est optimale avec λ défini à l'aide d'une somme harmonique H_d ($\sum_{i=1}^{i\leq d} \frac{1}{i}$) :

$$\forall i \in [2, d+1], \lambda(i) = \frac{1}{H_d(i-1)}$$

ρ suit une loi de poisson, dite loi des événements rares, de paramètre α. Ce

paramètre satisfait l'égalité suivante :

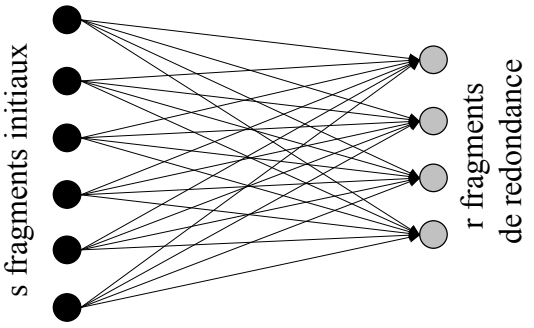
$$\frac{\alpha e^{\alpha}}{e^{\alpha}-1} = \frac{\beta H_d(d+1)}{d}$$

Et

$$\forall i \in [2, d+1], \rho(i) = e^{-\alpha}\frac{\alpha^{i-1}}{(i-1)!}$$

De cette manière, on obtient l'efficacité optimale de taux $\delta = \frac{\beta}{1+\frac{1}{d}}$.

FIG. 3.5 Caractérisation des graphes dans le codage Reed Solomon (exemple $s = 6$, $r = 4$)



On peut considérer que Reed Solomon est un cas particulier de Tornado, voir figure 3.5, où les graphes sont complets : le nombre d'arrêtes entrantes et sortantes sont fixes et de valeur maximale. Le code correcteur d'erreurs Tornado encode et décode beaucoup plus rapidement les informations, voir tableau comparatif 3.6 issu de l'article [6]. Tornado est environ 100 fois plus rapide sur les petits blocs de données et environ 10 000 fois plus rapide sur des gros blocs.

FIG. 3.6 Performances Tornado / Reed Solomon

Temps Encodage (en s), blocs de 1Ko		
Taille	Reed Solomon	Tornado
250 Ko	4.6	0.06
500 Ko	19	0.12
1 Mo	93	0.26
2 Mo	442	0.53
4 Mo	1717	1.06
8 Mo	6994	2.13
16 Mo	30802	4.33

Temps Décodage (en s), blocs de 1Ko		
Size	Reed Solomon	Tornado
250 Ko	2.06	0.06
500 Ko	8.4	0.09
1 Mo	40.5	0.14
2 Mo	199	0.19
4 Mo	800	0.40
8 Mo	3166	0.87
16 Mo	13829	1.75

En résumé, Tornado est un code correcteur très efficace en termes de performances et protection des données.

3.5 Efficacité des techniques de redondance

Plusieurs études théoriques ([76], [41]) dans le domaine du stockage pair à pair, ont permis de déterminer la probabilité qu'un bloc de donnée soit indisponible et le temps moyen théorique de la première perte d'un bloc *MTTF : Mean Time To Failure* [75] selon le mécanisme de redondance employé.

Bien qu'avec un facteur de redondance adapté, on peut résister à un grand nombre de défaillances, un mécanisme statique ne permet pas d'assurer la pérennité des données. En effet, si on considère que la durée de vie des pairs suit une loi classique de probabilité telle que la loi exponentielle, alors on a une forte probabilité d'avoir perdu plus de la moitié des pairs au bout du temps moyen de la durée de vie d'un pair. Pour assurer la pérennité, il est donc nécessaire d'introduire un mécanisme de réparation qui détecte et reconstruit les fragments de données perdus.

Dans [76], en prenant l'hypothèse qu'un mécanisme de réparation est associé au mécanisme de redondance et que ce processus scanne et reconstruit les réplicats des blocs perdus, la probabilité qu'un bloc soit disponible peut se calculer de la manière suivante :

Soit A, l'évènement qu'un bloc soit disponible. $P(A)$ la probabilité d'un tel évènement, n le nombre total de fragments, m le nombre de fragments nécessaires à la reconstruction, N, le nombre de pairs dans le système et M, le nombre de pairs indisponibles :

$$P(A) = \sum_{i=0}^{n-m} \frac{\binom{M}{i}\binom{N-M}{n-i}}{\binom{N}{n}}$$

La probabilité qu'un bloc soit disponible est égale au nombre de possibilités dans lesquelles on peut s'affranchir des fragments indisponibles sur les pairs indisponibles multiplié par le nombre de possibilités de récupérer les fragments disponibles sur les pairs disponibles, divisé par le nombre total de possibilités de récupérer tous les fragments sur tous les pairs.

Pour 1 million de machines (N=1M), et 10% sont actuellement indisponibles (M=100000), stocker 2 réplicats (r=2) fournit une probabilité qu'un bloc soit disponible avec une probabilité $P(A)$ de 0.99 tandis qu'avec un code correcteur (s=16, r=16), $P(A)$ est égale à 0.999999998.

Donc lorsque le but visé est la pérennité des données, un codage de type Reed Solomon semble plus approprié. Toutefois, il faut relativiser cette décision. D'autres facteurs, comme la disponibilité des pairs, peuvent influencer ce choix. La disponibilité est le fait que les pairs ne sont pas constamment connectés. Exemple : les pairs issus du grand public sont souvent connectés dans la journée, mais rarement la nuit.

Dans l'article [75], un modèle stochastique a été défini dans lequel on classifie les différents systèmes de type pair-à-pair en fonction de la disponibilité et des défaillances de ses pairs, et qui permet de déterminer l'efficacité des différents schémas de redondance (réplication simple, Reed Solomon, Shamir, Tornado, ...). Un résultat intéressant de cette étude est que les schémas sophistiqués de redondance, à priori meilleurs (tel que Reed Solomon), s'avèrent moins efficaces que la réplication classique pour les systèmes qui disposent d'un taux de disponibilité moyen.

3.6 Conclusion

Les techniques de redondance sont utilisées dans de nombreux domaines (stockage, transmission, audiovisuel). Dan notre cas, il est primordial de trouver la meilleure technique afin de garantir un haut degré de pérénité des données au sein de notre système de stockage. Pour cela, nous avons étudié différents type de codage et avons observé que Tornado est le meilleur code correcteur existant à ce jour.

D'autre part, la plupart des études sur l'efficacité des mécanismes de redondance font généralement l'hypothèse que les pairs sont non corrélés en ce qui concerne les pannes. Des études [78] ont montré que cette hypothèse n'est pas vérifiée en général. Des mécanismes pour détecter des corrélations entre les pairs ont été développées. Ils se basent sur la structure du réseaux et des observations en temps réels. Il faut alors distribuer les données de manière à éviter les corrélations trop fortes de pannes. Nous étudierons ce principe dans le chapitre 8.

Chapitre 4

Architecture fonctionnelle de Us

Le nombre d'ordinateurs PC augmente considérablement chaque jour, grâce à la démocratisation des équipements informatiques. La conséquence directe est qu'il existe une grande quantité de ressources telles que disque, mémoire ou temps de calcul inutilisés !

L'objectif du projet Us, "Ubiquitous Storage", est d'utiliser ces ressources disques inutilisées et d'offrir en supplément des fonctionnalités avancées, telles que pérennité et ubiquité des données. C'est dans le cadre de ce projet que nous avons conçu un système de stockage pair à pair [72, 53].

Mis bout à bout, ces deux termes peuvent paraître contradictoires, les environnements de type pair à pair sont par définition volatiles : on ne peut avoir aucune confiance sur la participation d'un pair à l'effort commun. Tandis que le stockage fait référence à des notions de pérennité, de constance dans le temps. Nous verrons donc par la suite comment contourner ce problème et utiliser au mieux les ressources de stockage disponibles.

Ce chapitre décrit le projet et dévoile l'architecture et les fonctionnalités du système de stockage pair à pair Us, en cours de développement au Laria de l'UPJV.

4.1 Description

Le système pair à pair Us offre un service de sauvegarde des données sur Internet. Chaque utilisateur dispose d'un espace personnel de stockage pérenne des données. Cet espace est issu de la mise en commun d'une partie de la mémoire disque de chaque utilisateur. Us est donc un système

mono-écrivain. Us est dédié au réseau de type asynchrone. Us se compose de deux entités principales : un Client qui est un demandeur d'espace et un Fournisseur qui alloue et gère l'espace. Son objectif premier est la pérennité des données. Us est un système de stockage bas niveaux, il peut être vu comme un disque dur virtuel. En fait, Us a le comportement d'un pilote de disque dur : c'est un espace de stockage dans lequel on peut stocker et effacer des blocs de données. Il peut être considéré comme une brique de base d'un système de fichiers ou peut être utilisé pour les accès aux données de programmes. Le Client Us peut donc être interfacé avec un système de fichiers spécifique, tel que NFS [66], CFS [15], pour permettre à l'utilisateur de gérer ses données plus simplement sous forme de fichier. De plus, Us est conçu pour être intéropérable avec d'autres systèmes de stockage, tel que IBP [49], PAST [16]. Les utilisateurs de Us peuvent utiliser une interface simplifiée (voir annexe A.2) pour stocker et récupérer leurs fichiers ou utiliser une interface dédiée de type système de fichier, nommée UsFS (voir chapitre 5). Cependant Us (et non UsFS) ne fournit pas de gestions complexes telles que gestion des méta-données, cohérence des données, gestion des droits d'accès. En résumé, Us fournit un espace de stockage dans lequel on peut stocker, lire et effacer des blocs de données. Us travaille donc avec des blocs immuables : on ne peut modifier les blocs. On efface les blocs qui contiennent des données obsolètes et de nouveaux blocs sont créés avec de nouvelles données. L'objectif principal de Us est le stockage pérenne de données brutes. Pour des raisons de portabilité, Us est écrit en Java.

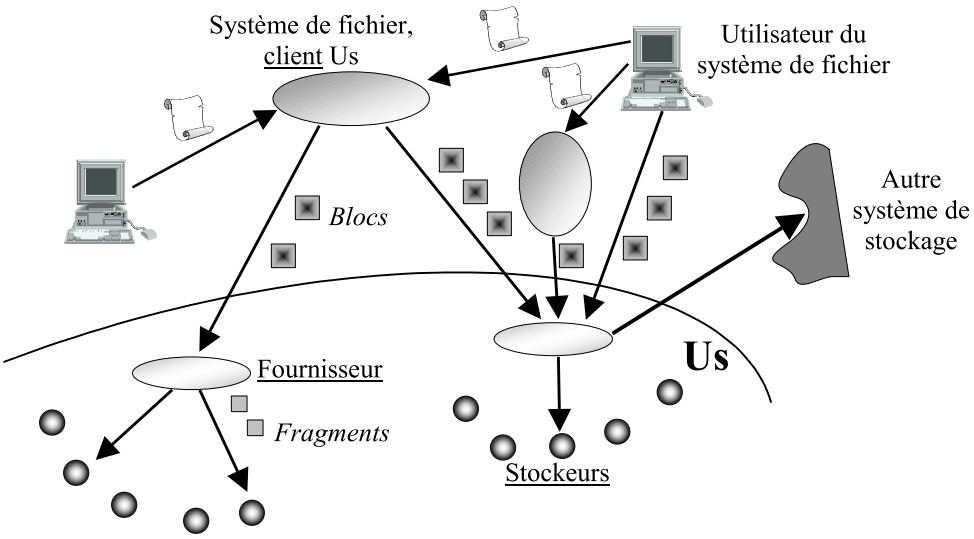
4.2 Architecture

Pour visualiser l'architecture globale de Us, se référer à la figure 4.1. Prenons un exemple de fichier distribué qui utilise Us : un utilisateur envoie son fichier au système de fichiers, lui même **client** de Us. Ce fichier est découpé en *blocs* et envoyé à un **fournisseur** de service Us. Ce dernier détient les informations sur les ressources disponibles et découpe les blocs en *fragments* en incluant des fragments de redondance Pour des raisons de facilité d'implémentation, notre prototype Us utilise l'algorithme de Reed Solomon pour générer les fragments de redondance. Une prochaine étape serait donc d'implémenter Tornado et d'intégrer ce code à la place de Reed Solomon, afin de gagner en performance. Finalement le fournisseur répartit ces fragments sur des pairs dédiés au stockage : les **stockeurs** qui sauvegardent physiquement les données de manière locale.

Donc, Us se compose de 3 types d'applications :

1. le **client** qui est l'interface utilisateur pour demander de stocker des blocs , de les récupérer ou de les effacer.

FIG. 4.1 Architecture globale de Us



2. le **fournisseur** qui est le fournisseur de service et c'est lui qui détient les informations sur les ressources et les gère.
3. le **stockeur** qui est le pair de stockage stockant les données sous forme de fragments.

Remarques : un pair peut-être à la fois client et stockeur. Le prototype de stockage Us dispose d'un contrôle centralisé (le fournisseur).

A présent, intéressons nous au fonctionnement de Us.

4.3 Fonctionnement

Us est écrit dans le langage Java pour des raisons de portabilité. Le but est ainsi de toucher un maximum de ressources. Pour que n'importe quel utilisateur disposant d'un ordinateur sous un OS quelconque puisse installer Us sans soucis.

Dans Us, les pairs communiquent entre eux. Cette communication est contrôlée et initialisée par le fournisseur. Le fournisseur a un rôle d'annuaire et authentifie les pairs communiquant.

D'un point de vue objet : Us dispose d'une classe, nommée **PeerInfo**. C'est elle qui contient toutes les informations permettant de communiquer avec un autre pair. Elle contient également un attribut statut qui indique l'état du pair : opérationnel, déconnecté, ou définitivement mort. Le prin-

cipe d'utilisation est le suivant : chaque pair construit son propre PeerInfo et l'envoie aux autres pairs pour qu'ils puissent communiquer entre eux.

Pour illustrer le fonctionnement du système de stockage Us, considérons les étapes suivantes : l'initialisation de Us où chaque pair s'enregistre auprès du fournisseur, l'écriture des données dans Us, la lecture de données dans Us, la tolérance aux pannes (gestion de la mort d'un pair).

Initialisation

Chaque nouveau stockeur Us contacte le fournisseur de service qui les enregistre en sauvegardant leur PeerInfo respectif et leur attribue un identifiant. Au sein du fournisseur de service se trouve un gestionnaire de ressources qui s'occupe aussi des allocations et attributions des espaces de stockages virtuels. Chaque stockeur sauvegarde localement son identifiant pour permettre lors d'une déconnexion temporaire de pouvoir réapparaître lors de sa prochaine connexion avec le même identifiant auprès du fournisseur qui grâce au gestionnaire vérifie l'existence de cet identifiant. Si cet identifiant existe dans le gestionnaire, alors le stockeur envoie juste son nouveau PeerInfo. Par contre si cet identifiant n'existe plus, alors le stockeur sera considéré comme nouveau, et par conséquent nouvel enregistrement et réattribution d'un nouvel identifiant. Lors d'une déconnexion / reconnexion d'un pair, son état passera juste de déconnecté à connecté sans incidence sur les données qu'il stockait. Il pourra ainsi à nouveau être disponible et assurer ses fonctionnalités de stockage.

FIG. 4.2 Détail des opérations du répartiteur Us

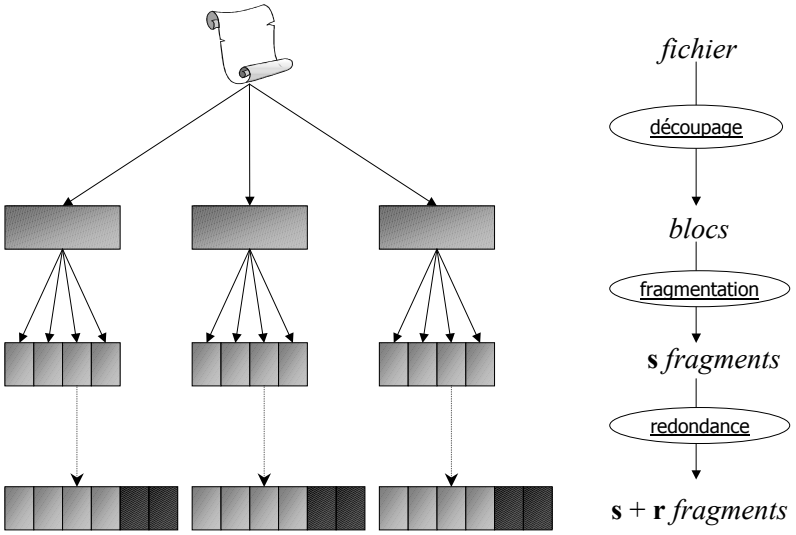
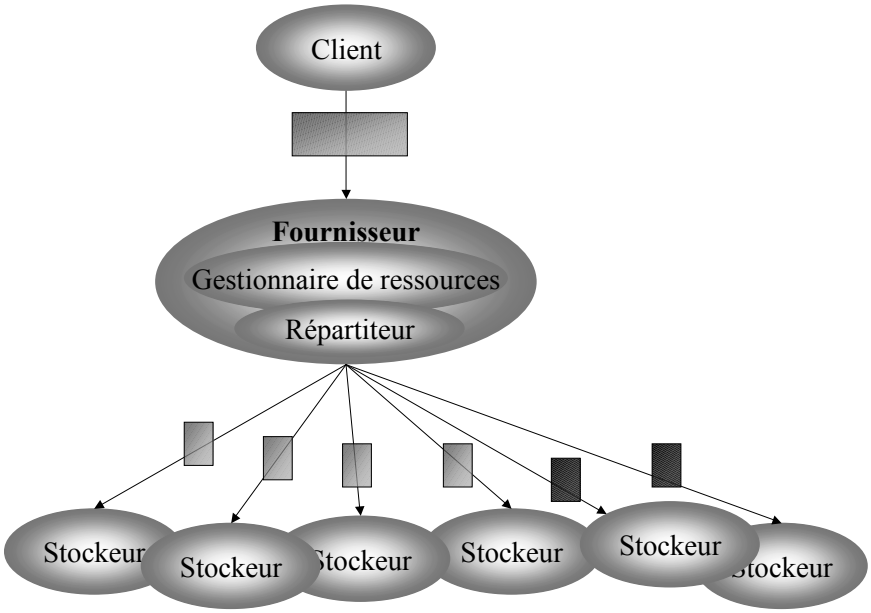


FIG. 4.3 Fonctionnement global de Us



Ecriture (PUT)

Lorsqu'un client contacte le fournisseur pour l'obtention d'un espace, le fournisseur contacte le gestionnaire qui attribue un espace qui est envoyé au client. A partir de ce moment lorsque le client veut stocker des blocs dans cet espace nouvellement créé : il émet une requête au gestionnaire pour obtenir une liste de stockeurs qui sera ensuite envoyée ainsi que le bloc à stocker à une entité secondaire du fournisseur, le **répartiteur** de fragments qui découpe le bloc à stocker en plus petits blocs nommés fragments, eux-mêmes répliqués suivant l'algorithme de Reed Solomon, voir figure 4.2. Tous ces fragments sont alors dispersés sur des stockeurs référencés par le gestionnaire, voir figure 4.3. Dans un premier temps, le choix de ces stockeurs est cyclique. Nous verrons dans les prochains chapitres 6, 7, 8 dédiés aux distributions, des choix plus intelligents et stratégiques prenant en compte divers critères. Le gestionnaire doit impérativement respecter la règle de stockage qui est : deux fragments d'un même bloc ne sont pas stockés sur le même stockeur afin de maximiser la tolérance aux pannes. Si un stockeur meurt pendant le transfert d'un fragment, alors un nouveau stockeur est donné par le gestionnaire pour stocker ce fragment. Par une approche intuitive de la phase d'écriture, on s'aperçoit que si le client et le répartiteur sont sur le même pair, on diminue les communications. Mais un problème de taille réside : l'accessibilité du répartiteur par les autres pairs reste problématique dans ce cas. D'où le

fait de laisser le répartiteur au sein du fournisseur dans un premier temps.

Lecture (GET)

Lorsqu'un client demande la récupération d'un bloc précis, il contacte un répartiteur qui récupère un nombre suffisant de fragments et reconstruit ce bloc suivant l'algorithme de Reed Solomon et le renvoie au client. Dans le système de stockage Us, c'est l'algorithme de Reed Solomon qui calcule les fragments r de redondance à partir des s fragments constituant le bloc d'origine. Lors de la reconstruction du bloc, il sera plus coûteux en temps si on ne peut obtenir tous les s fragments, car cette méthode va reconstruire les s fragments d'origine à partir de s fragments parmi les $s + r$ fragments. Pour la lecture, la possibilité d'échec est prise en compte, mais elle implique une notion de délai, le délai avant lequel on considère qu'il est impossible de récupérer le bloc. Le délai de lecture d'un bloc est variable, il dépend du temps de récupération des s fragments. S'il manque un ou des fragments s, on ajoute à ce temps, le temps de récupération du nombre de fragments manquants parmi les r fragments disponibles et la reconstruction des s fragments d'origine. Si cette dernière action échoue, un délai d'attente des stockeurs non connectés, disposant des précieux fragments, est appliqué. Passé ce délai, on considère que la lecture de ce bloc est un échec.

Tolérance aux pannes

La durée de vie d'un pair dans un système pair à pair est courte. Lorsqu'un stockeur disparaît, les fragments qu'il stockait doivent être reconstruits. Pour cela un répartiteur récupère un nombre suffisant de fragments des blocs endommagés. Puis, il demande au gestionnaire des nouveaux stockeurs. Ensuite, le répartiteur reconstruit les fragments manquants. Enfin, ces nouveaux fragments sont stockés sur les nouveaux stockeurs.

Pour détecter la panne d'un stockeur, on suppose qu'il existe un détecteur de pannes fiable qui gère les pannes de types franches et transitoires. En théorie, un tel détecteur ne peut exister (prouvé par Fisher Lynch Paterson en 1985 dans l'article [22]). En pratique, on considère qu'une panne sera toujours repérée. On peut imaginer un détecteur de pannes basé sur un modéle économique : ainsi dans le pire des cas, le système est prévenu de la panne par un appel de l'utilisateur. Par contre, à l'heure actuelle, les fautes de type byzantin ne sont pas prises en compte.

Deux stratégies pour la détection des pannes sont mises en place : la détection active du gestionnaire qui se base sur tous les stockeurs en

maîtrisant leurs temps de détection et le pulling au niveau du gestionnaire qui se base sur le temps depuis lequel un stockeur ne s'est plus connecté.

4.4 Utilisation

L'objectif de l'interface utilisateur de Us est de fournir un service Internet sous forme de requêtes simples, tout en restant proche de l'interface standard d'un disque dur. Pour stocker et gérer ses données, l'utilisateur de Us dispose de deux classes :

- **Broker**, pour la mise en relation du client avec un fournisseur de service distant.
- **Space**, pour représenter l'espace de stockage virtuel. Cet objet est composé d'un ensemble de blocs de taille fixe. La gestion de cet espace virtuel est indépendante de celle des fichiers.

Le Broker ne dispose que de deux méthodes. Les données nécessaires à sa construction sont écrites dans un fichier de configuration à l'installation. Les méthodes associées à ces classes sont :

- **Broker()**, constructeur par défaut qui met en relation le client pour son enregistrement auprès d'un fournisseur. Cet enregistrement correspond à l'attribution d'un identifiant client par le fournisseur.
- **malloc()**, méthode qui retourne au client un nouvel objet de classe Space représentant l'espace virtuel de stockage de la taille souhaitée. Cette méthode correspond donc à l'allocation mémoire du Space mis à disposition du client.

Un programme utilisateur du système Us doit contacter un Broker, qui est un gestionnaire de ressources, pour avoir un objet de classe Space, qui représente un espace de stockage virtuel. C'est par le biais de cet objet qu'il est possible de manipuler un ensemble de blocs de taille fixe.

Les méthodes à disposition de l'utilisateur sont les suivantes :

- **put()**, pour stocker un bloc.
- **get()**, pour récupérer un bloc préalablement stocké.
- **free()**,pour libérer un bloc préalablement stocké.

Les informations permettant de récupérer les blocs stockés sont enregistrées dans l'objet Space. La sauvegarde de cet objet sur disque local permet de récupérer les blocs ultérieurement.

Exemple

Afin de visualiser la simplicité d'utilisation du système Us, considérons une utilisation typique de Us. Dans cet exemple, un utilisateur va stocker un fichier dans Us et ensuite le récupérer.

L'utilisateur Us découpe son fichier en blocs. Puis, il les envoie sur Internet via Us. Pour récupérer son fichier, il doit récupérer tous les blocs et reconstruire le fichier.

Dans un premier temps, le client créé un objet broker qui va contenir toutes les informations nécessaires pour communiquer avec lui, IP, numéro de port :

```
Broker broker=new Broker();
```

Le client se met en contact avec le Broker qui va lui retourner le Space de taille FileSize qui représente dans ce cas la taille d'un fichier contenu dans FileIn que l'utilisateur souhaite stocker.

```
Space space=broker.malloc(FileSize);
```

Ensuite, le client stocke dans le Space chaque bloc de données de taille blockSize constituant le fichier. Si le nombre de blocs qu'il doit stocker est numBlocks, alors il stocke ses numBlocks blocs de données dans le Space, ce qu'il peut faire via l'objet Space.

```
for (i = 1;i ≤ inumBlocks;i++){
    buffer=fileIn.read(blockSize);
        space.put(i,buffer);
                }
```

Enfin, l'utilisateur lit le Space pour récupérer chaque bloc de données constituant le fichier désiré qui sera stocké physiquement dans le fichier FileOut. Il doit récupérer les numBlocks blocs de données constituant le fichier d'origine dans le Space, ce qu'il réalise grâce à la méthode get().

```
for (i = 1;i ≤ inumBlocks;i++){
        buffer=space.get(i);
  fileOut.write(buffer, blockSize);
                }
```

4.5 Problème de granularité

Tout d'abord, nous rappelons les différentes notions liées à l'utilisation du système de stockage Us. Un Space est un espace de stockage virtuel. Il est défini par un ensemble de x blocs. Un bloc est une concaténation de s fragments. L'utilisateur Us fournit un bloc en entrée à l'algorithme Reed Solomon qui en sortie créé les $s + r$ fragments résultant. Ce sont les fragments qui transitent sur le réseau.

Lorsqu'un utilisateur souhaite stocker un fichier, le fichier est découpé en blocs de taille fixe, et si la taille du fichier n'est pas un multiple de cette taille, le dernier bloc est tout de même rempli.

Maintenant, nous allons voir les différents paramètres du système Us. Les paramètres variables lors de l'utilisation du système sont en gras et les autres paramètres sont fixés au démarrage du système :

- fragmentSize : Taille d'un fragment
- Nb_Blocks : Nombre de Blocks dans un Space.
- s : Valeur de la fragmentation (Paramètre de l'Algorithme de Reed Solomon)
- r : Valeur de la redondance (Paramètre de l'Algorithme de Reed Solomon)
- Fact_Répli : Facteur de réplication = (s + r) / s
- **BlockSize** : Taille d'un Bloc = s * fragmentSize
- **SpaceSize** : Taille d'un Space = Nb_Blocks * BlockSize

Désormais, nous pouvons étudier l'impact des paramètres en fonction de leurs valeurs sur le système. Plus particulièrement sur les paramètres BlockSize, s,r qui sont les paramètres de premier ordre.

Pour le paramètre BlockSize, plus cette valeur est petite plus on va multiplier le nombre de blocs et donc le nombre de fragments et ainsi économiser de l'espace utile sur le dernier bloc du space qui sera non plein si BlockSize n'est pas un multiple de SpaceSize. De plus, on peut considérer que BlockSize représente la taille réelle minimale que peut avoir un fichier prêt à être stocké dans Us, car supposons que l'on est une taille de block de 1024 Ko, et que l'utilisateur souhaite stocker un fichier de 32 ko : si on exécute l'opération de stockage on perd 1024-32= 992 ko. De même il est nécessaire de prendre en compte la connexion de l'utilisateur mise à disposition pour paramétrer correctement cette valeur.

Pour le paramètre s, Plus cette valeur est petite, plus on va gagner du temps dans la reconstruction. L'algorithme de Reed Solomon est plus performant en terme de vitesse de calcul pour des petites valeurs de s ($s \leq 32$). Mais si s est petit, on perd en redondance. La cause est que l'on ne peut pas prendre $r = 16 * s$ par exemple, puisqu'il ne faut pas oublier le facteur de réplication qui dans ce cas approcherait le facteur 16 ! Et pour le paramètre r, sa valeur ne doit pas augmenter considérablement le facteur de réplication. Exemple : Si $s = r$, le facteur de réplication est égal à $s+s/s = 2$, et l'on tolère r pannes simultanées.

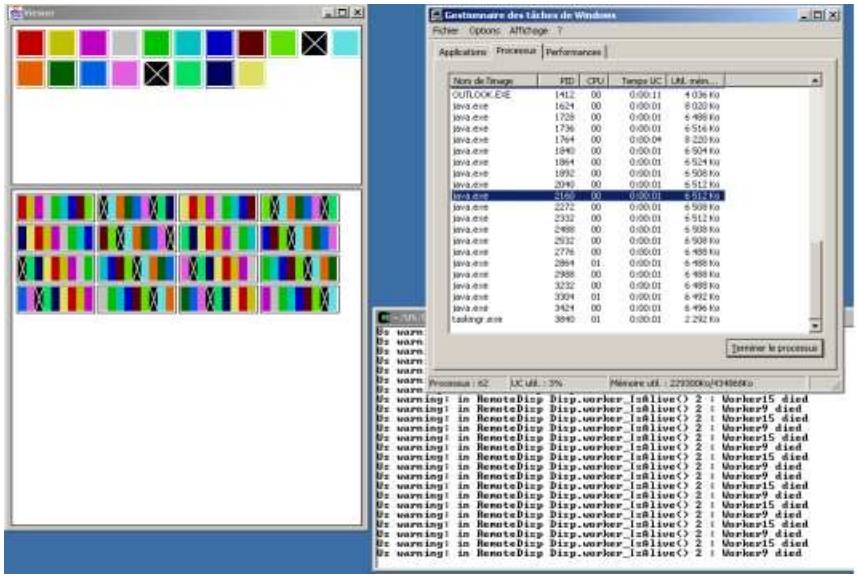
4.6 Déploiement

Nous avons envisagé trois modes de déploiement. Le premier mode est autonome, basé sur Java RMI [29]. C'est une bibliothèque qui offre une extrême simplicité d'utilisation, avec malgré cela des limites d'utilisation [53]. Le second mode est Internet/XtremWeb [21], pour utili-

ser Us à travers Internet en passant par XtremWeb. Ce dernier fournit principalement le gestionnaire de ressources. Le dernier mode est Internet/Jxta [74, 42, 1], qui procure un environnement de détection de ressources. Le but de ces différents modes de déploiement est de permettre à Us d'être multiplateformes pair à pair.

Us dispose également d'une interface graphique de monitoring rattachée au fournisseur Us, comme présenté à la figure 4.4 : les processus morts (en haut de la figure) sont représentés par une croix et induisent la perte de fragments (en bas) représentés pareillement. Lors de démonstrations, les processus sont tués en temps réél par le gestionnaire d'applications du système. Une version graphique plus élaborée de cette interface de monitoring existe, ainsi qu'une interface graphique cliente. Elles sont présentées respectivement dans les annexes A.1 et A.2.

FIG. 4.4 Moniteur du fournisseur Us



Mode autonome

Chaque pair doit lancer deux processus java : le **client**, c'est à dire l'interface utilisateur de Us, ainsi que le **stockeur**, qui est le processus de stockage contenant deux threads : un thread d'écriture sur le disque local et un thread de réparation. Indépendamment, le **fournisseur** doit être lancé sur un serveur.

Le mode de communication que nous avons choisi est Java-RMI [29] qui permet d'exécuter une méthode sur un processus distant. Il suffit de faire

tourner un processus de démon (rmiregistry) et d'y inscrire le processus appelé. Un thread est alors créé, rendant transparent les communications entre le démon et le processus. Cette bibliothèque intègre la gestion de l'envoi et réception des données, du contrôle du processus distant. Ce choix a été guidé par l'extrême simplicité d'utilisation du RMI. RMI est souvent critiqué pour ses piètres performances, cependant ces performances sont suffisantes pour le P2P, tel que l'indique l'article [26]. Nous avons pu vérifier que ces deux caractéristiques étaient exactes. En revanche, d'autres limites nous ont fait regretter ce choix :

- Résurrection des pairs : le démon a des difficultés à gérer les pannes des processus enregistrés . Il détecte bien que le processus est mort, mais il n'accepte pas toujours de le réenregistrer. Ainsi, il faut vérifié que l'on a bien tué tous les démons rattachés pour pouvoir le réinitialiser. Cette contrainte nous a particulièrement gêné lors des simulations de grandes tailles qui imposaient d'utiliser plusieurs stockeurs par processeur.
- Dérangement utilisateur : il restait un facteur que nous n'avions pas pris en compte, c'était le facteur humain . Le démon RMI peut générer plusieurs processus qui ne sont pas toujours détruits à la fin de l'exécution du processus java. Les utilisateurs qui acceptaient que leurs ordinateurs servent de stockeurs n'ont pas accepté cette "pollution" de leurs machines.
- Mode asynchrone inexistant : RMI ne prévoit pas de mode d'exécution asynchrone . Il faut le simuler à l'aide de thread chaque appel à une nouvelle méthode, ou redéfinir une classe héritée. Ainsi, un mode de fonctionnement plus raffiné annihile la simplicité de programmation qu'apporte la bibliothèque.
- Retour d'information : RMI ne dispose pas de retour d'information bas niveau, excepté IO exception, tel que pour connaître le statut du buffer de réception, la charge de travail du pair, erreur de sécurité, ...

En plus de ces limites, des problèmes ont été rencontrés avec Jxta-RMI comme décrits ultérieurement. Le RMI est donc une bibliothèque qui est bien adaptée dans son principe, mais qui manque de fonctionnalités pour être exploitable en P2P.

Nous en avons conclu qu'il fallait écrire notre propre mode d'appel de méthodes à distance, avec un mode asynchrone, surtout avec gestion des déconnexions, charges et sécurité.

XtremWeb

Pour l'utilisation de Us à travers Internet, XtremWeb est mis à contribution en tant que fournisseur de ressources. XtremWeb est une plate-

forme expérimentale de calcul distribué pair à pair, elle utilise des ressources distantes PCs des réseaux locaux LAN, stations de travail, serveurs, connectées à Internet. Les participants d'XtremWeb coopèrent dans un but commun, partager leur temps CPU quand le CPU est inactif. Cette plateforme permet d'exécuter des programmes de manière distribuée sur un grand nombre de ressources. XtremWeb peut donc être configuré pour exécuter une application voulue.

Fonctionnement XtremWeb

XtremWeb est composé de **client**, **serveur** et de **worker**. Le serveur est le point centralisé qui permet de contrôler les applications et les travaux initialisés par ces applications, après que le serveur ait lancé sur chaque Worker l'application désignée et les paramètres envoyés par le client , les workers envoient les résultats obtenus au serveur qui les renvoie au client. Les clients et les workers contactent le serveur pour chaque action, le client pour initialiser un travail et les workers pour demander du travail.

En résumé, le fonctionnement de XtremWeb est le suivant, dans un premier temps un client XtremWeb envoie N tâches au serveur XtremWeb, puis le serveur assigne les tâches aux workers XtremWeb qui se manifestent régulièrement au serveur en demandant du travail. Le serveur envoie le binaire correspondant, si il n'existe pas sur les workers XtremWeb. Finalement les workers XtremWeb reçoivent les tâches et le binaire associé, pour les exécuter, puis retourne au serveur XtremWeb les résultats obtenus.

Intégration XtremWeb

Pour que Us soit intéropérable avec XtremWeb, un client XtremWeb a été créé spécifiquement en sachant qu'une tâche XtremWeb est un travail paramétré associé à un binaire, dans notre cas, le binaire associé à chaque Tâche XtremWeb n'est autre que le programme du stockeur Us. Par conséquent, les pairs de Us sont des workers d'XtremWeb qui ont chacun un stockeur Us comme application locale à exécuter et le fournisseur de Us est un client d'XtremWeb qui demande des ressources, c'est-à-dire l'exécution de stockeurs Us sur les workers d'XtremWeb. Pour lancer XtremWeb et Us, il faut tout d'abord exécuter un serveur XtremWeb, puis un worker XtremWeb pour chaque individu voulant participer, lequel va s'enregistrer auprès du serveur XtremWeb. Ensuite exécuter un Client XtremWeb, qui est en fait le fournisseur de service Us, qui va contacter le serveur XtremWeb pour convertir les workers XtremWeb en Stockeurs Us. On exécute un Client Us, lequel va contacter le fournisseur de service Us pour stocker des données Chaque stockeur Us, engendré automatiquement

sur un worker XtremWeb, contactera le fournisseur de service Us pour s'enregistrer et ainsi créer la liste des ressources. Le Client XtremWeb peut engendrer à nouveau des Stockeurs Us à partir de Workers XtremWeb, et de ce fait augmenter la liste des ressources disponibles.

Jxta

La plupart des systèmes pair à pair existants ont développé leurs propres protocoles pour la découverte et l'exploitation des ressources disponibles sur le réseau. Le projet open source JXTA, initié par Sun, propose une plateforme générique dont l'objectif est de servir de couche de base pour la définition de services pair-à-pair spécialisés. JXTA propose un ensemble de protocoles de base pour la découverte et le monitoring des ressources, pour la communication inter-pairs en environnement hautement dynamique et permet à l'utilisateur de définir ses propres services. Notons que la découverte des ressources peut se faire aussi bien en mode centralisé que par diffusion.

Fonctionnement Jxta

Outre les concepts de pair et de groupes de pairs pouvant interagir de par les protocoles de bases inhérents à ce système, Jxta introduit deux concepts. Tout d'abord les Pipes, qui sont des tubes de communication unidirectionnels pour envoyer et recevoir des messages. Ils supportent tous types d'objets, et ils ont deux types de connexions : point-à-point ou par propagation. Un concept supplémentaire les Annonces : Message XML qui nomme, décrit et publie l'existence d'une ressource, une annonce a une durée de vie limitée. Jxta propose les propriétés suivantes à un pair : Un identifiant unique UUID, Adressable indépendamment de sa localisation : passage firewalls, NAT, plusieurs points d'accès réseau (TCP, HTTP, ...) De même Jxta propose plusieurs types de pairs : minimaux pour des pairs ayant des ressources minimales tel que PDA, simples, rendez-vous pour diffuser les annonces, routage pour router les messages.

Intégration Jxta

Us peut utiliser le protocole de découverte des pairs de Jxta pour fonctionner à travers Internet. Comme les deux précédentes versions utilisent un serveur identifié pour l'enregistrement, nous utiliserions ici le mode de découverte par diffusion. En ce qui concerne les communications, un projet de portage de RMI sur Jxta existe, mais il n'y avait pas de version disponible à l'époque. Nous n'avons donc pas pu utiliser les moyens de commu-

nications de Jxta et avons décidé de développer nos propres invocations distantes basées sur de simples envois de messages.

4.7 Conclusion

Les systèmes de stockage pair à pair reposent sur des nœuds très volatiles : le temps de connexion d'un pair sur Internet n'est que de quelques heures. Aussi, la pérennité des données est-elle difficile à garantir. Pour cela, les systèmes existants se basent sur des mécanismes de redondance des données. Dans le système Us, c'est l'algorithme de Reed Solomon qui est utilisé pour générer la redondance. Associé à cet algorithme, un processus de reconstruction exécuté continuellement en arrière plan assure l'intégrité des données. Le but principal du système est de toujours être capable de reconstruire les données et garantir ainsi la pérennité des informations stockées. Nous verrons dans les chapitres dédiés que pour atteindre ce but, il sera nécessaire de s'attarder sur la distribution des données.

Le premier prototype Us est opérationnel et fonctionnel. Ce prototype a donné lieu à quelques démonstrations, lors de conférences et diverses réunions. Il dispose de primitives très simple d'utilisation (PUT, GET) de blocs de données. Cependant, il réside au sein de l'architecture de ce prototype des fonctions centralisées. A l'heure actuelle, un nouveau prototype complètement décentralisé est en cours.

Chapitre 5

Interface Us : UsFS

UsFS est né de la volonté de créér un logiciel de backup (basé sur Us). Or Us travaille avec des blocs de données non mutables. Il était donc nécessaire de pouvoir manipuler des fichiers. Pour cela, nous avons développé une interface de type système de fichiers, nommée UsFS, qui interagit avec le prototype de stockage pair à pair Us défini précédemment. UsFS est un système mono-écrivain (puisque basé sur Us).

UsFS est basé sur le projet open source Fuse, "Filesystem in Userspace". Ce système de fichier intercepte les commandes de l'utilisateur et les interprète pour lancer le processus de sauvegarde des données par Us.

Dans un premier temps, UsFS stockaient les fichiers de manière primaire au sein de Us. Les fichiers étaient découpés en blocs et envoyés/stockés à l'aide de la primitive "put" dans Us. Mais lors d'opérations sur le fichier, tel que le move, il était nécessaire de supprimer les blocs du fichier et de réenvoyer/restocker le fichier intégralement.

Donc dans le but d'optimiser les performances et obtenir un gain de temps, nous avons introduit la notion de flogs. Les flogs sont les enregistrements des actions de l'utilisateur sur les fichiers. Ces fichiers flogs sont ensuite découpés en blocs, puis envoyés/stockés dans Us. Ainsi, on stocke uniquement les modifications du fichier sous forme de blocs. Un fichier de l'utilisateur sera donc stocké sous forme d'un ensemble d'enregistrements de flog. De cette manière, nous avons un contrôle des données plus fin et une optimisation des accès aux fichiers pour divers opérations sur les fichiers (delete, move, chmod).

De plus, grâce aux flogs, une fonctionnalité intéressante est offerte : le versioning. L'utilisateur peut ainsi retrouver les différentes versions de son fichier au cours du temps. Cette fonctionnalité est très proche de celle offerte dans le système de stockage réparti Ivy [44] (pour de plus amples

informations sur ce système, voir la section 2.4). La différence majeure avec Ivy est que Ivy est un système multi-écrivain, tandis que Us est mono-écrivain. Ivy utilise un système de journalisation d'écritures (versioning) pour résoudre le problème de la cohérence des données dans un environnement multi-utilisateurs. Tandis que UsFS utilise un système de flogs (versioning) pour optimiser les accès fichiers.

Nous présentons, en premier lieu, une brève description du projet Fuse. Ensuite, nous découvrirons l'architecture et les fonctionnalités ajoutées de UsFS. Puis, nous étudierons son implémentation et enfin nous verrons le fonctionnement de UsFS : les opérations d'écriture et lecture de fichiers et la gestion du versioning.

5.1 Présentation Fuse

Fuse [25] est un projet, codé en langage C, qui facilite l'implémentation d'un système de fichiers complet et fonctionnel sous le système d'exploitation Linux. Fuse dispose d'une librairie simple (API) et d'une installation accessible (pas besoin de recompiler le noyau depuis la version 2.x). Tout utilisateur peut utiliser Fuse (pas de privilèges ou autorisations requis par le root). Fuse est disponible pour les noyaux Linux 2.4.x et 2.6.X.

A l'origine, Fuse a été développé pour le projet open source AVFS [1]. AVFS est un système de fichiers qui autorise tous les programmes à naviguer dans des archives ou fichiers compressés (extensions tar, gzip, zip, bzip2, et rar). Cependant, Fuse est devenu très vite un projet séparé, utilisé par de nombreux autres projets satellites [2]. Les plus intéressants sont : FunFS [3] un système de fichiers réseau dont le but est "d'être meilleur que NFS", EncFS [4] un système de fichiers crypté. Le GmailFS [5], écrit en Python, qui transforme l'utilisation d'un compte email gmail en un medium de stockage. Et le projet Wayback [6] qui est un système de fichiers incluant un mécanisme de versioning.

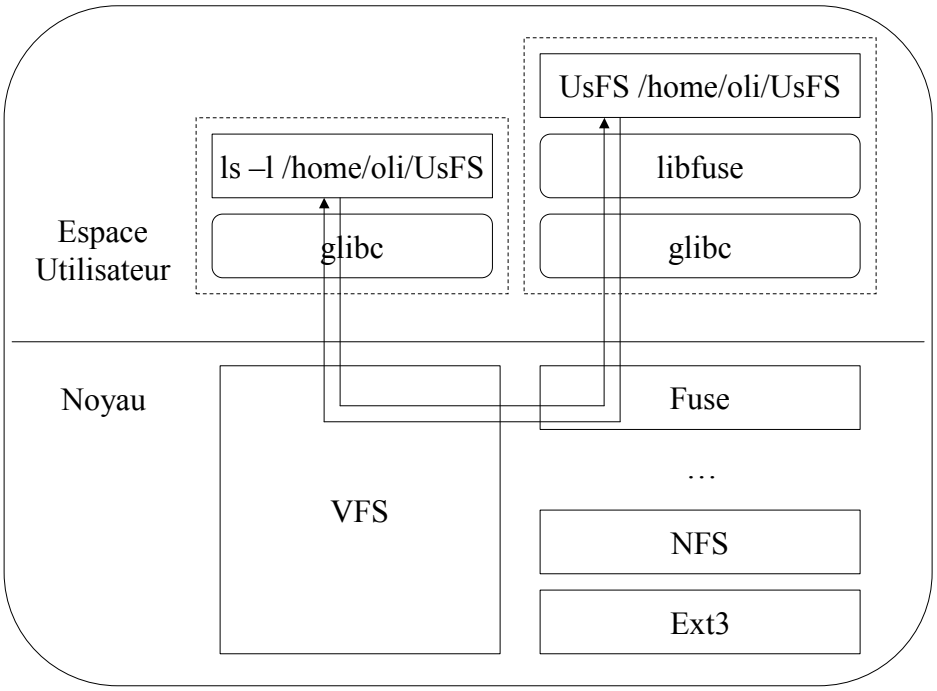
5.2 Architecture et fonctionnalités de UsFS

La figure 5.1 montre l'interaction entre UsFS et le module noyau Fuse (et les librairies Fuse et glibc), par le cheminement d'un appel au système

[1] http ://avf.sourceforge.net
[2] http ://fuse.sourceforge.net/filesystems.html
[3] http ://www.luminal.org/wiki/index.php/FunFS
[4] http ://arg0.net/users/vgough/encfs.html
[5] http ://richard.jones.name/google-hacks/gmail-filesystem/gmail-filesystem.html
[6] http ://wayback.sourceforge.net

de fichier UsFS (exemple par la commande stat).

FIG. 5.1 Interaction Fuse/UsFS



Le module de noyau Fuse et la librairie Fuse communique via un descripteur de fichier spécifique "/proc/fs/fuse/dev". Ce fichier peut être ouvert plusieurs fois de suite. Le descripteur de fichier est passé à l'appel système mount pour concorder avec le descripteur du système de fichiers monté.

Les fonctionnalités offertes par UsFs sont les suivantes :
- Accès transparent à Us : l'utilisateur dispose d'un système de fichier d'apparence ordinaire. Ainsi, il n'a pas besoin d'interagir directement avec Us.
- Sauvegarde pérenne : due au couplage avec Us, UsFS sauvegarde de manière pérenne et implicite les fichiers.
- versioning : accès aux différentes versions des fichiers réalisées au cours du temps.
- Cache local mémoire : optimisation des performance pour l'accès aux données.

Concernant l'**accès transparent à Us**, UsFS interprète de lui même toutes les commandes issues de l'utilisateur et les transcrit en actions concrètes

pour Us (sauvegarde / restauration des fichiers). La gestion et le découpage des blocs des fichiers sont assurés par le client Us dédié à UsFS.

La **sauvegarde pérenne** est assurée par Us. Dans le cas d'une totale perte des données sur le disque dur local, il sera toujours possible de récupérer et reconstruire l'intégralité des fichiers et répertoires stockés auparavant. Pour cela, une commande de restauration a été conçue. Cette commande dispose de diverses options, telle que la possibilité de régénérer l'intégralité du système à une date donnée. Cette commande est directement liée au mécanisme de versioning, expliqué ci-après.

Le **versioning** permet à l'utilisateur de récupérer différentes versions de son fichier stocké. Le changement de version d'un fichier a lieu, dès que l'utilisateur modifie son fichier. Il est possible aussi de récupérer une version d'un fichier en fonction d'une date donnée. Cette fonctionnalité est très similaire à la fonctionnalité offerte par un CVS. Enfin, cette fonctionnalité s'applique aussi à l'intégralité du système. Ainsi, il est possible de régénérer l'arborescence du système à partir d'une date donnée.

5.3 Implémentation UsFS

Dans un premier temps, nous décrirons la notion de flog (file logs). Ensuite, nous verrons comment s'effectue les communications entre les applications Us et UsFS par une interface dédiée. L'interface des deux logiciels est assuré par un démon externe. Ceci afin de faciliter la transition vers une nouvelle version des deux logiciels (Us et UsFS) ou utiliser facilement un autre système de fichier. Puis, une description et caractéristiques du cache local sont donnés. Enfin, nous verrons comment l'administration du système UsFS est gérée.

5.3.1 Flogs (File logs)

UsFS travaille sur deux catégories : les données des fichiers et les méta-informations liées à ces fichiers, selon le même principe que les inodes. Pour cela, Fuse fournit une API qui permet de connaître les évènements suivant :

- modification des données d'un fichier.
- écriture/modification des méta-informations d'un fichier. (les droits, dates, suppression, ...)
- lecture d'un fichier.
- lecture des méta-informations d'un fichier.

Au sein de UsFS, nous enregistrons chaque action dans un ”log de fichier” horodaté, nommé flog. Les flogs sont donc des fichiers. Ces fichiers flogs sont ensuite découpés en blocs, puis envoyés par des put dans Us. La définition de chaque flog est expliqué en paragraphe 5.4.1. Concernant l'opération d'horodatage, puisque UsFS n'est pas un système multi-utilisateur, on considère qu'il existe une horloge centralisée.

5.3.2 Interface Us/UsFS

Fuse ainsi que Us doivent être installés sur la machine dédiée au fonctionnement de UsFS. Afin de gérer les communications entre les applications, un démon nommé launchUs doit être exécuté, afin de lancer Us en arrière plan. Ce démon indique les actions à exécuter au système de stockage Us par le biais de l'application interface UsFS. Il existe une arborescence cachée du point de vue de l'utilisateur. Cette arborescence filaire se compose des répertoires suivants :

- **toPut** : UsFS indique au démon launchUs les fichiers flogs, indiquant les modifications apportées aux fichiers stockés par UsFS, à envoyer et stocker de manière pérenne via Us. Les fichiers flogs contiennent une estampille horodatée Ces fichiers apportent donc un historique des modifications effectuées sur les fichiers.
- **toGet** : UsFS indique au démon launchUs les fichiers à récupérer et stocker en local dans le cache.
- **toFree** : UsFS indique au démon launchUs les fichiers à effacer.
- **ack** : pour stocker les fichiers ack (accusés réception) fournis par Us, ces fichiers permettent de récupérer les données au sein de Us lorsqu'elles ne sont plus dans le cache local de UsFS. Les acks sont donc les métafichiers générés par Us lors de la sauvegarde des fichiers flog.

Un utilisateur UsFS peut alors créer un répertoire dans son compte. Par exemple : /home/user/UsFS. Puis, l'utilisateur monte le système de fichier UsFS sur ce répertoire, par le biais de la commande suivante :

```
/home/user$ UsFS repertoire_utilisateur &
```

Dès lors, l'utilisateur de UsFS a accès à un système de fichier classique. Toutes les commandes sont couplées à Us. Par exemple, lorsque l'utilisateur lance la commande ”df -H” , l'utilisateur visualise son quota d'espace total de stockage disponible alloué par Us et l'espace libre restant.

Dans un contexte multi-utilisateurs pour une machine, le lancement en arrière plan du démon launchUs pour envoyer et recevoir les données ne

doit pas être unique. Plusieurs instances de Us doivent donc pouvoir être lancées en même temps. Il est donc nécessaire d'autoriser plusieurs entités Us sur une même machine. Malgrè cela, il est impératif de prévoir une restriction dans les responsabilités et la charge de stockage de la machine : plusieurs exécutions de clients par machine sont autorisées, tandis qu'un seul stockeur doit être exécuté par machine. Cette restriction est due à des raisons de pérennité des données, pour ne pas rendre caduque l'utilisation des techniques de redondance.

5.3.3 Administration UsFS

Come tout logiciel, UsFS inclut une gestion des logs. Il génère à intervalles réguliers et en fonction d'événement précis (exemple : stockage d'un fichier) des fichiers logs qui sont stockés dans le répertoire **log**. Différents types de logs existent pour discerner les erreurs systèmes, avertissements, ... L'analyse de ces fichiers logs permettront d'améliorer la qualité du logiciel.

5.3.4 Cache local

Des informations sont stockées au sein de la machine de l'utilisateur, le cache local contient des fichiers, tandis que le répertoire exist contient les méta-informations des fichiers : Le répertoire **cache** sert pour stocker des fichiers et répertoires du système de manière temporaire, c'est un cache local (les fichiers stockés ne sont pas vides, ils sont "remplis" et prêts pour une éventuelle lecture). Le répertoire **exist** pour gérer l'existence des fichiers et répertoires stockés par le système. Ce répertoire contient donc l'arborescence entière des répertoires et fichiers (les fichiers sont vides). Il sert dans la gestion des méta-informations. UsFS est un système mono-écrivain (car basé sur Us). Un client UsFS (qui est donc un client Us) n'a accès qu'à son propre compte, à ses propres fichiers. Un autre client ne pourra pas accéder aux fichiers d'un autre client. Il n'y a donc pas de gestion de la cohérence du cache local.

Grâce au cache local , il n'est pas nécessaire de récupérer constamment les données à travers Us. Le cache local a une taille mémoire fixée. Cette limite doit donc être gérée. Cette gestion est assurée par UsFS, qui applique une politique particulière, telle que la politique LRU : si l'utilisateur demande un fichier n'ayant pas été accédé depuis longtemps, il ne se trouve plus dans le cache. UsFs envoie donc à Us une requête pour chaque bloc concernant le fichier et Us les télécharge. UsFS se charge de reconstruire le fichier et le rend à nouveau accessible par l'utilisateur.

En revanche, cette gestion de la taille du cache doit prendre en compte une taille maximale des fichiers traités. Sinon la politique de gestion du cache pourrait devenir absurde selon la taille d'un fichier. Exemple : pour un cache de 200 Mo, il est impossible de stocker un fichier de 650 Mo. Même si la limitation du cache a été prise en compte, la cache local est généralement de la taille du quota de l'utilisateur.

5.4 Fonctionnement de UsFS

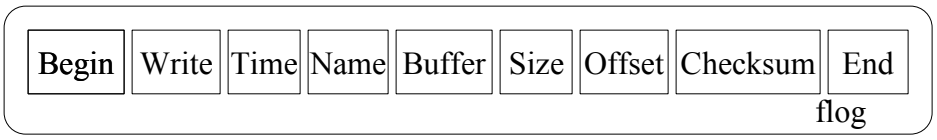
Nous allons aborder le fonctionnement des opérations d'écriture et de lecture d'un fichier dans UsFS. Ces opérations nécessitent l'utilisation des flogs. Les flogs sont des structures contenant les opérations à mener sur les fichiers. Nous verrons donc dans un premier temps l'opération d'écriture, puis l'opération de lecture et enfin comment s'effectue le versioning des fichiers.

5.4.1 Ecriture d'un fichier

Pour l'écriture d'un fichier, UsFs intercepte l'appel à la fonction WRITE et récupère le chemin et nom du fichier, le buffer, sa taille et l'offset. Puis, UsFS crée un flog à partir de ces informations.

Un flog est une structure qui contient un type de commande (WRITE, OPEN, MKNOD, CHMOD, ...) à exécuter et les paramètres d'entrée liés à cette commande, afin de pouvoir réexécuter entièrement cette commande ultérieurement. Donc un flog de type WRITE dispose du chemin et nom du fichier concerné, le buffer, sa taille et l'offset. Chaque flog contient une estampille horodatée. Cette estampille est primordiale pour retrouver les modifications au cours du temps d'un fichier. Un exemple de flog est donné en figure 5.2.

FIG. 5.2 Exemple de flog UsFS de type "Write" dans un fichier de nom "Name".



Une fois créé, UsFS exécute la lecture et exécution du flog en question. L'étape de lecture permet d'éprouver le lecteur de flog et de vérifier qu'il n'y ait pas d'erreurs lors de la création du flog. Cette vérification est possible grâce au checksum intégré dans le flog.

Au même emplacement relatif, dans le répertoire exist, le fichier de même extension modifié, indiquera les dates et types de modifications, ainsi que le nom du fichier flog correspondant. Ce fichier servira uniquement à fournir les méta-informations associées au fichier associé.

Enfin, ce sont les fichiers du répertoire log qui seront sauvegardés par Us. Ces fichiers permettent de retrouver l'intégralité de l'arborescence et contenus des fichiers.

5.4.2 Lecture d'un fichier

Dans ce cas, UsFS intercepte l'appel à la fonction READ. UsFS va d'abord lire les méta-informations du fichier souhaité et vérifier son existence dans le répertoire exist. Si le fichier existe, alors UsFS accède de manière locale directement au contenu du fichier dans le répertoire cache. Sinon UsFs doit le récupérer via Us. Il demande donc la récupération du fichier flog manquants, ce qui peut prendre un certain temps. Une fois, cette opération terminée, UsFS lit et exécute le flog afin de reconstruire le fichier et le place en créant si nécessaire l'arborescence associée dans le cache local. Puis, une fois le fichier reconstitué et stocké dans le cache, UsFS peut y accéder librement.

5.4.3 Gestion des flogs

Pour le stockage des versions des fichiers, les fichiers flog stockent les modifications apportées sur le fichier pour chaque version. De cette manière, il est toujours possible d'obtenir toutes les versions d'un fichier en parcourant les fichiers flogs. De même, il est possible de demander de restaurer un fichier à partir d'une date donnée. Le principe sera le même que pour les versions, une lecture et réexécution des fichiers flog jusqu'à la date désirée permettra d'obtenir l'effet escompté.

Enfin, des optimisations sont possibles en ce qui concernent le processus d'écriture et lecture des flogs. Par exemple un filtre de suppression d'opérations inutiles des flogs : si un utilisateur effectue deux écritures successives au même endroit d'un fichier, le système peut ne pas enregistrer la première. Un autre exemple serait un filtre de compression des flogs : si l'utilisateur effectue de nombreux WRITE consécutifs, il sera judicieux de fusionner l'ensembles des flogs générés en un seul.

5.5 Conclusion

L'interface UsFS définie précédemment interagit avec le prototype de stokage pair à pair Us. Elle est intuitive et l'utilisateur pense utiliser un système de fichier classique. Toutes les opérations de sauvegarde/restauration des fichiers et autres sont complètement masquées. De plus, un système de versioning des informations stockées au sein de UsFS ajoute la possibilité pour l'utilisateur de retrouver différentes versions de ces fichiers ou de son arborescence. Il peut ainsi retrouver l'intégralité de ses données à une date voulue.

UsFS est toujours en cours de développement et nécessite encore de nombreuses améliorations. Ces améliorations devront porter sur la cohérence des données et les interactions entre les divers processus en action au sein du système. Ceci afin d'éviter des situations de dead lock ou de causer des problèmes d'intégrité et cohérence sur des fichiers accédés en même temps par différents processus en action (Fuse est multithreadé). Afin d'annihiler ces problèmes, Fuse est lancé en mode séquentiel à l'heure actuelle. UsFS nécessiterait aussi une optimisation des performances de la commande de restauration : reconstruction de l'arborescence et contenu des fichiers à partir des fichiers flogs. A l'heure actuelle, la commande de restauration n'utilise pas de stratégie évoluée.

Chapitre 6

Problème de la distribution

Pour garantir la pérennité des données, un système de stockage pair à pair pérenne, tel que Us, doit exécuter un processus de reconstruction des données perdues dues à des pairs défaillants. Dans de précédents travaux, s'attachant à l'étude de la MTTF (Mean Time To Failure : durée moyenne de fonctionnement du système avant la première perte définitive d'une donnée) d'un système de stockage pair à pair , il a été montré [75] qu'un tel système doit faire face à un flux continu de données au sein du réseau afin d'assurer correctement la reconstruction de données perdues. La forte volatilité des pairs génère une très grande quantité de communications lors des reconstructions.

Dans Us, les utilisateurs stockent des blocs de données. Chaque bloc de donnée est découpé en fragments. Quand un pair tombe en panne, les fragments qu'il stockait sont perdus. Tous les fragments qu'il stockait doivent donc être reconstruits et redistribués sur d'autres pairs.

Soit f le nombre de fragments dans un bloc. Pour reconstruire chaque fragment, $f - 1$ fragments du bloc d'origine doivent être récupérés sur les pairs correspondants. On prend cette hypothèse forte pour ne considérer la mort que d'un seul pair et donc la reconstruction de l'ensemble des fragments qu'il stockait. Cela simplifie le problème en fixant un paramètre variable au départ. De plus, cette hypothèse permet de prendre en compte le critère de disponibilité. De cette manière, on considère que malgrè les problèmes de disponibilité, il sera toujours possible d'obtenir s fragments parmi les $f - 1$ (dans le cas de l'utilisation de ReedSolomon dans le processus de reconstruction), car il est impossible de prévoir la disponibilité des pairs au moment de la reconstruction.

Concernant la régénération d'un fragment, on considère qu'un pair se charge de récupérer les $f - 1$ fragments, régénère le bloc d'origine (phase

de décodage), puis régénère le fragment perdu (phase d'encodage). Lors du décodage, le fragment manquant est identifié et donc régénéré dans la phase d'encodage.

Considérons l'exemple suivant sur 100000 pairs, un pair stocke 100 fragments de 100Ko. Supposons que les blocs soient découpés en $f = 31$ fragments. Alors à la mort d'un pair, le nombre de fragments envoyés pour chaque pair concerné par la reconstruction est de $(f - 1) * 100$, approximativement 300Mo. Si on suppose que 1% des pairs tombe en panne par jour, environ 300Go doivent être envoyés par jour dans le réseau !

Parallèlement Us ne doit pas être intrusif pour les utilisateurs durant le processus continuel de réparation. Rappel : Us est dédié à un environnement constitué majoritairement de PC publics avec lignes ADSL, donc des débits asymétriques avec débit download plus élevé que l'upload. Donc la fraction de la bande passante en upload de l'utilisateur utilisée pour la reconstruction doit être la plus basse possible. Notre but est donc de minimiser la quantité de fragments envoyés par pair pour tout pair, quand une reconstruction est exécutée, i.e quand un pair tombe en panne.

La réception sur les pairs chargés de la reconstruction des fragments perdus est séquentielle. Mais dans le cas des lignes asymétriques, un pair peut recevoir x fois plus de données qu'il en envoie. Donc on ne considère pas le problème de la réception pour le moment.

Ainsi, dans un premier temps, nous définissons notre problème de minimisation du coût du processus de reconstruction. Il existe d'autres problèmes de minimisation de coût, tel que le coût de stockage des données et métadonnées, étudiés dans [36]. Puis, nous définissons une distribution basée sur la théorie des nombres premiers qui minimise ce coût pour tout pairs. Nous comparons cette distribution avec la distribution aléatoire, qui est la plus utilisée dans les systèmes de stockage distribué. Enfin, cette nouvelle distribution sera optimisée pour prendre en compte le comportement dynamique des pairs du système. Elle devra gérer les corrélations de panne, pour cela nous utiliserons des groupes de pairs liés à cette nouvelle distribution.

6.1 Définitions

Les notations sont résumées dans le tableau 6.1. Soient p un pair, N le nombre total de pairs, f le nombre de fragments d'un bloc, $f \leq N$, NB le nombre de blocs, α_i le nombre de fragments stockés par pair i, $\mathcal{P}$ l'ensemble des pairs, $\mathcal{B}$ l'ensemble des blocs stockés, B_p l'ensemble des blocs dont le pair p stocke un fragment, i.e l'ensemble des blocs à reconstruire à la panne de p. C_{max} est le coût de reconstruction. MP est le nombre de Métapairs.

N	Nombre total de pairs, $N = \|\mathcal{P}\|$
$\mathcal{P}$	L'ensemble des pairs [1–N]
f	Le nombre de fragments d'un bloc, $f \leqslant N$
NB	Nombre total de blocs, $NB = \|\mathcal{B}\|$
$\mathcal{B}$	L'ensemble des blocs [1–NB]
b	Un bloc : ensemble de f pairs
p	Un pair
B_p	Un ensemble de blocs dont le pair p stocke un fragment par bloc
α_i	Nombre de fragments stockés par le pair i
C_{max}	Coût de reconstruction (notion définie dans la section 6.1.4)
MP	Nombre de Métapairs (notion définie dans la partie 8.2)
MP_{Size}	Taille des Métapairs $MP_{Size} = N/MP$

TAB. 6.1 – Notations

MP_{Size} indique la taille des Métapairs.

Définissons quelques notions et définitions utilisées par la suite. Ensuite, une description de la distribution de donnée sera donnée et les définitions des coûts induits par le processus de reconstruction.

6.1.1 Distribution des données

La distribution des données répartit les fragments des blocs vers les pairs. La distribution est restreinte par la condition suivante : les f fragments d'un bloc de données sont stockés sur f pairs distincts. Aussi nous considérons $f \leq N$.

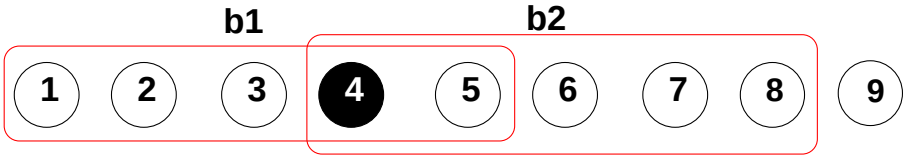
Chaque bloc b peut être représenté par une liste de f pairs. Les fragments d'un pair p appartiennent à des blocs distincts. La distribution des données, notée $\mathcal{D}$, se résume à :

$$\mathcal{D} : \mathcal{B} \mapsto \mathcal{P}^f$$
$$\forall b \in \mathcal{B}, p_{1,2}, ..., p_f \in \mathcal{P}, p_1 \neq p_2 \neq ... \neq p_f,$$
$$b \mapsto \{p_1, p_2, ..., p_f\}$$

Et cette relation est toujours vraie :

$$\forall b \in \mathcal{B}, \forall p \in \mathcal{P}, b \in B_p \Longleftrightarrow p \in b$$

FIG. 6.1 Stockage de deux blocs de taille f=5 sur N=9 pairs.



Une représentation matricielle d'une distribution est possible. Soit D une matrice de taille $NB * N$:

- $D[i][j] = 1$: le pair j stocke un fragment du bloc i.
- $D[i][j] = 0$: aucun fragment du bloc i n'est stocké sur le pair j.

Considérons l'exemple de stockage représenté par la figure 6.1. Deux blocs b_1 and b_2 de taille f=5 sont stockés sur N=9 pairs avec b_1={1,2,3,4,5} and b_2={4, 5, 6, 7, 8}. La matrice D représentant la distribution est :

$$D = \begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 \end{bmatrix}$$

Pour toutes distributions et pour tout nombre de blocs stockés, nous avons :

$$NB = \frac{1}{f} * \sum_{i=1}^{N} \alpha_i \tag{6.1}$$

$$\text{avec } \alpha_i = \sum_{j=1}^{N} D[i][j]$$

Maintenant nous allons introduire la notion de coût de communication pour les pairs lors du processus de reconstruction.

6.1.2 Coût local de communication d'un pair

Le coût local de communication $C_{loc(p,q)}$ est le nombre de fragments qu'un pair p envoie lors de la reconstruction des fragments d'un pair q :

$$\forall p, q \in \mathcal{P}, C_{loc(p,q)} = |B_p \bigcap B_q|$$

Et le nombre total de fragments utilisés lors de la reconstruction est égal à la somme de tous les coût locaux, à l'exception du pair en panne q. Aussi nous avons :

$$\forall q \in \mathcal{P}, \alpha_q * (f-1) = \sum_{p=1, p \neq q}^{N} C_{loc(p,q)} \tag{6.2}$$

L'impact du processus de reconstruction d'une distribution quelconque, indiquée par la matrice D, sur l'ensemble des pairs du réseau, peut être observé sur la matrice carrée R d'ordre N :

$$R = D^t * D$$

Cette matrice indique les informations suivantes :
- $R[i][i] = \alpha_i$: le nombre de fragments stockés par le pair i.
- $R[i][j] = C_{loc(j,i)}$: le nombre de fragments que le pair j doit envoyer à la panne du pair i.

Considérons le stockage de l'exemple présenté dans la figure 6.1, avec b_1={1,2,3,4,5} et b_2={4, 5, 6, 7, 8}. Nous avons B_1=B_2=B_3=$\{b_1\}$, B_4=B_5=$\{b_1,b_2\}$, B_6=B_7=B_8=$\{b_2\}$, B_9=$\emptyset$. La matrice R résultante de cet exemple est :

$$R = D^t * D \begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 2 & 2 & 1 & 1 & 1 & 0 \\ 1 & 1 & 1 & 2 & 2 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

Dans l'exemple donné par la figure 6.1, si le pair 4 tombe en panne, les coûts locaux sont :

$C_{loc(1,4)}$=$C_{loc(2,4)}$=$C_{loc(3,4)}$=$C_{loc(6,4)}$=$C_{loc(7,4)}$=$C_{loc(8,4)}$=1, $C_{loc(5,4)}$=2 et $C_{loc(9,4)}$=0.

6.1.3 Coût global de communication d'un pair

Pour chaque pair q tombant en panne, nous déterminons le pair le plus dérangé par le processus de reconstruction. Puisque deux pairs peuvent envoyer deux fragments simultanément, le coût global de communication est défini par le pair qui envoie le plus, i.e. le coût global de communication est le maximum de tous les coûts locaux. Soit $C_{glob(q)}$ le coût global pour reconstruire le pair q :

$$\forall q \in \mathcal{P}, C_{glob(q)} = max_{p \in \mathcal{P}, p \neq q} C_{loc(p,q)}$$

De retour sur l'exemple de stockage de la figure 6.1, le coût global de communication du pair 4 est $C_{glob(4)} = 2$.

6.1.4 Coût maximal de communication

Nous considérons par cette notion, le pire cas possible dans le processus de reconstruction. Dans toute distribution, le coût maximal de communication est le maximum parmi tous les coûts globaux, en considérant le fait que chaque pair puisse tomber en panne, ce qui est le cas dans un réseau pair à pair :

$$C_{max} = \max_{q \in \mathcal{P}} C_{glob(q)} = max_{q \in \mathcal{P}} max_{p \in \mathcal{P}, p \neq q} C_{loc(p,q)}$$

De retour sur l'exemple de stockage de la figure 6.1, le coût maximal de la reconstruction est $C_{max} = 2$.

6.2 Formulation du problème

Dans un premier temps, nous définissons nos objectifs, i.e. une distribution optimale. Ensuite, nous présentons la définition et propriétés d'une distribution idéale, un type de distribution optimale.

DÉFINITION 1 *Une* **distribution optimale** $\mathcal{D}_{opt}$, $\forall N$, $\forall f$, NB *le nombre de blocs, est une distribution des données qui minimise le coût maximal de reconstruction* C_{max} *tout en stockant un nombre maximal de blocs* NB, *i.e.* $\forall \mathcal{D}'$ *une distribution quelconque :*

$$C_{max}(\mathcal{D}_{opt}) \leq C_{max}(\mathcal{D}')$$

Soient N et f des paramètres fixés. Notre but est de trouver une distribution optimale pour toute valeur de NB. Par définition, ceci est équivalent à trouver une distribution optimale pour toute valeur de C_{max}.

LEMME 1 *Soit* f *le nombre de fragments d'un bloc,* N *le nombre total de pairs, et* $\mathcal{D}$ *une distribution de données, si* NB *est le nombre de blocs stockés avec* $\mathcal{D}$*, alors la borne inférieure de* C_{max} *est :*

$$NB * \frac{f * (f-1)}{N * (N-1)}$$

Preuve

Soient N, f, D *fixés, le nombre de blocs* NB *est fixé. Notre but est de minimiser* C_{max}*. Le coût maximal des communications peut être calculé de la manière suivante en partant de l'équation (6.1) :*

$$\sum_{q=1}^{N} \alpha_q = NB * f$$

Avec l'équation (6.2), nous avons :

$$\sum_{q=1}^{N} \alpha_q * (f-1) = \sum_{q=1}^{N} \sum_{p=1, p\neq q}^{N} C_{loc(p,q)}$$

Aussi :

$$NB * f * (f-1) = \sum_{q=1}^{N} \sum_{p=1, p\neq q}^{N} C_{loc(p,q)}$$

NB *et* f *sont des valeurs fixées, aussi il en est de même pour :*

$$\sum_{q=1}^{N} \sum_{p=1, p\neq q}^{N} C_{loc(p,q)}$$

De plus :

$$\forall q \in \mathcal{P}, C_{max} = \max_{p\in\mathcal{P}, p\neq q} C_{loc(p,q)}$$

i.e. C_{max} *est le maximum entre tout les coûts locaux de reconstruction. Remarquons que le nombre total de coûts locaux est de* $N*(N-1)$. *Aussi* C_{max} *est au moins égal à la moyenne de tout les coûts locaux de reconstruction :*

$$C_{max} \geq NB * \frac{f*(f-1)}{N*(N-1)}$$

❑

De plus, un coût fixé C_{max} impose des restrictions sur NB :

LEMME 2 *Soit* f *le nombre de fragments d'un bloc,* N *le nombre total de pairs, et* $\mathcal{D}$ *une distribution de données, si* C_{max} *est le coût maximal de reconstruction de la distribution* $\mathcal{D}$*, alors la bonne supérieure de* NB *est :*

$$\frac{N}{f} * \frac{N-1}{f-1} * C_{max}$$

Preuve

Soient N,f, D *fixés et* C_{max} *fixé. Notre but est de maximiser* NB. *Nous savons que :*

$$C_{max} = \max_{q\in\mathcal{P}} C_{glob(q)}$$

De plus :

$$\forall q \in \mathcal{P}, C_{max} = \max_{p\in\mathcal{P}, p\neq q} C_{loc(p,q)}$$

Alors :

$$\forall p, q \in \mathcal{P}, p \neq q, C_{loc(p,q)} \leq C_{max} \tag{6.3}$$

Aussi :

$$\forall q \in \mathcal{P}, \sum_{p=1, p\neq q}^{N} C_{loc(p,q)} \leq (N-1) * C_{max}$$

Nous rappelons l'équation (6.2) :

$$\forall q \in \mathcal{P}, \alpha_q * (f-1) = \sum_{p=1, p\neq q}^{N} C_{loc(p,q)}$$

Aussi :

$$\forall q \in \mathcal{P}, \alpha_q * (f-1) \leq (N-1) * C_{max}$$

Finalement, nous avons :

$$\forall q \in \mathcal{P}, \alpha_q \leq \frac{N-1}{f-1} * C_{max} \tag{6.4}$$

Nous rappelons aussi l'équation (6.1) :

$$NB = \frac{1}{f} * \sum_{q=1}^{N} \alpha_q$$

Donc le nombre de blocs stockés NB *est :*

$$NB \leq \frac{N}{f} * \frac{N-1}{f-1} * C_{max}$$

❑

Dans le meilleur des cas possibles, une distribution pourrait atteindre les bornes de NB et C_{max}.

DÉFINITION 2 *Soit* f *le nombre de fragments d'un bloc,* N *le nombre total de pairs, une* **distribution idéale** $\mathcal{D}$*, avec* NB *le nombre de blocs stockés et* C_{max} *son coût, est une distribution, telle que :*

$$NB = \frac{N}{f} * \frac{N-1}{f-1} * C_{max} \tag{6.5}$$

PROPOSITION 1
Soit $\mathcal{D}$ une distribution idéale, alors $\mathcal{D}$ est une distribution optimale.

Preuve

Soit N *et* f *fixés. Soit* $\mathcal{D}$ *une distribution idéale. Son coût est* C_{max} *et son nombre de blocs stockés est* NB*. Soit* $\mathcal{D}'$ *une autre distribution, son coût est* C'_{max} *est son nombre de blocs stockés est* NB'*. D'un côté, si*

$$C'_{max} \leq C_{max}$$

Puisque

$$NB' \leq \frac{N}{f} * \frac{N-1}{f-1} * C'_{max}$$

Alors

$$NB' \leq \frac{N}{f} * \frac{N-1}{f-1} * C_{max}$$

Aussi

$$NB' \leq NB$$

D'un autre côté, si

$$NB' \geq NB$$

Puisque

$$C'_{max} \geq \frac{f*(f-1)}{N*(N-1)} * NB'$$

Alors

$$C'_{max} \geq \frac{f*(f-1)}{N*(N-1)} * NB$$

Aussi

$$C'_{max} \geq C_{max}$$

Ainsi, $\mathcal{D}$ *est une distribution optimale.*

❑

DÉFINITION 3 *Soit* $\mathcal{D}$ *et* $\mathcal{D}'$ *des distributions, tels que :*

$$\mathcal{D} : \mathcal{B} \mapsto \mathcal{P}^f$$
$$\mathcal{D}' : \mathcal{B}' \mapsto \mathcal{P}^f$$

$\mathcal{D}'$ *est une* **distribution** k**-repliement(**$\mathcal{D}$**)**, *si il existe une partition* $\mathcal{A}$ *de* $\mathcal{B}'$ *où chaque sous-ensemble* s *appartenant à* $\mathcal{A}$ *est de telle manière que* $|s| = k$ *et il existe une bijection entre* $\mathcal{B}$ *et* $\mathcal{A}$*, tel que* $\forall b \in \mathcal{B}$, $s = g(b)$ *et* $\forall d \in s$, $\mathcal{D}(b) = \mathcal{D}'(d)$.

THÉORÈME 2
Une distribution idéale pour $C_{max} \geq 1$ peut être déduite d'une distribution idéale quelconque pour $C_{max} = 1$.

Preuve

Pour prouver ce théorème, nous utilisons une construction basée sur une distribution k*-repliement($\mathcal{D}$).*

Lors de l'utilisation de la distribution $\mathcal{D}$, la panne d'un pair engendre une quantité de communications de données envoyées v, la même panne génère k.v *communications pour la distribution $\mathcal{D}'$. Aussi, si C'_{max} est le coût maximal de reconstruction de $\mathcal{D}'$, alors :*

$$C'_{max} = k.C_{max}$$

De plus, la distribution $\mathcal{D}'$ stocke k fois plus de blocs que la distribution $\mathcal{D}$. Prenons NB′ *le nombre de blocs stockés par $\mathcal{D}'$, alors :*

$$NB' = k.NB$$

Soient N, f *fixés. Soit $\mathcal{D}1$ une distribution idéale et* $C1_{max}$ *son coût maximal de reconstruction. Soit* $C1_{max}$ *fixé à* 1*, donc le nombre de blocs stockés* NB1 *est de :*

$$NB1 = \frac{N}{f} * \frac{N-1}{f-1}$$

Considérons une distribution $\mathcal{D}^{\|}$ être une distribution k*-repliement($\mathcal{D}1$), C^k_{max} son coût maximal de reconstruction et* NB^k *son nombre de blocs stockés, alors :*

$$C^k_{max} = k\ NB^k = k * \tfrac{N}{f} * \tfrac{N-1}{f-1}$$

Ainsi

$$NB^k = C^k_{max} * \frac{N}{f} * \frac{N-1}{f-1}$$

Par définition, la distribution $\mathcal{D}^k$ est une distribution idéale.

❑

Puisqu'une distribution est construite sur une distribution idéale pour $C_{max} = 1$, nous nous intéressons donc à une distribution idéale pour $C_{max} = 1$. De ce fait, nous verrons que trouver une distribution idéale correspond à un problème ouvert. Donc nous donnerons une distribution très proche d'une distribution idéale.

Chapitre 7

Distributions de données

Dans ce chapitre, nous présentons comment définir une distribution asympotiquement idéale et nous la comparons à la distribution aléatoire. En premier lieu, nous présentons la distribution aléatoire, qui est la distribution généralement utilisée dans les systèmes de stockage distribués. Cette distribution servira de référence pour nos travaux. Puis nous indiquerons la marche à suivre pour trouver une distribution idéale. Nous donnerons quelques distributions optimales pour $C_{max} = 1$. Finalement, nous présenterons notre nouveau schéma de distribution [56, 54], nommé distribution Cas Général (la distribution CG), et les résultats expérimentaux associés.

7.1 Distribution aléatoire

La distribution aléatoire stocke les f fragments de chaque bloc de données sur f pairs distincts choisis aléatoirement parmi tous le pairs. De part les lois de probabilités et statistiques, cette distribution est efficace pour un grand nombre de pairs. En effet, la probabilité d'obtenir des listes égales de f pairs ou avec un grand nombre de pairs communs est faible (le coût de reconstruction augmente considérablement dans ces cas). Néanmoins, cette distribution nécessite une connaissance globale de tout le réseau, ce qui est difficile à implémenter dans un réseau pair à pair. Le système de stockage PAST [16] est un exemple d'utilisation d'une telle distribution. Chaque pair et toutes les ressources possèdent un identifiant unique. Un système de routage dynamique est associé à ces identifiants. Un fichier est stocké sur le pair dont l'identifiant est le plus proche numériquement de l'identifiant du fichier. La volatilité des pairs implique

qu'un nouveau pair avec un identifiant plus proche peut apparaître après le stockage. Des communications supplémentaires doivent alors être générées pour retrouver le fichier. La distribution aléatoire de données est dans la plupart des cas une distribution de données efficace non optimale pour minimiser le coût de reconstruction. Malheureusement, cette distribution n'exploite pas la topologie physique du réseau pour éviter les pannes d'ensemble. Pour gérer ce problème supplémentaire, des stratégies de distribution structurées doivent être appliquées.

7.2 Distributions idéales pour $C_{max} = 1$

La manière de trouver le nombre total de blocs stockés, quand C_{max} est fixé, indique de précieux renseignements pour trouver une distribution optimale. Soit $C_{max} = 1$, le plus petit coût maximal de reconstruction. De cette manière, tous les coûts locaux de reconstruction doivent être égaux au coût maximal de reconstruction, dans ce cas, un coût égal à 1. En d'autres termes, pour tout les blocs stockés, l'intersection entre deux différents blocs doit être au plus de 1. Par conséquent, des blocs distincts ne peuvent être stockés par la même liste de pairs. Aussi, une distribution idéale pour $C_{max} = 1$ doit avoir la propriété d'intersection suivante entre blocs :

$$\forall b_i, b_j \in \mathcal{B}, |b_i \bigcap b_j| \leq 1$$

Soit NB_1 le nombre total de blocs stockés par une distribution idéale avec un $C_{max} = 1$. NB_1 doit respecter la définition d'une distribution idéale, soit :

$$NB_1 = \frac{N}{f} * \frac{N-1}{f-1}$$

Nous pouvons remarquer que pour un nombre de pairs N fixé, il est possible de stocker au plus $\frac{N^2}{f^2}$ blocs, et inversement pour un nombre de blocs NB fixé, nous avons besoin de $\sqrt{NB \times f^2}$ pairs pour obtenir un C_{max} à 1.

Par exemple, considérons la situation illustrée par la figure 6.1. Le pair 4 et le pair 5 stocke deux fragments de b_1 et b_2. Si un des deux tombe en panne, celui qui reste doit envoyer deux fragments. Tandis qu'il aurait été plus judicieux pour obtenir un coût moindre d'avoir une intersection entre les blocs moins importante. Supposons que l'intersection entre les deux blocs ne soit que le pair 5, soit 1. Alors, le pair 5 génère un seul envoi pour les autres pairs. Une autre panne génère un seul envoi pour tous pairs encore vivant. Donc la taille de l'intersection entre deux blocs distincts doit être au plus de 1 pour obtenir $C_{max} = 1$.

7.2.1 Problème mathématique associé à la distribution

En fait, ce problème d'intersection se réfère à un problème mathématique connu. Il consiste à trouver un ensemble maximal de k-tuples [67], respectant la propriété d'intersection décrite auparavant. Dans notre cas, le problème peut être transcrit de la manière suivante :

- Les blocs sont des f-tuples.
- $\forall f$, $\forall N$, $\mathcal{B}$ est l'ensemble maximal des blocs stockés, avec $\forall b_i, b_j \in \mathcal{B}, |b_i \bigcap b_j| \leq 1$

De nombreux problèmes similaires sur les propriétés d'intersection existent. Certains sont résolus, d'autres restent toujours ouvert.

Par exemple, le fameux problème des "36 officiers" imaginé par Euler [30] en 1782 est le plus pertinent de la difficulté de tel problème. Le problème des "36 officiers" est : comment une délégation de six régiments, chaque régiment est composé de 6 officiers de grades distincts, peut être regroupée dans un tableau régulier 6*6, de sorte qu'aucune colonne ou ligne ne duplique un grade ou un régiment ?

Euler conjecture qu'un tel arrangement est impossible. En 1900, un français mathématicien Gaston Tarry [30] donna une démonstration de ce problème et confirma la conjecture d'Euler.

La première approche pour résoudre notre problème est de connaître le nombre maximum théorique de blocs pouvant être stockés par une distribution pour tout f et N pour un $C_{max} = 1$. Le but est de trouver $NB = NB_1$ pour obtenir une distribution idéale.

7.2.2 Limite théorique de NB

Dans le cas général avec une intersection inférieure ou égal à 1 ($C_{max} = 1$), le nombre théorique NB prouvé par J. Schonheim [67], est :

$$\forall f, \forall N, NB \leq NB_{max}, \text{ avec } NB_{max} = \left\lfloor \frac{N}{f} * \left\lfloor \frac{N-1}{f-1} \right\rfloor \right\rfloor$$

Pour commencer, on constate que $NB = NB_{max} = NB_1$ est vérifié uniquement pour certaines valeurs de f et N. De plus, NB_{max} est seulement une limite théorique : il n'existe pas de distribution théorique définie pour toute valeur de N et f qui pourrait atteindre cette limite. Il est prouvé [35] que cela est impossible pour certaines valeurs de f et N, de plus pour les cas où aucune solution n'est donnée, le problème est toujours ouvert.

Maintenant, regardons certains cas particuliers, où NB_{max} peut être atteint et $NB = NB_1$.

7.2.3 Distribution idéale basée sur les plans projectifs finis

Soit $N = f^2 - f + 1$. Dans ce cas, une distribution peut être définie par la construction d'un plan projectif fini d'ordre $(f - 1)$, quand une telle construction est possible. Nous allons montrer qu'une distribution basée sur ce type de construction est une distribution idéale.

DÉFINITION 4 *Un* **plan projectif fini** *est un plan projectif avec un ensemble fini de points. Un plan projectif fini d'ordre* n *est défini comme suit :*

1. *Un nombre de points qui est de* $n^2 + n + 1$ *et un nombre de lignes de* $n^2 + n + 1$.
2. *Toutes les lignes partagent* $n+1$ *points et tout les points partagent* $n+1$ *lignes.*
3. *L'intersection de deux lignes est un.*

Par exemple, le plan projectif fini d'ordre 2, appelé le plan de Fano, voir figure 7.1. Il est composé de 7 points et 7 lignes. Chaque ligne contient 3 points. Soit l'ensemble des points $\mathcal{P} = \{1, 2, 3, 4, 5, 6, 7\}$, alors l'ensemble des lignes est $\mathcal{L}$={{1,2,4}, {2,3,5}, {3,4,6}, {4,5,7}, {1,5,6}, {2,6,7}, {1,3,7}}. La ligne {2,6,7} de la figure 7.1 est représentée par un cercle.

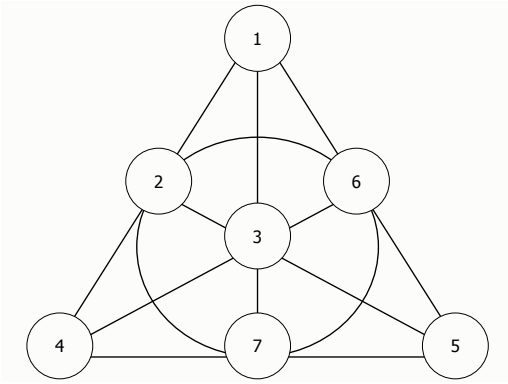
L'analogie avec notre problème est :

1. L'ordre n correspond au nombre $f - 1$: f est le nombre de pairs dans un bloc.
2. Les points d'un plan projectif fini d'ordre n sont des pairs, aussi $N = f^2 - f + 1$.
3. Les lignes d'un plan projectif fini d'ordre n sont des blocs, aussi $NB = f^2 - f + 1$.
4. L'intersection de deux blocs est 1.

Si l'ordre n est un nombre premier ou une puissance d'un nombre premier, l'existence d'un plan projectif fini d'ordre n est connu, voir les travaux de Albert et Sandler [2]. De plus, le théorème de Bruck-Ryser-Chowla [4] montre qu'un plan projectif fini d'ordre n existe, $n = 1$ ou $2 (\mathrm{mod} 4)$, alors l'ordre n est la somme de deux carrés. Ce théorème prouve par exemple qu'un plan projectif fini d'ordre 6, correspondant au problème des "36 officiers", ne peut exister. Une très longue démonstration informatisée, par Lam [35] en 1991, montre qu'un plan projectif fini d'ordre 10 n'existe pas non plus. Dans de nombreux autres cas, le problème de l'existence d'un plan projectif fini d'ordre n est toujours ouvert.

LEMME 3 *Une distribution basée sur la construction d'un projectif fini d'ordre* $f - 1$ *est une distribution idéale.*

FIG. 7.1 Le plan de Fano ou plan projectif fini d'ordre 2



Preuve

Par définition d'un plan projectif, $N = f^2 - f + 1$ *et* $NB = f^2 - f + 1$. *Cela implique :*

$$NB = \frac{N}{f} * \frac{N-1}{f-1} = NB_1$$

❑

Cette distribution est idéale, mais requiert N égal à $f^2 - f + 1$, c'est une trop grande restriction. De plus, pour certaines valeurs de f, trouver la construction d'un plan projectif d'ordre $f - 1$ reste un problème ouvert.

7.2.4 Distribution idéale basée sur les plans affines finis

Soit $N = f^2$. Dans ce cas, une distribution peut être définie à partir de la construction d'un plan affine fini d'ordre f, quand une telle construction est possible. Nous montrerons qu'une distribution basée sur une telle construction est une distribution idéale.

DÉFINITION 5 *Un* **plan affine fini** *est un plan affine avec un ensemble fini de points. Comme les lignes sont des ensembles de points finis et deux lignes sont similaires quand ils ont les même points, l'ensemble des lignes est fini. Les lignes d'un plan affine fini d'ordre* n *peut être divisé en* $n + 1$ *groupes déconnectés de* n *lignes parallèles. Ces groupes sont appelés classes parallèles. L'ordre d'un plan affine est le nombre* n, $n \geq 2$, *tel que :*

1. *Le nombre total de points est* n^2 *et nombre total de lignes est* $n(n+1)$.
2. *Toutes les lignes partagent* n *points et tout les points partagent* $n + 1$ *lignes.*
3. *L'intersection de deux lignes est au plus un.*

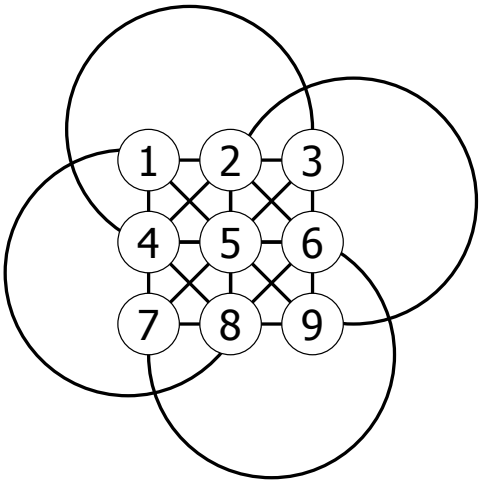
Par exemple, la figure 7.2 représente un plan affine fini d'ordre 3. Il a 9 points et 12 lignes. Soit $\mathcal{P}$={1,2,3,4,5,6,7,8,9} l'ensemble des points, alors l'ensemble des lignes est : $\mathcal{L}$={{1,2,3}, {4,5,6}, {7,8,9}, {1,4,7}, {2,5,8}, {3,6,9}, {1,5,9}, {2,6,7}, {3,4,8}, {3,5,7}, {4,2,9}, {6,8,1}}.

Aussi, l'analogie avec notre problème est :

1. L'ordre n correspond au nombre f de pairs dans un bloc.
2. Les points d'un plan affine fini d'ordre n sont les pairs, aussi $N = f^2$.
3. Les lignes d'un plan affine fini d'ordre n sont les blocs, aussi $NB = f^2 + f$.
4. L'intersection de deux blocs est au plus un.

La notion de plan affine fini d'ordre n est directement liée la notion de plan projectif fini d'ordre n. Un plan projectif fini d'ordre n existe, si et seulement si un plan affine fini d'ordre n existe.

FIG. 7.2 plan affine fini d'ordre 3



LEMME 4 *Une distribution basée sur la construction d'un plan projectif fini d'ordre* $f - 1$ *est une distribution idéale.*

Preuve

Par définition d'un plan projectif, $N = f^2$ *et* $NB = f^2 + f$. *Cela implique :*

$$NB = \frac{N}{f} * \frac{N-1}{f-1} = NB_1$$

❑

Les distributions basées sur les plans affine et projectif fini d'ordre n sont des distributions idéales. En considérant les restrictions sur les valeurs de N et f, et que les plans affine ou projectif fini n'existent pas toujours, une solution pour le cas général doit être trouvée. Trouver une solution pour le cas général est un problème mathématique ouvert, donc nous proposons une solution qui est proche de la distribution idéale.

7.3 Distribution CG

La distribution CG (distribution Cas Général) est conçue pour tout f nombre premier et pour tout N, $f^2 \leq N$. NB_1 est atteint dans certains cas particuliers, comme nous avons pu le constater auparavant. Aussi pour le cas général, nous proposons une distribution proche de l'optimal théorique. Afin de créer un ensemble de blocs ayant la propriété d'intersection souhaitée, nous allons dans un premier temps, construire les matrices M_i utilisées pour la construction de la distribution. Puis, nous prouverons que cette distribution est proche d'une distribution idéale à l'infini.

7.3.1 Construction matricielle

Soient r, s deux entiers, tel que $r^2 \leq s$. Considérons qu'il existe un entier premier p, tel que $p \times r \leq s$ et $r \leq p$. Trivialement, un tel entier p existe, dès lors que r est premier. Nous considérons les p matrices M_1, M_2, ..., M_p à p lignes et r colonnes définies par :

$$M_k = \left(a_{ij}^k\right)_{1\leq i\leq p; 1\leq j\leq r} \text{ où } a_{i1}^k = k$$

$$\text{et } a_{ij}^k = 1 + (j-1)\times p + ([i-1+(k-1)\times(j-2)] \bmod p)$$

$$\forall 1 \leq i \leq p \text{ et } \forall 2 \leq j \leq r$$

Par exemple, quand $r = 3$ et $s = 9$, nous avons $p = 3$, et les matrices M1, M2, M3 sont :

$$M_1 = \begin{bmatrix} 1 & 4 & 7 \\ 1 & 5 & 8 \\ 1 & 6 & 9 \end{bmatrix}, M_2 = \begin{bmatrix} 2 & 4 & 8 \\ 2 & 5 & 9 \\ 2 & 6 & 7 \end{bmatrix} \text{ et } M_3 = \begin{bmatrix} 3 & 4 & 9 \\ 3 & 5 & 7 \\ 3 & 6 & 8 \end{bmatrix}.$$

Remarquons que pour tout entier q, tel que $0 \leq q \leq r-1$, les entiers de l'intervalle $[1+p\times q; p\times(q+1)]$ apparaissent seulement dans la $(q+1)^{\text{eme}}$ colonne des matrices M_1, ..., M_k.

PROPOSITION 2
Deux différentes lignes des matrices M_1, ..., M_p ont au plus un élément en commun.

Preuve

Soient M_{k_1} et M_{k_2} deux matrices (pouvant être égales) parmi M_1, ..., M_p et soient $1 \leq i_1, i_2 \leq p$ deux entiers fixé, tels que $i_1 \neq i_2$ quand $k_1 = k_2$. Nous devons montrer que chaque ligne i_1^{eme} de la matrice M_{k_1} et la i_2^{eme} ligne de la matrice M_{k_2} ont au plus en commun un seul élément.

Puisque les ensembles d'entiers des colonnes sont indépendants. Nous prouvons que $\text{card}\{1 \leq j \leq r \mid a_{i_1 j}^{k_1} = a_{i_2 j}^{k_2}\} \leq 1$.

Si $j \geq 2$, $a_{i_1 j}^{k_1} = a_{i_2 j}^{k_2} \Leftrightarrow [i_1 - 1 + (k_1 - 1) \times (j-2)] \bmod p = [i_2 - 1 + (k_2 - 1) \times (j-2)] \bmod p$.

Alors, puisque nous considérons les classes modulo p, nous avons $\overline{i_1 - 1 + (k_1 - 1) \times (j-2)} = \overline{i_2 - 1 + (k_2 - 1) \times (j-2)}$.

Cela signifie que $\overline{k_1 - k_2} \times \overline{j-2} = \overline{i_2 - i_1}$.

Si $k_1 = k_2$, alors $1 \leq i_1, i_2 \leq p$, nous avons $i_1 = i_2$ cela contredit les hypothèses sur le choix des entiers i_1 and i_2 : aucun entier j vérifie l'équation. Alors nous avons nécessairement $k_1 \neq k_2$.

Puisque p est premier, $\overline{k_1 - k_2}$ admet un inverse et il existe exactement un seul entier $1 \leq j \leq p$ de telle sorte que $\overline{k_1 - k_2} \times \overline{j-2} = \overline{i_2 - i_1}$: nous avons le résultat dès lors $2 \leq j \leq r \leq p$.

Si $j = 1$, l'égalité $a_{i_1 j}^{k_1} = a_{i_2 j}^{k_2}$ implique $k_1 = k_2$: nous avons $a_{i_1 1}^{k_1} = a_{i_2 1}^{k_2}$ et, pour tout entier $j \geq 2$, $a_{i_1 j}^{k_1} \neq a_{i_2 j}^{k_2}$.

❑

7.3.2 Construction de la distribution

On rappelle que f est le nombre de fragments et N le cardinal de l'ensemble des pairs. Soit NB_P le nombre de blocs stockés par notre distribution.

On considère $f^2 \leq N$ et f premier. On définit deux entiers p_1 and p_2 de la manière suivante : l'entier p_1 est le plus grand nombre premier, tel que $p_1 \times f \leq N$ et $f \leq p_1$. L'entier p_2 est le plus grand entier, tel que $p_2 \times f \leq p_1$ et qui vérifie soit $p_2 < f$ soit $p_2 (\geq f)$ est premier.

PROPOSITION 3
$NB_P = p_1^2 + f \times p_2$ quand $p_2 < f$ et $NB_P = p_1^2 + f \times p_2^2$ quand $p_2 \geq f$

Preuve

Dans la section précédente, nous avons prouvé que nous pouvions construire p_1 matrices M_1, ..., M_{p_1} de telle sorte que deux différentes lignes aient au plus un élément en commun. Chaque ligne de ces matrices peut être considéré comme un bloc. Alors nous pouvons obtenir p_1^2 blocs.

Mais les ensembles d'entier p_1 sont construits de telle façon que les f colonnes des matrices soient indépendantes. Alors nous essayons de construire les blocs dans ces ensembles.

Quand $(1 \leq) p_2 < f$, un simple découpage en sous-ensembles de f éléments dans un ensemble avec p_1 éléments nous donne p_2 blocs pour chaque f colonnes.

Quand $p_2 (\geq f)$ est premier, pour chaque ensemble d'entier dans les f colonnes, nous construisons encore p_2 matrices de telle manière que deux différentes lignes de ces matrices aient au plus un élément en commun. Chaque ligne de ces matrices peuvent être considérées comme un bloc ($f \times p_2^2$ nouveaux blocs).

Donc $NB_P = p_1^2 + f \times p_2$ quand $p_2 < f$ et $NB_P = p_1^2 + f \times p_2^2$ quand $p_2 \geq f$.

❑

7.3.3 Des matrices à la distribution

En exécutant jusqu'à la fin notre algorithme avec $N = 9$ et $f = 3$, nous obtenons les matrices suivantes : $M_1 = \begin{bmatrix} 1 & 4 & 7 \\ 1 & 5 & 8 \\ 1 & 6 & 9 \end{bmatrix}$, $M_2 = \begin{bmatrix} 2 & 4 & 8 \\ 2 & 5 & 9 \\ 2 & 6 & 7 \end{bmatrix}$, $M_3 = \begin{bmatrix} 3 & 4 & 9 \\ 3 & 5 & 7 \\ 3 & 6 & 8 \end{bmatrix}$ et $M_4 = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$.

Le stockage des blocs à partir des matrices se déroule de la façon suivante : chaque ligne des matrices correspond a un ensemble de f pairs distincts devant stocker les f frements d'un bloc. Un pair stockant un seul fragment d'un même bloc.

Le schéma 7.3 représente un exemple de stockage de 4 blocs, indiqués par les lignes {1,4,7}, {1,2,3},{4,5,6} et {7,8,9}.

Pour le stockage additionnel de blocs, une fois toutes les lignes utilisées, on cycle à nouveau sur l'ensemble des lignes. De ce fait, le coût incrémente par pas de 1.

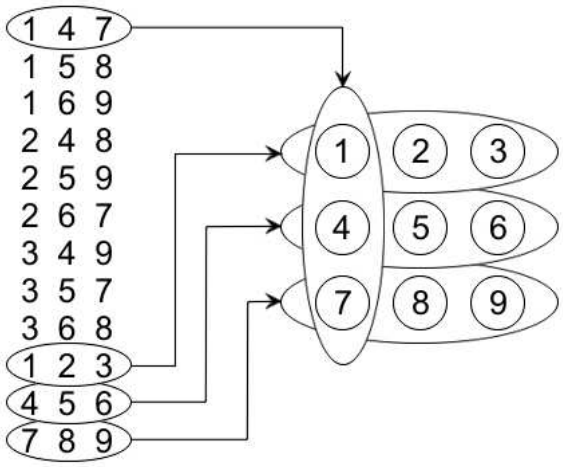
Pour plus d'exemples de matrices, se référer à l'annexe B.

7.3.4 Analyse de la distribution

PROPOSITION 4
Notre construction est idéale quand $N = f^2$ avec f un nombre premier.

Preuve

FIG. 7.3 Représentation de blocs stockés issus des lignes des matrices



Si $N = f^2$ *et* f *est un nombre premier, nous avons* $p_1 = f$ *et* $p_2 = 1$. *Alors* $NB_P = f^2 + f$, *et par définition de* NB_1, $NB_P = NB_1$.

❑

PROPOSITION 5
Lorsque f est fixé et N tend vers l'infini, le nombre de blocs NB_p and NB_1 sont équivalents.

Preuve

Pour montrer que notre distribution est équivalente à une distribution idéale, considérons que N *tend vers l'infini. Alors pour un coût de* $C_{max} = 1$, *nous avons juste besoin de montrer que* $NB_P \approx NB_1$.

Par définition, à l'infini, $NB_1 \approx \frac{N^2}{f^2}$. *Nous savons qu'à l'infini, l'écart entre deux nombre premiers est logarithmique, voir le théorème [19] de la densité des nombres premiers. Puisque* $\log(\frac{N}{f})$ *est négligeable devant* $\frac{N}{f}$ *à l'infini, nous avons* $p_1 \approx \frac{N}{f}$ *et* $NB_P \approx p_1^2$. *Ainsi,* $NB_P \approx \frac{N^2}{f^2}$ *à l'infini.*
Pour résumer, notre distribution est asymptotiquement idéale.

❑

0n peut également remarquer que $NB_P = p_1^2 + f \times p_2$ quand $p_2 < f$ et $NB_P = p_1^2 + f \times p_2^2$ quand $f^2 \leq p_2$.

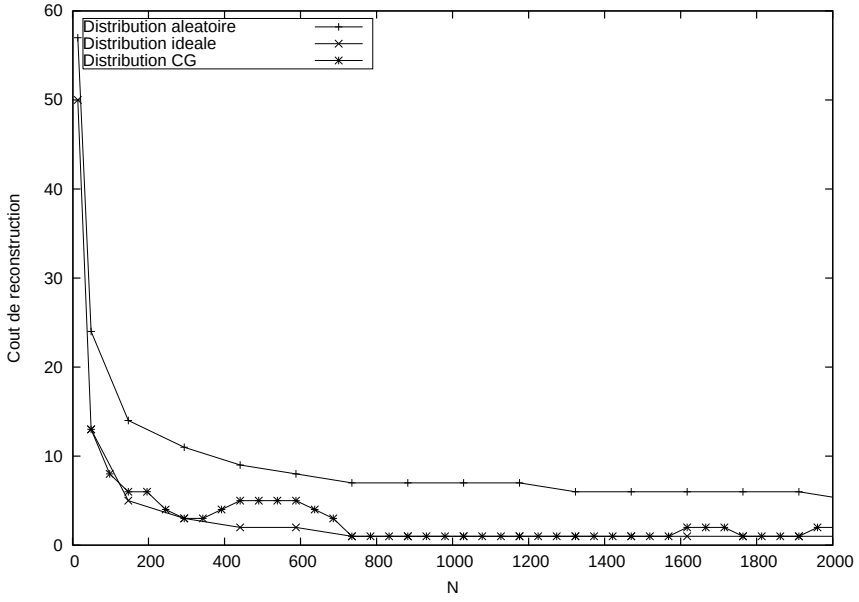
Il est possible de définir un nouveau entier premier p_3, tel que $p_3 \times f \leq p_2$. Avec un raisonnement similaire, nous pourrions obtenir $p_1^2 + f \times p_2^2 + f \times p_3^2$ blocs. Et ce raisonnement peut être à nouveau appliqué dès que N grandit avec p_4, un nouveau entier premier et ainsi de suite.

Pour résumer, l'algorithme de la distribution CG est asymptotiquement idéal.

Maintenant, à l'aide d'un simulateur de coût de reconstruction, une comparaison du coût entre différentes distributions est donnée.

7.4 Résultats expérimentaux

FIG. 7.4 Evolution du coût de reconstruction en fonction de N pour $f = 7$.



Dans un premier temps, comparons le coût de reconstruction en fonction de la valeur du nombre total de pairs N pour les trois distributions : la distribution aléatoire, la distribution idéale, et la distribution CG.

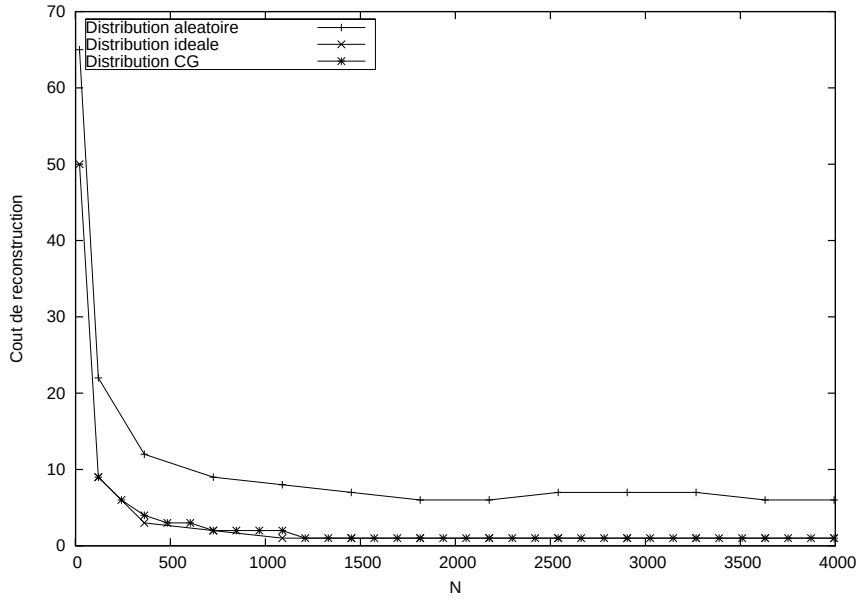
La distribution idéale est juste une limite théorique. Nous avons vu auparavant qu'il n'existe toujours pas à l'heure actuelle une telle distribution pour toute valeur de f et N (problème ouvert). Et la dernière distribution présentée : la distribution CG prouvée proche de la distribution idéale à l'infini.

Pour comparer les différentes distributions de données, nous avons écrit un simulateur. Ce simulateur calcule le coût de reconstruction en fonction d'une distribution donnée et de la valeur de f, N et NB.

Dans les simulations executées pour obtenir les figures 7.4 et 7.5, chaque pair stocke environ 100 fragments. Dans la figure 7.4, la valeur maximum de N est 2000. Dans la figure 7.5, la valeur maximum de N est 4000.

Comme prévu, nous pouvons voir que le comportement des coûts de

FIG. 7.5 Evolution du coût de reconstruction en fonction de N pour $f = 11$.



toutes les distributions sont toujours les mêmes. Le coût de la reconstruction diminue lorsque la valeur de N augmente.

La distribution aléatoire est très stable pour de grandes valeurs de N. Ceci est dû à la faible probabilité d'obtenir des groupes de f pairs égaux avec un grand nombre de pairs.

La distribution CG est toujours meilleure que la distribution aléatoire (voir figure 7.5), où le coût de reconstruction est environ dix fois moins que le coût de la distribution aléatoire. La distribution CG est proche de la distribution idéale. Mais, on peut noter sur la figure 7.4 quelques piques dans le coût de reconstruction de la distribution CG. Ceci est dû à la non optimalité de la distribution CG pour toutes valeurs de N, et spécialement pour les petites valeurs de f, comme dans la figure 7.4. Pour les plus grandes valeurs, supérieures ou égales à 11, les variations sont moins significatives (voir figure 7.5).

Comparons la distribution CG par rapport à la distribution idéale pour des valeurs usuelles. N est choisi entre 1000 et 50000, et f entre 5 et 53. Pour cela, pour chaque N et f, nous observons le nombre de pairs nécessaire pour stocker NB blocs pour la distribution CG et la distribution idéale avec $C_{max} = 1$. Nous rappelons que pour $C_{max} = k > 1$, le nombre de blocs est proportionnel au nombre de blocs obtenus avec $C_{max} = 1$. Tous les graphiques générés montrent que notre algorithme n'est jamais plus mauvais que 5% et souvent moins que 0.1%.

FIG. 7.6 Nombre de pairs N pour $C_{max} = 1$ et $f = 23$ en fonction du nombre de blocs stockés NB.

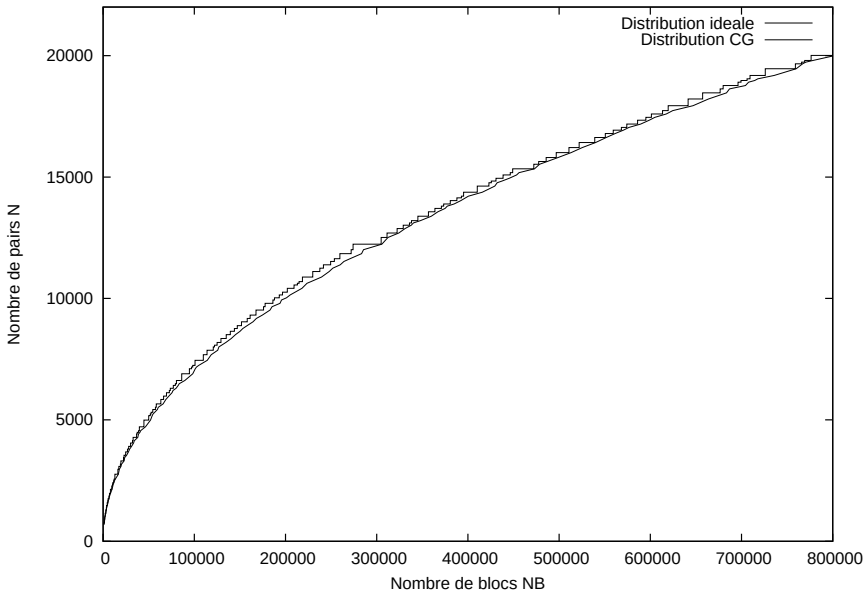
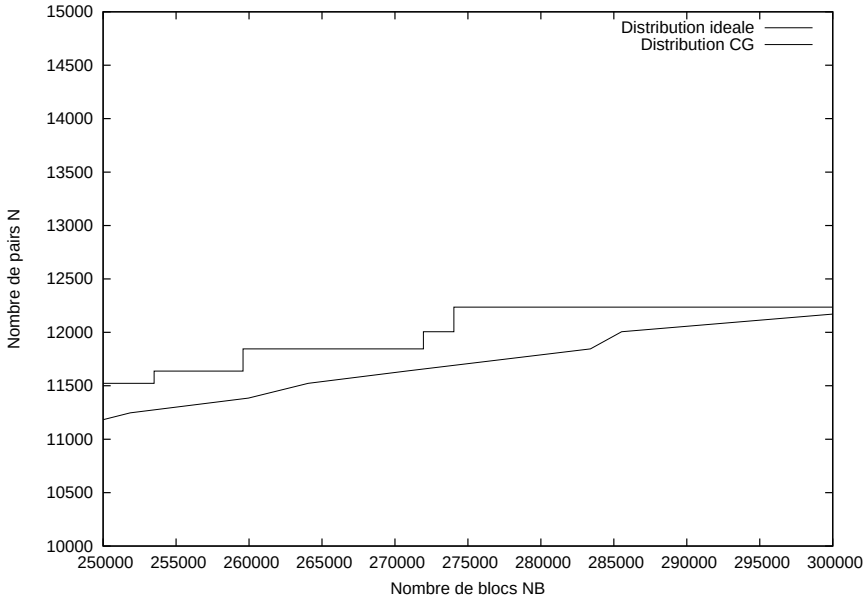


FIG. 7.7 Zoom de la figure 7.6 pour NB = 250000 à 300000



Pour un système de stockage pair à pair, il est intéressant de maximiser pour un coût de communication la capacité de stockage. Par exemple, la

figure 7.6 montre le nombre de pairs devant participer à la distribution des blocs pour un coût $C_{max} = 1$, dépendant du nombre de blocs stockés, f est fixé à 23. Dans un premier temps, nous observons que les courbes suivent la racine carrée de $NB * f^2$, comme cité précédemment dans l'analyse de la distribution CG.

La seconde observation est donnée par le zoom 7.7 de la figure 7.6, avec une intervalle de nombre de blocs variant de 250000 à 300000. La courbe résultante de l'algorithme de la distribution CG évolue en plateau. Cela vient de notre construction qui utilise des valeurs $p1$ et $p2$ qui n'évolue pas pour tout N, d'où une apparition d'un effet plateau.

Finalement, même pour des valeurs usuelles, la distribution CG est très proche d'une distribution idéale.

Après avoir étudié des distributions de données en environnement statique, i.e pour des valeurs de N fixées, nous voulons trouver des distributions qui prennent en compte le comportement dynamique des réseaux pair à pair.

Chapitre 8

Distribution de données en environnement dynamique

Un réseau pair à pair est un réseau dynamique. Les pairs se connectent et se déconnectent régulièrement, alors que d'autres tombent en pannes. Il faut donc adapter les distributions, vues précédemment, à cet environnement dynamique.

Dans la suite, nous introduisons la notion de structure. Une structure décrit comment les nœuds (ou pairs) sont arrangés de façon déterministe, de part la distribution utilisée. Une structure est définie par tous les liens existants.

Par exemple, un tel type de structure est visible sur la figure 7.2. Soit $\mathcal{P}$={1,2,3,4,5,6,7,8,9} l'ensemble des points, alors l'ensemble des liens est : $\mathcal{L}$={{1,2,3}, {4,5,6}, {7,8,9}, {1,4,7}, {2,5,8}, {3,6,9}, {1,5,9}, {2,6,7}, {3,4,8}, {3,5,7}, {2,4,9}, {1,6,8}}

Considérons une distribution optimale structurée, comme celle de la figure 7.2, qui optimise les envois de fragments. Lorsqu'un pair tombe en panne, les fragments reconstruits doivent être stockés sur d'autres pairs. D'une part, plusieurs pairs peuvent stocker de nouveaux fragments. La structure est alors explosée. D'autre part, de nouveaux fragments peuvent être stockés sur un même nouveau pair pour garantir que la structure endommagée soit reconstruite. L'effet de la parallélisation des envois est alors annulé par les réceptions séquentielles.

Ces deux conditions sont satisfaites en utilisant un groupe de pairs au lieu d'un unique pair pour chaque nœud de la structure. De tels groupes sont appelés Métapairs. C'est la distribution la plus adaptée à l'architecture de notre système de stockage pair à pair "Us". L'objectif de cette distribution Métapairs [55, 57] est de gérer le comportement dynamique des pairs

(arrivée/départ et pannes fréquents des pairs) tout en minimisant le coût de reconstruction.

8.1 Gestion des pannes : les pannes d'ensemble

Le système doit prendre en compte le comportement volatile des pairs du réseau. Il doit donc aussi gérer les pannes d'ensemble : selon la géographie, la panne d'un pair peut être en rapport avec les pannes d'autres pairs.

Par exemple, une défaillance électrique qui touche un quartier ou une inondation. La proximité géographique des pairs implique donc des pannes d'ensemble. Les pairs, qui appartiennent physiquement a un même réseau, sont plus "vulnérables". Si ce réseau tombe en panne, alors tous les pairs du réseau sont déconnectés.

Cette notion de pannes d'ensemble [78] est un facteur important à prendre en compte pour la technique de tolérance aux pannes. Les pairs sélectionnés pour la dissémination des fragments d'un bloc de données doivent faire face aux pannes d'ensemble. Sinon celles-ci peuvent rendre le mécanisme de redondance inopérant.

Pour intégrer nativement, au sein du système, la gestion des pannes d'ensemble est implicitement effectuée par les Métapairs.

8.2 Métapairs

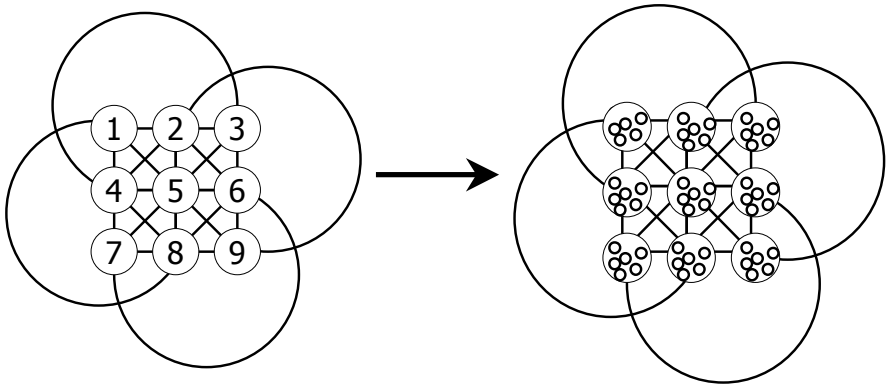
Dans cette distribution, les pairs sont arrangés selon leur géographie d'ensemble, en groupes appelés *Métapairs*. Chaque pair appartient exactement à un Métapair. Un couple de pairs qui montre une probabilité élevée de panne appartient à un même Métapair. Ainsi, un couple de pairs provenant de Métapairs distincts doit montrer une faible probabilité de panne commune. Quand un bloc de données est disséminé, les pairs choisis pour stocker les fragments sont sélectionnés parmi des Métapairs différents. Deux fragments d'un même bloc ne peuvent être stockés sur les pairs d'un même Métapair. Par conséquent, tous les pairs d'un même Métapair peuvent tomber en panne sans perte de données grâce à l'information de redondance.

8.3 Distribution basée sur les Métapairs

Dans cette distribution, l'ensemble des pairs est partitionné en groupes appelés Métapairs. Nous rappelons que la distribution basée sur les Métapairs gère les pannes d'ensemble. Due à ce problème de pannes d'ensemble, notre but est d'obtenir une distribution adaptée à la structure des Métapairs, et qui autorise des valeurs dynamiques pour N.

Définissons la distribution dynamique basée sur les Métapairs. Le procédé est simple : on remplace les pairs par les Métapairs. L'ensemble de ces Métapairs est structuré par une distribution, voir figure 8.1. Un nœud i de la distribution est remplacé par un Métapair i. Un fragment stocké sur un nœud i sera stocké sur un des pairs du Métapair i. Nous utilisons la distribution CG, ayant fait ses preuves précédemment.

FIG. 8.1 De la distribution à la distribution Métapairs dynamique



Ensuite, nous allons voir comment le routage s'effectue au sein des Métapairs et des explications sur le processus de reconstruction.

Grâce a cette distribution structurante, nous sommes en mesure de définir un mécanisme pour la gestion du comportement dynamique des systèmes de stockage pair à pair.

Lorsqu'un nouveau pair arrive, le Métapair qu'il intégrera est d'abord sélectionné. Le Métapair est choisi de telle manière que le nouveau pair se trouve, géographiquement parlant, loin des pairs des autres Métapairs (en respectant quelques critères d'équilibre), ceci pour optimiser le temps de vie des données. De même, le nouveau pair devra avoir une bonne bande passante avec les autres pairs du Métapair choisi. La raison de ce dernier critère est que, lorsqu'une reconstruction est enclenchée, le pair devra envoyer ses fragments aux autres pairs du Métapair en question.

Les fragments d'un bloc sont distribués sur les Métapairs de la struc-

ture. Pour chaque Métapair choisi, une fonction spécifique sélectionne les pairs stockeurs. Afin d'équilibrer le stockage, cette fonction est modifiée pour tendre à choisir les pairs qui stockent le moins de données. Cette fonction doit prendre en compte la topologie du réseau.

En résumé, un Métapair ne stocke pas plus d'un fragment d'un bloc. si ce fragment disparait, il est régénéré sur les autres pairs des Métapairs. Pour un même bloc, les fragments sont stockés parmi des Metapairs différents. Les fragments stockés par un Métapair appartiennent donc tous à des blocs différents. Les fragments sont reconstruits par les pairs du même Métapair.

Réception dans la recontruction

Pour le processus de reconstruction, si un pair tombe en panne au sein d'un Métapair, les fragments seront reconstruits par les pairs du même Métapair. L'ensemble des pairs participant à la reconstruction devra être choisi afin d'optimiser la réception des fragments utiles. Une réception optimale arrive dès lors que la taille des Métapairs est plus grande que le nombre de fragments à envoyer.

8.4 Distribution au dessus des Métapairs

Dans la distribution basée sur les Métapairs, un nœud i de la distribution CG est remplacé par Métapair i. Cette fois, l'objectif est de définir quel nœud doit être sélectionné dans le Métapair afin d'optimiser au mieux la distribution dynamique en terme de coût de reconstruction. Maintenant nous allons présenter une nouvelle stratégie basée sur la distribution CG.

Par la suite, nous utiliserons les notations suivantes : MP est le nombre de Métapairs, MP_{Size} est la taille des Métapairs (le nombre de pairs dans un Métapair), MP_i est le Métapair i, $\mathcal{P}_{MP_i}$ est l'ensemble des pairs du Métapair i, et p_j^i est le j^{eme} pair du Métapair i tel que $1 \leq i \leq MP_{Size}$.

Par exemple avec $MP = 9$ et $f = 3$, on obtient les matrices suivantes :

$$M_1 = \begin{bmatrix} MP_1 & MP_4 & MP_7 \\ MP_1 & MP_5 & MP_8 \\ MP_1 & MP_6 & MP_9 \end{bmatrix}, M_2 = \begin{bmatrix} MP_2 & MP_4 & MP_8 \\ MP_2 & MP_5 & MP_9 \\ MP_2 & MP_6 & MP_7 \end{bmatrix}, \text{etc...}$$

Ces matrices sont nommées matrices Métapairs. Chaque ligne représente un stockage d'un bloc dans la distribution CG appliquée sur les pairs.

Une fonction sélectionne le pair stockeur dans chaque Métapair. Cette fonction se réfère aux matrices Métapairs. Elle peut être basée sur l'algo-

rithme de la distribution CG, lorsque la taille des Métapairs est suffisante pour l'appliquer.

Dans ce cas, pour chaque ligne des matrices Métapairs, plusieurs blocs peuvent être stockés sans augmenter le coût de reconstruction. Par exemple, avec la première ligne (MP_1, MP_4, MP_7), les f fragments du premier bloc sont stockés sur les pairs (p_1^1, p_1^4, p_1^7). Ensuite, les fragments du prochain bloc sont stockés sur la même ligne sur les peers (p_2^1, p_2^4, p_2^7) et ainsi de suite jusqu'au dernier bloc qui sera stocké sur les pairs $(p^1_{MP_{Size}}, p^4_{MP_{Size}}, p^7_{MP_{Size}})$.

Dans un premier temps, nous calculons le nombre maximum de bloc pouvant être stocké sur une seule ligne des matrices Métapairs, sans augmenter le coût de reconstruction.

LEMME 5 *Soit* N *le nombre total de pairs groupé en Métapairs,* E *un ensemble de* f *Métapairs et* MP_{Size} *la taille de chaque Métapair, les Métapairs ont tous la même taille. Soit* NB *le nombre maximum théorique de blocs* NB *pouvant être stocké par* E*, un fragment par Métapair avec un coût de reconstruction* 1*, alors :*

$$NB = MP_{Size}^2$$

Preuve

Soit p *égal à* MP_{Size} *et* $E_1, E_2, ..E_f$ *les* f *ensembles de* E *, dont chaque ensemble dispose de* p *éléments. Nous voulons créer une famille* D *de cardinalité maximum de p-uples* $(x_1, x_2, ..., x_f)$ *qui vérife* $x_i \in E_i$ *et pour chaque couple disctinct de p-uples* $(a_1, a_2, ..., a_f)$ *et* $(b_1, b_2, ..., b_f)$ *de* D*. Le nombre d'indices* i *est tel que* $a_i = b_i$ *doit avoir au plus un élément en commun.*

Imaginons que cette famille D *existe. Nécessairement, si nous prenons deux ensembles dans* $E_1, E_2, ..E_f$*, tout les couples d'éléments de ces deux ensembles doivent avoir au maximum un élément dans* D*.*

Donc, le nombre maximum d'éléments de D *est le nombre maximum de couples, soit* $p \times p$*.*

Remarque : le nombre maximum peut être obtenu, si p *est un nombre premier et* $p \geq f$*.*

❑

Notre but est de trouver une distribution à l'intérieur de chaque Métapair qui autorise un nombre de blocs proche de la limite théorique. Pour se faire, nous utilisons la même stratégie que celle de la distribution CG qui sera appliquée sur les pairs à l'intérieur des Métapairs. Les matrices de distribution sont définies comme les matrices Métapairs vues dans la section 7.3.

Soit $E_1, E_2, ..E_f$ une ligne des premières matrices. Soit $p = MP_{Size}$ et $p_z^{E_i}$ un pair tel que $0 \leq i \leq f-1$. Nous considérons les p matrices MM_1, MM_2,

..., MM_p avec p lignes et f colonnes définies par $MM_k = \left(a_{ij}^k\right)_{1\leq i\leq p; 1\leq j\leq f}$ où $a_{i1}^k = p_k^{E_1}$ et $a_{ij}^k = p_z^{E_j}$ où $z = 1 + ([i - 1 + (k - 1) \times (j - 2)] \mathrm{mod} p)$ $\forall 1 \leq i \leq p$ et $\forall 2 \leq j \leq f$.

Par exemple, $f = 3$, $MP = 9$ et $MP_{Size} = 3$:

$$MM_1 = \begin{pmatrix} p_1^{E_1} & p_1^{E_2} & p_1^{E_3} \\ p_1^{E_1} & p_2^{E_2} & p_2^{E_3} \\ p_1^{E_1} & p_3^{E_2} & p_3^{E_3} \end{pmatrix}, MM_2 = \begin{pmatrix} p_2^{E_1} & p_1^{E_2} & p_2^{E_3} \\ p_2^{E_1} & p_2^{E_2} & p_3^{E_3} \\ p_2^{E_1} & p_3^{E_2} & p_1^{E_3} \end{pmatrix},$$

$$MM_3 = \begin{pmatrix} p_3^{E_1} & p_2^{E_2} & p_1^{E_3} \\ p_3^{E_1} & p_3^{E_2} & p_2^{E_3} \\ p_3^{E_1} & p_1^{E_2} & p_3^{E_3} \end{pmatrix}$$

Remarque : pour tout couple d'entiers $1 \leq z, t \leq MP_{Size}$, les pairs $p_z^{E_j}$ avec j dans l'intervale $[1+p\times i; p\times(i+1)]$ apparaissent seulement dans la $(i+1)^{eme}$ colonne des matrices MM_1, ..., MM_k. Et deux différentes lignes parmi les matrices MM_1, ..., MM_p ont au plus un élément en commun.

Comme dans la distribution CG, le nombre de blocs est optimal si MP_{Size} est un nombre premier. Donc l'implémentation de la distribution basée sur les Métapairs est très proche de la distribution CG existante, mais MP_{Size} doit impérativement respecter cette condition. Or dans un environnement dynamique, il est impossible de toujours valider cette condition. C'est pour cette raison que nous utilisons une distribution Métapair hybride. Cette distribution choisit la fonction de sélection des pairs stockeurs à l'intérieur des Métapairs en fonction de la valeur de MP_{Size}. Si MP_{Size} n'est pas un nombre premier, alors la fonction de sélection des pairs est la fonction aléatoire, sinon c'est l'algorithme de la distribution CG.

8.5 Coût intrinsèque de la distribution basée sur les Métapairs

Dans cette partie nous comparons le coût de reconstruction de la distribution aléatoire et de la distribution CG avec différentes tailles de Métapair.

Pour nos expérimentations, nous utilisons un simulateur qui calcule le C_{max} en fonction de la valeur de N, f, et d'une distribution donnée.

Les figures 8.2 et 8.3 montre l'impact du coût de reconstruction en fonction de la valeur N. Pour ces simulations, chaque pair stocke environ 100 blocs, i.e. chaque pair peut lire et stocker 100 blocs.

FIG. 8.2 Coût de la reconstruction en fonction de N et f = 5 pour la distribution aléatoire et différents nombres de Métapairs.

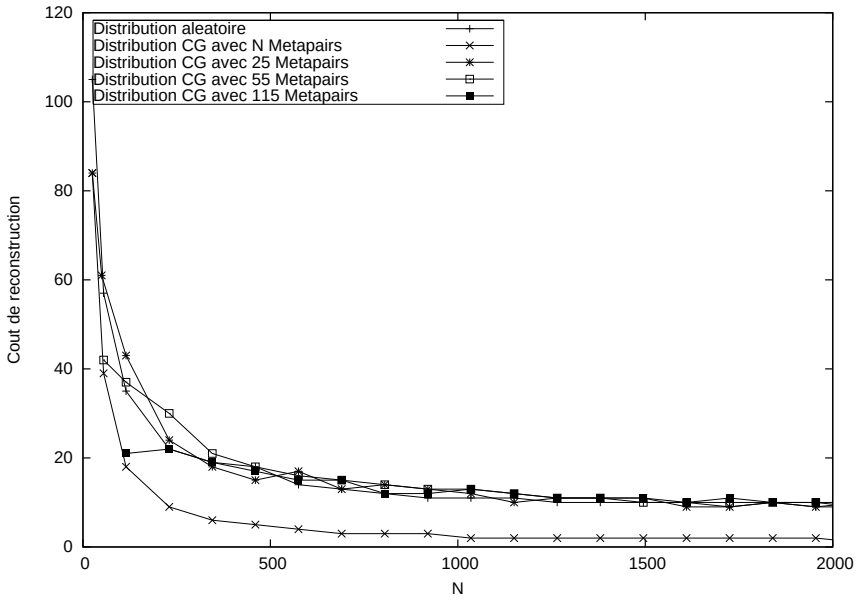
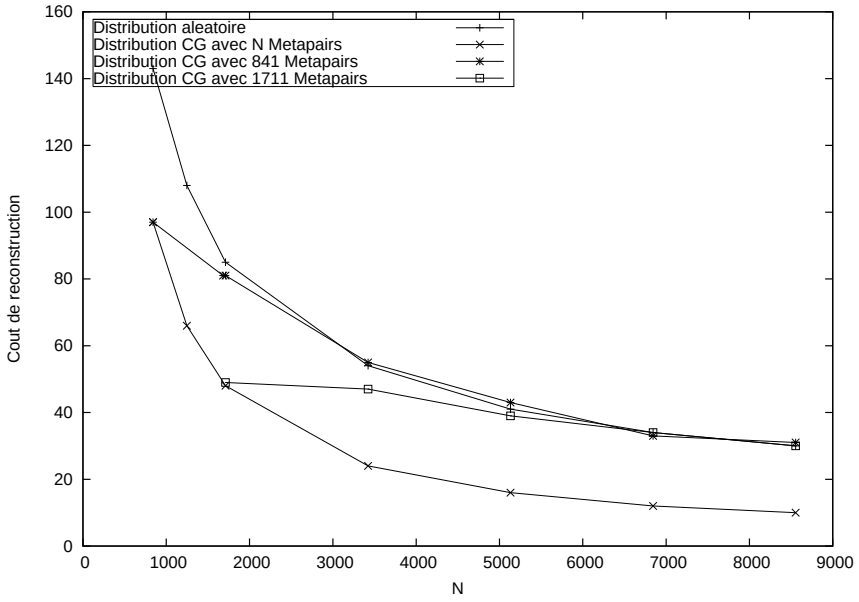


FIG. 8.3 Coût de la reconstruction en fonction de N et f = *29* pour la distribution aléatoire et différents nombres de Métapairs.



Les paramètres de la figure 8.2 sont $f = 5$, $N = 0$ à 2000, $NB = 0$ à 200000. Les paramètres de la figure 8.3 sont $f = 29$, $N = 0$ à 9000, $NB = 0$ à 900000.

Nous considérons toujours que la taille des Métapairs est la même pour tout les Métapairs. Par conséquence, les figures 8.2 et 8.3 montrent le coût de reconstruction pour différentes tailles de Métapairs en fonction de N. Par exemple, le premier point donné par la courbe de la distribution par Métapairs est obtenu avec une taille de Métapair égale à un, i.e un pair par Métapair. Par conséquence avec une valeur de N égale au nombre total de Métapairs.

La figure 8.3 montre que pour de petites valeurs de la taille des Métapairs, le coût de la distribution aléatoire est plus mauvais que celui de la distribution des Métapairs. Cela confirme l'avantage de calculer la distribution CG contre la distribution aléatoire. Une autre observation, donnée par la figure 8.2, est au sujet de la taille des Métapairs. Le coût de la distribution Métapairs est proche de celui de la distribution aléatoire, lorsque la taille des Métapairs est plus grande que deux. Aussi nous n'avons pas besoin d'avoir un grand nombre de Métapairs. Ainsi il n'est pas nécessaire d'avoir une grande structure pour gérer la corrélation des pannes.

La figure 8.2 montre que même si la taille des Métapairs grossit, le coût de la distribution Métapairs est toujours plus proche du coût de la distribution aléatoire. Nous pouvons conclure que le coût, induit pour gérer la corrélation des pannes et pour réussir à obtenir une distribution en environnement dynamique, n'est pas si élevé.

8.6 Analyse de la distribution Métapairs dans un environnement dynamique

Dans cette partie, nous observons l'évolution du coût de communication C_{max} durant une simulation du système. Pour se faire, nous simulons les distributions sur un ensemble de pairs, puis des pannes de pairs sont simulés. A chaque temps un pair meurt, les fragments qu'il stockait sont redistribués sur d'autres pairs. Ensuite un nouveau pair apparaît, afin de maintenir constant le nombre total de pairs dans le système.

La figure 8.4 montre l'évolution de C_{max} en fonction du taux de départ/arrivée des peers dans le temps, appelé "churn" [37]. Le nombre de pannes est fixé à 1000, C_{max} est mesuré toutes les 25 pannes. Chaque pair stocke 100 blocs. Les distributions sont :

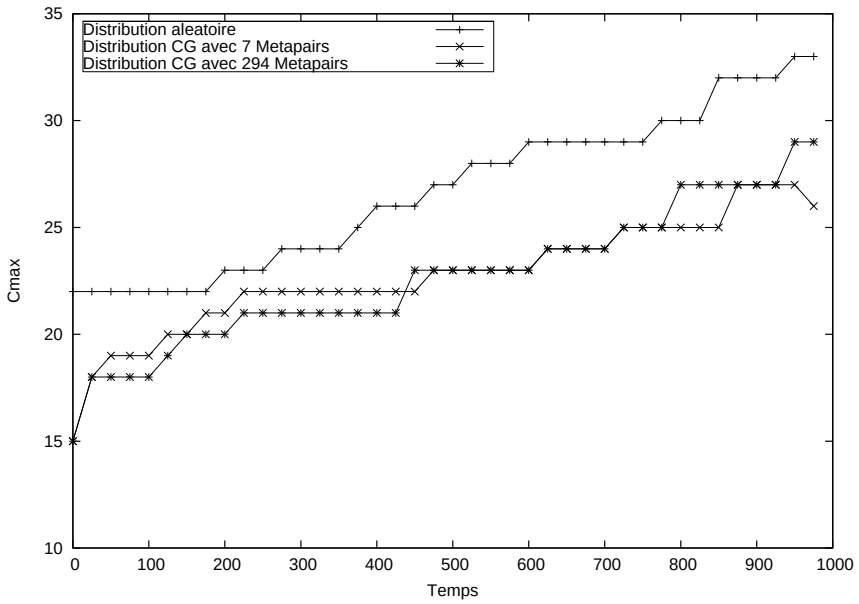
– **la distribution aléatoire :** distribution sur des pairs différents pour un même bloc, comme expliqué dans la section 7.1. Les reconstruc-

tions sont exécutées en utilisant un choix aléatoire parmi les pairs qui ne sont pas tombés en panne, et qui ne sont pas déjà utilisés pour le stockage des blocs endommagés.

– **Nombre de Métapairs = 7 :** distribution présentée dans la section 8.4, avec $f = 7$, et $MP = 7$ Métapairs de taille 285 et 286. Le nombre de Métapairs doit être un multiple de f pour obtenir de bons résultats avec la distribution CG. La reconstruction est exécutée en utilisant un choix aléatoire sur les pairs restant dans le Métapair du pair en panne.
– **Nombre de Métapairs = 284 :** même distribution, mais avec $MP = 284$ Métapairs de taille 7 et 8.

Plus de Métapairs aurait généré des tailles de Métapairs en dessous de un. Si la taille des Métapairs est en dehors de ces deux valeurs extrêmes, la distribution est moins efficace en terme de coût.

FIG. 8.4 Distribution CG avec différentes tailles de Métapair.



Au début, le coût de notre distribution est légèrement plus bas que celui de la distribution aléatoire. Pendant la première étape, jusqu'à 250 pannes environ, le coût de la distribution CG grossit plus rapidement que celui de la distribution aléatoire et atteint le même point. Puis il augmente plus doucement, augmentant ainsi le fossé entre les deux coûts. Ainsi, notre distribution a un meilleur comportement face aux pannes pour un coût réduit.

8.7 Conclusion

L'objectif de ce chapitre a été d'étudier le problème de la distribution de données au sein du système de stockage pair à pair Us. Lors d'une étude précédente [75], il a été prouvé qu'un tel système doit faire face à un continuel flux de reconstructions. Lorsque un pair tombe en panne, il faut reconstruire les fragments qu'il stockait. Les pairs qui stockent les fragments du même bloc envoient alors des données sur le réseau pour régénérer les fragments perdus sur un nouveau pair. Pour minimiser l'encombrement du réseau, il faut donc trouver une distribution qui minimise le coût de reconstruction des données.

Nous avons donc analysé comment obtenir une distribution optimale des données. Nous avons montré qu'une distribution idéale est une distribution optimale. Une distribution idéale de coût quelconque est toujours fondée sur une distribution idéale de coût 1. Nous avons présenté deux distributions de coût 1 pour des valeurs spécifiques de f et de N. Finalement, comme il s'agit d'un problème mathématique ouvert, nous proposons une solution dans le cas général (pour toutes valeurs de f et N) proche d'une distribution idéale pour un coût de 1 à l'infini. Dans la pratique, les résultats expérimentaux montrent que notre distribution est très proche de l'idéal. Mais une distribution optimale est trop stricte et n'est pas adaptée au comportement dynamique des systèmes pairs à pairs.

En ce qui concerne les distributions dynamiques, nous avons pu remarquer que la distribution aléatoire de données est, dans la plupart des cas, une bonne stratégie pour minimiser le coût de reconstruction. Mais malgré cela, cette distribution ne permet pas d'exploiter la topologie physique du réseau pour prendre en compte les pannes d'ensemble. Nous proposons donc une distribution où l'ensemble des pairs est partitionné en groupes appelés Métapairs et l'ensemble de ces Métapairs est structuré par la distribution CG. Chaque bloc de données est alors distribué dans les Métapairs en suivant un algorithme basé à la fois sur une sélection cyclique des pairs et une sélection selon notre algorithme de création de la distribution CG. Après le développement d'un simulateur pour tester nos distributions, nous avons pu constater lors de diverses exécutions en faisant varier certains paramètres (par exemple le nombre de Métapairs), que la distribution dynamique basée sur les Métapairs, selon le nombre de Métapairs, minimise le coût de reconstruction aussi bien que la distribution aléatoire, voir mieux dans certains cas, tout en prenant en compte les pannes d'ensemble.

Chapitre 9

Conclusion

L'objectif premier de cette thèse a été de concevoir un prototype de système de stockage pair à pair pérenne, nommé Us. Puis, le second objectif a été d'étudier le problème de la distribution de données au sein de ce système.

Le premier prototype Us est désormais opérationnel et fonctionnel. D'un point de vue utilisateur, il dispose de primitives très simples de manipulation (PUT, GET) de blocs de données. Un processus démon de reconstruction exécute en arrière plan les reconstructions des blocs endommagés en utilisant l'algorithme de ReedSolomon. Ceci afin de garantir la pérennité des données.

Nous avons créé aussi une interface plus usuelle, nommée UsFS. Cette interface offre les fonctionnalités d'un système de fichier avancé (avec gestion de cache, versioning).

Ensuite, nous nous sommes intéressés à la distribution des données au sein du réseau. Pour cela, nous avons analysé comment obtenir une distribution idéale des données prenant en compte une nouvelle mesure : le coût de reconstruction. Ce coût indique la perturbation maximale d'un pair en terme de communication lors d'une reconstruction. Comme il s'agit d'un problème mathématique ouvert, nous proposons une solution proche d'une distribution idéale. Dans la pratique, les résultats expérimentaux montrent que notre distribution est très proche de l'idéale. Mais une distribution optimale est trop stricte et n'est pas adaptée au comportement dynamique des systèmes pairs à pairs.

Finalement, nous nous sommes donc intéressés aux distributions dynamiques. Nous avons pu remarquer que la distribution aléatoire de données est, dans la plupart des cas, une bonne stratégie pour minimiser le coût de

reconstruction. Mais malgré cela, cette distribution ne permet pas d'exploiter la topologie physique du réseau pour prendre en compte les pannes d'ensemble. Nous proposons donc comme solution une distribution où l'ensemble des pairs est partitionné en groupes appelés Métapairs et l'ensemble de ces Métapairs est structuré par une distribution idéale.

Les perspectives, concernant l'architecture du système de stockage Us, seraient d'implémenter une version totalement distribuée. Puis, il serait souhaitable de tester Us à grande échelle en utilisant, par exemple, la grille expérimentale d'échelle nationale Grid'5000. De même, il serait intéressant de tester l'efficacité des différents mécanismes de redondance et intégrer le plus performant (exemple : Tornado).

Les perspectives dans le domaine de la distribution seraient d'étudier une implémentation distribuée de notre méthode pour l'intégrer au sein du système de stockage pair à pair Us. Ainsi, diverses études pourraient être menées, tel que l'impact de la gestion du coût de reconstruction sur les performances globales du système ou l'influence sur la pérennité des données.

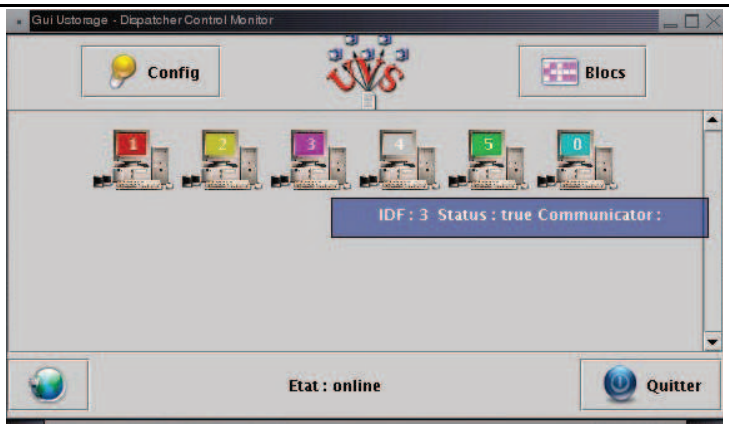
Enfin, un critère commun d'évaluation à l'ensemble des systèmes de stockage pair à pair est désormais à considérer : le taux de départ/arrivées des pairs (le churn). Des études préliminaires [60] ont été menées sur quelques systèmes. Il serait judicieux d'étudier l'influence de ce facteur sur Us et d'établir un comparatif de performance sur une sélection de divers systèmes de stockage pair à pair.

Annexe A

Interfaces graphiques Us

A.1 Interface graphique du dispatcher Us

FIG. A.1 GUI moniteur du dispatcher Us



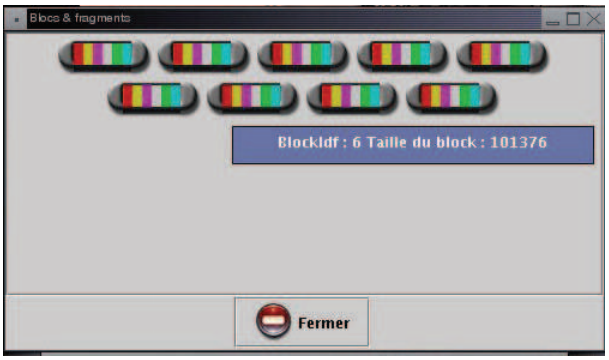
Cette interface A.1, codée en Java, est un moniteur de surveillance du dispatcher Us. Au niveau programmation, la flexibilité dont peuvent faire preuve les composants utilisés (API Swing) lui confèrent la possibilité de créer aisement de nouveaux modules au fur et à mesure de l'evolution du projet.

Cette interface a pour simple but d'afficher les informations sur la répartition des fragments sur les pairs connus du dispatcher. Le moniteur

envoie des requêtes de demande d'informations au dispatcher, car le dispatcher est considéré comme un démon du système. Une fois la communication établie, le moniteur établit la liste des pairs connus par le dispatcher et affecte une image et une couleur unique pour chaque pair. Lorsque la souris se déplace sur un pair, une fenêtre offrant les données personnelles de ce dernier est affichée. La forme de la fenêtre rappelle celle d'une bulle d'aide.

Ces fenêtres (bulles d'aide) ont été développées de façon à être le plus souple possible, elles sont totalement personnalisables.

FIG. A.2 GUI fenêtre affichage des blocs

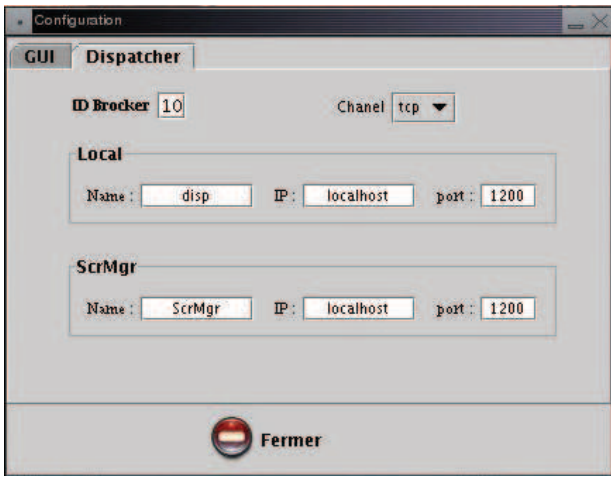


Une seconde fenêtre A.2 apparaît lorsque l'utilisateur clique sur le bouton «Blocs». Cette fenêtre affiche la liste des blocs qui ont été traités par le dispatcher. Chaque fragment (rectangle vertical de couleur) est de la même couleur que l'écran du pair sur lequel il est stocké. Une bulle d'aide affiche, tout comme les pairs, les informations liées au bloc pointé, telle que sa taille en Ko.

FIG. A.3 GUI fenêtre configuration de la transparence

Enfin, une fenêtre de configuration a été créée. Elle contient deux onglets permettant respectivement de configurer la transparence des bulles d'aide A.3 et les informations concernant le dispatcher A.4.

FIG. A.4 GUI fenêtre configuration du moniteur



A.2 Interface graphique du client Us

Cette interface A.5, codée en Java, concerne la partie client de Us. Son but est de rendre convivial l'envoi et la réception de fichiers. C'est une surcouche du client Us existant en ligne de commande.

FIG. A.5 GUI client fenêtre principale

Le bouton «envoyer» permet d'accéder à une boite de dialogue standard système pour sélectionner un fichier qui sera ensuite transféré vers le réseau Us.

Le bouton «recevoir» affiche une liste des fichiers qui ont été stockés par l'utilisateur. On sélectionne le fichier, puis on lance sa récupération à travers le réseau Us.

Le bouton «quota» indique à l'utilisateur l'espace libre restant à sa disposition.

Le bouton «config» permet d'accéder à la configuration du client Us. Comme dans l'interface graphique du moniteur Us, on accède à une fenêtre de configuration de communication du dispatcher.

Annexe **B**

Distribution CG

B.1 Construction des matrices

Dans cette section, nous donnons quelques exemples d'exécution de l'algorithme de distribution CG.

B.1.1 Cas idéal : $f^2 = N$

Nous prenons $N = 49$ et $f = 7$, ainsi nous obtenons les matrices suivantes :

$$M_1 = \begin{bmatrix} 1 & 8 & 15 & 22 & 29 & 36 & 43 \\ 1 & 9 & 16 & 23 & 30 & 37 & 44 \\ 1 & 10 & 17 & 24 & 31 & 38 & 45 \\ 1 & 11 & 18 & 25 & 32 & 39 & 46 \\ 1 & 12 & 19 & 26 & 33 & 40 & 47 \\ 1 & 13 & 20 & 27 & 34 & 41 & 48 \\ 1 & 14 & 21 & 28 & 35 & 42 & 49 \end{bmatrix}$$

$$M_2 = \begin{bmatrix} 2 & 8 & 16 & 24 & 32 & 40 & 48 \\ 2 & 9 & 17 & 25 & 33 & 41 & 49 \\ 2 & 10 & 18 & 26 & 34 & 42 & 43 \\ 2 & 11 & 19 & 27 & 35 & 36 & 44 \\ 2 & 12 & 20 & 28 & 29 & 37 & 45 \\ 2 & 13 & 21 & 22 & 30 & 38 & 46 \\ 2 & 14 & 15 & 23 & 31 & 39 & 47 \end{bmatrix}$$

$$M_3 = \begin{bmatrix} 3 & 8 & 17 & 26 & 35 & 37 & 46 \\ 3 & 9 & 18 & 27 & 29 & 38 & 47 \\ 3 & 10 & 19 & 28 & 30 & 39 & 48 \\ 3 & 11 & 20 & 22 & 31 & 40 & 49 \\ 3 & 12 & 21 & 23 & 32 & 41 & 43 \\ 3 & 13 & 15 & 24 & 33 & 42 & 44 \\ 3 & 14 & 16 & 25 & 34 & 36 & 45 \end{bmatrix}$$

$$M_4 = \begin{bmatrix} 4 & 8 & 18 & 28 & 31 & 41 & 44 \\ 4 & 9 & 19 & 22 & 32 & 42 & 45 \\ 4 & 10 & 20 & 23 & 33 & 36 & 46 \\ 4 & 11 & 21 & 24 & 34 & 37 & 47 \\ 4 & 12 & 15 & 25 & 35 & 38 & 48 \\ 4 & 13 & 16 & 26 & 29 & 39 & 49 \\ 4 & 14 & 17 & 27 & 30 & 40 & 43 \end{bmatrix}$$

$$M_5 = \begin{bmatrix} 5 & 8 & 19 & 23 & 34 & 38 & 49 \\ 5 & 9 & 20 & 24 & 35 & 39 & 43 \\ 5 & 10 & 21 & 25 & 29 & 40 & 44 \\ 5 & 11 & 15 & 26 & 30 & 41 & 45 \\ 5 & 12 & 16 & 27 & 31 & 42 & 46 \\ 5 & 13 & 17 & 28 & 32 & 36 & 47 \\ 5 & 14 & 18 & 22 & 33 & 37 & 48 \end{bmatrix}$$

$$M_6 = \begin{bmatrix} 6 & 8 & 20 & 25 & 30 & 42 & 47 \\ 6 & 9 & 21 & 26 & 31 & 36 & 48 \\ 6 & 10 & 15 & 27 & 32 & 37 & 49 \\ 6 & 11 & 16 & 28 & 33 & 38 & 43 \\ 6 & 12 & 17 & 22 & 34 & 39 & 44 \\ 6 & 13 & 18 & 23 & 35 & 40 & 45 \\ 6 & 14 & 19 & 24 & 29 & 41 & 46 \end{bmatrix}$$

$$M_7 = \begin{bmatrix} 7 & 8 & 21 & 27 & 33 & 39 & 45 \\ 7 & 9 & 15 & 28 & 34 & 40 & 46 \\ 7 & 10 & 16 & 22 & 35 & 41 & 47 \\ 7 & 11 & 17 & 23 & 29 & 42 & 48 \\ 7 & 12 & 18 & 24 & 30 & 36 & 49 \\ 7 & 13 & 19 & 25 & 31 & 37 & 43 \\ 7 & 14 & 20 & 26 & 32 & 38 & 44 \end{bmatrix}$$

$$M_8 = \begin{bmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 \\ 8 & 9 & 10 & 11 & 12 & 13 & 14 \\ 15 & 16 & 17 & 18 & 19 & 20 & 21 \\ 22 & 23 & 24 & 25 & 26 & 27 & 28 \\ 29 & 30 & 31 & 32 & 33 & 34 & 35 \\ 36 & 37 & 38 & 39 & 40 & 41 & 42 \\ 43 & 44 & 45 & 46 & 47 & 48 & 49 \end{bmatrix}$$

B.1.2 Cas quelconque

Nous prenons $N = 49$ et $f = 5$, nous obtenons les matrices suivantes :

$$M_1 = \begin{bmatrix} 1 & 8 & 15 & 22 & 29 \\ 1 & 9 & 16 & 23 & 30 \\ 1 & 10 & 17 & 24 & 31 \\ 1 & 11 & 18 & 25 & 32 \\ 1 & 12 & 19 & 26 & 33 \\ 1 & 13 & 20 & 27 & 34 \\ 1 & 14 & 21 & 28 & 35 \end{bmatrix}$$

$$M_2 = \begin{bmatrix} 2 & 8 & 16 & 24 & 32 \\ 2 & 9 & 17 & 25 & 33 \\ 2 & 10 & 18 & 26 & 34 \\ 2 & 11 & 19 & 27 & 35 \\ 2 & 12 & 20 & 28 & 29 \\ 2 & 13 & 21 & 22 & 30 \\ 2 & 14 & 15 & 23 & 31 \end{bmatrix}$$

$$M_3 = \begin{bmatrix} 3 & 8 & 17 & 26 & 35 \\ 3 & 9 & 18 & 27 & 29 \\ 3 & 10 & 19 & 28 & 30 \\ 3 & 11 & 20 & 22 & 31 \\ 3 & 12 & 21 & 23 & 32 \\ 3 & 13 & 15 & 24 & 33 \\ 3 & 14 & 16 & 25 & 34 \end{bmatrix}$$

$$M_4 = \begin{bmatrix} 4 & 8 & 18 & 28 & 31 \\ 4 & 9 & 19 & 22 & 32 \\ 4 & 10 & 20 & 23 & 33 \\ 4 & 11 & 21 & 24 & 34 \\ 4 & 12 & 15 & 25 & 35 \\ 4 & 13 & 16 & 26 & 29 \\ 4 & 14 & 17 & 27 & 30 \end{bmatrix}$$

$$M_5 = \begin{bmatrix} 5 & 8 & 19 & 23 & 34 \\ 5 & 9 & 20 & 24 & 35 \\ 5 & 10 & 21 & 25 & 29 \\ 5 & 11 & 15 & 26 & 30 \\ 5 & 12 & 16 & 27 & 31 \\ 5 & 13 & 17 & 28 & 32 \\ 5 & 14 & 18 & 22 & 33 \end{bmatrix}$$

$$M_6 = \begin{bmatrix} 6 & 8 & 20 & 25 & 30 \\ 6 & 9 & 21 & 26 & 31 \\ 6 & 10 & 15 & 27 & 32 \\ 6 & 11 & 16 & 28 & 33 \\ 6 & 12 & 17 & 22 & 34 \\ 6 & 13 & 18 & 23 & 35 \\ 6 & 14 & 19 & 24 & 29 \end{bmatrix}$$

$$M_7 = \begin{bmatrix} 7 & 8 & 21 & 27 & 33 \\ 7 & 9 & 15 & 28 & 34 \\ 7 & 10 & 16 & 22 & 35 \\ 7 & 11 & 17 & 23 & 29 \\ 7 & 12 & 18 & 24 & 30 \\ 7 & 13 & 19 & 25 & 31 \\ 7 & 14 & 20 & 26 & 32 \end{bmatrix}$$

$$M_8 = \begin{bmatrix} 1 & 2 & 3 & 4 & 5 \\ 8 & 9 & 10 & 11 & 12 \\ 15 & 16 & 17 & 18 & 19 \\ 22 & 23 & 24 & 25 & 26 \\ 29 & 30 & 31 & 32 & 33 \\ 34 & 35 & 36 & 37 & 38 \\ 39 & 40 & 41 & 42 & 43 \\ 44 & 45 & 46 & 47 & 48 \end{bmatrix}$$

On remarque que dans le cas non idéal, certains pairs du réseau ne sont pas utilisés pour le stockage de fragments.

Publications personnelles

Conférences

- Cyril Randriamaro and Olivier Soyez and Gil Utard and Francis Wlazinski. Data distribution for failure correlation management in a Peer to Peer Storage. ISPDC 2005, France, Lille. July 2005.
- Cyril Randriamaro and Olivier Soyez and Gil Utard and Fancis Wlazinski. Data distribution in a peer to peer storage system. GP2PC05 2005, UK, Cardiff. May 2005.
- Olivier Soyez. Us : Prototype de stockage pair à pair. RENPAR 2003, la Colle sur Loup, France. pages 214–218. October 2003.

Rapports de recherche

- Cyril Randriamaro and Olivier Soyez and Gil Utard and Francis Wlazinski. Data Distribution in a Peer to Peer Storage. Technical Report LaRIA 2006-03, Laboratoire de Recherche en Informatique d'Amiens, France. Jan 2006.
- Cyril Randriamaro and Olivier Soyez and Gil Utard and Francis Wlazinski. Structured data mapping in a Peer to Peer storage system. Technical Report LaRIA 2006-02, Laboratoire de Recherche en Informatique d'Amiens, France. Jan 2006.
- Cyril Randriamaro and Olivier Soyez and Gil Utard and Francis Wlazinski. On the load sharing in P2P data reconstruction process. Technical Report LaRIA 2005-01, Laboratoire de Recherche en Informatique d'Amiens, France. Jan 2005.
- Cyril Randriamaro and Olivier Soyez and Gil Utard and Francis Wlazinski. Data distribution in a peer to peer storage system. Technical Report LaRIA 2004-08, Laboratoire de Recherche en Informatique d'Amiens, France. Aug 2004.
- Cyril Randriamaro and Olivier Soyez and Gil Utard. Us : Prototype de stockage pair à pair. Technical Report LaRIA 2003-09, Laboratoire de Recherche en Informatique d'Amiens, France. Sep 2003.

Bibliographie

[1] Bernard Traversat Ahkil. Project jxta 2.0 super-peer virtual network.

[2] Albert and Sandler. An introduction to finite projective planes. In *An Introduction to Finite Projective Planes*, New York, 1968. Holt , Rinehart and Winston.

[3] David Bindel, Yan Chen, Patrick Eaton, Dennis Geels, Ramakrishna Gummadi, Sean Rhea, Hakim Weatherspoon, Westley Weimer, Christopher Wells, Ben Zhao, and John Kubiatowicz. Oceanstore : An extremely wide-area storage system. Technical Report Technical Report UCB CSD-00-1102, U.C. Berkeley, May 1999.

[4] Bruck, Ryser, and Chowla. The nonexistence of certain finite projective planes. In *Canadian Journal of Mathematics*, pages 88–93, 1949.

[5] John W. Byers, Michael Luby, and Michael Mitzenmacher. Accessing multiple mirror sites in parallel : Using tornado codes to speed up downloads. In *INFOCOM (1)*, pages 275–283, 1999.

[6] John W. Byers, Michael Luby, Michael Mitzenmacher, and Ashutosh Rege. A digital fountain approach to reliable distribution of bulk data. In *SIGCOMM*, pages 56–67, 1998.

[7] Y. Calas. Performance des codes correcteurs d'erreur au niveau applicatif dans les réseaux. Master's thesis, Université de Montpellier II, Montpellier, December 2003.

[8] Castro and Liskov. Practical byzantine fault tolerance. In *OSDI : Symposium on Operating Systems Design and Implementation*. USENIX Association, Co-sponsored by IEEE TCOS and ACM SIGOPS, 1999.

[9] Miguel Castro, Peter Druschel, Ayalvadi Ganesh, and Antony Rowstronand Dan S. Wallach. Security for structured peer-to-peer overlay networks. In *Proceedings of the Fifth Symposium on Operating Systems Design and Implementation (OSDI'02)*, Boston, December 2002.

[10] Yuan Chen, Jan Edler, Andrew Goldberg, Allan Gottlieb, Sumeet Sobti, and Peter Yianilos. A prototype implementation of archival intermemory. In *Proceedings of the Fourth ACM International Conference on Digital Libraries*, 1999.

[11] Ian Clarke, Oskar Sandberg, Brandon Wiley, and Theodore W. Hong. Freenet : A distributed anonymous information storage and retrieval system. In *Workshop on Design Issues in Anonymity and Unobservability*, pages 46–66, 2000.

[12] Brian F. Cooper and Hector Garcia-Molina. Bidding for storage space in a peer-to-peer data preservation system. In *International Conference on Distributed Computing Systems*, 2002.

[13] F. Dabek, J. Li, E. Sit, J. Robertson, M. Kaashoek, and R. Morris. Designing a dht for low latency and high throughput, 2004.

[14] Frank Dabek, M. Frans Kaashoek, David Karger, Robert Morris, and Ion Stoica. Wide-area cooperative storage with CFS. In *Symposium on Operating Systems Principles*, pages 202–215, 2001.

[15] 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.

[16] P. Druschel and A. Rowstron. PAST : A large-scale, persistent peer-to-peer storage utility. In *Procedings of HOTOS*, pages 75–80, 2001.

[17] eMule. Site internet emule : http ://www.emule.com.

[18] Entropia. Site internet entropia : http ://www.entropia.com.

[19] P. Erdős. On the difference of consecutive primes. *QJMOX*, 6 :124–128, 1935.

[20] G. Fedak, C. Germain, V. N'eri, and F. Cappello. Xtremweb : A generic global computing system, 2001.

[21] Gilles Fedak, Cécile Germain, Vincent Néri, and Franck Cappello. Xtremweb : A generic global computing system. In *CCGRID2001, workshop on Global Computing on Personal Devices*, May 2001.

[22] Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. Impossibility of distributed consensus with one faulty process. *j-ACM*, 32(2) :374–382, April 1985.

[23] Freenet. Site internet freenet : http ://www.freenet.sourceforge.net.

[24] Andrew C. Fuqua, Tsuen-Wan Ngan, and Dan S. Wallach. Economic behavior of peer-to-peer storage networks. In *Workshop on Economics of Peer-to-Peer Systems*, Berkeley, June 2003.

[25] FUSE. Site internet de fuse (filesystem in userspace) : http ://fuse.sourceforge.net.

[26] Cecile Germain, Vincent Neri, Gille Fedak, and Franck Cappello. Xtremweb : Building an experimental platform for global computing. In *GRID*, pages 91–101, 2000.

[27] Gnutella. Site internet gnutella : http ://www.gnutella.com.

[28] A. Goldberg and Peter N. Yianilos. Towards an archival intermemory. In *Proceedings of IEEE Advances in Digital Libraries, ADL 98*, pages 147–156, Santa Barbara, CA, 1998. IEEE Computer Society.

[29] Madhusudhan Govindaraju, Aleksander Slominski, Venkatesh Choppella, Randall Bramley, and Dennis Gannon. Requirements for and evaluation of RMI protocols for scientific computing. In *SC2000 : High Performance Networking and Computing. Dallas Convention Center, Dallas, TX, USA, November 4–10, 2000*, page 76, 2000.

[30] G.Tarry. Le problème des 36 officiers. *Compte Rendu de l'Assoc. Français Avanc. Sci. Naturel*, pages 122–123, 1900.

[31] John H. Hartman and John K. Ousterhout. The Zebra striped network file system. In Hai Jin, Toni Cortes, and Rajkumar Buyya, editors, *High Performance Mass Storage and Parallel I/O : Technologies and Applications*, pages 309–329. IEEE Computer Society Press and Wiley, New York, NY, 2001.

[32] M. Izal, G. Urvoy-Keller, E. Biersack, P. Felber, A. Hamra, and L. Garces-Erice. Dissecting bittorrent : Five months in a torrent's lifetime, 2004.

[33] Kazaa. Site internet kazaa : http ://www.kazaa.com.

[34] John Kubiatowicz, Divid 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.

[35] Lam. The search for a finite projective plane of order 10. In *American Mathematical Monthly*, pages 305–318, 1991.

[36] Viet-Dung Le, Gilbert Babin, and Peter Kropf. A structured peer-to-peer system with integrated index and storage load balancing. In *I2CS 2005, Innovative Internet Community Systems 2005, Paris, France*, June 2005.

[37] J. Li, J. Stribling, T. Gil, R. Morris, and F. Kaashoek. Comparing the performance of distributed hash tables under churn, 2004.

[38] M. Luby, M. Mitzenmacher, A. Shokrollahi, and D. Spielman. Efficient erasure correcting codes. In *IEEE Transactions on Information Theory, Special Issue : Codes on Graphs and Iterative Algorithms*, February 2001.

[39] M. Luby, M. Mitzenmacher, A. Shokrollahi, and V. Spielman. Practical loss-resilient codes. In *Proc. of the 29th ACM symposium on Theory of Computing*, 1997.

[40] M. Luby, L. Vicisano, J.Gemmel, L. Rizzo, M. Handley, and J. Crowcroft. The use of forward error correction (fec) in reliable multicast. In *The Internet Society*, dec 2002.

[41] Ricardo Marcelin-Jimenez. Improving reliability of distributed storage. In *I2CS 2005, Innovative Internet Community Systems 2005, Paris, France*, June 2005.

[42] Bernard Traversat Mohamed. Project jxta : A loosely-consistent dht rendezvous walker.

[43] MojoNation. Technology overview : http ://www.mojonation.net/docs/technical_overview.shtml.

[44] Athicha Muthitacharoen, Robert Morris, Thomer Gil, and Benjie Chen. Ivy : A read/write peer-to-peer file system. In *Proceedings of the 5th USENIX Symposium on Operating Systems Design and Implementation (OSDI '02)*, Boston, Massachusetts, December 2002.

[45] Napster. Site internet napster : http ://www.napster.com.

[46] F. Picconi, J-M. Busca, and P. Sens. Exploiting network locality in a decentralized read-write peer-to-peer file system. In *In International Conference on Parallel and Distributed Systems 2004 (ICPADS 04)*, New Port Beach, California, USA, July 2004.

[47] F. Picconi, J-M. Busca, and P. Sens. Pastis : un système de fichiers pair à pair multi-écrivain passant l'échelle. In *In DistRibUtIon de Données à grande Echelle 2004 (DRUIDE 04)*, Domaine du Port-aux-Rocs, Le Croisic, France, Mai 2004.

[48] James S. Plank. A tutorial on Reed-Solomon coding for fault-tolerance in RAID-like systems. *Software Practice and Experience*, 27(9) :995–1012, 1997.

[49] James S. Plank, Micah Beck, Wael R. Elwasif, Terence Moore, Martin Swany, and Rich Wolski. The internet backplane protocol : Storage in the network. In Micah Beck and Terry Moore, editors, *NetStore '99 : Network Storage Symposium*, October 1999.

[50] C. Greg Plaxton, Rajmohan Rajaraman, and Andrea W. Richa. Accessing nearby copies of replicated objects in a distributed environment. In *ACM Symposium on Parallel Algorithms and Architectures*, pages 311–320, 1997.

[51] D. Qiu and R. Srikant. Modeling and performance analysis of bittorrent-like peer-to-peer networks, 2004.

[52] M. O. Rabin. Efficient dispersal of information for security, load balancing, and fault tolerance. *Journal of ACM*, 38 :335–348, 1989.

[53] Cyril Randriamaro, Olivier Soyez, and Gil Utard. Us : Prototype de stockage pair à pair. Technical report LaRIA 2003-09, Laboratoire de Recherche en Informatique d'Amiens, September 2003.

[54] Cyril Randriamaro, Olivier Soyez, Gil Utard, and Francis Wlazinski. Data distribution in a peer to peer storage system. Technical report LaRIA 2004-08, Laboratoire de Recherche en Informatique d'Amiens, August 2004.

[55] Cyril Randriamaro, Olivier Soyez, Gil Utard, and Francis Wlazinski. Data distribution for failure correlation management in a peer to peer storage. In *ISPDC 2005, France, Lille*, July 2005.

[56] Cyril Randriamaro, Olivier Soyez, Gil Utard, and Francis Wlazinski. Data distribution in a peer to peer storage system. In *GP2PC05 2005, UK, Cardiff*, May 2005.

[57] Cyril Randriamaro, Olivier Soyez, Gil Utard, and Francis Wlazinski. On the load sharing in p2p data reconstruction process. Technical report LaRIA 2005-01, Laboratoire de Recherche en Informatique d'Amiens, January 2005.

[58] Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard Karp, and Scott Shenker. A scalable content addressable network. In *Proceedings of ACM SIGCOMM 2001*, 2001.

[59] Sean Rhea, Patrick Eaton, Dennis Geels, Hakim Weatherspoon, Ben Zhao, and John Kubiatowicz. Pond : the oceanstore prototype. In *Proceedings of the 2nd USENIX Conference on File and Storage Technologies (FAST '03)*, 2003.

[60] 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.

[61] Sean C. Rhea and John Kubiatowicz. Probabilistic location and routing. In *Proceedings of the 21st Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM 2002)*, June 2002.

[62] Matei Ripeanu. Peer-to-peer architecture case study : Gnutella network. In *Proceedings of International Conference on Peer-to-peer Computing*, August 2001.

[63] L. Rizzo. On the feasibility of software fec, deit technical report lr-970131. available as http ://www.iet.unipi.it/ luigi/softfec.ps.

[64] Antony Rowstron and Peter Druschel. Pastry : Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *IFIP/ACM International Conference on Distributed Systems Platforms (Middleware)*, pages 329–350, November 2001.

[65] 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.

[66] Russel Sandberg, David Goldberg, Steve Kleiman, Dan Walsh, and Bob Lyon. Design and implementation of the Sun Network Filesystem. In *Proc. Summer 1985 USENIX Conf.*, pages 119–130, Portland OR (USA), 1985.

[67] J. Schonheim. On maximal systems of k-tuples. *Studia Sci. Math. Hungar.*, pages 363–368, 1966.

[68] Seti@home. Site internet seti@home :
http ://setiathome.ssl.berkeley.edu.

[69] A. Shamir. How to share a secret. *Communications of the ACM*, 22(11) :612–614, November 1979.

[70] A. Shamir and E. Biham. Differential cryptanalysis snefru, kharfe, redoc-ii, loki and lucifer. In *Advances in Cryptology - Crypto'91*, 1991.

[71] E. Sit and R. Morris. Security considerations for peer-to-peer distributed hash tables, 2002.

[72] Olivier Soyez. Us : Prototype de stockage pair à pair. In *RENPAR 2003, la Colle sur Loup, France*, pages 214–218, October 2003.

[73] I. Stoica, R. Morris, D. Karger, M. Kaashock, and H. Balakrishman. Chord : A scalable peer-to-peer lookup protocol for internet applications, 2001.

[74] Sun. Project jxta. http ://www.jxta.org.

[75] Gil Utard and Antoine Vernois. Data durability in peer to peer storage systems. In *4th IEEE Workshop on Global and Peer to Peer Computing*, Chicago, April 2004.

[76] Hakim Weatherspoon and John Kubiatowicz. Erasure coding vs. replication : A quantitative comparison. In *Proceedings of the First International Workshop on Peer-to-Peer Systems*, March 2002.

[77] Hakim Weatherspoon and John Kubiatowicz. Erasure coding vs. replication : A quantitative comparison. In *Proceedings of the First International Workshop on Peer-to-Peer Systems*, March 2002.

[78] Hakim Weatherspoon, Tal Moscovitz, and John Kubiatowicz. Introspective failure analysis : Avoiding correlated failures in peer-to-peer systems. In *Proceedings of International Workshop on Reliable Peer-to-Peer Distributed Systems*, 2002.

[79] 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.

[80] Shelley Q. Zhuang, Ben Y. Zhao, Anthony D. Joseph, Randy H. Katz, and John Kubiatowicz. Bayeux : An architecture for scalable and fault-tolerant wide-area data dissemination. In *Proceedings of the Eleventh International Workshop on Network and Operating System Support for Digital Audio and Video*, June 2001.

Stockage dans les systèmes pair à pair

Olivier Soyez

Cette thèse a pour objectif de définir un système de stockage pair à pair, nommé Us. Le but principal de Us est de garantir la pérennité des données. Pour cela, Us associe un mécanisme de redondance des données à un processus dynamique de reconstruction.

Dans un premier temps, nous avons créé un prototype Us et conçu une interface utilisateur, nommée UsFS, de type système de fichiers. Un procédé de journalisation des données est inclus dans UsFS.

Ensuite, nous nous sommes intéressés aux distributions de données au sein du réseau Us. Le but de ces distributions est de minimiser le dérangement occasionné par le processus de reconstruction pour chaque pair. Enfin, nous avons étendu notre schéma de distribution pour gérer le comportement dynamique des pairs et prendre en compte les corrélations de panne.

Mots clefs : Stockage ; Pair à Pair ; Redondance ; Fiabilité ; Pérennité ; Distribution de données ; Plan projectif fini ; Corrélation de panne

---

Storage in peer to peer systems

Olivier Soyez

The objective of this thesis is to define a peer to peer storage system, named Us. The main aim of Us is to guarantee the data durability. In this way, Us associates a data redundancy mechanism with a dynamic process of reconstruction.

In a first time, we have created a prototype Us and made a user interface, named UsFS, like a files system. A functionality of data versioning is included in UsFS.

Next, we interested in data distributions inside the Us network. The aim of these distributions is to minimise the disturbance because of the reconstruction process for each peer. Finally, we extended our distribution scheme to manage the dynamic behaviour of peers and to take into account the failure correlations.

Keywords : Storage ; Peer to Peer ; Redondancy ; Fiability ; Durability ; Data distribution ; Finite projective plane ; Failure correlation