Adapnet : Stratégies adaptatives pour la gestion de données distribuées sur un réseau P2P

Nicolas Bonnel

THESE / UNIVERSITE DE BRETAGNE SUD
sous le sceau de l'Université Européenne de Bretagne
pour obtenir le grade de :
DOCTEUR DE L'UNIVERSITE DE BRETAGNE SUD
Mention : Sciences et Technologies de l'Information et de la Communication
Ecole Doctorale SICMA

présentée par
Nicolas Bonnel
Préparée au Laboratoire de Recherche en Informatique et ses Applications de Vannes et Lorient (VALORIA)
N^o d'ordre : 134

Adapnet : Stratégies auto-adaptatives pour la gestion de données distribuées sur un réseau P2P

Thèse soutenue le 5 décembre 2008
devant le jury composé de :
Abdelkader Hameurlain
Pr Université de Toulouse 2 / *Rapporteur*
Lionel Brunie
Pr. INSA Lyon / *Rapporteur*
Achour Mostefaoui
HDR Université de Rennes 1 / *Examinateur*
Frédéric Guinand
Pr. Université du Havre / *Examinateur*
Gildas Ménier
Mcf. Université de Bretagne Sud / *Encadrant*
Pierre-François Marteau
Pr. Université de Bretagne Sud / *Directeur de thèse*

Résumé

Depuis quelques décennies, la quantité d'information numérique produite ne cesse de croître exponentiellement, ce qui soulève des difficultés de plus en plus critiques en terme de stockage, d'accessibilité et de disponibilité de cette information. Les architectures logicielles et matérielles construites autour du modèle pair-à-pair (P2P) semblent répondre globalement aux exigences liées au stockage de l'information mais montrent leurs limites en ce qui concerne les exigences d'accessibilité et de disponibilité de l'information.

Nous présentons dans cette thèse différents apports concernant les architectures P2P pour la gestion de grands volumes d'information. Les stratégies algorithmiques que nous proposons exploitent des topologies virtuelles dédiées sur lesquelles nous développons des protocoles de maintenance et de gestion du réseau efficaces. En particulier, pour assurer le passage à grande échelle, nous proposons des solutions pour lesquelles les coûts des opérations de maintenance et de gestion des topologies virtuelles sont constants en moyenne pour chaque noeud du réseau, et ceci, quelle que soit la taille du réseau.

Nous analysons les principaux paradigmes de la répartition d'information sur un réseau P2P, en considérant successivement, le problème de l'accès à de l'information typée (semi-structurée) et le cas général qui dissocie entièrement la nature des requêtes du placement de l'information. Nous proposons une méthode d'aiguillage de requêtes portant sur la structure et le contenu de documents semi-structurés ainsi qu'une technique plus générale dans le cas le plus défavorable où aucune connaissance n'est disponible *a priori* sur la nature des informations stockées ou sur la nature des requêtes.

Dans l'optique de la gestion d'une qualité de service (qui s'exprime en terme de rapidité et de fiabilité), nous nous intéressons également au problème de la disponibilité pérenne de l'information sous l'angle de la réplication des données stockées dans le réseau. Nous proposons une approche originale exploitant une mesure locale de densité de réplicas estimée sur une topologie virtuelle dédiée.

Abstract

In the last few years, the amount of digital information produced has exponentially increased. This raises problems regarding the storage, the access and the availability of this data. Software and hardware architectures based on the peer-to-peer (P2P) paradigm seem to satisfy the needs of data storage but cannot handle efficiently both data accessibility and availability.

We present in this thesis various contributions on P2P architectures for managing large volumes of information. We propose various strategies that operate on dedicated virtual topologies that can be maintained at low cost. More precisely, these topologies scale well because the cost for node arrival and node departure is on average constant, whatever the size of the network.

We analyze the main paradigms of information sharing on a P2P network, considering successively the problem of access to typed information (semi-structured) and the general case that completely separates the nature of the queries and data location. We propose a routing strategy using structure and content of semi-structured information. We also propose startegies that efficiently explore the network when there is no assumption on the nature of data or queries.

In order to manage a quality of service (which is expressed in terms of speed and reliability), we also investigate the problem of information availability, more precisely we replicate data stored in the network. We propose a novel approach exploiting an estimation of local density of data replica.

Remerciements

Je tiens en premier lieu à remercier la région Bretagne, sans laquelle cette thèse n'aurait pas été possible, ainsi que les deux rapporteurs, Abdelkader Hameurlain et Lionel Brunie, qui ont accepté d'examiner mon travail, et qui malgré les imperfections qu'il présentait, ne m'ont fait que des remarques constructives. Je remercie également Frédéric Guinan et Achour Mostefaoui d'avoir accepté de participer au jury de cette thèse.

Je tiens enfin à remercier mes encadrants de m'avoir supporté et fait confiance pendant ces trois années. Je remercie mon directeur de thèse, Pierre-François Marteau, qui malgré les lourdes charges administratives qu'il avait, ne m'a jamais fermé la porte de son bureau quand j'avais une question à lui poser. Je remercie également mon autre encadrant, Gildas Ménier, avec qui en plus de pouvoir travailler efficacement, j'ai pu agréablement discuter d'autre sujets variés.

Je remercie tous mes autres collègues, Sebastien, Alban , ainsi que tous les autres doctorants, avec lesquels ces trois années ont été très agréables ; je signerais sans réfléchir pour faire une autre thèse dans les mêmes conditions.

Je tiens également à remercier tous ceux qui ont cru en moi. Que ce soit les amis ou la famille, plus particulièrement mes parents qui m'ont toujours soutenu dans mes études, mais aussi mon frère et ma sœur, ainsi que ma grand-mère Eliane. Je tiens enfin à remercier ma bien-aimée Lucie, qui a du me supporter dans la dernière ligne droite.

Table des matières

Table des figures

Liste des tableaux

Chapitre 1

Introduction

L'utilisation de l'informatique comme moyen de stockage et de traitement de l'information s'est démocratisée très rapidement, promouvant par la même occasion le phénomène du "tout numérique" : la majorité des médias de l'information est de plus en plus exploitée plus ou moins exclusivement sous une forme numérique qu'il s'agisse de musiques, films, livres, courriers, etc ...

La réduction des coûts des machines [33] a favorisé cette démocratisation qui se traduit par un accroissement quasi exponentiel des postes de travail (et de stockage). Les besoins liés à la communication et à l'accès à l'information, ainsi que le développement d'Internet accentuent encore les exigences des utilisateurs en terme de fiabilité, rapidité et volume de stockage.

La quantité globale de données numériques à été estimée à 161 exaoctets (10^{18} octets) au début de l'année 2007 [35]. Cette quantité croît de manière exponentielle : plus les avancées technologiques permettent d'espérer un traitement et un stockage massif d'information, plus de nouveaux projets devenus envisageables voient le jour, repoussant les limites déjà extrêmes. Les experts prévoient une augmentation de cette quantité de données à 988 exaoctets en 2010 [35]. En 1999, "seulement" deux exaoctets de données numériques ont été générés.

La construction du Grand Collisionneur Hadronique (LHC), le plus gros accélérateur de particules au monde, vient d'être achevée à la fin de cette année 2008 [46]. Le LHC produit en fonctionnement entre 500 Mo et $1,5$ Go de données par seconde, et à peu près 15 pétaoctets par an (ce qui correspond à une pile de CD de 21 kilomètres de haut). Pour pouvoir mener des expérimentations, les données collectées devront être accessibles pendant 15 ans. Ce genre d'application nécessite des capacités de stockage, de traitement et d'accessibilité de l'information qui donne un avant-goût des besoins de demain. Il est donc absolument crucial de proposer de nouvelles stratégies de gestion de grande quantité d'information.

Les disques durs actuels permettent de stocker au mieux quelques téraoctets de données. Étant donnés les chiffres avancés précédemment, il paraît donc difficile de centraliser le stockage de grosses quantités de données. La répartition des données entre différentes machines offre plusieurs avantages. Outre le fait que cette approche

permet de gérer de grandes quantités de données, elle est aussi plus résistante aux pannes, pour autant qu'une forme de redondance soit prise en compte. Si une machine tombe en panne, d'autres machines peuvent encore rester accessibles. Ce genre d'approche nécessite bien évidement un mécanisme de communication assurant la cohésion fonctionnelle de l'ensemble des machines utilisées.

Les grilles d'ordinateurs [8, 6] permettent le partage de la puissance de calcul et des capacités de stockage sur Internet. Elles permettent d'envisager une qualité de service optimale, au prix évidemment d'un coût proportionnel aux problèmes de maintenance et de service associés à la plate-forme utilisée.

Bigtable [14] est un système de stockage distribué qui a été conçu pour stocker de grosses quantités de données structurées (de l'ordre de plusieurs pétaoctets) sur un réseau de plusieurs milliers de serveurs dédiés. Ce système est donc prévu pour tourner sur des grilles d'ordinateurs. C'est le système de stockage utilisé par plus de 60 applications développées par la société Google.

Google possède une des plus grandes grilles d'ordinateurs au monde. La quantité de machines la composant a été estimée entre 150000 et 170000 en 2005 [3] et à 450000 en 2006, réparti dans 30 centres de données. En 2005, Google indexait 8 milliards de pages. Le nombre de clusters est estimé à 200, chacun possédant entre 1000 à 5000 machines pour un stockage total estimé à 5 Pétaoctets.

La particularité de la grille de Google est d'être composée d'ordinateurs à peu près identiques aux ordinateurs personnels. La solution retenue par cette société a donc été d'assembler énormément d'ordinateurs bon marché pour obtenir une énorme puissance de calcul et une grosse capacité de stockage à un prix raisonnable.

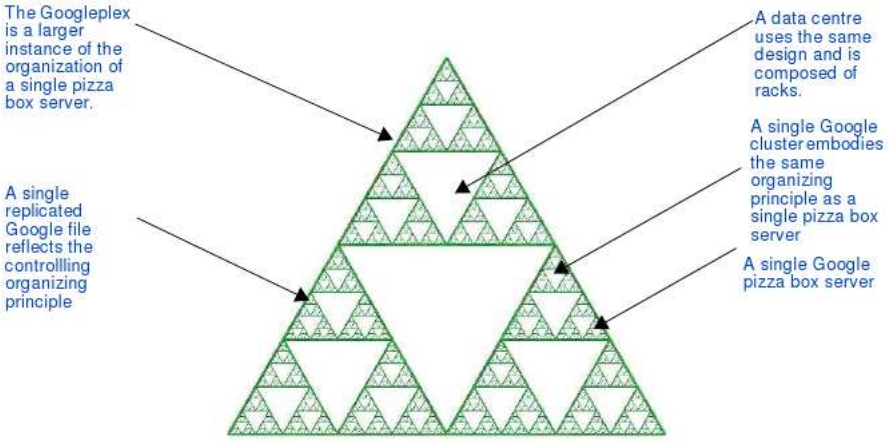


FIG. 1.1 – Organisation supposée des serveurs de Google [3].

La figure 1.1 illustre l'organisation supposée [3](la société ne communique pas sur cet aspect) des machines formant la grille de Google. La grille dans sa globalité peut être vue comme un unique super calculateur, qui est en fait composé de plusieurs super calculateurs. Les bases de ces super calculateurs sont des grappes d'ordinateurs. Cette

organisation “fractale” rend le système très résistant aux pannes : plusieurs ordinateurs voir plusieurs grappes d'ordinateurs peuvent tomber en panne sans que le système en soit affecté du point de vue fonctionnel.

Cette approche permet aussi de rajouter aisément des ordinateurs au réseau. Les ressources d'une machine sont disponibles pour le fonctionnement global de l'application moins de 72H après son installation physique. Ceci explique comment Google a pu presque tripler la taille de son parc de machines en un an.

Il est établi que la plupart des ordinateurs domestiques ne seraient jamais utilisés au maximum de leur potentiel. L'utilisation de ces ressources laissées vacantes pour construire un système de gestion de masse d'information distribué permet, en sacrifiant une partie des performances, de réduire considérablement les coûts d'acquisition et d'entretien de matériel : ce sont les utilisateurs qui contribuent au bon fonctionnement du système.

Il faut cependant tenir compte des caractéristiques de ces systèmes pour pouvoir les utiliser : la nature du parc des unités de traitement et de stockage est en pratique éparse, les postes de travail sont connectés de manière intermittente, communiquent de manière plus ou moins efficace, dans une hétérogénéité totale à la fois de capacité de calcul et de volume de stockage.

La gestion (stockage et traitement) d'une grande quantité d'information semble dépendante de l'exploitation efficace de cette nébuleuse d'unités de traitement et des échanges. L'idée d'exploiter cette répartition pour repousser encore les limites fonctionnelles à partir de limitations matérielles est légitime et appelle à la réalisation de nouvelles stratégies de stockage et d'échange s'appuyant sur une architecture matérielle dont les fonctions se doivent d'être bien plus que la somme des fonctions de ses constituants.

Les exigences sont multiples mais hiérarchiquement équivalentes : assurer un stockage massif, sans perte d'information ainsi qu'un accès rapide et fiable à une information délocalisée par nécessité de stockage (et/ou de traitement) dans un cadre de ressources fluctuantes et difficiles à contrôler.

Ce travail de thèse s'intéresse tout particulièrement à ces aspects. Dans la mesure où la nature du support de traitement rend le contrôle heuristiquement supervisé délicat à grande échelle, nous abordons des approches favorisant une organisation non supervisée des informations et de leur accès. L'idée principale est de parier sur l'autonomie d'unités de déplacement d'information (et de réplica) pour permettre une adaptation automatique aux fluctuations liées au réseau support ou aux données elles-mêmes.

Dans le chapitre 3, nous montrons comment la résolution de ce problème ne dépend pas seulement d'une architecture de communication, mais aussi de la nature de l'information elle-même : nous proposons ainsi une approche pouvant exploiter la nature des documents semi-structurés pour favoriser le routage de requêtes dédiées. Nous explorons également en contre-partie la notion de localité d'information pour proposer une stratégie d'exploration compacte qui permet de s'affranchir de la nature de la requête.

Nous étendons ensuite ce modèle dans la partie 4 pour généraliser cette exploration locale en exploitant un nouveau mécanisme de contrôle de propagation basé sur une topologie dédiée. Nous présentons une évaluation des coûts de maintenance de cette

stratégie.

Le chapitre 5 aborde la question complémentaire que constitue la réplication d'information pour assurer l'efficacité d'une recherche sans heuristique (c'est-à-dire autorisant n'importe quel type de requête), ainsi que la disponibilité des données stockées (résistance aux pannes locales) : nous proposons une stratégie de réplication proactive pour contrôler de manière dynamique une densité assurée de réplica localement. Nous évaluons l'efficacité de cette approche.

Nous concluons par une discussion (partie 6) sur les choix stratégiques proposés et extensions envisageables des travaux réalisés.

Dans le chapitre suivant, nous présentons les principales stratégies pair-à-pair (P2P) existantes et discutons des options choisie par ces approches pour proposer une exploitation efficace d'un ensemble de machines connectées (efficace en terme de stockage, traitement et accès à l'information). Ces stratégies sont généralement indépendantes des données à traiter.

Chapitre 2

Principales architectures pair-à-pair (P2P)

Sommaire

Les architectures pair-à-pair (P2P, de l'anglais *peer-to-peer*) désignent un ensemble d'utilisateurs, appelés nœuds (ou pairs), ainsi que les protocoles utilisés par ces nœuds pour communiquer entre eux. Plus précisément, les nœuds communiquent d'égal à égal, au contraire d'autres protocoles hiérarchiques type client/serveur, comme par exemple FTP.

Définition : La *valence* d'un pair (ou d'un nœud) est le nombre de voisins que ce pair possède, c'est-à-dire le nombre de pairs avec lesquels il est directement connecté.

Les architectures P2P permettent aux utilisateurs de mettre en commun des ressources comme de la mémoire ou du CPU [4, 24]. Nous traiterons plus particulièrement dans ce chapitre des architectures P2P dédiées au partage de mémoire. La majorité des architectures P2P présentent les caractéristiques suivantes :

Passage à l'échelle : ces systèmes sont complètement distribués et peuvent atteindre des tailles très importantes, de l'ordre de plusieurs milliers voir plusieurs millions de nœuds. Même en atteignant des tailles très importantes, ces architectures conservent de bonnes performances en terme de temps d'accès à l'information.

Adaptabilité : ces architectures ont été conçues pour être déployées dans des environnements dynamiques (arrivées et départs des nœuds fréquents dans le réseau). Elles sont très résistantes aux pannes : même si une partie du réseau tombe en panne, ces architectures peuvent continuer de fonctionner (éventuellement en mode dégradé).

Ces architectures sont devenues très populaires depuis le début des années 2000. D'après une étude menée en 2007 [34], le partage de fichiers via des réseaux P2P est responsable de 48% à 80% du trafic internet suivant les régions. D'après la même étude, deux architectures (eDonkey [30] et BitTorrent [58]) sont responsables à elles seules de 70% à 97% de tout le trafic P2P mondial.

Il existe différents types d'architectures P2P : centralisées, structurées et non structurées [50, 28]. Historiquement, les architectures P2P centralisées ont été les premières proposées. Les systèmes pair-à-pair ont depuis évolué et se décomposent aujourd'hui principalement en deux catégories : les architectures structurées et celles non structurées. Les premières présupposent une localisation des données sur certains nœuds, ce qui permet de retrouver l'information plus vite. Les secondes n'imposent pas cette localisation ce qui généralement rend l'information plus lente à retrouver.

Enfin, certaines architectures tirent partie des bénéfices apportés par les deux familles : nous présentons à la fin de ce chapitre quelques architectures hybrides.

2.1 Architectures P2P centralisées

Dans les architectures P2P centralisées, les pairs qui se connectent au réseau envoient une copie de leurs indexes au serveur central. Quand une requête est émise depuis un pair, elle est acheminée jusqu'au serveur qui traite cette requête puis retourne au pair "émetteur" une liste des pairs qui contiennent l'information recherchée, comme illustré sur la figure 2.1. Le pair "émetteur" contacte ensuite directement les pairs qui possèdent les fichiers correspondant aux critères de recherche et les télécharge (ou télécharge des fragments de ceux-ci).

La copie des fichiers est donc décentralisée, alors que la gestion des requêtes reste centralisée, ce qui rend le système fragile vis-à-vis d'une panne du serveur, et pose des problèmes de passage à l'échelle (ce mécanisme limite la taille maximale du réseau). Nous présentons dans la suite de cette section quelques architectures P2P centralisées.

2.1.1 Napster

Napster [54] est apparu au début des années 2000. C'est la première architecture P2P à avoir été très populaire. Elle permet aux utilisateurs d'échanger de la musique, un morceau se téléchargeant auprès d'un seul autre pair. Suite à des problèmes juridiques, Napster est aujourd'hui un réseau privé et payant.

L'utilisation d'une base de données centralisée sur un serveur pour répondre aux requêtes des utilisateurs rend le passage à l'échelle difficile et le système très vulnérable aux pannes du serveur. En plus de cet inconvénient, le fait que le transfert de fichiers ne se fasse qu'auprès d'un seul autre pair limite la vitesse de téléchargement et en cas

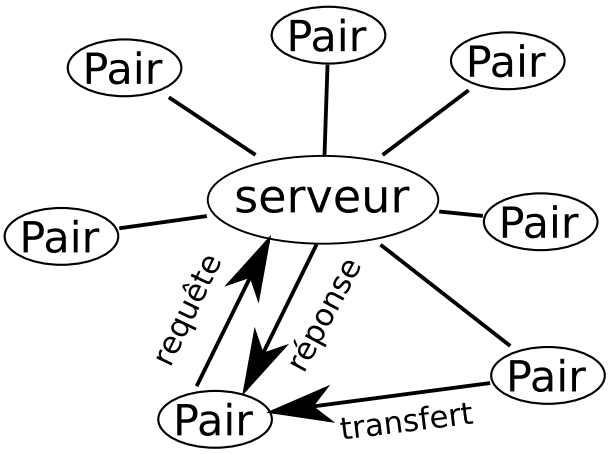


FIG. 2.1 – Architecture P2P centralisée.

de déconnexion durant le transfert, il faut recommencer le téléchargement depuis le début.

2.1.2 eDonkey (version initiale)

eDonkey [30] repose aussi sur des serveurs qui étaient initialement isolés les uns des autres. Le principal avantage qu'il présente par rapport à Napster est le protocole de transfert de fichier multi-source. Ce protocole décompose les fichiers en petits blocs, ce qui permet le téléchargement de différents blocs auprès de différents pairs simultanément, augmentant ainsi la vitesse de transfert. De plus les téléchargements peuvent être interrompus puis repris à tout moment auprès d'autres pairs disposant du fichier. Les serveurs étant aujourd'hui connectés entre eux, nous reviendrons sur eDonkey dans la suite de ce chapitre.

2.1.3 BitTorrent

BitTorrent [58] a été créé dans le but de diffuser rapidement de gros fichiers. L'idée générale est d'accélérer la diffusion des fichier en les fragmentant. Dès qu'un utilisateur a téléchargé un fragment de fichier, les autres utilisateurs peuvent télécharger ce fragment. Il a été estimé que BitTorrent a été responsable de 53% du trafic P2P mondial en 2004 [58], et il existe à ce jour plus d'une vingtaine de logiciels qui permettent d'utiliser ce protocole.

La particularité de BitTorrent par rapport aux autres architectures P2P est qu'il est en fait constitué de mini réseaux P2P. Pour chaque fichier présent dans le réseau, il y a un fichier *.torrent* qui contient l'adresse d'un ou plusieurs *tracker*. Ces derniers sont des points d'entrée dans le réseau et référencent les utilisateurs qui possèdent soit le fichier en totalité (on appelle ces utilisateurs des *sources* (*seeds*)), soit des fragments de ce fichier, comme illustré sur la figure 2.2.

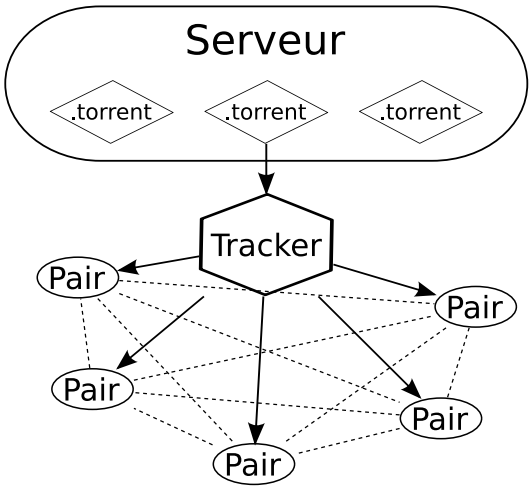


FIG. 2.2 – Architecture de BitTorrent. Les fichiers *.torrent* sont généralement accessibles depuis des sites web.

La recherche d'information passe donc par des serveurs qui hébergent les *.torrents*. Cela rend la recherche d'information plus difficile car il faut interroger les serveur un par un. De plus l'entretien et la mise à jour des *.torrent* n'est pas automatisée, ce qui impose une charge de travail supplémentaire. En contrepartie, ces interventions "manuelles" permettent d'avoir très peu de faux fichiers (*fake*).

Le principal atout de BitTorrent est son protocole de téléchargement avancé. L'utilisation d'un ratio de partage (quantité de données envoyées divisée par la quantité de données reçues) permet de décourager les profiteurs (*free-riders*) et récompense les utilisateurs qui contribuent au bon fonctionnement du réseau. Ceci permet d'obtenir des taux de transfert relativement élevés, et un chargement des fichiers assez rapide. C'est d'ailleurs pour cette raison que BitTorrent est devenu très populaire.

Par contre, la disponibilité des informations est fortement liée aux *trackers* ; si le tracker responsable d'un fichier devient indisponible, alors le fichier ne sera plus disponible dans le réseau. C'est aussi pour cette raison que la durée de vie des informations dans le réseau est assez courte (de l'ordre de quelques mois). Ceci montre clairement la spécialisation du réseau Bit-torrent dans la diffusion rapide de fichiers récents.

2.1.4 Comparatif

Le tableau 2.1 résume les différences entre les architecture P2P centralisées qui ont été abordées dans cette section. Napster est dédié à l'échange de fichier audio, donc de petite taille (de 3 à 5 Mo a peu près). Le téléchargement monosource sans possibilité de reprendre un transfert interrompu n'a donc pas été un frein à son développement. eDonkey et BitTorrent qui permettent un téléchargement plus rapide, ainsi que la possibilité de l'interrompre, se sont spécialisés dans le partage de fichiers plus volumineux,

comme par exemple des films, des albums (archives contenant plusieurs fichiers audio) ou des jeux vidéos.

Architecture	Sources multiples	Reprise du transfert	Taille des fichiers	Qualité de service sans contrepartie	Durée de vie des données
Napster	non	non	Mo	oui	++
eDonkey	oui	oui	Go	oui	++
BitTorrent	oui	oui	Go	non	+

Tab. 2.1 – Comparaison des différentes architectures P2P centralisées.

2.2 Architectures P2P structurées (DHT)

La plupart des architectures P2P structurées implémentent une Table de Hachage Distribuée (DHT) et associent la localisation des informations à la topologie du réseau. Les DHT fournissent une opération de base : étant donnée une clé, elles font correspondre cette clé avec un nœud du réseau.

Ces systèmes sont particulièrement adaptés pour retrouver des informations peu répliquées. Cependant, le hachage détruit l'ordre sur les clés, c'est-à-dire que deux clés semblables peuvent avoir des valeurs de hachage très différentes. Ainsi, il est très coûteux de répondre à des requêtes approchées ou portant sur un intervalle.

Nous présentons rapidement à la fin de cette section des architectures P2P ayant une structure en arbre. Elles permettent d'acheminer efficacement les requêtes dans un intervalle.

De plus, même si les clés sont distribuées de manière homogène, la taille des informations liées à ces clés peut varier énormément, ce qui peut engendrer des problèmes d'équilibrage de charge, notamment en terme de stockage et de bande passante. Enfin, pour effectuer une recherche, il faut *à priori* une connaissance complète de la clé associée à cette recherche.

Mis à part CAN [60] qui repose sur un espace cartésien et Viceroy [52] qui repose sur un espace en "papillon", que nous décrivons tous deux plus loin dans ce chapitre, les autres architectures présentées dans cette section utilisent une topologie inspirée du maillage de Plaxton [57], semblable à celle d'un hypercube, comme illustré sur la figure 2.3.

Le maillage de Plaxton impose certaines contraintes assez limitantes notamment la nécessité d'une connaissance globale pour établir les liaisons uniques entre les identifiants de documents et leur nœud racine, ce qui complique beaucoup les processus d'ajout et de suppression de nœuds dans le réseau. La nature statique du maillage de Plaxton entraîne une faible capacité d'adaptation aux changements dynamiques dans le réseau.

Plus précisément, étant donné un réseau avec un espace d'adressage de taille N

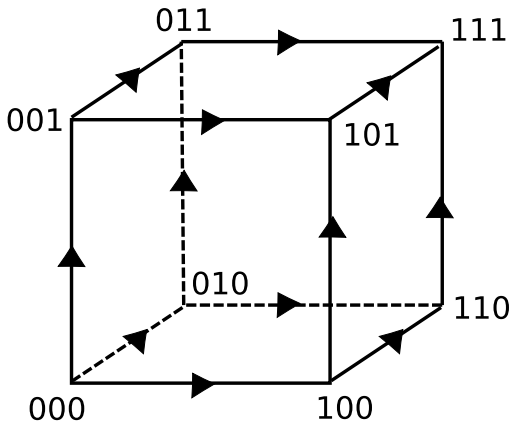


FIG. 2.3 – Exemple d'architecture utilisant un maillage en hypercube en dimension 3. Les flèches illustrent les différentes routes que peuvent prendre des requêtes concernant les clés stockées sur le nœud 111 à partir du nœud 000.

(les nœuds du réseau ont leur adresse sur $log(N)$ bits), le principe est de connecter chaque nœud à $O(log(N))$ autres nœuds. La manière dont les nœuds sont connectés dans le réseau varie suivant les différents protocoles. La recherche d'information consiste à faire suivre la requête vers un nœud voisin ayant une adresse d'un bit plus proche de la requête. Ceci permet de répondre à toutes les requêtes en $O(log(N))$ sauts (transition d'un nœud à un autre) dans les conditions optimales.

2.2.1 Pastry

Dans l'architecture de Pastry [63], inspirée de celle de Plaxton [57], l'identifiant unique des nœuds est codé sur 128 bits. Quand un nœud rejoint le réseau, il reçoit un identifiant généré aléatoirement, en supposant que l'ensemble des identifiants ainsi généré est réparti de manière uniforme dans l'espace de nommage. On associe également à chaque donnée une clé de 128 bits, et la donnée est stockée dans le réseau sur le nœud ayant l'identifiant le plus proche de cette clé.

Chaque nœud dans Pastry maintient une table de routage, un ensemble de nœuds voisins et un ensemble de nœuds feuilles. En utilisant des tables de routage contenant B colonnes et $log_B(N)$ lignes, Pastry permet d'acheminer les requêtes en $O(log_B(N))$ sauts, comme illustré sur la figure 2.4.

L'ensemble des nœuds voisins contient les identifiants et adresses IP des nœuds les plus proches. La mesure de proximité est fournie par un programme externe qui se base sur l'adresse IP du nœud cible ; on peut facilement utiliser une autre mesure, comme le plus court chemin en terme de nombre de sauts, la plus haute bande passante, la plus faible latence, ou même une combinaison de ces différents critères.

L'ensemble des nœuds feuilles est composé pour moitié des nœuds ayant les identifiants inférieurs les plus proches, et pour moitié des nœuds ayant les identifiants

0354	**1474**	2314	3441	4687	5317	6321	7644	8138	9344
1032	1184	1298	**1347**	1489	1515	1664	1789	1814	1998
1309	1317	1322	**1335**	1347	1359	1366	1378	1384	1393
1330	1331	1332	1333	1334	1335	1336	**1337**	1338	1339

FIG. 2.4 – Exemple de table de routage pour le nœud 1337 (en gras), avec $B = 10$ et $N = 10000$. A chaque nouvelle ligne, la longueur du préfixe commun est augmentée de un. Le tableau contient $log_{10}(10000) = 4$ lignes.

supérieurs les plus proches. Plus la taille de cet ensemble est importante, plus le système est fiable, mais plus les coûts en mémoire et maintenance du système sont importants. En général, la taille de ces deux ensembles est B ou $2 \times B$.

Plusieurs applications utilisent Pastry :

- **Scribe** [64] qui est un système d'abonnement / publication à des thématiques. Quand un utilisateur crée une thématique, elle est stockée dans le système en calculant la valeur de hachage du nom de la thématique concaténée avec le nom du créateur de cette thématique. Cet utilisateur peut ensuite publier des nouvelles qui seront transmises aux abonnés à cette thématique via un arbre de diffusion multicast.
- **PAST** [25] est un système de fichiers distribué. Les fichiers sont insérés ou retrouvés dans le système en calculant une clé, qui est la valeur de hachage du nom du fichier, et en hébergeant le fichier sur le nœud x qui a l'identifiant le plus proche de la clé. Des copies sont également créées sur les autres nœuds ayant les identifiants les plus proches de la clé, la plupart de ces nœuds étant dans l'ensemble des nœuds feuille du nœud x.
- **Squirrel** [36] permet à ses utilisateurs de partager le cache de leur navigateur web.
- **Pastiche** [19] est un système de sauvegarde qui permet d'utiliser l'espace disque libre des utilisateurs pour réaliser des sauvegardes de fichier.

2.2.2 Tapestry

Tapestry [75] est lui aussi inspiré du maillage de Plaxton [57] et utilise donc un mécanisme de routage assez semblable à celui de Pastry. Par contre, l'identifiant unique des nœuds est codé sur 160 bits. La principale différence entre Tapestry et Pastry est la manière dont sont pris en compte le positionnement dans le réseau, ainsi que la réplication des données.

Chaque nœud maintient une table de routage à plusieurs niveaux, semblable à celle du tableau 2.4, où chaque niveau contient les adresses des nœuds qui possèdent le même préfixe que le nœud courant, la longueur du préfixe en commun dépendant du niveau dans le tableau. Tout comme Pastry, chaque nœud maintient aussi un ensemble de pointeurs sur les nœuds voisins.

Parmi les applications qui utilisent Tapestry, on peut citer :

- **Oceanstore** [43] qui est un système de stockage distribué. L'utilisation de fragmentation, redondance et dispersion des données le rend très résistant aux pannes. De plus il supporte le travail collaboratif et utilise des techniques cryptographiques pour résister aux attaques malveillantes.
- **Bayeux** [67] est une application auto-organisée pour faire de la diffusion multicast.
- **SpamWatch** [76] est un filtre anti-spam décentralisé qui utilise le mécanisme de recherche par similarité implémenté dans Tapestry.

2.2.3 Chord

Le protocole Chord [71] organise les nœuds sur un anneau, en codant l'identifiant des nœuds sur 160 bits. Ce protocole est basé sur une métrique circulaire. Chaque nœud est connecté aux nœuds qui ont leur adresse juste avant ou juste après la sienne. Pour accélérer le routage (en $O(log(N))$ si on s'en tient à une topologie en anneau, des raccourcis sont créés.

Chaque nœud se connecte à $O(log(N))$ autres nœuds. Si x est l'adresse du nœud, alors il se connecte aux nœuds ayant pour adresse $x + 2^i \ mod \ N, 0 < i < N$. Si le nœud doit se connecter à un nœud qui n'est pas présent dans le réseau, il se connecte au nœud ayant l'adresse supérieure la plus proche (modulo N). Un exemple de réseau de nœuds ayant leur adresse sur trois bits est illustré à la figure 2.5.

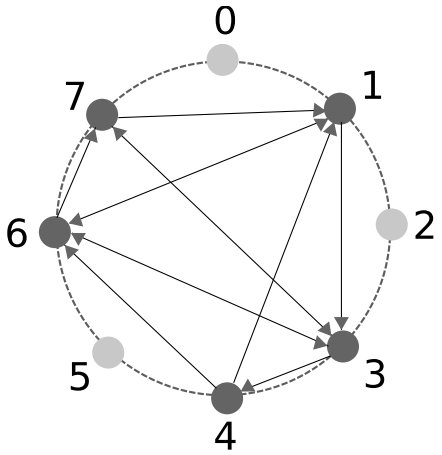


Fig. 2.5 – Exemple de réseau de cinq nœuds (en noir) utilisant le protocole Chord. Les adresses des nœuds sont codées sur trois bits, chaque nœud a donc trois connexions sortantes au maximum.

Grâce a une fonction de hachage consistante, les nœuds reçoivent approximativement le même nombre de clés. Quand un nœud rejoint le réseau, une partie de $O(1/N)$ des clés est déplacée vers des endroits différents. C'est le minimum nécessaire pour assurer que les nœuds conservent approximativement le même nombre de clés.

La topologie de Chord est asymétrique : un nœud peut avoir des informations de routage pour atteindre un autre nœud, même si ce dernier ne connaît même pas l'existence du premier. Les nœuds dans Chord ne peuvent donc pas recevoir des informations de routage à partir des requêtes qu'ils reçoivent. Cette asymétrie a pour conséquence des tables de routage rigides, qui ne sont pas très adaptées à un environnement très dynamique.

Chord est utilisé dans les applications suivantes :

– **Cooperative File System (CFS)** [23] qui est un système de stockage de fichiers distribué. Ce système répartit la charge de manière équitable sur tous les nœuds du réseau.
– **Résolution de DNS** de manière distribuée [20].

O-Chord [41] étend le protocole Chord pour pouvoir traiter des requêtes plus complexes que des simples mots-clés, comme par exemple des requêtes SQL. O-Chord permet aux pairs du réseau d'échanger plus efficacement leurs données grâce à l'utilisation d'ontologies de domaine de connaissance, tout en gardant les performances de routage et de passage à l'échelle du protocole Chord.

2.2.4 Kademlia

Le protocole de Kademlia [53] est basé sur la métrique du OU exclusif, et tout comme la plupart des architectures présentées dans cette section, l'identifiant des nœuds est codé sur 160 bits. La topologie obtenue est donc symétrique, ce qui fait que chaque nœud reçoit à peu près la même quantité de requêtes de ses différents voisins.

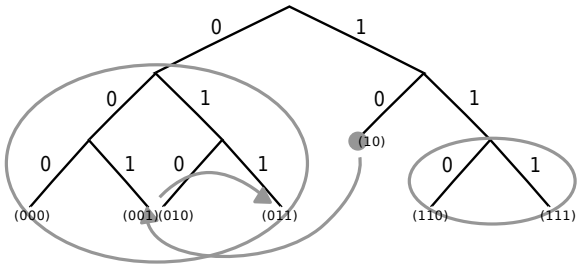


FIG. 2.6 – Le nœud 10 (point gris) possède un voisin dans chaque sous-arbre entouré. Les flèches indiquent un message routé de 10 vers 011, en passant par 001.

La figure 2.6 illustre l'arbre binaire de Kademlia. Chaque nœud doit posséder un contact dans chaque sous-arbre. Le plus haut sous-arbre correspond à la moitié de l'arbre binaire qui ne contient pas le nœud, le sous-arbre suivant correspond à la moitié de l'arbre restant qui ne contient pas le nœud, etc ....

Le protocole Kademlia est un peu plus souple que Chord, un nœud peut choisir ses contacts dans chaque sous-arbre en fonction de leur latence par exemple. Un nœud peut même avoir plusieurs contacts dans chaque sous-arbre, ce qui permet de lancer des requêtes en parallèle, ou d'optimiser les requêtes pour qu'elles emploient les chemins avec les meilleures latences.

Kademlia a récemment été incorporé dans certaines applications P2P de partage de fichier, notamment eDonkey [30] (Overnet), eMule (Kad [69]), ou certains clients de Bit-torrent [58], nous en reparlons brièvement dans la dernière section de ce chapitre.

2.2.5 CAN

CAN (Content Addressable Network) [60] repose sur un espace cartésien d-dimensionnel sur un tore. Chaque nœud occupe une portion de cet espace et possède un nombre de voisins en $O(d)$, d étant la dimension de l'espace virtuel. Plus la dimension de cet espace est grande, plus le routage est rapide, en revanche plus le coût de maintien des tables de routage est élevé.

Le routage d'une clé vers un nœud se fait de manière gloutonne en $O(d.N^{\frac{1}{d}})$, en envoyant le message vers le voisin qui a les coordonnées les plus proches de celle du nœud vers lequel on veut aller, comme illustré sur la figure 2.7.

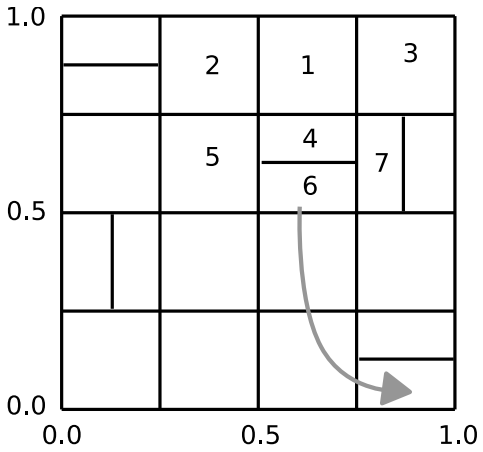


Fig. 2.7 – Topologie sur un espace à 2 dimensions. La flèche indique un routage du nœud 6 vers un point situé dans $([0.75; 1.0], [0.0; 0.25])$.

Un nouveau pair rejoignant le réseau doit être responsable d'une partie de l'espace : il reçoit en conséquence la moitié d'une zone dont était responsable un autre pair (qui conserve l'autre moitié pour lui). Quand une panne est détectée par un nœud voisin, ce dernier récupère l'espace dont était responsable le nœud défaillant, met à jour ses tables de routage et envoie un message à ses voisins, pour s'assurer que leurs tables de routage ont bien été mises à jour.

2.2.6 Viceroy

La topologie virtuelle maintenue par Viceroy [52] est celle d'un graphe en papillon, comme illustré sur la figure 2.8. Le diamètre de cette topologie est inférieur à celui de CAN, et le degré moyen des nœuds est inférieur à celui observé sur le maillage de Plaxton.

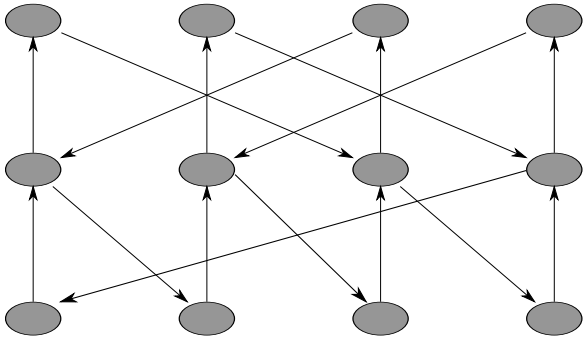


FIG. 2.8 – La topologie de Viceroy est un graphe en papillon.

Le routage s'effectue en remontant verticalement dans le graphe, puis ensuite en redescendant jusqu'au nœud cible. Ce processus ce fait en $O(log(N))$ sauts, N étant le nombre de nœuds dans le réseau.

2.2.7 Comparatif

Le tableau 2.2 résume les différences entre les architecture P2P structurées abordées dans cette section.

Architecture	Taille des tables de routage	Performances du routage	Coût d'ajout ou de suppression de nœud	Flexibilité des tables de routage
Pastry	$2.B.log_B N$	$O(log_B N)$	$log_B N$	élevée
Tapestry	$log_B N$	$O(log_B N)$	$log_B N$	élevée
Chord	log_N	$O(log_N)$	$(log_N)^2$	faible
Kademlia	$B.log_B N + B$	$O(log_B N) + c$	$log_B N + c$	élevée
CAN	$2.d$	$O(d.N^{\frac{1}{d}})$	$2.d$	élevée
Viceroy	log_N	$O(log_N)$	log_N	faible

TAB. 2.2 – Comparaison des différentes architectures P2P structurées.

Mis à part CAN qui présente des propriétés différentes, toutes les autres architectures P2P structurées présentées ici offrent à peu près des performances en $O(log(N))$ par rapport à la taille des tables de routage, des performances de routage et des coûts de maintien (les chiffres avancés concernent en général les cas les plus favorables) de ces tables de routage.

Certaines architectures permettent de régler la base B du logarithme utilisé, et donc la taille des tables de routage. Cela permet de favoriser soit la vitesse du routage, soit l'occupation mémoire des tables de routage (ainsi que le nombre de connexions ouvertes par nœud).

2.2.8 Autres architecture P2P structurées

Comme nous l'avons mentionné précédemment, un inconvénient des DHT est leurs faibles performances pour résoudre des requêtes portant sur un intervalle. Des travaux plus récents se sont attelés à la résolution de ce problème.

2.2.8.1 P-Tree

P-Tree [21] adopte une structure d'arbre B+ (cas particulier d'arbre équilibré dans lequel les données ne sont stockées que sur les nœuds feuille) et utilise Chord pour réaliser le routage des requêtes. P-Tree permet l'acheminement des requêtes en $O(log(N))$, tant pour les requêtes exactes que les requêtes portant sur un intervalle. La structure de l'arbre B+ est par contre assez coûteuse à maintenir : quand un nœud rejoint le réseau, en plus des coûts en $O(log(N))$ pour la recherche du prédécesseur dans l'anneau et en $O(log^2(N))$ pour mettre à jour les tables de routage, le coût de récupération de la structure de l'arbre auprès du prédécesseur est assez important. Enfin P-Tree est conçu pour assigner une donnée par pair, et n'est donc pas adapté au stockage de grands ensembles de données.

2.2.8.2 P-Grid

P-Grid [1] possède une structure d'arbre binaire dans laquelle chaque nœud maintient des références vers les nœuds de même préfixe de longueur l, mais avec une valeur différente pour la position $l+1$. Cependant, suivant la distribution des données, il peut arriver que l'arbre ne soit pas équilibré du tout et que l'acheminement des requêtes ne soit plus garantit en $O(log(N))$. De plus, P-Grid ne supporte que les requêtes portant sur le préfixe, et non les requêtes portant sur un intervalle en général.

2.2.8.3 Baton

Baton [38] maintient une structure d'arbre B qui est équilibrée quelle que soit la distribution des données. Il permet lui aussi l'acheminement des requêtes exactes et dans un intervalle en $O(log_2(N))$. Cependant, quand la taille du réseau est grande, la faible base du logarithme peut entraîner des coûts de recherche non négligeables. Baton* [37] offre des performances en $O(log_d(N))$, mais ne fournit aucune assurance quand à l'équilibrage de charge.

2.3 Architectures P2P non-structurées

Les architectures P2P non structurées n'imposent aucune contrainte entre la localisation des données et la topologie du réseau. Ces systèmes sont particulièrement adaptés pour retrouver de l'information ayant un grand nombre de copies, mais montrent leur limite pour la recherche d'information peu répliquée.

Du fait des faibles contraintes imposées sur la topologie virtuelle, ces systèmes sont particulièrement adaptés aux environnements très dynamiques.

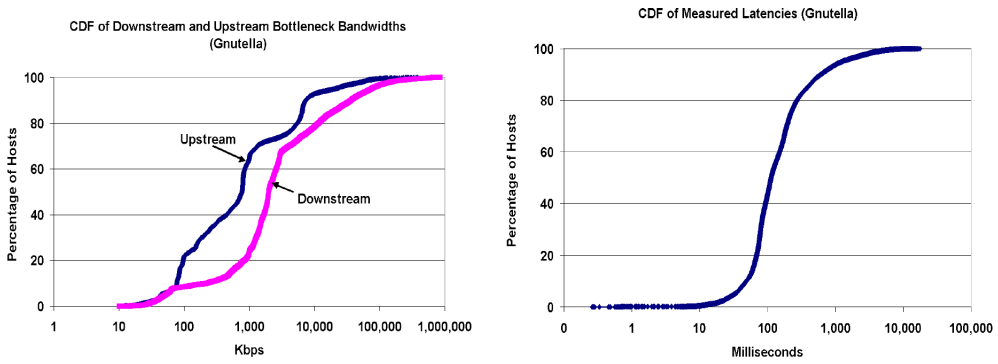
2.3.1 Gnutella

Gnutella [17] possédait à l'origine (version 0.4 du protocole) une topologie en graphe aléatoire, comme Freenet. Aujourd'hui, l'architecture de Gnutella repose sur un réseau hiérarchique avec des super-pairs (version 0.6 du protocole). Du fait de protocoles simples et ouverts, mais aussi du fait de ses faibles performances concernant la recherche d'information, Gnutella est le système P2P non structuré qui a le plus fait l'objet de recherches dans les dernières années.

Beaucoup de mesures ont été effectuées sur Gnutella [66]. Elle mettent notamment en évidence l'hétérogénéité naturellement présente dans ce type de réseau, tant du point de vue de la répartition des ressources (CPU, mémoire, bande passante) que de la répartition des informations.

2.3.1.1 Caractéristiques des connexions

La figure 2.9 illustre l'hétérogénéité des connexions des différents pairs qui participent au bon fonctionnement du réseau. On retrouve la même hétérogénéité au niveau de la bande passante (figure 2.9(a)) ou de la latence (figure 2.9(b)) des différents nœuds.



(a) Répartition cumulée de la bande passante des différents pairs

(b) Latence cumulée des différents pairs

FIG. 2.9 – Caractéristiques des connexions, tirée de [66]

On observe ainsi quelques nœuds avec des connexions soit très mauvaises, soit excellentes, tandis que la majorité des nœuds possède une connexion de qualité moyenne. Les mesures présentées ici datent de 2003 mais nous pensons qu'elles sont toujours d'actualité, non pas en terme de données absolues mais de répartition, et que ces distributions suivent toujours une gaussienne (le cumul de ces distribution suit donc une sigmoïde) : même si l'ADSL a remplacé le modem classique dans les campagnes, on peut avoir des connexions de bien meilleure qualité dans les grandes villes.

2.3.1.2 Durée des sessions

La figure 2.10 illustre la longueur des sessions des différents pairs présents dans le réseau. La répartition cumulée de la longueur de ces sessions suit une loi logarithmique : la majorité des pairs se connecte pendant de courtes durées (en général inférieures à une heure), tandis que peu de pairs restent connectés longtemps.

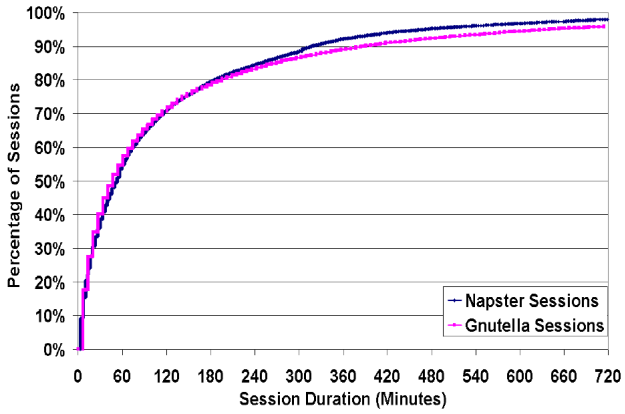


FIG. 2.10 – Longueur moyenne des sessions, tirée de [66].

Comme les mesures décrites à la section précédente, celle-ci date aussi de 2003. Cependant, nous pensons qu'il n'y a pas eu d'évolution notable du comportement des utilisateurs, et que cette répartition sur la durée des sessions entre les différents pairs, même si elle a pu légèrement évoluer, reste d'actualité.

2.3.1.3 Données partagées

On peut voir sur la figure 2.11 la quantité de données partagées par chaque utilisateur. On observe là encore une répartition très hétérogène : peu d'utilisateurs partagent beaucoup ou peu de fichiers, tandis que la majorité partage une quantité moyenne de fichiers. On voit aussi que 20% des utilisateurs ne partagent pas du tout de fichiers : ce sont des profiteurs (en anglais *free rider* [2]) qui, comme ils ne contribuent pas, gênent le bon fonctionnement du réseau.

On voit là encore que la répartition suit une densité gaussienne, ce qui laisse à penser que si l'on regroupe les capacités des différents pairs en terme de mémoire et bande passante, on obtient là encore une répartition cumulée qui suit une sigmoïde. Nous adoptons cette hypothèse pour simplifier la suite de notre étude et nous caractérisons les différents pairs par un critère de "capacité", qui désigne tant leurs capacités en terme de bande passante, que de mémoire et même de CPU.

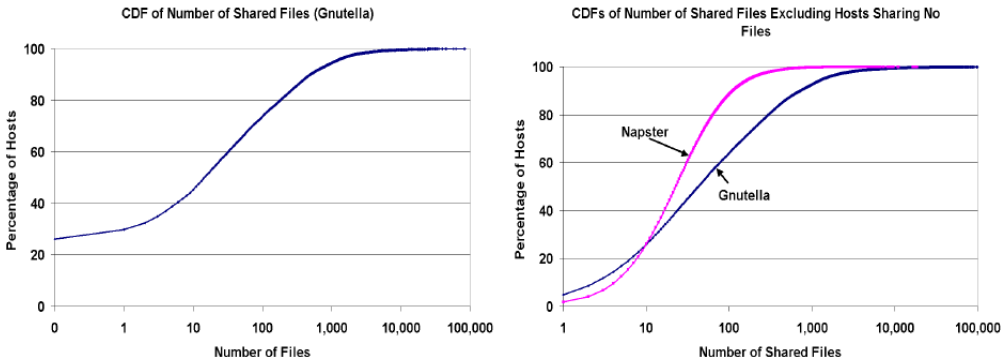


FIG. 2.11 – Distribution du partage des fichiers, tirée de [66].

2.3.2 eDonkey (version2)

eDonkey [30] qui relevait à l'origine d'une architecture P2P centralisée a déjà été abordé à la section 2.1.2. Les différents serveurs ont ensuite été connectés entre eux : c'est maintenant un réseau P2P hiérarchique dans lequel on peut considérer les serveurs comme les super-pairs, et les clients des serveurs comme des pairs simples.

eDonkey est actuellement l'un des réseau P2P les plus utilisés, nous présentons ici des mesures sur les fichiers partagés effectuées par [27]. Pour des raisons légales, des serveurs (super-pairs) ont été mis hors service, et les logiciels permettant de se connecter au réseau eDonkey implémentent maintenant des DHT, pour permettre un fonctionnement complètement décentralisé.

2.3.2.1 Données partagées

La figure 2.12 présente la quantité de données partagées par les utilisateurs du réseau, d'une part en tenant compte des profiteurs (personnes ne partageant aucune donnée), d'autres part en les ignorant. On voit que la majorité des personnes partagent peu de fichiers, mais par contre très peu de personnes partagent moins de 1 Go de données, ce qui illustre la spécialisation du réseau dans le partage de gros fichiers.

2.3.2.2 Quantité de réplicas

La figure 2.13 présente la répartition des fichiers dans le réseau en fonction de leur nombre de copies. Ces mesures ont été faites sur une période de cinq jours et coïncident avec les mesure faites sur d'autres systèmes P2P décrites par la figure 2.11. Elles mettent en évidence une réplication très hétérogène : quelques fichiers sont très répliqués, alors que la majorité de ces fichiers n'est pas répliquée.

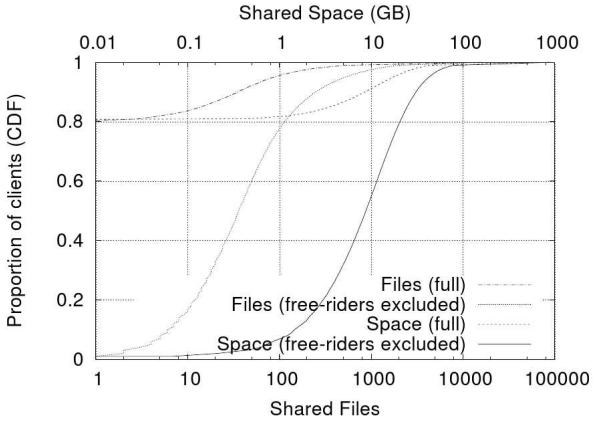


FIG. 2.12 – Distribution du partage des fichiers, tirée de [27].

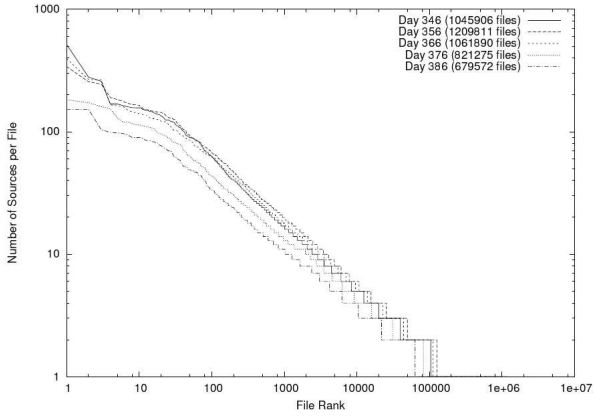


FIG. 2.13 – Distribution du nombre de replicas, tirée de [27].

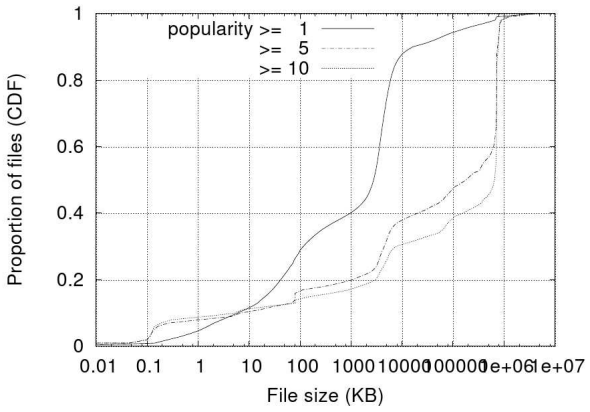


FIG. 2.14 – Distribution de la taille des fichiers, tirée de [27].

2.3.2.3 Taille des fichiers

La figure 2.14 décrit la répartition cumulée des fichiers en fonction de leur taille et de leur popularité. On remarque que les fichiers les plus populaires sont les plus volumineux, ce qui illustre encore une fois la spécialisation du réseau eDonkey dans le partage de gros fichiers.

2.3.3 Freenet

Freenet [16] possède une topologie en graphe aléatoire comme illustré sur la figure 2.15 : les nœuds qui se connectent au réseau choisissent leurs voisins de manière aléatoire. Contrairement aux autres architectures P2P non structurées présentées dans cette section, Freenet fait correspondre à chaque information une clé (obtenue par hachage). Tout comme la plupart des DHT, on ne peut envisager que des requêtes exactes, mais ceci permet d'obtenir des bonnes performances de routage.

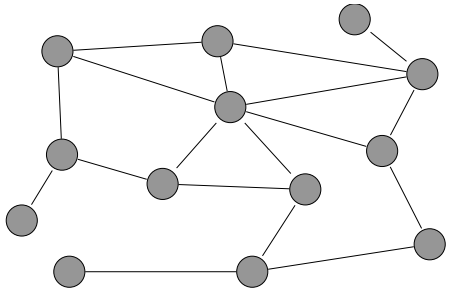


FIG. 2.15 – Exemple de graphe aléatoire.

Dans cette architecture, les utilisateurs mettent un cache à disposition du système. Quand un utilisateur A souhaite télécharger un fichier, il n'y a pas de connexion directe

entre lui et l'utilisateur B qui possède le fichier. Ce dernier est transmis de proche en proche à tous les pairs qui sont sur le chemin qu'a pris la requête entre les nœuds A et B. Ce mécanisme induit un temps de récupération des information plus long et l'utilisation supplémentaire de bande-passante, mais permet d'assurer une forme d'anonymat, d'accélérer les temps de réponse et enfin d'introduire de la redondance qui fiabilise le système.

Freenet [16] permet aussi de construire des réseaux ami-à-ami (F2F de l'anglais friend-to-friend), qui sont des cas particuliers de réseaux P2P dans lesquels tous les voisins d'un nœud sont des nœuds "amis". Ces réseaux permettent une meilleure régulation du contenu global du réseau, car on peut pénaliser un pair qui a contribué au transfert d'un fichier frauduleux et ainsi n'avoir confiance qu'en des pairs transmettant des fichiers corrects.

2.3.4 FastTrack

Le réseau FastTrack , dont le client le plus connu est Kazaa [48], est un réseau P2P non-structuré hiérarchique. FastTrack est un protocole propriétaire, utilisant des techniques de cryptage ; il est donc assez difficile d'obtenir des informations précises à son sujet. Cette section s'appuie sur les mesures et recherches effectuées par [47] sur ce réseau. FastTrack était très utilisé en 2003, avec plus de trois millions d'utilisateurs qui partageaient cinq pétaoctets de données.

La topologie du réseau FastTrack est une topologie hiérarchique à deux niveaux, comme illustré sur la figure 2.16. Dans le niveau du haut, il y a les super-nœuds (SN) et dans le niveau du bas les nœuds ordinaires (NO). Nous nous intéressons particulièrement à ce réseau P2P car il était l'un des premiers à tirer partie de l'hétérogénéité des capacités de calcul, mémoire ou bande-passante entre les différents pairs naturellement présente dans le réseau.

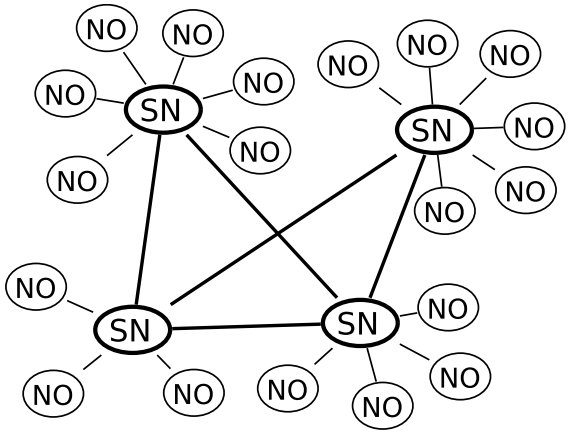


Fig. 2.16 – Topologie à deux niveaux du réseau FastTrack.

Plus particulièrement, les SN ont généralement des capacités de traitement, mémoire

et bande-passante au-dessus de la moyenne. Il ont aussi la plupart du temps une durée de connexion plus stable dans le temps. Du fait de leur rôle, ces SN ont aussi des responsabilités plus importantes dans le réseau. Chaque NO a un parent SN, qu'il choisit quand il se connecte au réseau FastTrack. Il maintient une connexion TCP intermittente avec son SN et lui envoie les métadonnées des fichiers qu'il partage. Les SN possèdent donc toutes les informations d'indexation des NO dont ils sont responsables, et peuvent répondre aux requêtes qui concernent tous ces NO.

Pour que les requêtes puissent être envoyées dans tout le réseau, les SN maintiennent également des connexions TCP longue durée avec d'autres SN. Ainsi, une requête émise par un NO est dans un premier temps traitée par son SN, qui regarde si tous les NO dont il est responsable ont des fichiers qui correspondent à la requête. Il relaye ensuite la requête à d'autres SN auxquels il est connecté.

2.3.5 Gia

Gia a été proposé par [15] et propose plusieurs améliorations à Gnutella. Contrairement au réseau FastTrack qui utilise des protocoles propriétaires, Gia utilise des protocoles ouverts. Par contre, tout comme FastTrack, il tire partie de l'hétérogénéité entre les capacités des nœuds présents dans le réseau. Quatre améliorations majeures sont proposées pour accroître les performances du système et garantir un passage à l'échelle : maintenance d'une topologie particulière qui tient compte des capacités des nœuds, contrôle de flot actif pour éviter la surcharge de certaines régions du réseau, pointeur sur le contenu des nœuds voisins, et amélioration du protocole de recherche d'information.

2.3.5.1 Topologie

Gia prend en compte les différents niveaux de capacité des nœuds dans le réseau. Il possède une topologie adaptative qui fait en sorte que les nœuds ayant les plus grandes capacités sont ceux ayant le plus de voisins. De plus les nœuds ayant de faibles capacités sont rapprochés d'au moins un nœud à forte capacité.

La topologie de Gia ressemble donc à celle de FastTrack, dans le sens ou les nœuds ayant le plus de voisins sont ceux ayant le plus de capacité. Il n'y a par contre aucune distinction pairs / super-pairs, et donc pas de topologie à deux niveaux.

2.3.5.2 Contrôle de flot

Pour éviter que certains nœuds ne soient surchargés de requêtes, Gia utilise un mécanisme de contrôle de flot actif. Un nœud qui désire propager une requête ne peut le faire que si le voisin vers lequel il veut propager la requête l'a explicitement informé qu'il pouvait recevoir des requêtes.

Ce mécanisme de contrôle proactif contraste avec des techniques réactives, comme par exemple l'abandon d'une requête par un nœud s'il est surchargé. Cette dernière n'est d'ailleurs pas du tout adaptée à Gia qui, contrairement à Gnutella qui propage les

requêtes par inondation, utilise un mécanisme de marche aléatoire, détaillé à la section 2.3.5.4.

Le contrôle de flot est mis en œuvre en utilisant un mécanisme de jetons : les nœuds donnent régulièrement des jetons à leurs voisins, qui peuvent utiliser ces jetons pour propager des requêtes vers le nœud qui leur a donné ces jetons. Si un nœud devient surchargé de requêtes, il réduit la fréquence à laquelle il donne des jetons à ses voisins.

2.3.5.3 Connaissance du contenu des nœuds voisin

Les nœuds du réseau échangent périodiquement avec leurs voisins les index des fichiers qu'ils possèdent. Cela permet d'améliorer l'efficacité de la recherche d'information car un nœud qui reçoit une requête peut y répondre, non seulement pour lui, mais aussi pour tout ses voisins. Contrairement à FastTrack où seulement les SN peuvent répondre aux requêtes des NO dont ils sont responsables, tout les nœuds de Gia possèdent les informations d'indexation de leurs voisins.

Bien sûr, quand un nœud perd un voisin (la perte étant détectée quand il ne reçoit pas de message PING de ce nœud depuis un certain délai), soit à cause de son départ, soit à cause de l'adaptation de la topologie, il supprime les informations d'indexation de ce voisin.

2.3.5.4 Recherche d'information

La combinaison de la topologie adaptative qui fait en sorte que n'importe quel nœud du réseau est à proximité d'un nœud ayant beaucoup de voisins, et de la connaissance du contenu des voisins fait que les nœuds ayant des capacités importantes peuvent répondre de manière efficace aux requêtes. Ces nœuds peuvent d'ailleurs être plus ou moins vus comme les SN du réseau.

Comme il n'y a pas de distinction de niveau entre les SN et les NO comme dans FastTrack, Gia utilise un mécanisme de propagation des requêtes basé sur une marche aléatoire modifiée. Au lieu de transmettre les requêtes à des voisins choisis au hasard, les nœuds dans Gia essayent d'aiguiller les requêtes vers les nœuds ayant le plus de voisins, tout en respectant bien sur le mécanisme de contrôle de flot décrit à la section 2.3.5.2.

2.3.6 BubbleStorm

BubbleStorm [72] est une architecture P2P qui a été proposée récemment. Le principe est assez simple : les données et les requêtes sont répliquées suivant un schéma qui ressemble à une inondation locale. L'ensemble des réplicas d'une donnée ou d'une requête forme alors une bulle. Quand la bulle d'une requête entre en contact avec la bulle d'une donnée correspondant au critère de cette requête, alors cette dernière peut être (partiellement) résolue. Le pair qui possède un réplica d'une donnée et qui reçoit un réplica d'une requête correspondant à la donnée qu'il possède est appelé *pair rendez-vous*.

Ainsi, si les données sont suffisamment répliquées, elles forment des bulles de taille suffisante pour que les bulles des requêtes correspondant à ces données ne soient pas trop importantes. Si cette contrainte de forte réplication de l'information est respectée, Bubblestorm offre des performances de recherche d'information et de tolérance aux pannes très élevées.

Enfin, plusieurs principes énoncés dans [72] nous semblent très pertinents :

1. Dissocier le transport des requêtes et le langage utilisé pour les formuler.
2. Attribuer aux nœuds une charge, et donc une valence, proportionnelle aux ressources dont ils disposent.

2.3.7 Comparatif

Le tableau 2.3 résume les différences entre les architecture P2P non structurées qui ont été abordées dans cette section.

Architecture	Topologie	Langage de requête	Réplication	Anonymat
Freenet	Aléatoire	Mots clé	Système	oui
FastTrack	Hiérarchique	Libre	Utilisateur	non
eDonkey	Hiérarchique	Libre	Utilisateur	non
Gnutella 0.4	Aléatoire	Libre	Utilisateur	non
Gnutella 0.6	Hiérarchique	Libre	Utilisateur	non
Gia	Aléatoire	Libre	Hybride	non
Bubblestorm	Aléatoire	Libre	Système	non

Tab. 2.3 – Comparaison des différentes architectures P2P non structurées.

Les performances de routage de Freenet, initialement faibles, s'améliorent de manière incrémentale au cours de son utilisation, pour atteindre des performances semblables à celles des DHT. Mais tout comme pour ces dernières, les requêtes sont limitées aux mots clés. Bien que les implémentations des protocoles Gnutella et eDonkey supportent des langages de requêtes prédéfinis, ces architectures peuvent théoriquement supporter n'importe quel langage de requête. La réplication dans Gia se distingue car d'une part les utilisateurs créent des réplicas en téléchargeant les données, d'autre part le système créé des pointeurs vers ces données sur les nœuds voisins.

2.3.8 Storm Botnet

Storm [31] est un *ver* qui est apparu le 17 janvier 2007. Il doit son nom à son mode initial de diffusion : c'était un *trojan* en pièce jointe d'un email parlant de tempête (storm en anglais) en Europe. Ce virus a été conçu pour profiter de failles de sécurité dans le système d'exploitation Windows et se servir de machines infectées pour d'autres tâches (contamination d'autres machines, envoi de spam ...).

Le nombre d'ordinateurs infectés par ce virus a été estimé fin 2007 entre 1 et 20 millions et sa puissance de calcul potentielle le place au dessus des super-calculateurs les plus puissants. Ce virus a été utilisé pour envoyer 75% de la totalité du spam mondial en 2007.

La particularité de Storm est qu'au lieu de renvoyer directement des informations à un serveur central, toutes les machines infectées par ce virus sont connectées dans un réseau P2P. Plus particulièrement, l'architecture P2P utilisé par Storm est Overnet (i.e. la même architecture que eDonkey).

Ce virus illustre malheureusement l'utilisation détournée que l'on peut faire des techniques P2P. Du fait de sa nature complètement décentralisée, le réseau Storm Botnet est extrêmement résistant aux attaques qui sont tentées pour le détruire. Il faut ajouter à ceci qu'il semble y avoir derrière ce virus une équipe de développeurs compétents qui le mettent régulièrement à jour pour contrecarrer les dispositions prises par les compagnies produisant des logiciels anti-virus.

Les concepteurs de Storm utilisent une technique appelée *fast-flux-DNS* pour envoyer leurs instructions au réseau de machines infectées. Cette technique consiste en gros à établir une connexion d'une machine infectée vers un serveur des concepteurs. Cette connexion est très brève : toutes les minutes, les concepteurs se connectent à une autre machine infectée. De plus les connexions entre machines infectées étant cryptées, on ne peut pas savoir si le virus communique avec le serveur des concepteurs ou une autre machine infectée.

2.4 Architectures P2P hybrides

Cette section présente quelques architectures P2P qui combinent des caractéristiques venant des architectures structurées et non structurées.

2.4.1 eDonkey et BitTorrent

eDonkey [30] et BitTorrent [58] sont les deux architectures P2P les plus populaires. Elles ont toutes les deux des éléments centralisés : trackers et sites les référençant pour BitTorrent, et serveurs pour eDonkey. Ces architectures sont utilisées pour faire du partage de données, les fichiers partagés étant la plupart du temps sous copyright, et contribuent donc, dans une proportion assez importante étant donné le trafic internet qu'elles génèrent, au piratage de fichiers.

La RIAA (Recording Industry Association of America) a ainsi saisi en 2006 le plus gros serveur d'eDonkey (portant le nom de Razorback), et certains sites web recensant des *.torrent* [5] ont été mis (temporairement) hors-service. Les deux réseaux ont cependant pu continuer de fonctionner en mode dégradé. Pour faire face aux problèmes liés à leurs éléments centralisés, ces deux systèmes implémentent maintenant la même DHT : Kademlia [53].

Les utilisateurs d'eDonkey [30] peuvent ainsi se connecter au réseau Overnet et les utilisateurs d'eMule au réseau Kad [69]. Les utilisateurs de ces réseaux peuvent donc

maintenant choisir soit de passer par les éléments centraux, soit d'envoyer leurs requêtes sur la DHT, soit d'utiliser les deux.

2.4.2 JXTA

JXTA [32] pour *juxtapose* est un environnement P2P développé en java. Il peut servir de support pour construire différentes applications P2P allant du stockage distribué au calcul réparti. JXTA tire parti de l'hétérogénéité des machines présentes dans le réseau pour construire un réseau hybride, combinant DHT et architecture P2P non structurée hiérarchique, comme illustré sur la figure 2.17. JXTA offre aussi des fonctionnalités pour que les nœuds derrière un pare-feu puissent participer au réseau P2P.

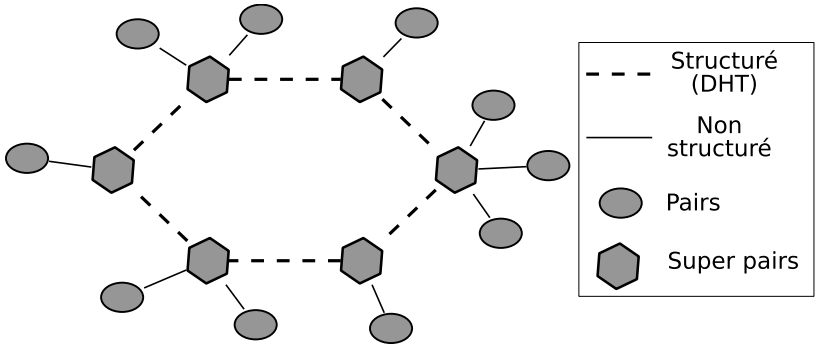


FIG. 2.17 – Architecture de JXTA : les nœuds ayants des ressources suffisantes deviennent super-pairs (hexagones) et sont connectés entre eux via une DHT. Ils gèrent chacun plusieurs pairs simples, qui sont des nœuds disposant de ressources moindres.

L'architecture de JXTA est composée de nœuds ayant différents rôles suivant les ressources qu'ils possèdent. Les nœuds peuvent avoir plusieurs rôles simultanément (ex : relais et rendez-vous).

- **Les pairs simples** : ils offrent et utilisent des services. Cette catégorie d'utilisateur est décomposée en deux parties : les pairs simples minimaux, qui ont de faibles ressources (ex : PDA ou téléphone portable) et des fonctions limitées, et les pairs simples complets, qui représentent la majorité des utilisateurs.
- **Les pairs rendez-vous** : ce sont des pairs qui ont des ressources supérieures à la normale, et qui ne sont pas derrière un pare-feu. Ils permettent aux autres pairs de découvrir les ressources dans le réseau, et peuvent être vus comme les super-pairs du réseau.
- **Les pairs relais** : ils permettent de trouver des chemins de communication avec les autres pairs.

2.4.3 Autres

Une architecture hybride a été proposée par [49] dans laquelle l'inondation (flooding) est utilisée pour retrouver les informations hautement répliquées et une DHT pour

retrouver les informations rares.

2.5 Choix de l'architecture

Du fait de leur nature complètement distribuée et de l'utilisation de techniques adaptatives, les architectures P2P sont bien adaptées aux environnements dynamiques. Suivant la nature de cet environnement (peu ou très dynamique, homogène ou hétérogène ...), certaines architectures sont plus adaptées que d'autres. Nous avons déjà présenté dans le chapitre 1 les raisons qui nous ont motivés dans le choix d'une architecture P2P. Cette section présente les raisons qui nous ont poussés à nous orienter vers une architecture P2P non-structurée plutôt que structurée.

2.5.1 Localisation des informations

La localisation des informations est ce qui différencie le plus les architectures structurées des architectures non-structurées. Alors que les premières font correspondre l'emplacement des informations à des propriétés topologiques et permettent ainsi d'acheminer très rapidement les messages vers leur destination, les secondes n'imposent aucune contrainte à ce niveau.

Il faut bien différencier la localisation des informations de leur accès. Dans le premier cas on utilise la topologie du réseau, et les performances des recherches sont donc dépendantes de celle-ci. Dans le second cas, une fois l'information localisée, une liaison directe entre le ou les nœuds contenant l'information et celui désirant y accéder est créée et le transfert s'effectue. Quand l'information recherchée est localisée, son accès est donc complètement dissocié de la topologie du réseau.

Bien sûr, certaines architectures proposant des mécanismes garantissant l'anonymat [16] ne permettent pas la copie de fichiers par création de liaisons directes entre le nœud possédant l'information et celui désirant y accéder, ce qui entraîne un surcoût en terme d'utilisation de mémoire et de bande-passante non négligeable pour ces architectures.

Dans une architecture structurée, c'est le système qui décide où sera stockée l'information. Cela permet entre autres de localiser l'information que l'on recherche plus rapidement. Dans les architectures non-structurées, ce sont les actions de l'utilisateur qui déterminent où sera stockée l'information, par exemple en téléchargeant l'information qui l'intéresse sur sa propre machine.

2.5.2 Hétérogénéité entre les pairs

La plupart des architectures P2P structurées proposées considèrent un réseau contenant un ensemble de machines homogènes du point de vue des capacités de traitement, stockage ou bande passante. Tous les nœuds d'une architecture P2P structurée sont égaux les uns par rapport aux autres. Comme la plupart de ces systèmes fournissent une table de hachage distribuée, un effort particulier est fait pour utiliser une fonction de hachage *consistante* qui distribue équitablement les clés entre les différents nœuds

du réseau, de sorte qu'ils disposent tous à peu près de la même quantité d'information à gérer.

Comme décrit dans le chapitre 1, nous souhaitons concevoir un système fonctionnant sur réseaux hétérogènes de machines, du point de vue des capacités de traitement, stockage ou bande passante.

2.5.2.1 Stockage

Dans les architectures P2P structurées, la localisation des informations est liée à l'identifiant d'un nœud. Il n'y aucune corrélation entre les identifiants des nœuds et leur capacité de stockage. L'insertion d'une information dans un réseau P2P structuré se fait donc sans la connaissance de la charge du nœud censé héberger cette information. Comme mentionné précédemment, les nœuds recevront à peu la même quantité d'information. Ainsi, les nœuds ayant des capacités de stockage en dessous de la moyenne seront en surcharge, alors que les nœuds ayant des capacités de stockage au dessus de la moyenne seront en sous-charge.

On peut solutionner ce problème en dissociant la localisation des clés de la localisation des informations. Au lieu de correspondre a des données, les clés sont associées à des pointeurs sur les données. Tous les nœuds du réseau auront alors à peu près le même nombre de pointeurs, mais un nombre de données différent. Cette solution est utilisée dans le mode dégradé de certaines applications P2P [69].

2.5.2.2 Traitement et bande passante

Associer aux clés un pointeur sur les données, au lieu des données elles-mêmes, permet de résoudre le problème du stockage de données en utilisant une architecture P2P structurée sur un réseau hétérogène. Cela n'a par contre aucun effet sur la quantité de requêtes traitées par les nœuds. Comme tous les nœuds du réseau traitent à peu près la même quantité de requêtes, les nœuds ayant des capacités de traitement ou bande passante en dessous de la moyenne seront en surcharge par rapport aux autres nœuds du réseau.

Un autre point à prendre en compte est le maintien de ces pointeurs. Ainsi, chaque fois qu'une donnée est copiée, déplacée ou supprimée, il faut mettre à jours les pointeurs de la clé qui correspond à cette donnée, et cela entraîne des coûts de traitement et de bande passante supplémentaires.

2.5.3 Acheminement des requêtes

Dans un réseau contenant N nœuds, les architectures P2P structurées permettent de localiser une clé en $O(log(N))$ messages, alors que les architectures P2P non-structurées ont généralement des temps de localisation en $O(N)$. Cette différence est à tempérer par le fait qu'il faut dans les architectures structurées d'une part devoir connaître *a priori* la clé que l'on recherche et d'autre part par le surcoût entraîné par des recherches approximatives (par exemple dûe à une erreur lors de la saisie d'un mot clé), alors que architectures non-structurées sont très bien adaptées à ces deux cas de figure.

2.5.3.1 Connaissance *a priori* de la clé

Les réseaux P2P structurés permettent d'acheminer efficacement une clé, mais il faut déjà être capable de l'obtenir, car les clés sont générées à partir de l'information recherchée. Dans le cas où le transfert d'un fichier est déjà commencé, qu'il a été mis en pause (par exemple lorsque l'on quitte le réseau pour revenir ultérieurement), et que l'on souhaite reprendre le téléchargement, il n'y a bien sûr aucun problème, puisque l'on dispose déjà de la connaissance de la clé.

Ce n'est bien sûr pas le cas quand on veut localiser l'information pour la première fois. Pour contourner ce problème, il y a plusieurs solutions, qui ont cependant leurs limitations. La première est de générer les clés à partir du nom du fichier. Ainsi l'utilisateur qui voudra retrouver un fichier dans le réseau devra au préalable connaître son nom exact. La deuxième solution est d'utiliser un catalogue, fournissant par exemple la description des fichiers, et la clé qui leur est associée. L'inconvénient de cette solution est le caractère centralisé du catalogue, car il serait trop coûteux d'avoir une copie du catalogue sur chaque machine.

2.5.3.2 Requêtes approximatives

Dans la mesure où un utilisateur ne sait pas forcément ce qu'il recherche exactement, nous souhaitons pouvoir gérer dans le système des requêtes "approximatives" (i.e. plus générales qu'un ou plusieurs mots clés). Pour une raison de liberté au niveau de la formulation des requêtes, nous souhaitons dissocier le langage de requête de l'architecture P2P utilisée. Il s'agit en fait de considérer une notion de distance entre les données, ou entre les descripteurs de ces données. Dans les systèmes P2P structurés, les clés sont obtenues en hachant soit les données, soit les descripteurs de ces données, et ce hachage brouille complètement la distance évaluable entre deux données, ou leur descripteur.

Ainsi, deux clés "semblables" correspondront à des données complètement différentes, de même que deux données "semblables" auront des clés complètement différentes. Même si certaines architecture P2P structurées supportent efficacement les requêtes dans un intervalle, les requêtes approximatives sont très coûteuses à gérer pour les systèmes P2P structurés.

2.5.4 Maintien de la topologie

Dans les réseaux P2P utilisés pour le partage de données, les pairs ne restent, en moyenne, pas très longtemps dans le réseau (la durée d'activité est de l'ordre d'une heure). Dans les réseaux de grande taille, cela entraîne une fréquence globale de connexion et déconnexion très importante. Cela ne pose quasiment aucun problème aux réseaux P2P non structurés, l'essentiel pour un nœud étant de rester connecté à au moins un autre nœud du réseau. Si au pire le nœud devient complètement déconnecté, il peut recommencer la procédure d'entrée dans le réseau.

Cette dynamicité pose par contre des problèmes de surcharge dans les réseaux P2P structurés. Pour entretenir leur topologie, la plupart de ces systèmes demande un

nombre d'opérations de réparation de l'ordre de $O(log(N))$. C'est la quantité nécessaire dans les cas les plus favorables, quand les nœuds préviennent leurs voisins de leur départ du réseau.

Il faut ajouter des opérations pour détecter les pannes, et faire des copies des données perdues. Si la fréquence globale des connexions et départs des nœuds est trop importante, la surcharge induite par ces opérations de maintenance peut devenir rapidement trop importante pour les nœuds ayant une faible bande-passante.

2.6 Principes de conception de réseaux P2P non structurés

Suite à des mesures et expérimentations réalisées sur les architectures P2P non structurées, certaines recommandations pour concevoir des systèmes P2P non-structurés émergent [47, 74].

2.6.1 Architecture complètement distribuée

Se reposer sur un ou plusieurs serveurs dédiés entraîne des coûts en matière d'infrastructure ou de maintenance plus élevés. On peut s'affranchir de ces coûts en distribuant complètement l'architecture, et en demandant à certains nœuds d'assumer des rôles clés.

2.6.2 Utilisation des super-pairs

Comme nous l'avons mentionné dans le chapitre 1, notre approche consiste à pouvoir utiliser un maximum de ressources disponibles pour accroître les capacités de stockage. A la différence de solutions dédiées, comme par exemple des clusters constitués de machines dont les capacités de traitement, mémoire et bande-passante sont équivalentes, nous voulons exploiter nos applications sur un ensemble de machines qui ne sont pas dédiées à priori à cette tâche.

Nous nous trouvons donc dans un réseau très hétérogène du point de vue des différentes capacités de traitement, mémoire et bande-passante des machines le composant. Cette hétérogénéité, comme on pourrait le croire à première vue, n'est pas nécessairement un handicap. Au contraire, en attribuant aux différents nœuds du réseau une charge de travail correspondant plus ou moins à leurs capacités, on peut améliorer les performances globales de l'architecture.

Une architecture hiérarchique à deux niveaux, avec des nœuds ordinaires (NO) et des super-nœuds (SN) a déjà prouvé son efficacité [47, 15]. Dans une telle architecture, les SN sont responsables de plusieurs NO : ils récupèrent les méta-données des informations stockées par les NO et peuvent donc répondre aux requêtes à leur place.

Ceci permet d'une part de réduire la quantité des messages présents dans le réseau car seuls les SN échangent des messages. D'autre part, ceci permet d'accélérer le traitement des requêtes, un SN pouvant répondre d'un seul coup pour tous les NO dont il est responsable.

Les NO étant fortement dépendants de leur SN, il peut être intéressant d'ajouter de la redondance pour fiabiliser encore plus le système : un NO peut alors avoir plusieurs SN [15].

2.6.3 Maintien efficace de la topologie

Pour que la topologie puisse être maintenue de manière efficace, il faut que les SN aient des informations à jour concernant un sous ensemble des autres SN. Cela permet d'acheminer les requêtes efficacement, et de changer de voisins correctement, ainsi que d'éviter la fragmentation du réseau (un SN qui ne dispose pas d'informations à jour sur ses autre voisins SN pourrait facilement se retrouver isolé).

Il faut donc d'une part que les SN échangent des information régulièrement, mais d'autre part que cet échange ne soit pas trop fréquent pour ne pas générer trop de trafic additionnel risquant de surcharger la bande passante des SN ayant les plus petites capacités. Enfin pour améliorer le bon fonctionnement du réseau, il vaut mieux privilégier des connexions avec les SN ayant des informations à jour.

2.6.4 Changement régulier de voisinage

Le changement périodique de voisins dans la partie haute du réseau (liaison SN-SN) peut avoir plusieurs effets bénéfiques. Cela peut tout d'abord permettre d'éviter le fractionnement du réseau en îlots, dans la mesure ou un réseau dont les liaisons sont dynamiques est beaucoup plus difficile à fractionner qu'un réseau dans lequel les liaisons entre SN ne changent pas.

Un autre avantage de cette stratégie est qu'un nœud faisant une recherche à un moment donné et n'ayant pas de réponse peut refaire la même recherche à un moment ultérieur et avoir des résultats, car sa recherche n'aura pas été effectuée dans la même partie du réseau.

Il faut cependant faire attention à ce que ce changement de voisins ne soit pas trop fréquent pour ne pas engendrer un excès de trafic réseau, comme nous l'avons déjà mentionné dans la section 2.6.3.

2.6.5 Choix des voisins en fonction de leur proximité

Pour éviter de surcharger les infrastructures, il parait logique que deux nœuds voisins dans la topologie logique soient aussi rapprochés dans la topologie physique, et que la liaison entre ces deux nœuds ait une faible latence. Ceci permet d'avoir des temps de réponse plus rapides, et des débits plus élevés.

La conséquence directe de cette corrélation entre topologie physique et logique est que des machines dans une même région ou dans un même pays seront dans la même zone du réseau.

Il faut cependant contraster ce gain de performance avec la disponibilité des fichiers : privilégier uniquement les connexions en fonction de leur localité rendrait la totalité du réseau difficilement accessible depuis n'importe quel point, et il serait alors beaucoup

plus difficile de localiser des informations sur des machines dans des régions ou pays étrangers.

2.6.6 Équilibrage de charge

Les SN gérant la majorité du trafic réseau (maintenance de la topologie, propagation des requêtes ...) et représentant les points d'entrée dans le réseau pour les NO, il est important qu'ils ne soient pas surchargés. Dans un réseau comme eDonkey, ceci est réalisé en mettant une limite au nombre de connexions qu'un serveur (ici considéré comme un SN) peut gérer.

Dans des réseaux utilisant des mécanismes adaptatifs pour désigner les SN, il faut réguler de manière distribuée la charge de ces SN. Ceci peut être fait au niveau des NO, qui doivent privilégier des connexions aux SN ayant les charges les plus basses. On peux aussi réguler la charge au niveau des SN de plusieurs manières, par exemple en utilisant le principe des vases communicants : les SN comparent régulièrement leur charge avec celle de leurs voisins, ceux qui ont la charge la moins élevée récupèrent des NO de ceux qui ont une charge plus élevée.

Si la charge globale de tous les SN devient trop importante, il faut alors un mécanisme pour promouvoir des NO en SN. De la même manière, on peut imaginer un mécanisme pour rétrograder un SN en NO quand sa charge devient quasiment nulle.

Chapitre 3

Recherche d'information dans les réseaux P2P non structurés

Sommaire

Dans les réseaux P2P non structurés, les requêtes sont évaluées localement puis transférées à d'autre pairs. Cette approche permet de pouvoir dissocier le langage de requête utilisé du mode de transport des requêtes. Pour le traitement de requêtes exactes sur des mots-clés, les DHT restent les plus adaptées. Dans les architectures P2P non structurées, les propriétés topologiques du réseau ne sont pas utilisées pour la recherche d'information, ce qui a pour conséquence une latence et/ou un nombre de messages relativement importants pour localiser l'information recherchée. Par contre le placement des données est libre et les nœuds peuvent gérer eux-mêmes le contenu qu'ils hébergent.

Si un nœud reçoit une requête qu'il ne peut satisfaire, il peut la propager dans le réseau de différentes manières. L'*inondation* (*flooding*) consiste à transmettre la requête à tous les voisins du nœuds, excepté celui par lequel la requête est arrivée. Dans le cas de la marche aléatoire, la requête est transmise à un voisin choisi au hasard. Enfin la requête peut être transmise à quelques voisins, soit choisis au hasard, soit choisis en fonction d'heuristiques.

Ce chapitre et le suivant présentent trois stratégies de propagation de requêtes. Les requêtes formulées portent non seulement sur des caractéristiques décrivant le contenu des documents recherchés (par exemple des mots clés), mais aussi leur structure (balises HTML [59] ou XML [12] par exemple) dans le cas de documents semi-structurés. Ces requêtes pouvant porter simultanément sur la structure et le contenu des documents semi-structurés peuvent être formulées dans des langages dédiés comme XPath [62] ou XQuery [13].

La première approche proposée est une marche aléatoire avec heuristique permettant

d'améliorer l'acheminement des requêtes portant sur la structure des documents. Elle permet d'obtenir de bonnes performances pour l'acheminement des requêtes exactes mais les performances se dégradent d'autant plus que les requêtes sont approximatives. Les deux autres approches que nous proposons ne font aucune hypothèse sur le langage de requête utilisé, les requêtes étant évaluées localement par les nœuds. La première stratégie permet de propager les requêtes dans le réseau sans redondance. La deuxième approche est une extension de la première qui permet d'atteindre rapidement un grand nombre de nœuds tout en tirant partie des capacités hétérogènes des nœuds. Bien que ces deux approches nécessitent des topologies dédiées, celles-ci n'emploient pas de cache pour stocker les requêtes déjà vues et ainsi diminuer la redondance. La création et le maintien de ces topologies faiblement structurées sont également présentés dans ces chapitres.

3.1 Propagation de requêtes dans les réseaux P2P non structurés

3.1.1 Inondation

Ce type de routage était initialement utilisé dans Gnutella [17]. Quand une requête R_i est créée, on lui assigne une durée de vie TTL_{R_i} (Time To Live), qui est par exemple un compteur que l'on initialise avec un entier strictement positif. TTL_{R_i} correspond en fait à la portée de la requête R_i : plus il est important, plus il y aura de nœuds qui seront visités, et plus la requête aura de chances d'être satisfaite. Un grand TTL entraîne aussi des temps moyens de réponse plus importants. Quand un nœud reçoit une requête, il la traite puis la fait suivre à tous ses voisins, en décrémentant le compteur de un. Si le compteur atteint zéro, le nœud ne fait pas suivre la requête.

En utilisant un grand TTL, ce type de routage permet de retrouver un maximum d'éléments correspondants aux critères de la recherche. De plus, cette approche est rapide (on utilise en général un TTL inférieur à dix) et fiable (elle se comporte très bien dans les réseaux très dynamiques comportant beaucoup d'arrivées ou de départs de nœuds). Par contre, ce type de routage est très coûteux, notamment en terme de bande passante, dans la mesure où il génère un nombre important de messages. Les nœuds peuvent recevoir de différents voisins la même requête et, bien qu'ils oublient les requêtes déjà vues (stockée dans un cache), cette redondance augmente avec le rayon de l'exploration (le nombre de cycles potentiels dans le graphe augmente avec le rayon de l'exploration). Cette stratégie entraîne donc une utilisation plus importante de ressources réseau sans toutefois augmenter les chances de trouver des éléments correspondant au critère de la recherche [51].

3.1.1.1 Inondation adaptative

Un problème particulièrement délicat avec ce type de routage est de fixer le bon TTL : il est par exemple inutile d'inonder tout le réseau si l'élément que l'ont recherche se trouve dans le voisinage du nœud qui émet la requête. Une solution consiste à faire

une requête avec un petit TTL, puis d'attendre la réponse. Si on n'a aucune réponse, on peut alors refaire la même requête avec un TTL plus important, et ainsi de suite. Suivant la stratégie utilisée, on parle d'anneaux croissants (en anglais *expending rings*) [51] ou d'approfondissement itératif (en anglais *iterative deepening*) [73]. Cette approche solutionne le problème du choix du bon TTL et réduit de manière significative le trafic réseau pour retrouver des données très répliquées. Par contre, elle est plus coûteuse que l'inondation classique pour retrouver les données rares, dans la mesure ou les premières itérations sont souvent inefficaces.

3.1.1.2 Inondation légère

Dans [40], Song Jiang et al. ont proposé de diminuer la redondance provoquée par l'*inondation* en utilisant une topologie virtuelle en arbre. Comme la redondance dans l'inondation ne devient importante qu'à partir d'un certain rayon, l'algorithme de [40] est divisé en deux phases. Dans la première phase, une inondation classique est utilisée pour les premiers sauts. Ensuite dans la deuxième phase, les messages ne sont propagés que dans l'arbre de diffusion.

La construction et la maintenance de l'arbre de diffusion se font très simplement : chaque nœud désigne sa connexion vers son voisin de plus haute valence comme étant une branche de l'arbre de diffusion. La racine de cette arborescence est donc le nœud ayant le plus haut degré dans le réseau. Si les nœuds de haut degré ne sont pas directement connectés entre eux, et c'est très souvent le cas en pratique, on obtient alors plusieurs arborescences disjointes. Il est par contre très fréquent dans une topologie en graphe aléatoire suivant une loi de puissance que deux nœuds de haute valence soient joignables via un troisième nœud de faible valence. Pour réduire le nombre d'arbres disjoints, l'heuristique utilisée pour la construction de l'arborescence n'est donc pas la plus haute valence, mais le nombre maximum de voisins à deux sauts.

3.1.2 Marche aléatoire

Le principe du routage par marche aléatoire est de faire suivre à chaque fois la requête à un nœud choisi au hasard parmi les voisins jusqu'à ce qu'on obtienne une réponse positive ou bien jusqu'à ce que le TTL associé à la requête expire. On considère alors que la requête est acheminée par un marcheur aléatoire. Cette technique permet de réduire de manière significative le trafic dans le réseau, mais entraîne un délai en moyenne plus important pour obtenir la réponse.

Il est possible d'augmenter le nombre de marcheurs, ce qui permet de réduire le délai pour avoir une réponse correcte, sans augmenter de manière trop importante le trafic réseau. Cette technique s'appelle la k-marche aléatoire, k étant le nombre de marcheurs utilisés [51].

3.1.3 Utilisation d'heuristiques

Ce type de routage permet d'aiguiller les requêtes vers les zones du réseau où est en principe localisée l'information que l'on recherche. Ces heuristiques sont liées au

contenu de la requête, par exemple à ses mots clés ou à sa catégorie. Pour des raisons de performance (soit en terme de consommation CPU, mémoire ou bande-passante), on ne peut pas utiliser des heuristiques trop complexes, ce qui limite l'expressivité du langage de requête que l'on peut utiliser avec cette technique.

3.1.3.1 Indices de routage

Les indices de routage (RI) ont été proposés par [22]. Les nœuds catégorisent les documents qu'ils possèdent. Ils informent ensuite leurs voisins du nombre de documents qu'ils ont dans chaque catégorie. Quand un nœud doit propager une requête, il la fait suivre à un sous-ensemble de ses voisins qui possèdent beaucoup de documents appartenant à la ou aux mêmes catégories que cette requête, ou qui permettent d'atteindre d'autres nœuds ayant beaucoup de documents appartenant à la ou aux catégories de la requête.

Les nœuds peuvent agréger les informations relatives aux catégories des documents que possèdent leurs voisins (et les voisins de leurs voisins, ...) de différentes manières.

Trois types d'indices de routage sont ainsi proposés :

- **compound** : étant donné une requête, on peut savoir en allant dans chaque direction le nombre de documents pertinents. Cette approche est la plus simple mais elle ne prend pas en compte les informations de distance (notamment en terme de sauts) pour atteindre les données correspondant au critère de recherche.
- **hop-count** : en plus d'indiquer la quantité de documents pertinents suivant un certain lien, on associe la distance (en nombre de sauts) à laquelle se trouvent ces documents. Pour des raisons d'occupation mémoire, les informations maintenues ne concernent que les documents situés à une moins d'une certaine distance appelée *horizon*. Si des documents correspondant au critère de recherche se situent derrière cet horizon, on ne dispose d'aucune information pour aiguiller les requêtes dans la bonne direction.
- **Exponentially aggregated** : cette structure permet de contourner le problème d'horizon. Les nœuds possèdent des connaissances sur le contenu des autres nœuds qui sont d'autant plus approximatives que le nœud est loin. L'approximation augmente de manière exponentielle avec la distance au nœud.

3.1.3.2 Filtres de Bloom

Un filtre de Bloom [7] est une structure de données qui permet de répondre de manière approximative à des questions portant sur l'appartenance d'un objet à un ensemble. Il est constitué d'un tableau de m bits et de k fonctions de hachage $h_1, ..., h_k$, comme illustré sur la figure 3.1.

Quand le filtre est vide, tous les bits sont mis à 0. L'insertion d'un élément x dans le filtre est réalisée en mettant les positions des bits correspondants aux valeurs $\{h_i(x)_{i=1..k}\}$ à 1. Une requête est traitée en regardant si tous les bits correspondants aux positions renvoyées par les k fonctions de hachage sont mis à 1. Les faux-positifs sont possibles mais pas les faux-négatifs. La probabilité d'avoir une réponse positive

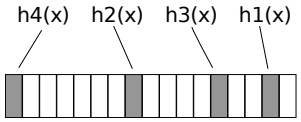


FIG. 3.1 – Ajout d'un élément x dans un filtre de Bloom de taille m=16 et k=4 fonctions de hachage. Les fonctions de hachage associent à chaque objet de l'univers considéré un nombre de 1 à m.

concernant un élément qui n'a pas été inséré dans le filtre (faux positif) est de $(1-(1-\frac{1}{m})^{n.k})^k$, n étant le nombre d'éléments insérés dans le filtre.

Différentes sortes de filtres de Bloom ont été utilisées comme heuristique de routage, notament les filtres de Bloom atténués [61] ou les filtres de Bloom à atténuation exponentielle (EDBF : Exponentially Decaying Bloom Filter) [44], qui sont présentés plus en détail à la section 3.2.1.1.

3.2 Routage de requêtes de chemin

Les données semi-structurées, de type XML, sont de plus en plus populaires. XML est utilisé pour représenter des documents textuels (par exemple de la mise en forme avec XHTML), mais aussi pour décrire des services. L'accroissement de la popularité de ce format soulève le problème du stockage et de l'indexation de ces fichiers. On peut donc envisager son exploitation dans des réseaux de grande taille, type réseaux P2P. En plus de pouvoir formuler et acheminer des requêtes concernant le contenu de documents XML, il est intéressant de pouvoir réaliser les mêmes opérations sur la structure des documents. Des langages tels que XPath [62] et XQuery [13] permettent de formuler des requêtes portant sur l'organisation arborescente des documents XML, en particulier sur des chemins ou des sous-arbres des structures arborescentes.

Ce chapitre présente un mécanisme de routage de requêtes de chemin XML adapté aux réseaux P2P non structurés. L'utilisation de filtres de Bloom permet d'aiguiller les requêtes vers les nœuds possédant l'information recherchée. Bien sûr, comme mentionné à la section 3.1.3, l'utilisation d'une heuristique impose des contraintes sur l'expressivité des requêtes. Le langage de requêtes proposé ici est un sous-ensemble du langage XPath limité aux requêtes de chemin. Les résultats présentés ici concernent une topologie en graphe aléatoire sans super-pair.

3.2.1 Filtres de Bloom multi-niveaux à atténuation exponentielle

Cette section présente une heuristique utilisant des filtres de Bloom pour aiguiller des requêtes de chemin XML dans les réseaux P2P. C'est plus précisément une combinaison de deux heuristiques qui sont rapidement présentées ci-après.

3.2.1.1 Filtres de Bloom à atténuation exponentielle (EDBF)

Ces filtres ont été proposés par [44]. L'introduction d'un élément est réalisée comme dans un filtre de Bloom classique, en mettant les positions des bits renvoyées par les k fonctions de hachage $h_i(x)$ à 1. Par contre, l'interrogation du filtre concernant un élément x ne renvoie pas *vrai* ou *faux* mais le nombre $\theta(x)$ de bits se rapportant à l'élément x mis à un. Ces filtres sont ensuite utilisés pour coder des tables de routage probabilistes, dans lesquelles $\frac{\theta(x)}{k}$ est la probabilité de trouver l'élément x en suivant un certain nœud dans le réseau (chaque nœud maintien un filtre par voisin qu'il possède). Cette probabilité décroît de façon exponentielle avec le nombre de sauts existants entre le nœud courant et le nœud possédant l'élément x.

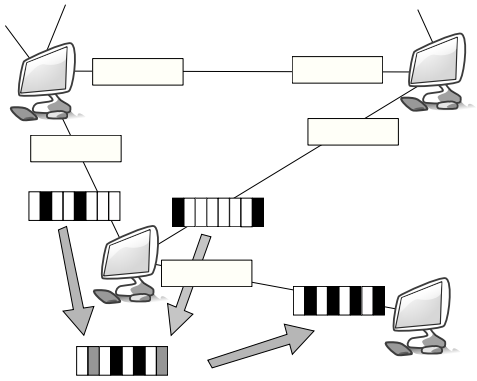


FIG. 3.2 – Mise à jour des filtres.

Les nœuds mettent à jour leurs filtres périodiquement à partir des filtres envoyés par leurs voisins. Un filtre proposé à un nœud voisin est créé à partir des informations atténuées de tous les autres voisins et de l'information locale du nœud sans atténuation, comme illustré sur la figure 3.2. L'atténuation d'un filtre est réalisée en remettant à zéro chaque bit avec une probabilité de $1/d$, d correspondant au facteur d'atténuation du filtre, la création des mises à jour des filtres étant détaillée dans l'algorithme 1. L'opération d'agrégation des informations des différents filtres est un OU bit à bit.

La propagation des requêtes est réalisée en utilisant l'algorithme SQR (Scalable Query Routing) [44]. Si la requête est propagée d'un nœud pour la première fois, elle est envoyée vers le voisin qui a le plus haut score en interrogeant l'EDBF du lien vers ce nœud. Si la requête a déjà été vue, elle est propagée vers un voisin tiré au hasard. Cette approche est synthétisée dans l'algorithme 2.

3.2.1.2 Filtres de Bloom à plusieurs niveaux (MLBF)

Ces filtres ont été introduits par [42]. L'idée principale est d'utiliser des filtres de Bloom pour décrire les propriétés structurelles d'ensembles de documents XML. Il y a deux catégories de filtres de Bloom : les filtres en largeur (BBF : Breadth Bloom Filter) et les filtres en profondeur (DBF : Depth Bloom Filter), comme illustré sur la

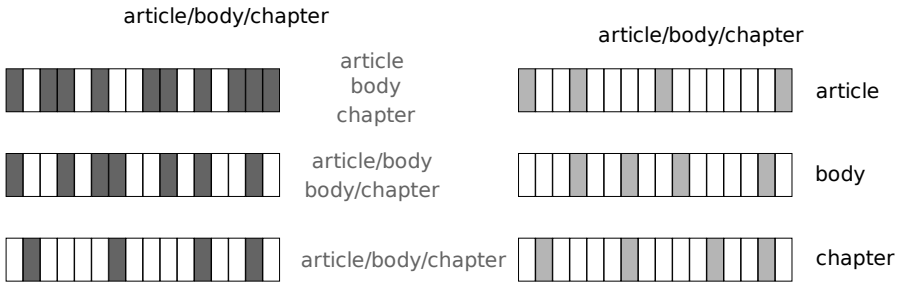
Algorithme 1 : Création des mises à jour des filtres pour SQR

```
// Création de l'EDBF A en y ajoutant tous les éléments x du nœud X
pour tous les x ∈ X faire
    pour i de 1 à k faire
        A[h_i(x)] ← 1
    fin
fin
// Création de la MAJ du filtre (pour le voisin J)
// Copie du filtre local puis ajout de l'information des autres voisins que J
   atténuée
U_j ← A
pour tous les I voisins de X, I ≠ J faire
    pour r de 1 à m faire
        si A_i[r] = 1 alors
            U_j[r] ← 1 avec une probabilité 1/d
    fin
fin
Envoie de U_j au nœud J
```

Algorithme 2 : Propagation des requêtes avec SQR

```
// Propagation pour une requête Y
si dejaVu(Y) alors
    faireSuivre(Y,voisinAleatoire())
sinon
    max ← −∞
    Node next
    pour tous les i voisin faire
        Θ ← interrogation(EDBF[i],Y)
        si Θ ≥ max alors
            max ← Θ
            next ← i
        fin
    fin
    faireSuivre(Y,next)
fin
```

figure 3.3. Un BBF est composé de i filtres de Bloom $\{BBF_1, ..., BBF_i\}$. L'ajout d'un document XML est réalisée en insérant tous les nœuds du niveau l dans BBF_l. Un DBF est composé de j filtres de Bloom $\{DBF_1, ..., DBF_j\}$. L'insertion d'un document XML est réalisé en ajoutant tous les sous-chemins du document de longueur l dans DBF_l. Selon [42], les BBF permettent un meilleur acheminement des requêtes que les DBF dans le cas général, mais les derniers permettent d'aiguiller des requêtes comportant des relations ancêtre-descendant, ce qui n'est a priori pas possible pour les BBF.



(a) DBF (b) BBF

Fig. 3.3 – Filtre de Bloom à plusieurs niveaux avec trois BBF et trois DBF. Pour cet exemple, chaque sous-filtre a une taille $m = 16$ et $k = 4$ fonctions de hachage. Cette figure illustre l'insertion du chemin "article/body/chapter" dans le BBF et le DBF.

Pour réaliser le routage des requêtes en utilisant une telle structure, les nœuds du réseau sont regroupés dans une organisation hiérarchique. Plusieurs nœuds sont désignés comme *nœuds racines* en fonction des ressources qu'ils possèdent (ce sont l'équivalent de super-pairs), et sont connectés sur un *canal principal* [42] qui leur permet de communiquer entre eux. Chaque nœud racine a un super-filtre qui contient tous les éléments des sous-pairs dont il est responsable, et un filtre local correspondant au contenu qu'il héberge. Quand un nœud racine reçoit une requête, il vérifie d'abord son filtre local puis, s'il a une réponse positive de son super-filtre, propage la requête vers ses sous-pairs.

3.2.1.3 Combinaison des deux types de filtres

Une organisation hiérarchique du réseau est bien adaptée à des nœuds ayant des capacités hétérogènes, mais quand la taille du réseau devient très importante il devient délicat de relier les super-pairs sur un *canal principal*. Par contre, un réseau de super-pairs formant un graphe aléatoire passe très bien a l'échelle, comme l'a montré le réseau FastTrack décrit à la section 2.3.4. Les filtres de Bloom à atténuation exponentielle se prêtent bien à des topologies en graphe aléatoire, mais ne permettent pas d'aiguiller des requêtes de chemin. La combinaison de ces deux types de filtres, en apportant les quelques modifications décrites ci-après, permet de pallier cette lacune et de guider les requêtes de chemin dans des topologies en graphe aléatoire.

Un filtre de Bloom multi-niveaux à atténuation exponentielle (EDMLBF) [10] est

composé d'un filtre en largeur (BBF) de taille l et d'un filtre en largeur inversé (RBBF : Reversed Breadth Bloom Filter) lui aussi de taille l. La figure 3.4 illustre un exemple de tel filtre. L'insertion de chaque chemin de longueur k, $P = /e_1/.../e_k/$ dans les filtres se fait en mettant e_i dans BBF_i et $RBBF_{k-i+1}$, $\forall i \in [1, \ldots, k]$.

Les sous-filtres composants ces filtres en largeur sont réalisés avec des filtres de Bloom à compteur (CBF) [26], de sorte que les éléments puissent facilement être retirés de ces filtres : le tableau de bits est remplacé par un tableau d'entiers, chaque entier ayant le rôle d'un compteur. L'insertion d'un élément est réalisée en incrémentant de un les entiers aux positions renvoyées par les k fonctions de hachage. Le retrait est réalisé en décrémentant de un ces entiers. La question d'appartenance d'un élément au filtre est traitée en regardant si tous les entiers aux positions renvoyées par les k fonctions de hachage sont strictement positifs.

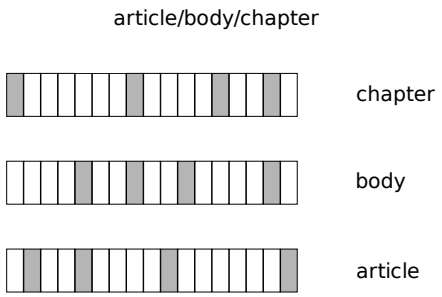


Fig. 3.4 – Exemple de filtre de Bloom en largeur inversé composé de trois sous filtres.

Nous n'utilisons pas de filtre en profondeur (DBF) pour deux raisons. La première, invoquée par [42], est une perte d'efficacité au niveau de l'aiguillage des requêtes comparativement au BBF. La seconde raison est que les DBF gèrent très mal des spécifications incomplètes de chemin. Par exemple, une requête de chemin du type /A/ ?/C/ peut être aiguillée en utilisant les informations des premiers et troisièmes niveaux des filtres en largeur, alors que seule l'information du premier niveau d'un filtre en profondeur (c'est-à-dire les chemins de longueur un) peut être utilisée.

Par ailleurs, il n'y a pas de prise en compte dans les filtres des relations entre les différents éléments qui composent les chemins. Par exemple si les chemins /A/B/ et /C/D/ sont insérés dans les filtres, alors ils répondront positivement pour les chemins /A/D/ et /C/B/, alors que ceux-ci n'ont pas été insérés dans le filtre. On peut imaginer atténuer ce problème en réalisant du clustering sur la structure des documents XML et regrouper sur un même nœud ou des nœuds voisins des documents XML étant structurellement semblables.

Ces filtres de Bloom multi-niveaux à atténuation exponentielle sont utilisés pour encoder des tables de routage probabilistes, de la même manière que ce qui est décrit dans [44]. La figure 3.5 illustre un exemple d'utilisation de tels filtres. La mise à jour de ces filtres et le routage des requêtes se font de manière similaire aux principes décrits à la section 3.2.1.1.

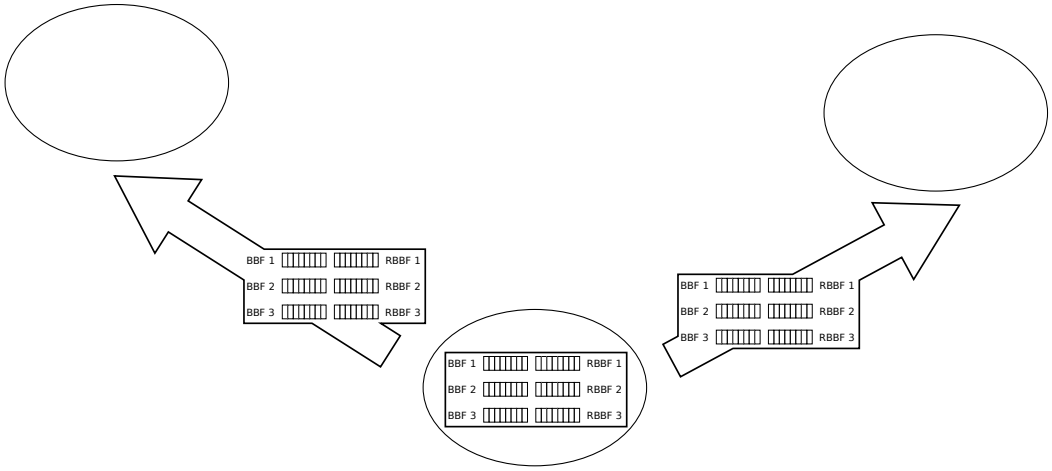


FIG. 3.5 – Exemple de configuration de filtres d'un nœud ayant deux voisins. Chaque filtre est composé de trois sous-filtres.

3.2.1.4 Interrogation des filtres

Le langage de requête utilisé est un sous-ensemble de XML Path Language (XPath) [62]. Soit E l'ensemble de tous les noms de balise XML et " ?" un élément quelconque. Soit $E^? = E \cup \{"?"\}$ et S l'ensemble des symboles de relation, $S = \{"/","//"\}$. "/" correspond à une relation parent-enfant et "//" à une relation ancêtre-descendant, "/" étant un cas particulier de "//". Une requête de chemin P de longueur k est définie comme un mot sur l'alphabet $E^? \cup S$, avec $P = s_0\ e_1\ s_1\ ...\ e_k\ s_k$, $e_i \in E^?$ et $s_i \in S$. $/e_1$ correspond à l'élément racine. Nous supposons que chaque nœud possède localement un mécanisme pour répondre correctement aux requêtes. Pour pouvoir utiliser au moins un des deux filtres en largeur, il faut connaître soit le début, soit la fin du chemin recherché, s_0 et s_k ne peuvent donc pas être "//" simultanément.

Soit P une requête de chemin contenant uniquement des relations parent-enfant, $P = /e_1/.../e_k/$. Soit $Q(F, x)$ la réponse à l'interrogation de la présence de l'élément x dans le filtre F. L'interrogation des filtres se déroule de la manière suivante :

$$Q(EDMLBF, P) = Q(BBF, P).Q(RBBF, P) \tag{3.1}$$

$$Q(BBF, P) = \prod_{i=1}^{k} Q(BBF_i, e_i) \tag{3.2}$$

$$Q(RBBF, P) = \prod_{i=1}^{k} Q(RBBF_i, e_{k-i+1}) \tag{3.3}$$

L'interrogation des filtres à décroissement exponentiel retourne un résultat dans l'intervalle $[0, 1]$ et le "." dans les trois équations correspond à un produit.

Si la requête de chemin contient des relations ancêtre-descendant, alors le sous-chemin avant le premier "//" est traité par le BBF et le sous-chemin après le dernier "//" est traité par le RBBF. S'il n'y a que des relations parent-enfant, alors l'intégralité de la requête de chemin est traitée par les deux filtres en largeur.

3.2.2 Expérimentations

Pour mesurer l'efficacité de l'utilisation de filtres de Bloom pour aiguiller des requêtes de chemin, nous avons conduit quelques expérimentations avec un simulateur que nous avons écrit en Java. A chaque nœud du réseau est attribuée une capacité qui correspond au nombre de fichiers qu'il peut héberger. Pour simuler une répartition hétérogène des capacités, nous avons utilisé une répartition cumulée des capacités décrite par la figure 3.6, et il y a en moyenne 10 fichiers XML par nœud. Le réseau est statique : nous n'avons pas pris en compte ici l'arrivée ou le départ de nœuds. Pour les deux algorithmes d'exploration (marche aléatoire et SQR), nous avons utilisé 10 marcheurs. Les paramètres que nous avons utilisés sont résumés dans la table 3.1.

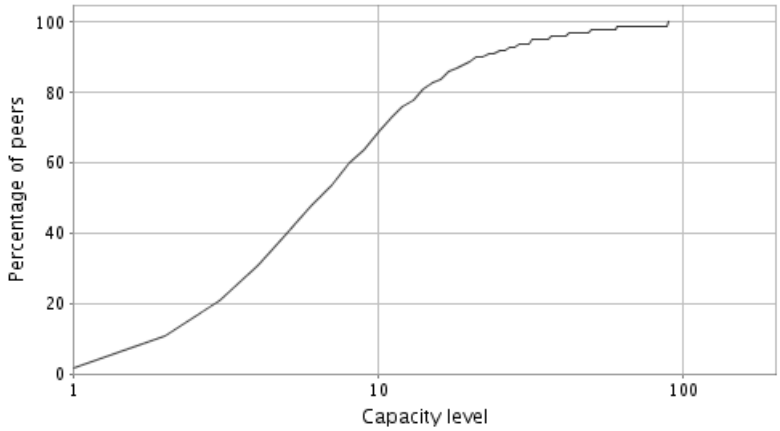


FIG. 3.6 – Répartition cumulée des capacités des différents pairs.

Nous avons utilisé une collection de fichiers XML tirée de *lastfm* [45]. Les documents possèdent une profondeur d'arbre de trois, ce qui signifie que les éléments du second niveau sont insérés dans le BBF et le RBBF. Les éléments du même niveau ont tous le même nom, l'hétérogénéité dans la structure est liée aux attributs de ces éléments. C'est pourquoi nous insérons dans les filtres la chaîne de caractères qui correspond au nom de l'élément et de ses attributs, comme par exemple la chaîne dans la balise $< author\ name =" Bob\ Dylan" >$. Nous avons utilisé la fonction de hachage proposée par [39] qui prend en paramètre une valeur d'initialisation, chaque fonction de hachage $h_t()$ étant initialisée avec la valeur t.

Nous avons choisi d'illustrer ici le gain que peut apporter cette approche pour la marche aléatoire. Pour réaliser la comparaison, 1000 requêtes de chemin de longueur 3 sont générées à chaque itération (qui correspond à un certain TTL). Pour chaque

Paramètre	Valeur
Taille du réseau	10000
Valence moyenne	6
Documents XML	100000
Taux moyen de réplication	$\frac{3}{10000}$
Nombre de sous filtres	2
Nombre de fonctions de hachage	8
Atténuation	8
Nombre de marcheurs	10
Nombre de requêtes	1000

Tab. 3.1 – Paramètres utilisés pour les expérimentations

requête un nœud du réseau est choisi aléatoirement puis 10 marcheurs de chaque type (aléatoire et SQR) sont propagés dans le réseau à partir de ce nœud. Dans le cas de l'utilisation de filtres, nous avons testé deux configurations l'une pour laquelle les sous-filtres ont une taille de 2^{13} (SQR 8k) et l'autre pour laquelle une taille double est utilisée, c'est à dire de 2^{14} (SQR 16k).

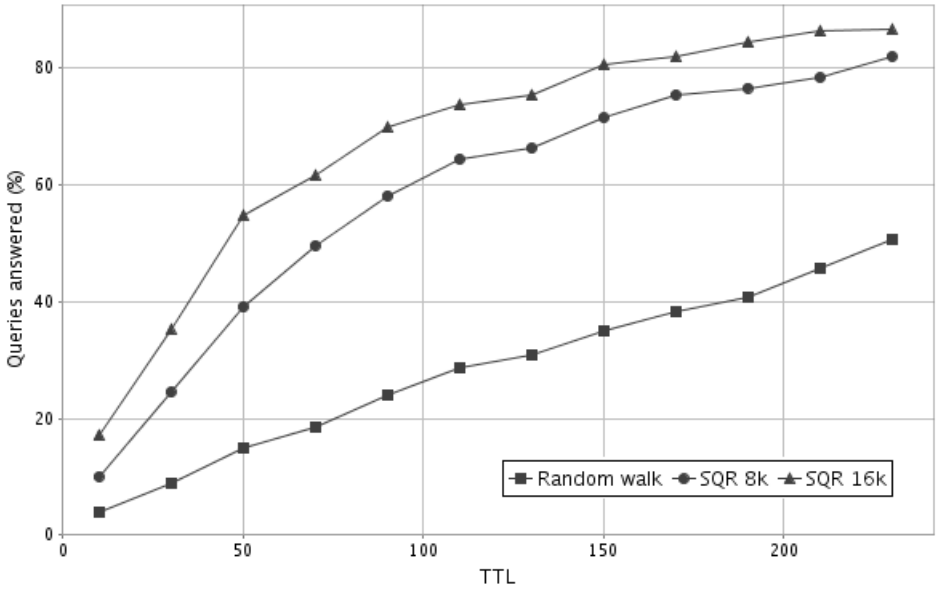


Fig. 3.7 – Pourcentage de requêtes de chemin satisfaites en fonction du TTL imposé.

La figure 3.7 illustre le nombre de requêtes satisfaites en fonction du TTL des marcheurs. On peut clairement voir le gain de performance apporté par l'utilisation des filtres de Bloom multi-niveaux à atténuation exponentielle. Le fait de doubler la taille des filtres offre aussi un gain notable à TTL faible, l'écart se réduisant avec

l'augmentation du TTL. Nous avons assigné à chaque nœud 10 documents XML en moyenne. Il faut préciser que, d'une part la taille des filtres est relativement faible (un peu moins de 64 ko pour tous les filtres d'un nœud en moyenne pour le second cas), ce qui permet d'augmenter la taille de ces filtres si l'on stocke plus de documents XML, et d'autre part que les documents XML utilisés pour cette expérimentation sont assez larges (plus d'une centaine de chemins par document).

Par ailleurs, nous n'avons utilisé que deux niveaux de sous-filtres, ce qui permet de ne traiter efficacement que les requêtes de chemins portant sur des documents de profondeur 4. On peut bien sûr utiliser cette configuration même si les documents ont une profondeur supérieure à 4, mais l'aiguillage efficace de requêtes complexes perd alors en performance.

Pour pouvoir gérer des documents XML plus profonds et en plus grande quantité, on peut imaginer utiliser quatre niveaux de sous-filtres et avoir avec des sous-filtres de taille $2^{14} octets$. Chaque EDMLBF ayant alors 8 sous filtres (4 BBF et 4 RBBF) aurait une taille de $8 \times 2^{14} octets = 128 Ko$. En considérant une valence moyenne de 7 (chaque nœud aurait donc 8 EDMLBF), les filtres occuperaient au total $8 \times 128 Ko = 1 Mo$.

Cela permettrait d'aiguiller efficacement des requêtes de chemin dans des réseaux dont les nœuds contiennent une centaine de documents XML. On peut supposer qu'un plus grand nombre de sous-filtres donnerait des performances supérieures à celles de la figure 3.7 dans le cas de requêtes comportant plus de trois éléments connus, et qu'au contraire, le gain de performance diminuerait s'il y a des éléments inconnus dans la requête. Dans le cas où un seul élément de la requête est connu, on retombe dans le cas des filtres de Bloom à atténuation exponentielle [44].

3.2.3 Synthèse

Les filtres de Bloom multi-niveaux à atténuation exponentielle permettent donc d'aiguiller efficacement des requêtes de chemin, l'efficacité de cet aiguillage baissant avec la précision des requêtes. La mise à jour des filtres se faisant périodiquement, il est envisageable de régler cette période soit de manière à avoir des filtres régulièrement à jour, soit de manière à diminuer le trafic réseau. On peut imaginer pouvoir gérer une centaine de documents XML de taille moyenne avec des filtres nécessitant un espace mémoire de l'ordre du mégaoctet. On peut aussi envisager l'utilisation des filtres de Bloom multi-niveaux à atténuation exponentielle avec un mécanisme d'inondation contrôlée selon lequel les messages ne seraient transmis qu'aux voisins ayant la plus haute probabilité de conduire à l'information recherchée. Cela permettrait de localiser les informations plus rapidement, mais génèrerait aussi d'avantage de trafic réseau.

Il y a cependant quelques inconvénients à l'utilisation de filtres de Bloom multi-niveaux à atténuation exponentielle. La première limitation étant le nombre de documents XML (ou plus précisément le nombre total de chemins) qu'un nœud peut stocker. Bien sûr on peut toujours augmenter la taille des filtres, mais cela ne change pas le fait que la quantité de documents XML qu'un nœud peut héberger reste bornée. On pourrait pallier cette limitation en faisant du clustering de données : les documents XML pourraient être déplacés sur des nœuds contenant d'autre documents XML, avec

comme critère d'optimisation le nombre minimum de bits mis à 1 dans les filtres.

La seconde limitation est l'expressivité du langage de requêtes. Nous avons mentionné précédemment que le langage que nous utilisions était un sous-ensemble du langage XPath [62]. Il est par exemple impossible d'utiliser l'approche que nous proposons pour aiguiller des requêtes de chemin dont on ne connaît pas les extrémités (le nombre d'élément au début et à la fin du chemin étant indéterminé), comme par exemple une requête *//chapter//*. On pourrait résoudre ce problème soit en introduisant des structures de routage supplémentaires (par exemple des filtres de Bloom en profondeur), soit en générant plusieurs requêtes (par exemple *//chapter/* peut être déclinée en */chapter/*, */?/chapter/*, */?/?/chapter/* ...), mais cela entraînerait dans les deux cas un surcoût (d'espace dans le premier cas et de calcul dans le second cas) non négligeable, pour un faible gain de performances.

Le fait d'utiliser cette approche avec une marche aléatoire biaisée permet d'alléger le trafic réseau. Cependant elle induit aussi des temps de réponse très longs : même avec un TTL de 200 il reste des requêtes non résolues. Enfin cette approche reste très liée au langage de requête, donc peu flexible si ce dernier évolue.

3.3 Marche en spirale

La seule solution qui permette réellement de s'affranchir du langage de requête est d'interroger tous les nœuds du réseau. Cette solution n'est clairement pas envisageable car elle serait trop coûteuse dans des réseaux de grande taille. On peut alors se restreindre à interroger un sous-ensemble de nœuds, solution d'autant plus efficace que l'information est répliquée. Comme nous l'avons mentionné au début de ce chapitre, l'inondation permet d'interroger rapidement un grand nombre de nœuds. Le principal inconvénient de cette approche est que les nœuds peuvent être sollicités plusieurs fois pour une même requête, ce qui entraîne une surcharge inutile du réseau et des nœuds.

On peut diminuer la redondance de messages en rajoutant un cache sur les nœuds pour stocker les requêtes récemment traitées. Cela leur permet de ne pas faire suivre des requêtes déjà traitées et évite ainsi que le nombre de messages générés par inondation ne croisse de manière exponentielle avec le rayon de l'inondation. On peut aussi agir sur la topologie virtuelle : en éliminant des cycles, la redondance de messages diminue. C'est l'approche utilisée par Lightflood qui construit un arbre pour la propagation des requêtes.

Plutôt que d'utiliser un cache pour stocker les requêtes déjà vues, nous proposons ici d'agir sur la topologie pour propager des messages sans redondance. La marche en spirale que nous proposons permet de réaliser l'exploration du voisinage d'un nœud de manière compacte et exhaustive. Elle est assez proche d'une marche aléatoire avec heuristique, en se différenciant de celle-ci principalement sur sa capacité à cloner certains marcheurs, parallélisant ainsi la recherche. Dans tous les cas, les nœuds ne sont visités qu'une seule fois en environnement stable.

3.3.1 Marche en spirale

Une marche aléatoire modifiée a été introduite par [29]. Elle est destinée à être utilisée dans des réseaux de capteurs, que l'on peut considérer comme un cas particulier de réseaux P2P non structurés. La modification introduite dans la marche aléatoire consiste à visiter les nœuds les plus proches du nœud source avant ceux situés plus loin. Sur un réseau de capteurs dans le plan, et sous certaines conditions, cette marche décrit a peu près une spirale. Cependant elle ne garantit pas que tous les nœud à l'intérieur d'un certain rayon du nœud source sont effectivement visités.

Dans les réseaux de capteurs, les connexions entre les pairs se font en fonction de leur proximité. Dans un réseau P2P non structuré qui repose sur la topologie d'internet, les pairs peuvent théoriquement se connecter à n'importe quel autre pair, quelle que soit sa localisation. Ceci offre la possibilité de contrôler la topologie virtuelle du réseau P2P. En utilisant l'algorithme proposé par [29] sur un maillage triangulaire régulier, avec tous les nœuds ayant une valence de six, on peut observer que cet algorithme réalise une exploration compacte du voisinage d'un nœud. En fait, on observe que cette marche fonctionne assez bien quand les nœuds ont des voisins en commun.

3.3.1.1 Principe de la marche en spirale

La marche en spirale repose sur le même principe que les courbes de remplissage [65]. L'idée est de pouvoir parcourir le voisinage d'un nœud du graphe de manière dense. La marche en spirale possède en plus l'avantage de mesurer la distance (en termes de sauts) au nœud qui a initié la requête. La marche en spirale peut être vue comme un cas de k marche aléatoire biaisée et permet donc de visiter le voisinage d'un nœud en générant une quantité de messages linéairement proportionnelle aux nombre de nœuds visités. L'exploration du réseau réalisée est une exploration en largeur : les nœuds les plus près sont visités en premier et tous les nœuds à une certaine distance du nœud source sont visités en générant un nombre fini de messages.

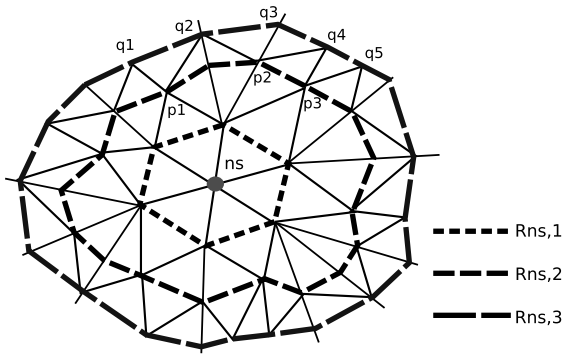


Fig. 3.8 – Les trois premiers anneaux d'une marche en spirale. Quand un marcheur a fini le parcours d'un anneau, il passe à l'anneau suivant, décrivant ainsi une spirale.

La marche en spirale repose sur une topologie virtuelle ayant la propriété suivante :

deux nœuds voisins dans le graphe ont exactement deux voisins distincts en commun. Cet algorithme fonctionne de manière optimale sur un maillage triangulaire sur une sphère. Dans un premier temps, la topologie du réseau est considérée comme stable durant une marche en spirale, c'est-à-dire qu'il n'y a pas de nœud qui rejoint ou quitte le réseau.

- Soit $\delta(n_1, n_2)$ la distance entre deux nœuds n_1 et n_2 dans le réseau. $\delta(n_1, n_2)$ est le nombre de sauts du chemin le plus court qui relie n_1 à n_2.
- Soit $R_{n,r} = \{n_i / \delta(n_i, n) = r\}$.
- Soit $N(n) = R_{n,1}$

La marche en spirale repose sur la construction d'anneaux concentriques centrés sur le nœud n_s qui a lancé la marche, et qui permet ainsi une exploration exhaustive du voisinage des nœuds, comme le fait l'inondation. Chaque anneau est construit en se servant de l'anneau précédent comme support, comme illustré sur la figure 3.8. $R_{n_s,i}$ est le i^{eme} anneau de la marche en spirale partant de n_s. $R_{n_s,0}$ est le premier anneau et contient uniquement le nœud n_s qui a initié la marche en spirale. $R_{n_s,max}$ est le dernier anneau de la marche, max étant le rayon de la marche.

Avec la propriété suivante : $\forall n \in R_{n_s,i-1}, |N(n) \cap R_{n_s,i-1}| = 2$, on peut alors décrire séquentiellement $R_{n_s,i-1}$, car deux voisins $\in R_{n_s,i-1}$ ont nécessairement un voisin commun dans $R_{n_s,i-2}$ et un autre voisin commun dans $R_{n_s,i}$. La manière dont $R_{n_s,i+1}$ est construit, en s'appuyant sur la connaissance de $R_{n_s,i-1}$ et en visitant $R_{n_s,i}$ est décrite dans l'algorithme 3.

Algorithme 3 : Construction de l'anneau suivant.

$p \leftarrow n / n \in R_{n_s,i-1}, |N(n) \cap R_{n_s,i}| > 1$
$q \leftarrow n / n \in R_{n_s,i}, n \in N(p), |N(n) \cap R_{n_s,i-1}| > 1$
$P \leftarrow R_{n_s,i-1}$ `// Ensemble des pivots à utiliser`
$R_{n_s,i+1} \leftarrow \emptyset$
tant que $P \neq \emptyset$ **faire**
  $R_{n_s,i+1} \leftarrow R_{n_s,i+1} \cup \{n / n \in N(q), n \notin R_{n_s,i-1}, n \notin R_{n_s,i}\}$
  $q \leftarrow n / n \in (R_{n_s,i} \cap N(q) \cap N(p))$ `// On explore l'anneau suivant`
  si $|N(q) \cap R_{n_s,i-1}| > 1$ **alors**
    $P \leftarrow P - p$
    $p \leftarrow n / n \in P, |N(n) \cap R_{n_s,i}| > 1$ `// On déplace le pivot`
fin
$R_{n_s,i-1} \leftarrow R_{n_s,i}$
$R_{n_s,i} \leftarrow R_{n_s,i+1}$

Chaque fois qu'un nœud dans $R_{n_s,i}$ est visité, tous ses voisins qui ne sont pas dans $R_{n_s,i-1}$ ou $R_{n_s,i}$ sont mémorisés dans $R_{n_s,i+1}$. Quand $R_{n_s,i}$ a été complètement parcouru, tout $R_{n_s,i+1}$ est connu. La marche en spirale se termine soit quand la durée de vie du marcheur atteint 0, cette durée de vie pouvant être comptée en unité de

temps ou en nombre de nœuds visités, soit quand le marcheur atteint un certain rayon (mesuré en nombre de sauts minimum nécessaires pour revenir au nœuds source n_s).

3.3.1.2 Retour à la source

Quand un marcheur a terminé son exploration, il ne revient pas au nœud source en parcourant le chemin inverse. Durant son exploration, le marcheur mémorise les anneaux successifs dans une table de hachage, en associant à chaque ensemble de nœuds visités la distance à laquelle ces nœuds se situent par rapport au nœud source n_s. En utilisant les informations dans cette table de hachage, le marcheur peut revenir au nœud source en suivant ce gradient de distance. La longueur du chemin de retour est donc égale au rayon de la marche en spirale.

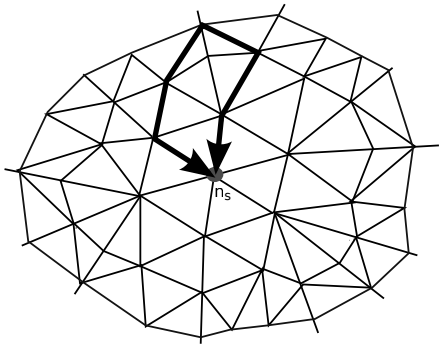


FIG. 3.9 – Différents chemin de retour possible à partir d'un nœud du 3^{eme} anneau.

Cette approche permet au marcheur de retourner au nœud source en utilisant différents chemin, comme illustré à la figure 3.9, ce qui la rend assez résistante aux perturbations (arrivées et départs des nœuds) dans la topologie. On peut aussi utiliser cette propriété pour répondre aux requêtes au fur et à mesure de la recherche à faible coût, les réponses étant renvoyées par différents chemins répartissant ainsi le trafic dans le réseau.

Si l'on souhaite réaliser la même chose avec une marche aléatoire classique, ce manque de raccourci diminue la fiabilité de l'approche et augmente le nombre de messages générés. Soit k le nombre de réponses que l'on renvoie au fur et à mesure et n la longueur de la marche pour obtenir ces k réponses. En supposant que les k réponses soient sur le dernier anneau de rayon r, on a au pire $n + k.r$ messages générés avec la marche en spirale, cette quantité étant bien sûr plus faible dans la mesure où il est fort probable d'obtenir des réponses avant le dernier anneau.

Sur un maillage triangulaire homogène où tous les nœuds ont une valence de 6, chaque anneau contient $6.r$ nœuds. On a donc $n = 1 + \sum_{i=1}^{r} 6.i = 3.r^2 + 3.r + 1$ et on peut dire que si r est suffisamment grand, $r \approx \sqrt{n}$. En pratique il y a plus de nœuds par anneau du fait des irrégularités dans la courbure de la topologie donc le rayon est même plus faible pour un n donné. En répondant aux requêtes progressivement, ont

génère donc avec la marche en spirale $\approx n + k\sqrt{n}$ messages.

Dans le cas de la marche aléatoire classique, en supposant que l'on renvoie une réponse tous les $\frac{n}{k}$ nœuds visités, on génère $n + \sum_{i=1}^{k} i.\frac{n}{k} = n\frac{k+3}{2}$ messages.

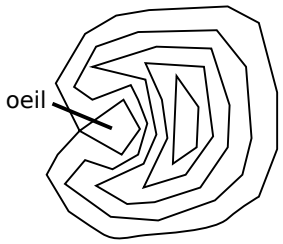


FIG. 3.10 – Apparition d'un œil pendant la marche en spirale.

3.3.1.3 Irrégularités dans la courbure de la topologie

Il peut arriver durant le parcours d'un anneau que des nœuds distants dans cet anneau soient voisins dans la topologie. Ce phénomène crée un raccourci dans la construction de l'anneau suivant et laisse une partie du voisinage inexploré que nous appelons *œil*, comme illustré sur la figure 3.10.

La probabilité d'apparition de ce phénomène augmente avec le rayon de la marche. Un œil est détecté quand un nœud dans $R_{n_s,i}$ a plus de deux voisins dans $R_{n_s,i}$. Pour résoudre ce problème, un clone du marcheur est créé et explore le sous-anneau.

Le clone d'un marcheur hérite de la mémoire du marcheur initial, il peut donc revenir au nœud n_s à l'origine de la marche en utilisant le gradient de distance. Quand un marcheur se clone pour gérer ce cas de figure, la seule différence entre les deux marcheurs est que celui qui est hors de l'oeil construit des anneaux de plus en plus grand, tandis que celui à l'intérieur de l'oeil construit des anneaux de plus en plus petits.

En utilisant cette technique de clonage, la marche en spirale peut ainsi être utilisée dans des topologies avec une courbure très déformée, où des *yeux* peuvent contenir d'autres *yeux*. Dans de telles configurations, si k *yeux* sont rencontrés, la marche en spirale peut être vue comme un arbre d'exploration avec k feuilles. Parmi celles-ci, une seule feuille correspond au dernier nœud exploré par un marcheur ayant construit des anneaux de plus en plus grands (seul un marcheur n'a jamais été piégé dans un *œil*).

L'avantage de l'arbre obtenu par rapport aux arbres que l'on peut obtenir avec des stratégies comme l'inondation est que tous les nœuds sont différents : les marcheurs ne visitent jamais les même nœuds, dans la mesure ou leur espace d'exploration est délimité par l'anneau sur lequel ils reposent. Mais contrairement à l'inondation, l'arbre obtenu est très profond ce qui entraîne des délais d'exploration relativement longs.

3.3.1.4 Cas des environnements très dynamiques

Jusqu'a présent, nous avons fait l'hypothèse que le réseau était stable durant la marche en spirale. On peut imaginer différentes solutions pour que la marche puisse se poursuivre même s'il y a des changements dans la topologie pendant que la marche s'exécute. Cependant ces solutions produisent une marche approximative, risquant une perte d'exhaustivité dans le parcours. La première solution est d'utiliser l'algorithme de [29] décrit au début de cette section. De par sa simplicité, cette marche peut être utilisée dans des environnements très dynamiques.

La seconde solution est d'utiliser l'algorithme 3 décrit précédemment. Chaque fois qu'une information correspondant aux critères de recherche est trouvée, un message est renvoyé au nœud source n_s en utilisant le gradient de distance mémorisé. Quand le marcheur détecte une anomalie pendant la construction de ses anneaux, il s'arrête. Cette technique fournit une solution approchée dans le cas où un nœud dans $R_{n_s,i-1}$, $R_{n_s,i}$ ou $R_{n_s,i+1}$ est déconnecté pendant que le marcheur parcourt $R_{n_s,i}$. Dans les autres cas cette technique fonctionne correctement, dans la mesure où l'utilisation du gradient pour revenir au nœud source n_s est une technique relativement résistante aux changements dans la topologie du réseau.

3.3.2 Topologie virtuelle

Nous faisons l'hypothèse d'unicité des identifiants des nœuds dans le réseau. Nous imposons en plus une relation d'ordre entre ces identifiants. L'algorithme de parcours présenté dans la section précédente repose grandement sur la notion de voisins communs. Toutes les connexions dans le réseau sont bi-directionnelles. Une connaissance du voisinage à deux sauts permet aux nœuds de savoir très rapidement quels voisins ils ont en commun avec tel ou tel autre voisin. Cela évite des échanges de messages supplémentaires pour déterminer les voisins communs, ce qui rend le parcours plus efficace du point de vue de la consommation de ressources.

Pour avoir une connaissance de leur voisinage à deux sauts, les nœuds dans le réseau envoient à leurs voisins des messages à intervalles de temps réguliers. Ces messages contiennent leur identifiant, ainsi que les identifiants de tous leurs autres voisins. On peut concevoir cet échange de messages comme un *ping* plus élaboré.

Lorsque des nœuds rejoignent ou quittent le réseau, la topologie virtuelle doit être maintenue. Il peut être également intéressant de modifier localement la topologie, pour modifier le comportement de l'algorithme d'exploration, et influer ainsi sur le parallélisme de celui-ci.

3.3.2.1 Invariant topologique

La topologie utilisée par la marche en spirale est un maillage triangulaire sur une sphère, la topologie minimale correspondant à un icosaèdre. Pour cette topologie, quel que soit le couple de nœuds (n_1, n_2) du réseau considéré, si n_1 et n_2 sont voisins, alors ils ont exactement deux voisins en communs. Cette propriété induit que chaque nœud

dans le réseau possède au moins 4 voisins. L'algorithme de parcours qui est présenté dans la section suivante repose en grande partie sur cette propriété.

On pourrait qualifier la topologie du réseau de *faiblement structurée*, dans la mesure où il y a bien des contraintes imposées à chaque nœud sur le choix de ses voisins mais que cela n'influe en rien sur le placement des données. Quelle que soit la taille du réseau, la quantité de messages générée lors de l'arrivée ou de la panne d'un nœud reste globalement constante, ce qui nous suggère que le maintien de la topologie n'entraîne pas une consommation de ressources trop importante.

3.3.2.2 Arrivée d'un nœud

Un nœud n souhaitant se connecter au réseau contacte un nœud n_1 pris au hasard dans le réseau. La manière d'obtenir ce nœud n_1 n'est pas décrite ici, il peut par exemple s'enregistrer auprès d'un serveur qui fournit un point d'accès au réseau. Une fois n_1 obtenu, n_2, un voisin de n_1 est choisi au hasard. La connexion entre n_1 et n_2 est coupée et des connexions entre n, n_1 et n, n_2 sont créées. Pour finir, n contacte les deux voisins n_3 et n_4 que n_1 et n_2 avaient en commun et créé une connexion avec eux, comme décrit dans la figure 3.11 et dans l'algorithme 4.

Algorithme 4 : Arrivée du nœud n dans le réseau.

$n_1 \leftarrow randomNode()$
$n_2 \leftarrow randomNeighboor(n_1)$
$\{n_3, n_4\} \leftarrow N(n_1) \cap N(n_2)$
$sendMessage(n_1, DISCONNECT, n_2)$
$sendMessage(n_2, DISCONNECT, n_1)$
$neighbors \leftarrow \{n_1, n_2, n_3, n_4\}$
pour tous les $n_n \in N(n)$ **faire**
  $sendMessage(n_n, CONNECT, n)$
  $sendMessage(n, CONNECT, n_n)$
fin

Ce protocole de connexion rajoute un nœud et trois connexions au réseau, rapprochant la valence moyenne des nœuds dans le réseau de six. Cependant, comme n a une valence de quatre et n_3, n_4 ont leur valence augmentée d'un, cela créée des déformations locales dans la courbure de la topologie du réseau. Si l'on souhaite corriger cette déformation, le nœud n peut ensuite tenter d'acquérir d'autres voisins en essayant d'aplatir localement la courbure de la topologie. Ce processus d'aplatissement de la courbure est décrit plus tard en section 3.3.2.4.

3.3.2.3 Départ d'un nœud

Quand un nœud quitte le réseau, son départ est considéré comme une panne, il n'y a donc pas de distinction entre panne ou départ d'un nœud. Cela permet de réagir plus rapidement aux variations dans le réseau. Par exemple un nœud qui n'a plus de

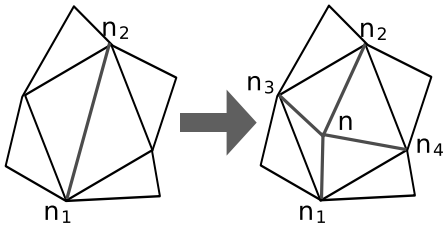


FIG. 3.11 – Connexion d'un nouveau nœud au réseau.

ressource disponible peut éventuellement quitter le réseau, sans attendre que des copies du contenu qu'il partage soient effectuées vers d'autres nœuds du réseau : l'idée étant de réaliser des copies des fichiers *a-priori*. La panne d'un nœud n est détectée par ses anciens voisins qui ne reçoivent plus de messages de ping de la part de n depuis un certain temps. Quand la panne est détectée, la topologie a besoin d'être localement réparée, pour préserver le maillage triangulaire et l'invariant décrit à la section 3.3.2.1.

Algorithme 5 : Réparation du maillage suite à la panne du nœud n.

$n_1 \leftarrow min(N(n))$
$bordure \leftarrow N(n)$
n_1 envoie un message à l'un de ses 2 voisins
Ce message est propagé le long de l'anneau $bordure$ et revient à n_1
si $\exists x \in bordure / |N(x)| = 3$ **alors**
  On cherche un nœud remplaçant dans le réseau
sinon
  tant que $|bordure| > 3$ **faire**
    $n_2 \leftarrow randomElement(2HopNeigbors(n_1) \cap bordure)$
    $sendMessage(n_1, CONNECT, n_2)$
    $sendMessage(n_2, CONNECT, n_1)$
    $bordure \leftarrow bordure \setminus \{N(n_1) \cap N(n_2)\}$
    $n_1 \leftarrow n_2$
  fin
  si *Un nœud détecte que l'invariant topologique est brisé* **alors**
    On défait toutes les connexions établies précédemment
    On cherche un nœud remplaçant dans le réseau
  fin
fin

Quand la panne d'un nœud n est détectée par ses anciens voisins, celui qui a le plus petit identifiant active un agent de réparation. Cela est possible car chaque nœud possède une connaissance du réseau à deux sauts. L'agent ainsi créé se déplace ensuite entre les anciens voisins de n et crée des connexions, comme illustré sur la figure 3.12 et dans l'algorithme 5.

Il y a deux sortes de trous dans le maillage : certains sont réparables, d'autres

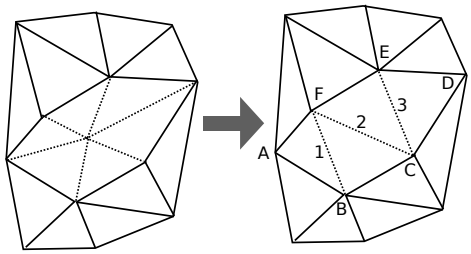


FIG. 3.12 – Réparation de la topologie après la panne d'un nœud. La bordure du trou est initialement délimitée par l'ensemble de nœuds $\{A, B, C, D, E, F\}$. Après la création de la connexion 1, la bordure du trou est délimitée par $\{B, C, D, E, F\}$, puis $\{C, D, E, F\}$. Une fois la troisième connexion créée, la bordure ne contient que 3 élément : C, D et E, il n'y a donc plus de trou dans le maillage triangulaire et la réparation est terminée.

non. Si le nœud qui vient d'avoir une panne avait un voisin qui avait une valence de quatre, alors le trou dans le maillage ne sera pas réparable, car la procédure décrite précédemment pourrait créer une configuration où l'un des nœuds aurait une valence de trois, ce qui briserait l'invariant de la topologie. Si un tel cas se présente, un message est envoyé dans le réseau pour trouver un nœud remplaçant à celui qui vient de partir. Ce message est propagé dans le réseau en suivant une marche aléatoire. Le nœud ainsi choisi devra pouvoir quitter sa zone dans le réseau en laissant un trou réparable, il doit donc avoir des voisins avec une valence minimale de cinq.

Cependant, même dans ce dernier cas, l'agent de réparation peut occasionnellement briser l'invariant topologique. Si l'agent n'arrive pas à réparer le trou, il casse toutes les connexions qu'il vient de créer et un message est envoyé dans le réseau pour trouver un nœud remplaçant pour boucher le trou, comme précisé dans l'algorithme 5.

3.3.2.4 Courbure de la topologie

D'après le théorème de Gauss-Bonnet [70] et la caractéristique d'Euler, la courbure locale de la topologie du réseau autour du nœud n est positive si n a cinq voisins ou moins, nulle si n a exactement six voisins et négative si n a sept voisins ou plus.

Les déformations locales dans la courbure de la topologie sont dûes à l'accumulation dans une même zone du réseau d'arrivées ou départs de nœuds. Les marcheurs en spirale ont tendance à se cloner dans les zones localement déformées. Une topologie déformée augmente donc le parallélisme de la marche en spirale.

Quand un nœud rejoint le réseau, il a un degré de quatre, si un de ses voisins quitte le réseau, il laisse un trou dans le maillage non réparable. Le processus d'aplatissement local de la topologie lui permet d'acquérir de nouveaux voisins, et puisqu'il réduit le nombre de nœuds ayant une valence de quatre dans le réseau, il facilite la procédure de réparation du maillage. La modification de la courbure de la topologie permet de favoriser soit la réparation du maillage, soit le parallélisme de la marche en spirale.

Pour réduire la courbure de la topologie, les nœuds vérifient périodiquement leur

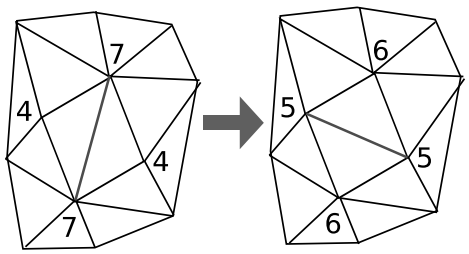


FIG. 3.13 – Aplatissement local de la topologie.

valence. Ceux qui ont une valence supérieure à six essaient alors d'aplatir localement la courbure de la topologie. Ils sélectionnent leur voisin avec la plus haute valence. Si la somme de leur valence moins deux est supérieure à la somme de la valence des deux voisins qu'ils ont en commun, ils cassent leur connexion et connectent leurs deux voisins en commun l'un à l'autre, comme décrit sur la figure 3.13 et dans l'algorithme 6. Bien sûr, ceci est réalisé uniquement si la création du lien entre les deux voisins en commun ne conduit pas à une configuration locale brisant l'invariant topologique.

Algorithme 6 : Aplatissement de la courbure de la topologie autour du nœud n.

si $|N(n)| > 6$ **alors**
  $n_1 \leftarrow x \in N(n) / \forall y \in N(n), |N(x)| \geq |N(y)|$
  $\{n_2, n_3\} \leftarrow N(n) \cap N(n_1)$
  si $|N(n)| + |N(n_1)| - 2 > |N(n_2)| + |N(n_3)|$ *ET possibilité de préserver l'invariant* **alors**
    $sendMessage(n, DISCONNECT, n_1)$
    $sendMessage(n_1, DISCONNECT, n)$
    $sendMessage(n_2, CONNECT, n_3)$
    $sendMessage(n_3, CONNECT, n_2)$
  fin
fin

De la même manière, la courbure locale de la topologie peut être accentuée. Les nœuds qui ont une valence égale à six et qui possèdent un voisin ayant une valence de six peuvent casser leur connexion avec ce voisin, et connecter leurs deux voisins communs ensemble. De manière plus générale, étant donné deux nœuds, si la somme de leur valence est inférieure à la somme de la valence des deux voisins qu'ils ont en commun, ils cassent leur connexion et connectent leurs deux voisins en commun l'un à l'autre (si ceci ne brise bien sûr pas l'invariant topologique).

3.3.3 Synthèse

La marche en spirale permet donc, sous certaines conditions, l'exploration exhaustive du voisinage d'un nœud, et sans messages redondants. Cette stratégie d'exploration peu être vue comme une marche aléatoire biaisée et permet donc de localiser l'information en générant peu de trafic réseau. En contrepartie, les délais pour localiser l'information sont comparables à ceux d'une marche aléatoire, et donc plus longs qu'en utilisant une stratégie comme l'inondation. Le mécanisme de retour de la marche en spirale est plus résistant aux pannes que celui d'une marche aléatoire classique, ce qui rend l'approche bien adaptée aux réseaux dynamiques.

Par contre, comme nous l'avons vu à la section précédente, on ne peut pas paralléliser la marche et garder une topologie réparable. La deuxième caractéristique étant essentielle au bon fonctionnement du système, cela soulève plusieurs problèmes.

Tout d'abord la topologie ne peut pas être trop déformée, ce qui empêche la parallélisation massive de l'algorithme et rend impossible l'exploration rapide d'un grand nombre de nœuds. De plus le fait d'avoir une topologie *plate* induit une valence moyenne homogène, proche de six, ce qui rend cette approche peu adaptée à la mise en profit de l'hétérogénéité naturellement présente dans les réseau P2P.

Ces problèmes pourraient être résolus en modifiant le processus de réparation du maillage. Ainsi, la topologie pourrait être très déformée, ce qui paralléliserait grandement la marche et serait plus adapté a un environnement hétérogène. La section suivante présente une évolution de cette approche, qui simplifie les processus de connexion et déconnexion des nœuds car elle requiert un invariant topologique moins contraignant à maintenir.

Chapitre 4

Exploration par arbre de remplissage

Sommaire

Cette section présente une stratégie d'exploration qui permet de parcourir un maillage triangulaire statique sans redondance de messages. Contrairement à l'exploration par marche en spirale que nous avons décrite précédemment, la topologie nécessaire a l'exploration par arbre de remplissage tire partie de l'hétérogénéité de ressources dans le réseau. Comme pour l'approche précédente, le nombre de messages générés lors de l'arrivée ou du départ d'un nœud reste globalement constant quelle que soit la taille du réseau.

4.1 Principe de fonctionnement

L'exploration par arbre de remplissage (EAR) [11] proposée est un compromis entre une exploration par marche aléatoire et une exploration par inondation. Tout comme la marche en spirale, la stratégie EAR utilise des marcheurs pouvant se cloner. Chaque marcheur mémorise la séquence des nœuds qu'il a visités, cette séquence représentant une sorte de fil d'Ariane. Elle permet de ne visiter les nœuds qu'une seule fois.

4.1.1 Exploration par arbre de remplissage

Sur un graphe planaire, si l'on interdit à un marcheur d'aller vers les nœuds déjà explorés, la séquence des nœuds visités constitue pour lui une frontière infranchissable. Quand les marcheurs atteignent cette frontière, ils se clonent, comme illustré sur la

figure 4.1. Chaque nouveau marcheur créé par clonage hérite de la séquence de nœuds visités mémorisée par son marcheur parent. L'algorithme étant initialisé avec un seul marcheur, cet ensemble de séquences de nœuds visités forme un arbre qui remplit le voisinage du nœud d'où a été lancée l'exploration.

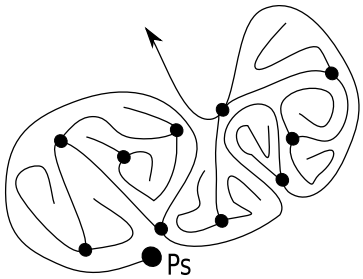


Fig. 4.1 – Fonctionnement de l'algorithme d'arbre de remplissages en commençant l'exploration à partir du nœud P_S

Étant donnée que les marcheurs se clonent, cette approche peut être vue comme une sorte de k-marche aléatoire. Mais contrairement à cette dernière, les nœuds ne seront visités qu'une seule fois. Contrairement à l'inondation (ou même à LightFlood [40]), cette stratégie ne nécessite pas l'utilisation d'une mémoire cache sur les nœuds pour mémoriser les requêtes déjà vues et pour éviter de les propager à nouveau.

Soit $N(p)$ le voisinage immédiat du pair p. Du fait du maillage triangulaire, $N(p)$ est un anneau dont on peut visiter tous les nœuds séquentiellement, en ne passant qu'une seule fois par chacun des nœuds. Quand un marcheur visite p, il analyse son voisinage immédiat et regroupe les nœuds non visités qui sont connectés entre eux. En retirant de l'anneau $N(p)$ les nœuds déjà visités, il reste entre 0 et $x = \lfloor \frac{|N(p)|}{2} \rfloor$ composantes connexes. Un clone du marcheur est envoyé dans chaque composante connexe, la figure 4.2 illustrant un exemple avec deux composantes connexes. Le comportement du marcheur est décrit dans la procédure 7. Une requête R_i de TTL TTL_{R_i} émise depuis le nœud p se fait avec l'appel suivant : $lancerMarcheur(p, \emptyset, TTL_{R_i})$. Le deuxième argument de cette procédure est l'ensemble des nœuds déja visités par les ancêtres du marcheur cloné. Cet ensemble est donc vide lors du lancement du marcheur initial.

Sur les x composantes connexes détectées, $x - 1$ composantes sont entourées d'une séquence de nœuds visités (le nœud sur lequel se situe le marcheur est considéré comme visité). Les marcheurs qui explorent ces composantes connexes sont donc emprisonnés dans des cul-de-sac et ne peuvent pas explorer le reste du réseau. Cela garantit que chaque nœud ne pourra pas être visité plus d'une fois. Le marcheur qui est lancé dans la dernière composante connexe explore le reste du réseau encore accessible.

Quand une réponse satisfaisante a été trouvée pour la requête transportée par le marcheur, celui-ci retourne vers le nœud source en utilisant le chemin inverse du chemin qu'il a mémorisé. Un marcheur a terminé son exploration quand il arrive sur un nœud dont tous les voisins ont déjà été visités (il n'y donc pas de composante connexe détectée) ou que son TTL atteint zéro.

Procedure 7 `lancerMarcheur(`$start$,$Visited_p$,TTL_p`)`

Données : H(x) Résultat donné par l'heuristique pour la propagation vers le nœud x

$current \leftarrow start$ `// nœud courant`
$Visited \leftarrow Visited_p$ `// nœuds déja visités`
$TTL \leftarrow TTL_p$ `// durée de vie du marcheur`
tant que $TTL > 0$ **faire**
  $Visited \leftarrow Visited \cup \{current\}$
  `// On détermine l'ensemble des voisins non visités`
  $NotVisited \leftarrow N(current) \setminus Visited$
  $TTL \leftarrow TTL - 1$
  si $NotVisited = \emptyset$ **alors**
    $TTL \leftarrow 0$
  sinon si $TTL > 0$ **alors**
    `// Ensemble des nœuds sur lesquels un clone va être envoyé`
    $ToSend \leftarrow \emptyset$
    tant que $NotVisited \neq \emptyset$ **faire**
      `// On détermine les composantes connexes`
      $Cluster \leftarrow \{randomElement(NotVisited)\}$
      $clusterGrow \leftarrow true$
      tant que $clusterGrow$ **faire**
        $clusterGrow \leftarrow false$
        pour tous les $p \in cluster$ **faire**
          si $N(p) \cap NotVisited \neq \emptyset$ **alors**
            $Cluster \leftarrow Cluster \cup (N(p) \cap NotVisited)$
            $clusterGrow \leftarrow true$
          fin
        fin
      fin
      `// On choisi dans chaque composante connexe un nœud qui maximise l'heuristique de propagation`
      $x \leftarrow x \in Cluster / \forall y \in Cluster, H(x) > H(y)$
      $ToSend \leftarrow ToSend \cup \{x\}$
      $NotVisited \leftarrow NotVisited \setminus Cluster$
    fin
    $current \leftarrow randomElement(ToSend)$
    $ToSend \leftarrow ToSend \setminus \{current\}$
    pour tous les $p \in ToSend$ **faire**
      lancerMarcheur(p,V,TTL)
    fin
  fin
fin

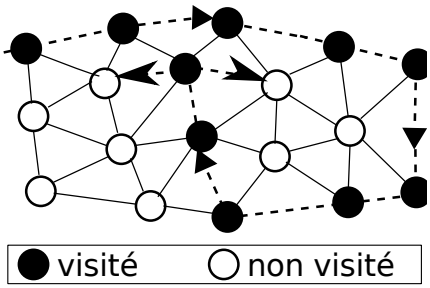


FIG. 4.2 – Exemple de scénario de clonage.

4.1.2 Heuristique pour la propagation

Comme décrit précédemment, les marcheurs analysent systématiquement le voisinage du nœud sur lequel ils se trouvent et regroupent les nœuds non visités connectés entre eux. Pour chaque regroupement, un clone du marcheur est activé pour réaliser l'exploration. Cependant, le choix du nœud de chaque groupe qui servira à l'initialisation du marcheur "clone" a un grand impact sur le parallélisme de la marche, et donc sur le nombre de nœuds visités quand le TTL est réduit.

Différentes heuristiques peuvent être utilisées pour le choix du nœud dans chaque regroupement :

Aléatoire : Le nœud est sélectionné au hasard dans la liste. Si le regroupement comporte k nœuds, chaque nœud a une probabilité d'être sélectionné égale à $\frac{1}{k}$.

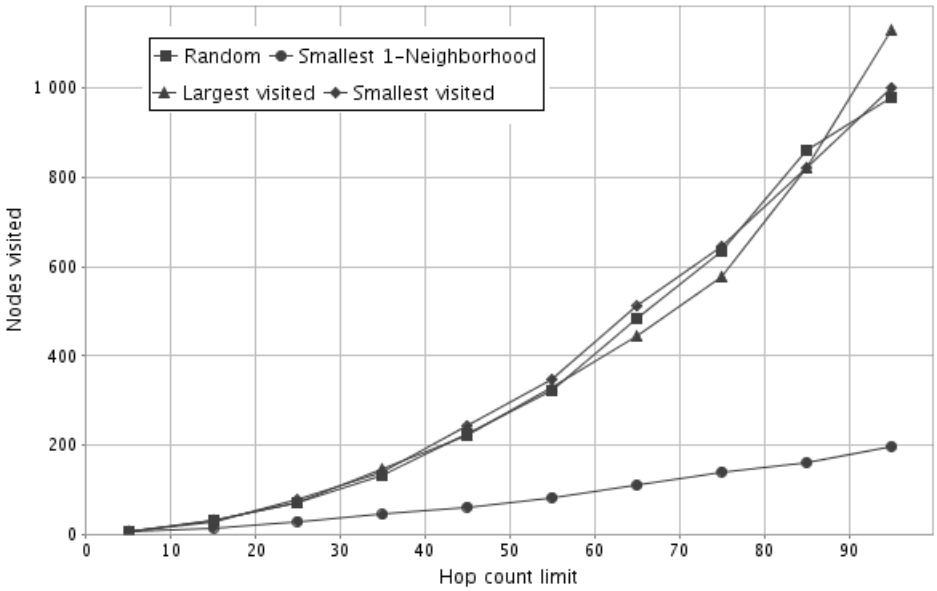
Plus grand voisinage : Le nœud sélectionné est celui ayant la plus grande valence. Si plusieurs nœuds ont la même plus grande valence, l'un d'entre eux est tiré au hasard.

Plus grand 2-voisinage : Le nœud sélectionné est celui ayant le plus de voisins à deux sauts. Si plusieurs nœuds ont un 2-voisinage de même taille, l'un d'entre eux est tiré au hasard.

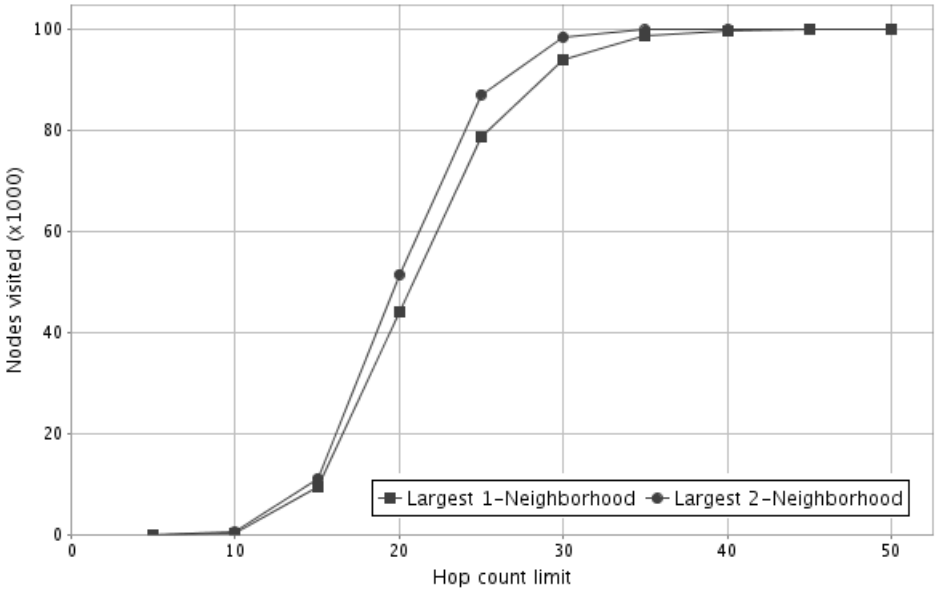
Plus petit voisinage : Le nœud sélectionné est celui ayant la plus petite valence. Si plusieurs nœuds ont la même plus petite valence, l'un d'entre eux est tiré au hasard.

Plus de voisins visités : Le nœud sélectionné est celui ayant le plus de nœuds voisins ayant déjà été visités. Si plusieurs nœuds ont la même quantité de voisins déjà visités, l'un d'entre eux est tiré au hasard.

Moins de voisins visités : Le nœud sélectionné est celui ayant le moins de nœuds voisins ayant déjà été visités. Si plusieurs nœuds ont la même quantité de voisins déjà visités, l'un d'entre eux est tiré au hasard.



(a) Aléatoire, plus petit voisinage, plus et moins de voisins visités



(b) Plus grand voisinage à un ou deux sauts

FIG. 4.3 – Impact de l'heuristique utilisée sur le nombre de nœuds explorés en fonction du TTL fixé.

Paramètre	Valeur
Taille du réseau	100000
Heuristique de connexion	Choix du triangle aléatoire
Forme initiale du réseau	Triangle

Tab. 4.1 – Paramètres utilisés pour les expérimentations

La figure 4.3 illustre la quantité de nœuds visités en fonction du TTL fixé, avec les paramètres de simulation de la table 4.1. Il y a très clairement plusieurs ordres de grandeurs entre les heuristiques utilisant le plus grand voisinage (à un ou deux sauts), et les autres heuristiques. Plus un nœud a de voisins, plus il peut y avoir de regroupement différents et donc de marcheurs clonés. L'utilisation du 2-voisinage offre des performances légèrement meilleures, car elle permet de ne visiter en premier que des nœuds de valence élevée, deux nœuds de valence élevée pouvant être liés l'un à l'autre par un troisième nœud de faible valence.

Il est intéressant de noter que les heuristiques *plus petit voisinage* et *plus de voisins visités* offrent des performances similaire à celle du choix d'un nœud sélectionné aléatoirement. L'heuristique *moins de voisins visités* offre les moins bonnes performances, ce qui paraît logique vu que celle-ci favorise moins le clonage des marcheurs.

Au vu des résultats présentés dans cette section, nous utiliserons donc comme heuristique pour la suite des expérimentations le choix du voisin ayant le plus grand voisinage à deux sauts. On peut supposer qu'il est possible d'améliorer encore les performances en utilisant comme heuristique un voisinage à trois sauts. Cependant il est significativement moins coûteux que les nœuds maintiennent une connaissance locale du réseau à deux sauts plutôt qu'à trois. D'autre part nous supposons que le gain en performances avec l'utilisation d'un voisinage à trois sauts est probablement négligeable, la différence entre l'utilisation du voisinage à un ou deux saut étant déjà relativement faible.

4.2 Topologie virtuelle

Comme pour la topologie dédiée à la marche en spirale présentée au chapitre précédent, nous faisons l'hypothèse d'unicité des identifiants des nœuds dans le réseau et nous imposons une relation d'ordre entre ces identifiants. Nous supposons aussi que tous les pairs partagent une référence temporelle. Les nœuds possèdent une connaissance locale du réseau et échangent périodiquement différents messages pour mettre à jour cette connaissance.

La topologie virtuelle est maintenue via des protocoles dédiés pour l'arrivée et le départ (ou la panne) des pairs. Ces protocoles induisent un nombre de messages proportionnel au degré du nœud, et comme dans un maillage triangulaire la moyenne des valences des nœuds est quasiment constante (ou tend vers six), le nombre de messages générés par ces protocoles reste en moyenne constant par nœud, quelle que soit la taille du réseau.

En plus de son intérêt pour l'algorithme EAR, cette topologie offre une autre propriété intéressante : elle permet d'assigner plus de connexions aux nœuds possédant beaucoup de ressources et permet donc d'adapter la charge des nœuds en fonction de leurs capacités de traitement.

4.2.1 Exploitation de l'hétérogénéité

Même si les statistiques publiées concernant les pairs des réseaux eDonkey, Gnutella ou Napster [66, 27] commencent à dater un peu, nous pensons que les écarts de capacités entre les différents pairs, en terme de mémoire, bande-passante ou CPU, restent du même ordre de grandeur. Nous considérons que chaque client possède une configuration homogène du point de vue de ces trois ressources : les pairs ayant le plus de mémoire sont aussi ceux qui ont le plus de bande-passante et de CPU. Nous assignons donc à chaque pair un niveau de capacité qui caractérise ces trois ressources. Ce niveau de capacité est notamment utilisé dans ce chapitre pour déterminer le nombre maximum de voisins qu'un nœud peut avoir.

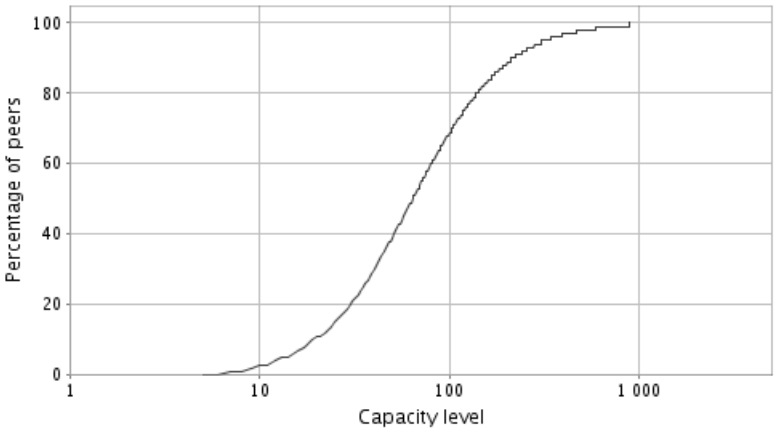


Fig. 4.4 – Répartition cumulée des capacités des différents pairs.

Suite aux mesures présentées au chapitre précédent, la répartition cumulée des capacités que nous utilisons suit une forme sigmoïdale et est présentée à la figure 4.4. La capacité moyenne obtenue est légèrement inférieure à 100.

4.2.2 Connaissance locale du réseau

Les nœuds possèdent une connaissance de leur voisinage à deux sauts et maintiennent en plus quelques informations concernant leurs voisins immédiats. La figure 4.5 illustre un exemple d'informations retenues pour un nœud ayant quatre voisins.

Voisin : identifiants des nœuds voisins.

Nœud	Âge	Remplacements	Voisins	Capacité
7331	8447	null	452, 178, 6813, 78451	15

Voisin	Âge	Remplacements	Triangles	2-voisins	Ping	Capacité
452	13487	null	178,6813	654,748,9456,45784	17587	25
178	15789	null	452,78451	9456,17845	17547	20
6813	10475	(3458,326)	452,78451	748,35987,12478,45,9874	17598	50
78451	413	null	178,6813	17845,963,6478,45	17589	10

FIG. 4.5 – Exemple de connaissance locale du nœud 7331 et tables de routage associées.

Âge : date à laquelle le nœud a occupé sa position dans le réseau. Deux nœuds qui échangent leur position échangent aussi leur âge.

Remplacements : identifiants des nœuds avec lesquels des connexions ont été échangées, ainsi que l'âge du changement de cet échange. Une fois un âge limite atteint, l'entrée est supprimée du tableau. Cette colonne est utilisée pour préserver les fils d'Ariane des marcheurs suite à des réparations ou optimisations du maillage.

Triangles : triangles formés avec les autres voisins. Par exemple le nœud 7331, 178 et 452 forment un triangle dans le maillage. Il y a toujours deux éléments dans cette colonne.

2-voisins : autres voisins du nœud voisin qui sont à deux sauts du nœud courant.

Ping : date de la réception du dernier message de ping reçu de ce voisin.

Capacité : nombre maximum de connexions que le nœud peut avoir. Il faut noter que ce n'est pas une contrainte forte et que certains nœuds peuvent de temps en temps avoir plus de voisins que leur capacité ne leur permet. Nous décrivons plus loin dans cette section la technique que nous utilisons pour réguler le nombre de voisins d'un nœud en fonction de cette capacité. Cette capacité peut être un paramètre réglé par l'utilisateur ou calculé par le système.

4.2.3 Messages utilisés

Pour maintenir la topologie dans le réseau, les nœuds échangent différents messages. Chaque message débute par un code qui permet d'identifier le type du message.

4.2.3.1 Ping

Ce type de message est envoyé périodiquement et comporte les informations suivantes :

PING	n	age_n	$remplacements_n$	V_n	$capacite_n$

Un pair qui reçoit ce message vérifie et met à jour sa table de routage pour le nœud n. La liste des voisins du nœud n est mise à jour en retirant l'identifiant du nœud qui reçoit le message, ainsi que celui des deux nœuds qui forment des triangles avec le nœud qui reçoit le message et le nœud n.

4.2.3.2 Obtention d'un triangle

Ce type de message comporte les informations suivantes :

GETOLDESTTRIANGLE	n

Les messages GETOLDESTTRIANGLE sont utilisés pour la connexion des nœuds au réseau. n correspond à l'identifiant du nœud désirant récupérer un triangle, ainsi qu'à son adresse. Nous décrivons cela plus en détail dans la section 4.2.5.

4.2.3.3 Récupération d'un triangle

Ce type de message comporte les informations suivantes :

TRIANGLE	*age*	n_1	n_2	n_3

Les messages TRIANGLE sont utilisés pour la connexion des nœuds au réseau. n_1, n_2 et n_3 correspondent aux identifiants des trois nœuds formant le triangle, ainsi qu'à leur adresse. Nous décrivons cela plus en détail dans la section 4.2.5.

4.2.3.4 Séparation d'un triangle en deux triangles

Ce type de message comporte les informations suivantes :

SPLIT	n	n_1	n_2

Un pair qui reçoit ce message créé une nouvelle entrée dans sa table de routage pour le nœud n. Son âge est la date actuelle, et n_1 et n_2 sont ajoutés dans la colonne *triangles*. Les informations des triangles pour les entrées n_1 et n_2 sont mises à jour : n remplace n_2 dans le premier cas et n_1 dans le second cas.

4.2.3.5 Fusion de deux nœuds

Cette opération permet de fusionner deux nœuds voisins. Elle est nécessaire pour certaines configuration du maillage à réparer. Ce type de message comporte les informations suivantes :

MERGE	n_1	n_2

Un pair qui reçoit ce message fusionne les contenus des entrées dans sa table de routage pour les nœuds n_1 et n_2 dans l'entrée n_1 et supprime l'entrée n_2. En considérant que les deux nœuds formant des triangles pour l'entrée n_2 sont n_1 et n_3, les informations des triangles pour l'entrée n_1 sont mises à jour : n_2 est remplacé par n_3. n_2 est rajouté dans la colonne remplacement de l'entrée n_1 avec la date actuelle.

4.2.3.6 Remplacement dans un triangle

Ce type de message comporte les informations suivantes :

REPLACE	n_1	n_2

Un pair qui reçoit ce message remplace l'entrée (si elle existe) dans sa table de routage correspondant au nœud n_1 par n_2. Il met également à jour les informations la colonne triangle. n_1 est rajouté dans la colonne remplacement avec la date actuelle. Les autres informations associées ne sont pas mises à jour : elle le seront avec la réception du prochain message de ping venant de ce pair. Enfin un nœud qui reçoit ce message renvoie deux messages TRIANGLE à n_2, les messages contenant les nouveau triangles auxquels n_2 appartient.

4.2.3.7 Départ d'un nœud

Ce type de message comporte les informations suivantes :

LEAVE	n

Un pair qui reçoit ce message quitte le réseau puis se reconnecte à nouveau, en évitant d'avoir n comme nouveau voisin. Ce message est utilisé par les pairs pour la régulation de leur charge.

4.2.4 Forme initiale du réseau

Le protocole utilisé pour l'arrivée des pairs nécessite un maillage triangulaire initial déjà existant comportant au moins trois nœuds connectés ensembles. On peut cependant imaginer des maillages triangulaires initiaux différents. Nous présentons ici l'impact du réseau initial sur les performances des arbres de remplissage. Les quatre maillages triangulaires initiaux que nous avons choisi sont les suivants : triangle, tétrahèdre, octahèdre et dodécahèdre. Les paramètres de simulation utilisés sont donnés par la table 4.2.

Paramètre	Valeur
Taille du réseau	100000
Heuristique de connexion	Choix du plus vieux triangle
Heuristique de propagation	Plus grand voisinage à 2 sauts

Tab. 4.2 – Paramètres utilisés pour les expérimentations

La figure 4.6 nous montre que la forme initiale la plus adaptée à l'exploration rapide du réseau est un triangle. Il est intéressant de noter que les performances de couverture avec une forme initiale en dodécahèdre sont bien inférieures à celles des trois autres formes.

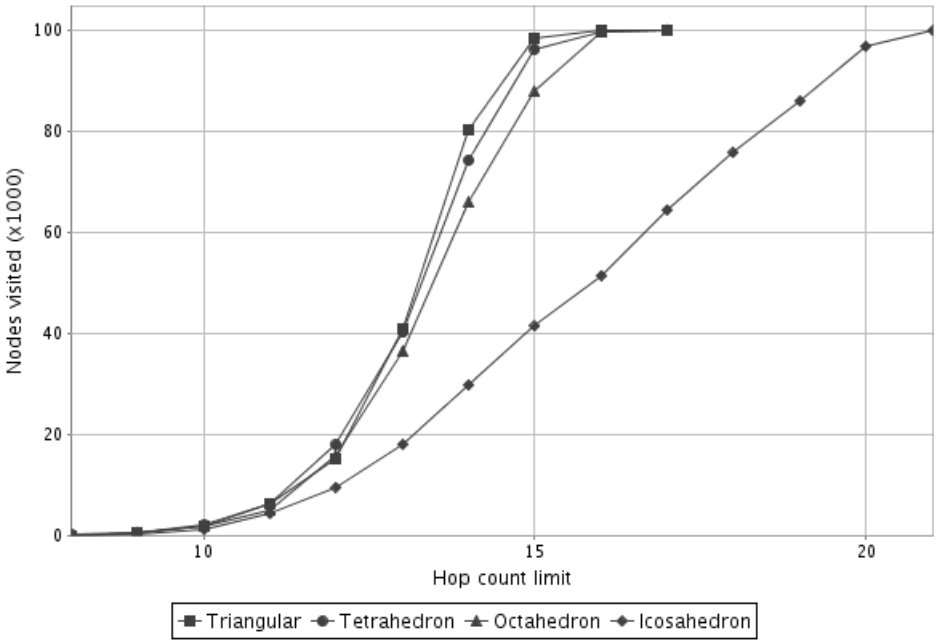


FIG. 4.6 – Impact de la forme initiale du réseau.

4.2.5 Arrivée d'un pair

Un pair p qui souhaite rejoindre le réseau obtient un triangle (p_1, p_2, p_3), nous détaillons plus loin la manière d'obtenir ce triangle. De nouvelles connexions sont créées entre $p - p_1$, $p - p_2$ et $p - p_3$, comme illustré à la figure 4.7.

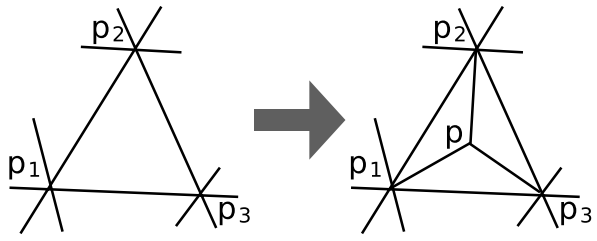


FIG. 4.7 – Arrivée d'un pair.

Les listes des triangles de p_1, p_2 et p_3 sont ensuite mises à jour. Quand un nœud se connecte ainsi au réseau, il a un degré de trois et augmente le degré de trois autres nœuds de un. Ainsi, quelle que soit le degré de connectivité moyen initial, plus la taille du réseau augmente, plus la valence moyenne des nœuds se rapproche de six.

4.2.5.1 Heuristique pour la connexion des nœuds

Nous étudions dans cette section différentes heuristiques pouvant être utilisées pour le choix des triangles lors des phases de connexion au réseau. Le choix du triangle peut impacter grandement sur la forme générale du réseau, et donc sur les performances de l'exploration par arbre de remplissage. Nous étudions dans un premier temps les heuristiques en supposant que l'on dispose d'une connaissance globale du réseau (tous les triangles sont connus), puis nous proposons dans un second temps une mise en œuvre de l'heuristique offrant les meilleures performances.

Dans la liste des triangles disponibles, ne sont conservés que les triangles dont les sommets peuvent accepter au moins une nouvelle connexion chacun. Les différentes heuristiques pour le choix d'un triangle lors de la connexion d'un nœud que nous étudions sont les suivantes :

Aléatoire : Le triangle est sélectionné au hasard dans la liste de tous les triangles du réseau. Si le réseau comporte k triangles, chaque triangle a une probabilité d'être sélectionné égale à $\frac{1}{k}$.

Plus grande valence : Le triangle est sélectionné aléatoirement parmi les triangles dont fait partie le nœud ayant la plus grande valence. Cette heuristique n'est bien sûr pas adaptée si des nœuds ont une limite très élevée de nombre de connexions, dans la mesure ou elle conduirait à une forme de réseau dégénérée.

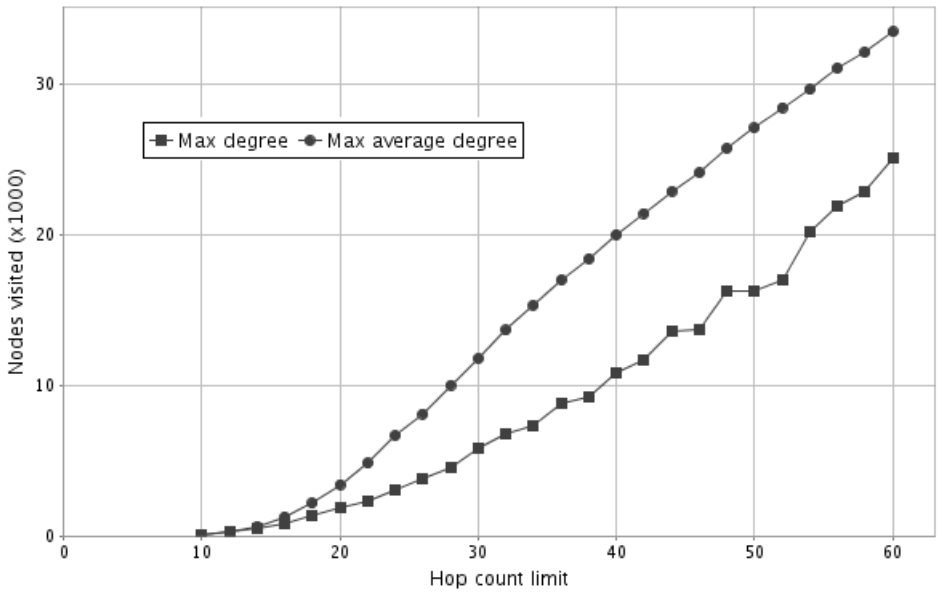
Plus grande valence moyenne : Le triangle sélectionné est celui dont la somme des valences des nœuds est la plus grande. Comme la précédente, cette heuristique n'est pas adaptée si des nœuds ont une limite très élevée de nombre de connexions.

Plus petite valence et plus petite valence moyenne : Ces heuristiques homogénéisent la valence des nœuds du réseau. Cela pose problème pour exploiter efficacement l'hétérogénéité des capacités des nœuds dans le réseau. De plus les performances de couverture d'EAR deviennent très mauvaises car ces heuristiques aboutissent à un réseau ayant un très grand diamètre, c'est pourquoi nous ne les étudions pas d'avantage.

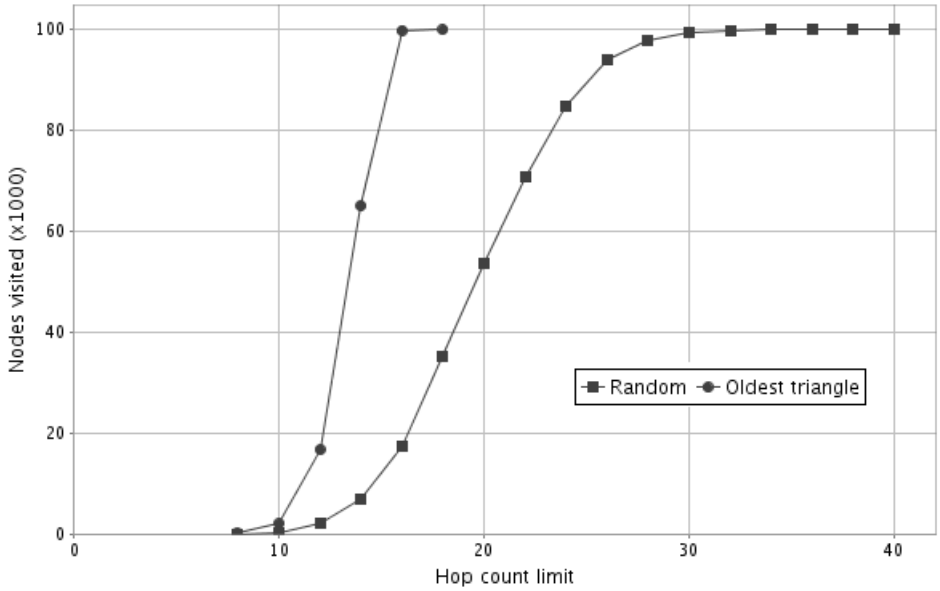
Plus vieux triangle : Le triangle sélectionné est le plus ancien dans la liste des triangles. L'âge d'un triangle est donné par le minimum des âges des trois sommets qui le composent.

Les paramètres de simulation utilisés sont donnés par la table 4.3. Comme l'illustre la figure 4.8, la maximisation de la valence des nœuds ayant les plus grandes capacités n'offre pas les meilleures performances. Préférer le triangle ayant la plus grande valence moyenne plutôt que la plus grande valence maximale permet d'équilibrer un peu mieux le réseau et double à peu près les performances.

Par ailleurs, le choix aléatoire du triangle réalise encore un meilleur équilibrage du réseau et offre des performances qui sont meilleures d'un ordre de grandeur par



(a) Plus grande valence et plus grande valence moyenne



(b) Aléatoire et plus vieux triangle

FIG. 4.8 – Impact de l'heuristique utilisée pour le choix du triangle lors de la connexion des nœuds au réseau sur la couverture par EAR.

Paramètre	Valeur
Taille du réseau	100000
Heuristique de propagation	Plus grand voisinage à 2 sauts
Forme initiale du réseau	Triangle

Tab. 4.3 – Paramètres utilisés pour les expérimentations

rapport aux deux précédentes heuristiques. La meilleure approche est celle consistant à prendre le plus vieux triangle : celle-ci améliore significativement les performances comparativement au choix aléatoire car c'est celle qui équilibre le mieux le réseau.

4.2.5.2 k plus vieux triangles

Bien que l'heuristique du choix du plus vieux triangle lors de la connexion offre les meilleures performances, elle est malheureusement impossible à mettre en œuvre pour deux raisons : la nature même des architectures P2P fait qu'il est impossible d'avoir une connaissance complète du réseau, et donc de savoir quel est le plus vieux triangle du réseau, et d'autre part les nœuds ayant des capacités limitées, le plus vieux triangle peut ne pas être utilisable comme point d'entrée, l'un des trois nœuds étant déjà saturé en terme de nombre de connexions.

Pour résoudre le problème de la connaissance limitée que l'on a du réseau pour le choix du plus vieux triangle, nous proposons la solution suivante : k nœuds pris au hasard sont contactés et chaque nœud renvoie le plus vieux triangle dont il fait partie. Le triangle sélectionné est le plus ancien parmis les k triangles renvoyés. Nous avons choisi d'appeler cette heuristique les *k plus vieux triangles*. Le cas ou k est égal au nombre de triangles dans le réseau correspond à l'heuristique du plus vieux triangle proposée au début de cette section.

Pour savoir quel triangle est le plus ancien, les nœuds utilisent les informations locales qu'ils ont mémorisées. L'âge d'un triangle est donné par le minimum des âges des trois nœuds qui le composent. En supposant que les nœuds ne se connectent jamais exactement en même temps, on peut associer à chaque nœud un plus vieux triangle. On peut aussi faire un tirage aléatoire si on sélectionne deux nœuds qui se sont connectés au réseau en même temps.

La figure 4.9 illustre l'impact du nombre de nœuds pris au hasard dans le choix du plus vieux triangle sur les performances des arbres de remplissage. La topologie initiale est cette fois un tétrahèdre, dans la mesure ou il faut au moins quatre triangles différents pour qu'un nœud se connecte au réseau dans le cas ou $k = 4$. On peut voir qu'avec $k = 4$, on obtient déjà des performances proches de l'optimal, et qu'avec $k = 1$ on obtient de meilleures performance qu'un choix du triangle aléatoire (qui couvre totalement le réseau avec un TTL de 30).

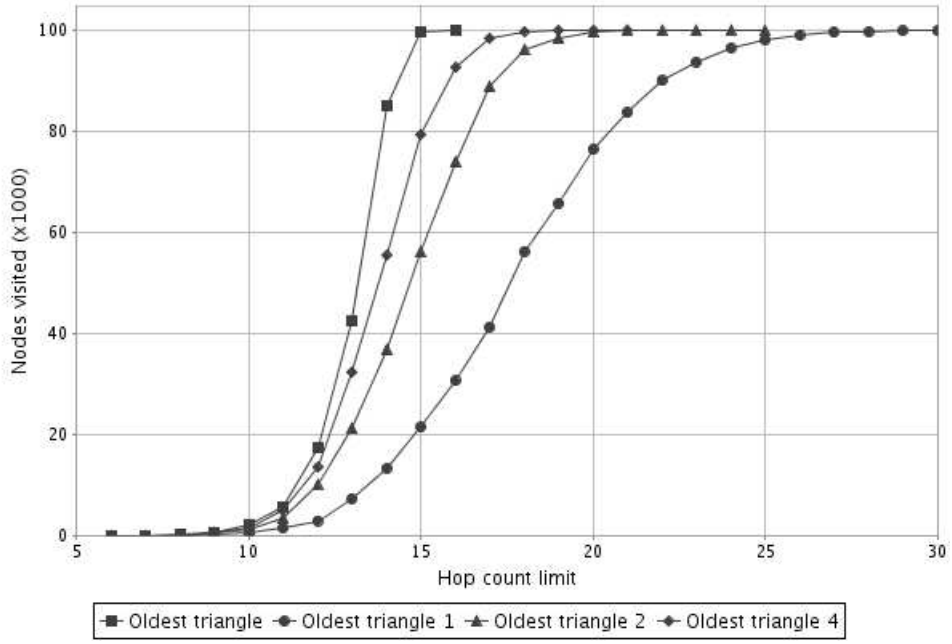


FIG. 4.9 – Impact de la connaissance limitée du réseau dans le choix du plus vieux triangle.

4.2.5.3 Protocole de connexion

Un pair p souhaitant intégrer le réseau contacte k autres pairs pris au hasard. La procédure pour obtenir les adresses de ces pairs peut être réalisée de différentes manières (utilisation d'un serveur central répertoriant quelques nœuds du réseau pris au hasard, cache gardant les adresses des voisins des précédentes sessions, nœuds dans le réseau connectés en permanence, ...) et n'est pas traitée ici. Une fois cette liste de pairs $P = \{p_i, i \in [1..k]\}$ définie, le pair se connecte au réseau en suivant le protocole décrit dans l'algorithme 8

Certains pairs peuvent ne plus pouvoir accepter de nouvelles connexions, soit parce qu'ils sont saturés, soit parce que tous les triangles auxquels ils appartiennent possèdent au moins un nœud saturé. Dans ce cas un autre pair est contacté, cette procédure se répétant jusqu'à avoir le choix parmi k triangles pour la connexion.

4.2.6 Départ ou panne d'un pair

Nous considérons le départ d'un pair comme une panne de celui-ci et nous utiliserons dorénavant ce terme pour désigner les deux cas de figure. La panne d'un nœud p_f est détectée par ses voisins quand ils n'ont plus reçu de message de type *ping* issu de p_f depuis un certain temps. Quand la panne d'un nœud est détectée, il se peut qu'il faille réparer la topologie virtuelle pour maintenir le maillage triangulaire. En fait, la topologie n'a pas besoin d'être réparée si p_f a trois voisins, car sa panne peut être

Algorithme 8 : Arrivée du nœud p dans le réseau.

pour i *de* 1 *à* k **faire**
  $sendMessage(p_i, GETOLDESTTRIANGLE, p)$
fin
// On attend de récupérer toutes les réponses
// Chaque pair p_i renvoie le plus vieux triangle t_i qui lui est associé avec un message TRIANGLE
$t \leftarrow t_x / \forall i \in [1..k], t_x.age \geq t_i.age$
$sendMessage(t.n_1, SPLIT, t.n_2, t.n_3)$
$sendMessage(t.n_2, SPLIT, t.n_1, t.n_3)$
$sendMessage(t.n_3, SPLIT, t.n_1, t.n_2)$
// Mise à jour des tables de routage du nœud p

vue comme l'inverse de son arrivée dans le réseau, c'est-à-dire en lisant la figure 4.7 de droite à gauche. Le processus de réparation présenté ici conserve la forme générale de la topologie.

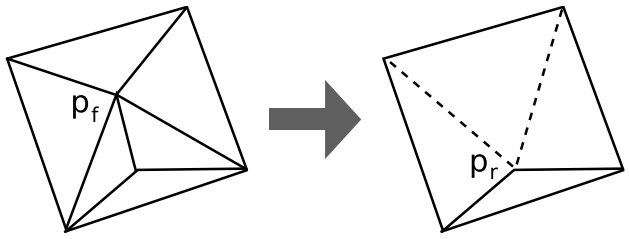


FIG. 4.10 – Réparation de la topologie après la panne du nœud p_f. Les deux voisins de p_r ne peuvent pas être candidats pour prendre en main la réparation du trou car ils ont trois voisins dans $N(p_f)$.

Définition : Un *trou polygonal* dans le maillage est une face qui contient strictement plus de 3 sommets (nœuds). Un trou polygonal est délimité par l'ensemble des sommets que cette face contient. La taille d'un trou polygonal est le nombre de connexions à créer pour retrianguler complètement la face correspondant à ce trou.

Propriété : Soit $s(T)$ la taille d'un trou polygonal délimité par un ensemble T de sommets. Alors $s(T) = |T| - 3$. On peut aisément démontrer cette propriété par récurrence. Une fois que l'on a connecté deux sommets opposés d'un quadrilatère, la retriangulation est terminée, un trou polygonal contenant 4 sommets a donc une taille de 1. Soit un trou polygonal délimité par t sommets (donc de taille $t - 3$). Alors si on connecte 2 sommets non voisins, on obtients deux trous polygonaux délimités par u et v sommets, tel que $u + v = t + 2$ (2 sommets délimitent els contours des 2 trous). La somme s de la taille de ces trou est données par $s = (u - 3) + (v - 3) = t - 4$, soit une

unité de moins que la taille du trou avant la création de la connexion.

La panne d'un pair p_f avec une valence strictement supérieure à trois créé un trou polygonal dans le maillage. Ce trou polygonal est délimité par $N(p_f)$ et sa taille est $s = |N(p_f)| - 3$. Ce trou peut être réparé en créant s connexions. Le processus de réparation est pris en charge par un nœud p_r qui se connecte à tous les nœuds de $N(p_f)$, à l'exception de ses voisins et de lui même, comme illustré sur la figure 4.10. L'âge de p_r est ensuite mis à jour : c'est le maximum des âges de p_r et p_f. p_r doit posséder deux propriétés :

1. Comme il créé $|N(p_f) \setminus N(p_r)| - 1$ connexions, p_r ne doit avoir que deux voisins dans $N(p_f)$ pour pouvoir créer s connexions.
2. Ses deux voisins dans $N(p_f)$ doivent avoir un degré minimum de trois, de sorte qu'il n'y ait pas de nœud avec un degré de deux après le processus de réparation.

4.2.6.1 Caractérisation de p_r

Si on casse la deuxième propriété, il y a alors deux cas de figure. Le premier est le cas où tous les nœuds dans $N(p_f)$ ont un degré de deux, ils sont donc tous déconnectés du reste du réseau (sauf bien sûr dans le cas d'un réseau de trois nœuds disposés en triangle). Dans le cas contraire, la première propriété est également invalidée : du fait du maillage triangulaire, un nœud avec un degré de deux aura ses deux voisins dans $N(p_f)$ mutuellement connectés. En considérant un réseau sans îlot, la première propriété est donc suffisante et le le nœud p_r choisi pour prendre en charge la réparation doit donc satisfaire le critère $|N(p_f) \cap N(p_r)| = 2$.

4.2.6.2 Existence de p_r

Soit p_c un nœud qui possède une connexion avec un autre nœud n'appartenant pas à $N(p_f)$. A cause du maillage triangulaire, les deux voisins de p_c dans $N(p_f)$ sont connectés à cet autre nœud et il existe donc une séquence de trois nœuds dans $N(p_f)$ qui ont au moins une valence de trois. La planarité du graphe fait que p_c ne peut pas être connecté à d'autres nœuds de $N(p_f)$ et satisfait donc la contrainte $|N(p_f) \cap N(p_c)| = 2$.

4.2.6.3 Choix de p_r

Pour sélectionner p_r, nous utilisons la relation d'ordre entre les identifiants. Le nœud dans $N(p_f)$ ayant le plus petit identifiant détermine s'il peut assurer la réparation du maillage. La connaissance de son voisinage à deux sauts qu'il a obtenu précédemment de p_f lui permet de savoir qu'il possède le plus petit identifiant. Si ce pair ne peut pas assurer la réparation, il lance alors un jeton sur l'anneau $N(p_f)$. Chaque nœud sur cet anneau examine s'il peut assurer la réparation, et si ce n'est pas possible, fait suivre le jeton sur l'anneau.

4.2.6.4 Complément pour la réparation

Lorsqu'un nœud du réseau tombe en panne, le fil d'Ariane de tous les marcheurs qui sont passés par ce nœud est brisé. Cependant, tant que le trou n'est pas réparé, cela n'est pas gênant. Un nœud qui assure la réparation du maillage en créant des connexions avec d'autres nœuds peut ouvrir des brèches dans les fils d'Ariane de certains marcheurs. Ces brèches peuvent alors provoquer des clonages inutiles et donc de la redondance au niveau des messages, chose que nous souhaitons éviter.

Pour palier ce problème, un nœud qui a assuré la réparation diffuse un message REPLACE à tous ses voisins, contenant l'identifiant du nœud tombé en panne qu'il a remplacé, ainsi que le sien. Ses voisins mettent alors à jour leur table de routage dans la colonne correspondante, comme décrit dans la section 4.2.3. Les nœuds remplacés sont supprimés des tables de routage au bout d'une certaine durée que l'on peut définir expérimentalement en fonction de la durée de vie moyenne des marcheurs par exemple. Quand les marcheurs doivent analyser le voisinage du nœud sur lequel ils se trouvent, ils prennent aussi en compte les informations de remplacement pour déterminer les composantes connexes et ainsi conserver un fil d'Ariane pertinent.

4.2.6.5 Surcharge d'un nœud

Le processus de réparation peut conduire à une surcharge du nœud qui assure la réparation du maillage. C'est notamment le cas quand un nœud de haute valence tombe en panne. Un nœud en surcharge est allégé progressivement. Nous décrivons cette opération dans la section suivante.

4.2.6.6 Protocole de réparation

Les algorithmes 9 et 10 décrivent le protocole de réparation. Le premier correspond au cas où p_f avait une valence de trois, il n'y a donc pas besoin de créer de nouvelle connexion pour réparer le maillage. Si ce n'est pas le cas, alors il faut dans un premier temps déterminer p_r, ce qui est toujours décrit dans le même algorithme. Le second algorithme correspond à la procédure de réparation lancée par p_r une fois que ce dernier a été déterminé.

4.2.6.7 Nœuds de valence 3

Comme illustré sur la figure 4.11, la construction du réseau se fait de manière récursive : l'insertion d'un nœud dans un triangle forme trois nouveaux triangles (l'ancien disparaissant), et permet ainsi l'insertion de trois nouveaux nœuds.

Définition : La construction du réseau permet de définir des *niveaux de récursivité*. Tous les nœuds appartenant à la forme initiale du réseau se situent au niveau 0. Tous les nœuds ajoutés dans les triangles de niveau 0 appartiennent au niveau 1 et forment des triangles de niveau 1, etc.

Algorithme 9 : Détermination de p_r suite au départ du nœud p_f.

```
pour tous les p ∈ N(p_f) faire
    si |N(p_f)| = 3 alors
        {p, p_1, p_2} ← N(p)
        n.merge(p_f, p_1, p_2)
    sinon
        si p a le plus petit identifiant dans son 2-voisinage alors
            si |N(p_f) ∩ N(p)| = 2 alors
                reparer()
            sinon
                envoyer un jeton vers l'un des deux voisins qui formait un triangle
                avec p_f
            fin
        sinon
            répéter
                attendre
            jusqu'à jeton reçu OU autre nœud répare le maillage
            si |N(p_f) ∩ N(p)| = 2 alors
                reparer()
            sinon
                envoyer un jeton vers l'autre voisin qui formait un triangle avec p_f
            fin
        fin
    fin
fin
```

Algorithme 10 : Réparation assurée par p_r suite au départ du nœud p_f.

```
pour tous les p ∈ {N(p_f) ∩ N(p_r)} faire
    sendMessage(p, MERGE, p_r, p_f)
fin
pour tous les p ∈ N(p_f) \ {N(p_f) ∩ N(p_r)} faire
    sendMessage(p, REPLACE, p_f, p_r)
    // Réception des messages TRIANGLE
    // Mise à jour des tables de routage du nœud p_r
fin
```

Définition : Un *réseau parfaitement équilibré* est un réseau dans lequel tous les triangles d'un niveau sont remplis avant de commencer à remplir les triangles du niveau suivant.

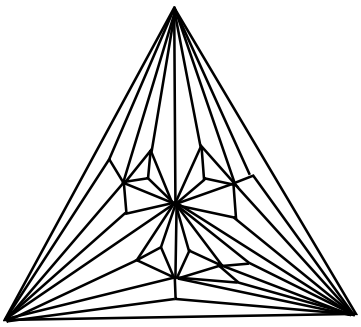


FIG. 4.11 – Forme générale du réseau, en partant d'une forme du réseau initiale en triangle.

Soit k un niveau de récursivité dans le maillage et n le dernier niveau de récursivité. Sachant que le premier niveau de récursivité nécessite déjà un triangle, le nombre de nœuds dans un réseau parfaitement équilibré en partant d'un triangle est $3 + \sum_{k=0}^{n} 3^k$. Cela correspond à une série géométrique de raison 3 dont le n-ième terme vaut $3 + \frac{3^{n+1}-1}{3-1} = \frac{3^{n+1}+5}{2}$. Seuls les nœuds du dernier niveau de récursivité ont une valence de trois, ce qui correspond à 3^n nœuds. Ainsi, la quantité de nœuds dont le départ ne nécessite pas de réparation pour conserver la triangulation du maillage, dans un réseau parfaitement équilibré, est de $\frac{2\times 3^n}{3^{n+1}+5}$. Cette quantité tend vers $\frac{2}{3}$ quand n tend vers l'infini.

On a donc théoriquement, en utilisant l'heuristique du plus vieux triangle pour la connexion, deux tiers des nœuds qui ont une valence de trois, ce qui signifie que leur panne ne nécessite pas de réparation du maillage. En pratique, sur un réseau de 100000 nœuds, il y a entre 60000 et 64000 nœuds qui ont une valence de trois en utilisant les 4 plus vieux triangles. On peut donc avoir un réseau qui se répare presque aussi bien (au sens du nombre de départs ne nécessitant pas de réparation) qu'un maillage parfaitement équilibré. Cette heuristique de connexion, en plus d'offrir les meilleures performances pour les arbres de remplissage, en terme de couverture réseau par rapport au TTL, offre aussi les meilleures performances en terme de réparabilité du réseau.

4.2.7 Optimisation de la valence des pairs

Comme nous l'avons mentionné dans le chapitre précédent, les réseaux P2P sont caractérisés par une hétérogénéité de capacité au niveau des pairs qui les composent. Certains pairs ont plus de bande passante ou de capacité de calcul et peuvent donc traiter plus de requêtes que les autres. C'est pourquoi les pairs devraient avoir un nombre de voisins proportionnel à leurs capacités. Cette adéquation entre les capacités

des pairs et leur connectivité est réalisée en faisant des permutations de voisinage entre les pairs.

Algorithme 11 : Échange de voisinage entre p_1 et p_2.

```
// échange de voisinage entre p1 et p2
pour tous les p ∈ N(p1) \ {p2} faire
  | sendMessage(p, REPLACE, p1, p2)
fin
pour tous les p ∈ N(p2) \ {p1} faire
  | sendMessage(p, REPLACE, p2, p1)
fin
// Réception des messages TRIANGLE par p1 et p2
// Mise à jour des tables de routage de p1 et p2
```

Si un pair p_1 a plus de ressources et une valence inférieure à un de ses voisins p_2, il peut se déconnecter de tous ses voisins sauf p_2, et se connecter à tous les voisins de p_2, excepté lui-même. p_2 réalise la même opération et échange ainsi ses voisins avec ceux de p_1, comme décrit dans l'algorithme 11. Comme la réparation du maillage, ce processus d'optimisation permet au réseau de conserver la même forme générale.

Cette optimisation se fait sur un seul critère (la quantité de ressources), mais étant donné la taille du réseau, il est peu probable d'arriver à une configuration optimale. Par contre, comme nous n'utilisons pas d'heuristique pour sortir des maxima locaux, cette approche converge assez rapidement. Pour étudier cet aspect, nous simulons un réseau ayant une forme initiale en triangle, et nous ajoutons des nœuds progressivement jusqu'à avoir un réseau de 100000 nœuds. Périodiquement les nœuds essayent d'ajuster leur valence en fonction de leurs capacités : tous les 100 nœuds ajoutés, chaque nœud compare sa capacité et sa valence avec un de ses voisins choisi au hasard et le processus d'optimisation est réalisé comme décrit à la section 4.2.7.

La figure 4.12 illustre la valence des pairs obtenue en fonction de leur capacités. Très peu de nœuds de haute capacité ont une faible valence. Par contre, beaucoup de nœuds ayant des capacité au-dessus de la moyenne (qui est de 100 à peu près) ont une faible valence : ces nœuds sont des candidats idéaux pour prendre en main la réparation du maillage suite à la panne de nœuds de haute valence. On peut voir aussi qu'il peut être intéressant de ne pas optimiser le maillage trop régulièrement pour justement maintenir une quantité acceptable de ce type de nœuds pour la réparation.

4.2.7.1 Complément pour l'optimisation

Comme la réparation d'une panne, cette opération d'optimisation pose un problème aux arbres de remplissage : elle peut casser le fil d'Ariane de certains marcheurs. Ce problème est résolu de la même manière que pour la réparation : les nœuds envoient un message REPLACE à leur voisins les informant qu'ils ont effectué un remplacement (le message contient bien sûr l'identifiant du nœud remplacé).

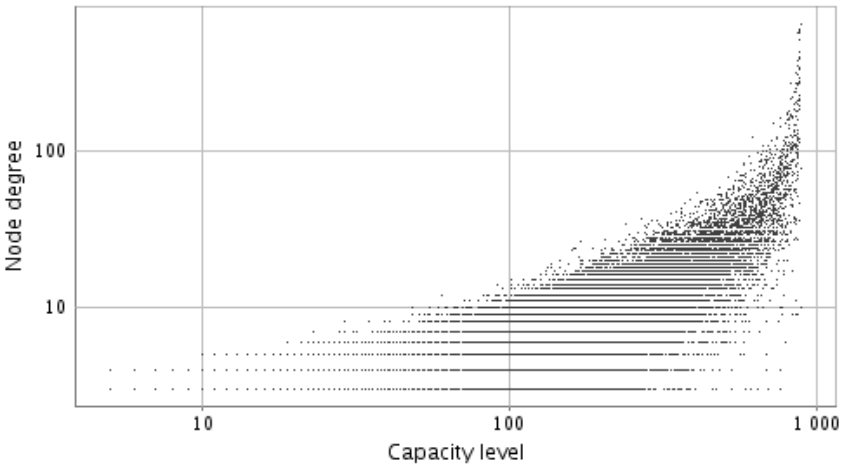


FIG. 4.12 – Valence des pairs en fonction de leurs capacitées.

4.2.7.2 Surcharge d'un nœud

Comme nous l'avons mentionné précédemment, certains nœuds peuvent se retrouver en surcharge à la suite de la réparation d'un trou dans le maillage, et ne pas avoir de voisin capable de les remplacer. On peut réguler la charge des nœuds en utilisant le processus décrit à la section 3.3.2.4. Cependant cette approche altère la forme générale de la topologie et réduit considérablement les performances des arbres de remplissage. Pour maintenir la forme du réseau intacte, un nœud en surcharge perd progressivement des voisins. Plus précisément, les voisins qu'il perd sont des nœuds ayant une valence de trois qui quittent le réseau puis le rejoignent à nouveau, comme décrit dans l'algorithme 12.

Algorithme 12 : Allègement progressif de la charge du nœud p.

répéter
  prendre un voisin x au hasard $\nearrow |N(x)| = 3$
  $sendMessage(x, LEAVE, p)$
jusqu'à *p n'est pas surcharge* $OU\ \forall n \in N(p), |N(n)| > 3$

On peut imaginer des cas de figure où un nœud en surcharge n'ait pas de voisin ayant une valence de trois. Ceci n'est en pratique pas gênant pour deux raisons : presque deux tiers des nœuds ont une valence de trois, et donc la probabilité qu'un nœud en surcharge (donc ayant beaucoup de voisins) n'ait pas de voisins de valence trois est très faible. D'autre part il est intéressant d'optimiser localement la topologie en utilisant l'algorithme 11 juste après une réparation, de sorte qu'il y ait idéalement très peu de nœuds en surcharge ou que cette dernière reste minime.

4.3 Études de différents aspects des arbres de remplissage

Nous étudions dans cette section d'autre aspects des arbres de remplissage. Comme pour les expérimentations précédentes, nous simulons des réseaux de 100000 nœuds. La figure 4.13 illustre un exemple d'arbre de remplissage déroulé, obtenu lors de l'exploration d'une partie du réseau.

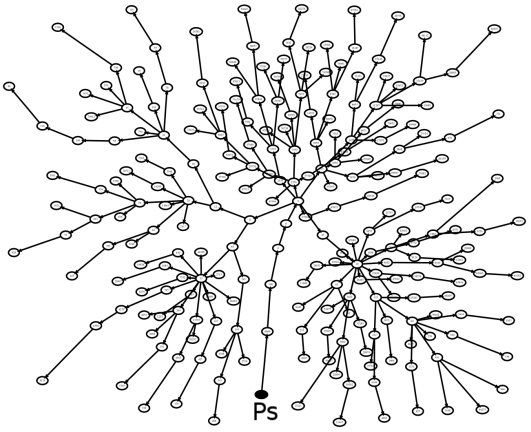


FIG. 4.13 – Arbre de remplissage résultant d'une exploration avec un TTL de 12, en partant du nœud P_s. La position des nœuds dans l'arbre déroulé ne reflète pas leur position dans le maillage.

4.3.1 Limitation des capacités et connectivité des nœuds

La figure 4.14 illustre l'impact du fait que les nœuds ont, ou n'ont pas, des capacités limitées, en considérant que l'on a accès à tous les triangles du réseau. Même si certains nœuds ont une connectivité réduite, il est possible, grâce au processus d'optimisation, de conserver un réseau ayant une forme générale équilibrée qui garantit des performances de couverture presque aussi bonnes que sur un maillage parfaitement équilibré.

4.3.2 Conservation de la forme générale du réseau en situation dynamique

Nous considérons l'expérience suivante : à chaque itération, 10% des nœuds sont retirés, puis la même quantité est ajoutée. L'heuristique de connexion utilisée est celle des 4 plus vieux triangles. L'optimisation du maillage est par contre réalisée moins fréquemment que dans les expérimentations précédentes : elle n'est réalisée que tous les 1000 nœuds ajoutés ou enlevés, c'est-à-dire dix fois moins fréquemment que précédemment. Le TTL des marcheurs est fixé à 14. Quand un trou dans le maillage a été réparé, une optimisation locale est réalisée, de sorte que le nœud qui au final assure la réparation est celui qui a la plus grosse différence entre sa capacité et sa valence.

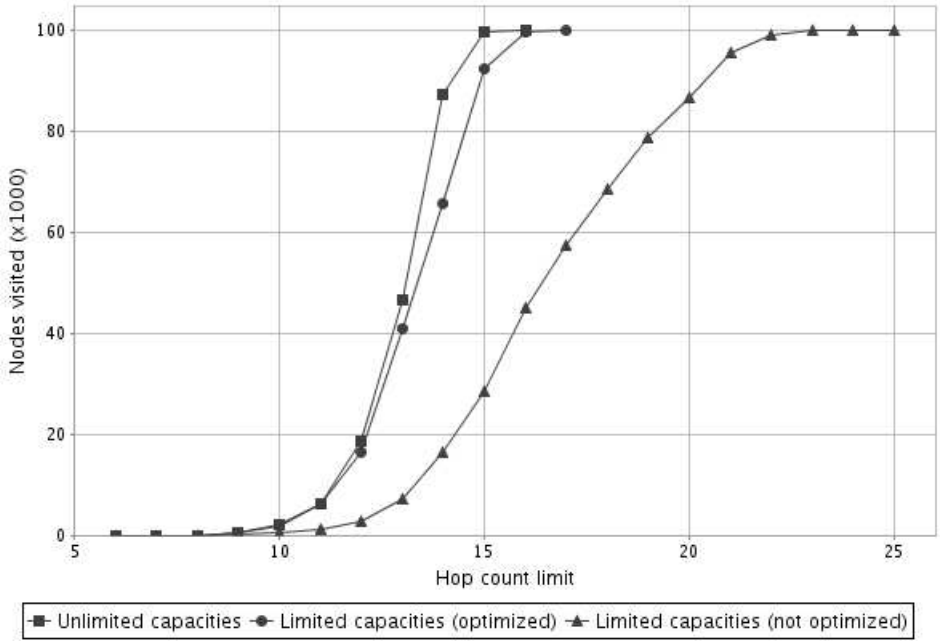


FIG. 4.14 – Impact de la connectivité limitée de certain nœuds en ayant une connaissance globale du réseau.

Les nœuds retirés sont sélectionnés au hasard en suivant une probabilité uniforme (la capacité n'est pas utilisée pour déterminer la durée de connexion des nœuds). Enfin il n'y a pas de régulation effectuée quand un nœud est en surcharge.

La figure 4.15 illustre l'évolution de la couverture du réseau réalisée par arbre de remplissage à TTL constant (14). Bien que la quantité de nœuds reste à peu près stable au cours du temps on peut observer une légère amélioration des performances dans les 10 premières itérations. Les performances restent ensuite globalement stables, ce qui illustre l'efficacité du processus de réparation du maillage : le réseau reste équilibré et conserve sa forme globale. L'amélioration du début s'explique par le fait que l'optimisation est réalisée moins régulièrement que dans les expérimentations précédentes et donc que pendant les premières itérations, le maillage peut encore être optimisé.

La figure 4.16 fait état de la valence des pairs en fonction de leurs capacités à la fin de l'expérimentation. Nous pouvons voir que quelques nœuds sont en très légère situation de stress, mais que globalement la repartition de la charge des pairs en fonction de leur capacité reste très acceptable. On peut bien sûr n'avoir aucun nœud en situation de stress en utilisant le processus de régulation mentionné à la fin de la section 4.2.7, mais nous souhaitions observer l'impact de ce phénomène. Enfin il y a toujours des nœuds de haute capacité avec une faible valence qui peuvent être susceptibles de prendre en main des réparations locales du maillage, comme nous l'avons mentionné à la section 4.2.7.

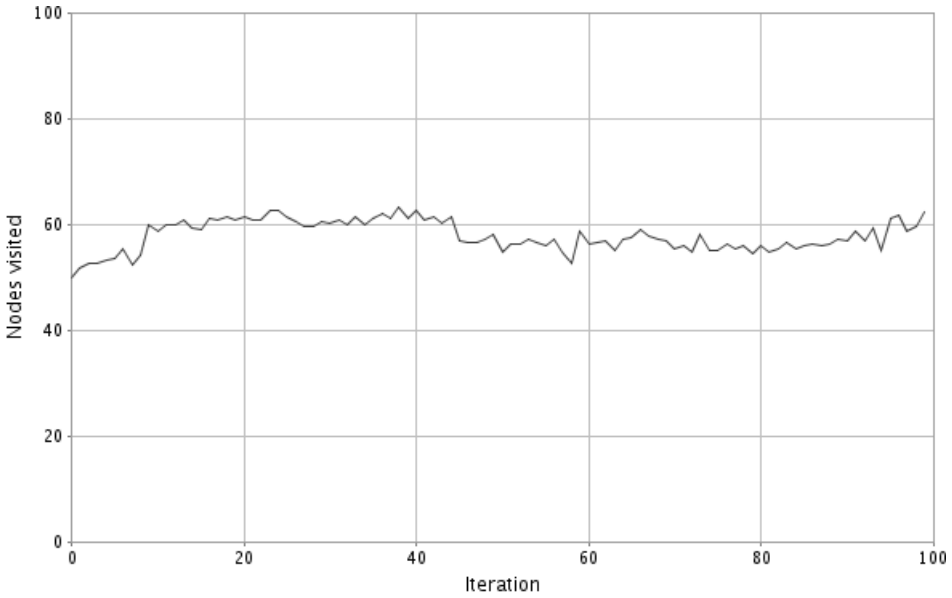


FIG. 4.15 – Couverture du réseau à TTL constant (14) suite au départ et à l'arrivée de nouveaux nœuds.

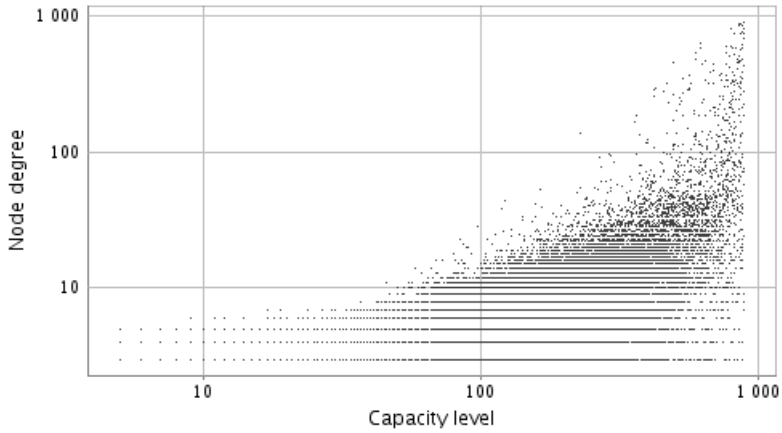


FIG. 4.16 – Valence des pairs en fonction de leurs capacités à la fin de l'expérimentation.

4.3.3 Charge en fonction de la valence

La stratégie EAR est efficace, en terme de couverture réseau à TTL fixé, avec des heuristiques de propagation privilégiant les nœuds de haute valence. Ces nœuds-là voient donc passer beaucoup de requêtes, alors que d'autres nœuds sont très peu sollicités. Nous mesurons ici la charge des nœuds en fonction de leur valence. Chaque nœud dans le réseau émet une requêtes avec un TTL de 13 et nous mesurons le nombre de requêtes vues par nœud en fonction de la taille de son voisinage. Nous avons fixé le TTL à 13 en se basant sur les précédentes expérimentations : cela permet d'avoir une couverture du réseau partielle, et donc de mieux mesurer les différences de quantité de requêtes vues entre les nœuds.

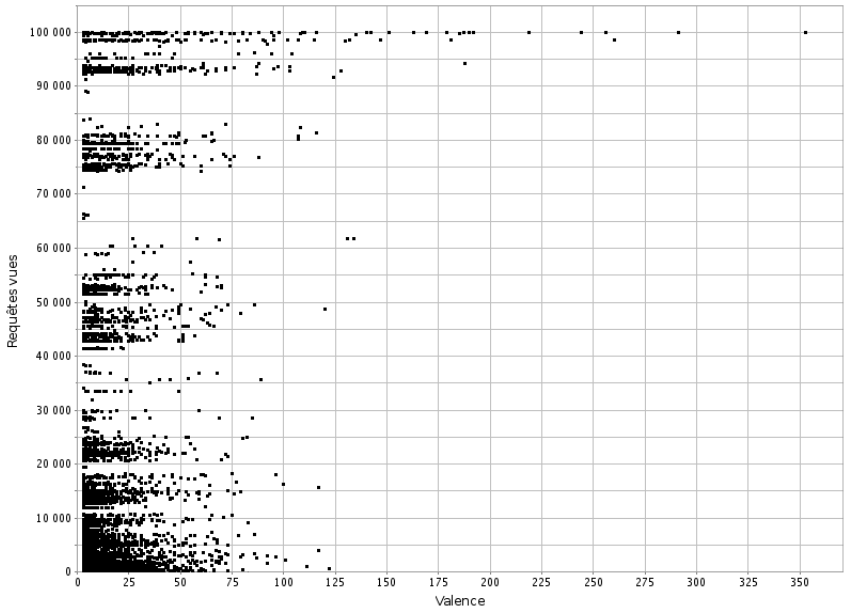


Fig. 4.17 – Nombre de requêtes vus par les nœuds en fonction de leur valence, avec un TTL fixé à 13.

La figure 4.17 illustre le résultat obtenu. On peut voir que tous les nœuds du réseau de valence supérieure à 130 voient passer toutes les requêtes qui ont été émises. La majorité des nœuds de faible valence ne voit passer qu'une partie des requêtes et sont donc peu sollicités. Il y a par contre quelques nœuds de faible valence qui voient passer toutes les requêtes : ce sont des nœud connecté à un ou deux nœuds de haute valence.

Comme nous l'avons mentionné à la section 4.3.2, certains nœuds possèdent de grosses capacités mais une faible valence et sont des candidats idéaux pour la réparation. La figure 4.17 nous montre qu'il faut placer ces nœuds à certains endroits du réseau, de sorte que même s'ils ont une faible valence, ils voient quand même passer une quantité de requêtes correspondant à leur capacités. On peut imaginer prendre ce paramètre en compte dans les phases d'optimisation du maillage.

4.4 Comparaison avec d'autres approches

Nous comparons ici les arbres de remplissage avec d'autres approches : l'inondation [17], la k-marche aléatoire [51] et LightFlood [40], avec les 4 premiers sauts réalisés en inondation classique. Nous faisons tourner ces algorithmes sur un graphe aléatoire suivant une loi de puissance, avec une valence moyenne de six. Ce type de graphe correspond assez bien aux topologies aléatoires qui peuvent se former dans des réseaux tels que Gnutella : peu de pairs ont une très haute connectivité, et la majorité des pairs a une faible connectivité. Les réseaux simulés contiennent 100000 nœuds.

4.4.1 Couverture

La figure 4.18 représente le nombre de nœuds différents visités en fonction du TTL de l'exploration. L'inondation a la meilleure couverture à petit TTL, mais ceci entraîne aussi beaucoup de redondance, comme nous allons le voir par la suite. LightFlood a lui aussi une bonne couverture avec un faible TTL : 95% des nœuds sont visités avec un TTL de 10.

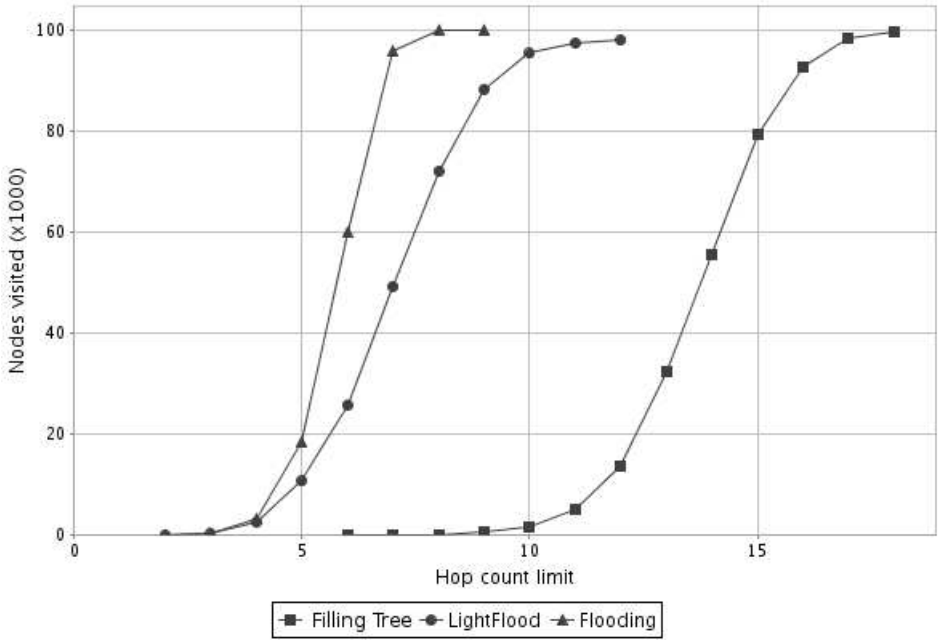


Fig. 4.18 – Couverture réseau.

Les arbres de remplissage nécessitent un TTL plus important pour couvrir le réseau, de l'ordre du double de l'inondation classique. Par contre, ils n'ont pas besoin de cache pour savoir quelles requêtes ont déjà été traitées par les nœuds. La marche aléatoire n'est pas mentionnée ici, dans la mesure où ses performances en terme de couverture réseau sont vraiment très faibles si le TTL n'est pas très grand.

En plus de cela, il ne faut que 20 sauts aux arbres de remplissage pour couvrir complètement le réseau, alors qu'avec un TTL infini LightFlood couvre 98.2% du réseau. Ce manque d'exhaustivité dans LightFlood est expliqué dans [40] : les messages propagés peuvent se heurter les uns aux autres dans l'arbre de diffusion et être perdus.

4.4.2 Redondance

Soit v le nombre de nœuds visités et m le nombre de messages générés. Le pourcentage de messages redondants est évalué à $100 \times (\frac{m}{v} - 1)$. La figure 4.19 illustre le pourcentage de messages redondants, donc inutiles, en fonction du nombre de nœuds visités, sur un graphe statique.

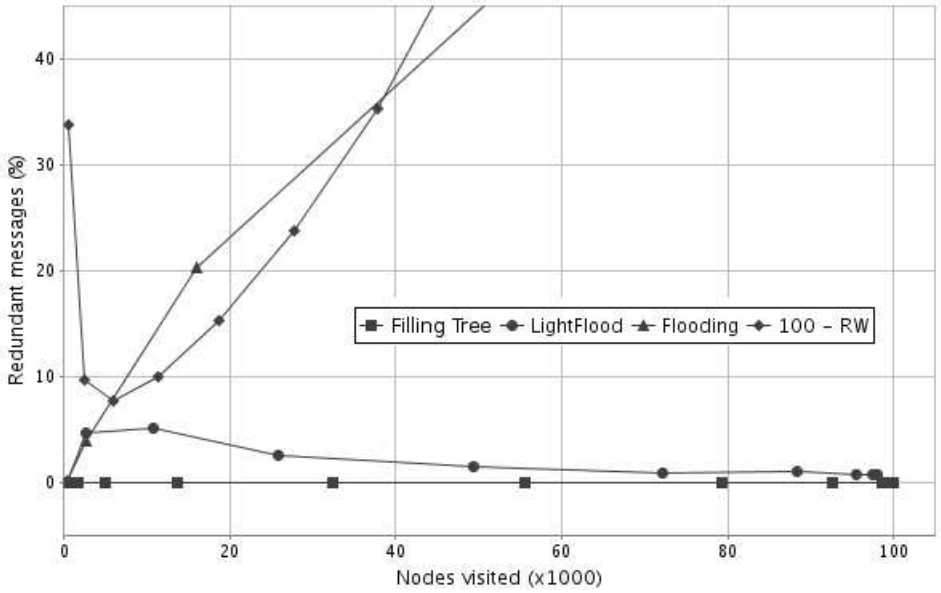


Fig. 4.19 – Messages redondants.

Nous voyons bien ici que la marche aléatoire et l'inondation ne passent pas bien à l'échelle, le pourcentage de messages redondants augmentant avec le nombre de nœuds visités. Ces deux approches sont par contre assez bien adaptées pour visiter peu de nœuds (jusqu'à 20000 dans notre simulation). LightFlood génère jusqu'à 5% de messages redondants, et ce au bout de quatre sauts d'inondation classique. Ensuite, dès lors que les messages sont propagés dans l'arbre de diffusion, la redondance diminue jusqu'à atteindre à peu près un pourcent pour la visite de 95000 nœuds. Comme illustré sur la figure 4.19, les arbres de remplissage ne génèrent pas de messages redondants.

4.4.3 Départ des nœuds

Le départ des nœuds peut avoir un impact sur la couverture des arbres de remplissage. Nous étudions cet aspect en mesurant cette couverture avec un TTL fixé à 14 dans deux scénarios. Dans le premier la topologie est réparée en suivant les procédures décrites précédemment et dans le second la topologie n'est jamais réparée : le nombre de trous dans le maillage augmente au cours du temps. La encore, les nœuds retirés sont choisis aléatoirement suivant une probabilité uniforme.

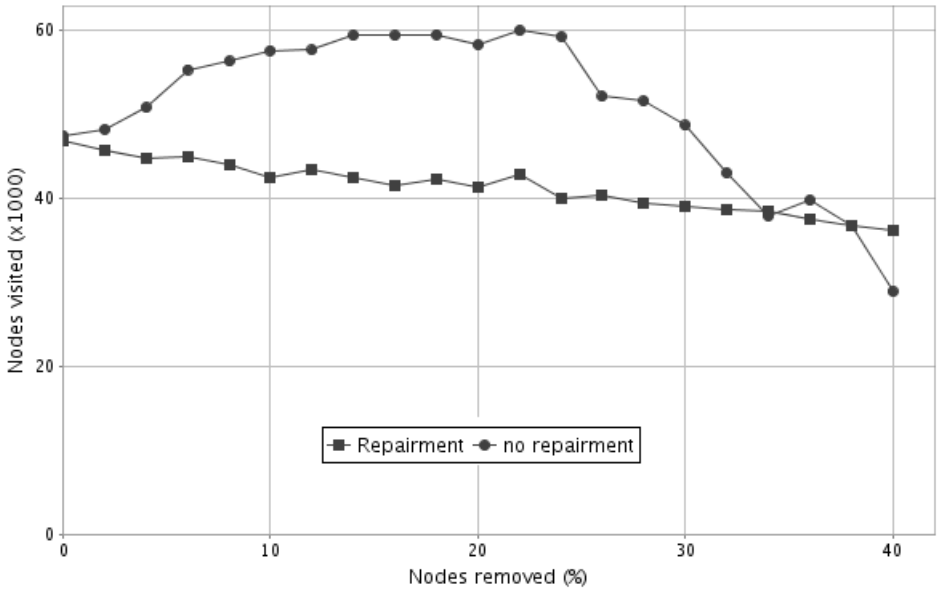


FIG. 4.20 – Évolution de la couverture moyenne à TTL fixe (14) suite au départ de nœuds

La figure 4.20 illustre le résultat obtenu. La couverture des arbres de remplissage dans une topologie réparée diminue petit à petit au cours du temps. Par contre cette couverture augmente puis diminue ensuite plus rapidement quand la topologie n'est pas réparée. Le fait d'avoir des trous dans la topologie augmente le taux de clonage des marcheurs, car le nombre de composantes connexes détectées à l'analyse du voisinage d'un nœud augmente également. La construction récursive du réseau garantit que malgré ce clonage accru, il n'y a toujours pas de messages redondants. Bien sûr, tandis que le nombre de nœuds couverts à TTL constant augmente, le nombre de nœuds atteignables avec un TTL infini diminue. C'est d'ailleurs pour cette raison que les performances des arbres de remplissage s'effondrent quand la topologie possède trop de trou non réparés.

Cette propriété est très intéressante pour un réseau dynamique car comme il y a tout le temps des nœuds qui rejoignent ou quittent le réseau, il est difficile de considérer le maillage comme toujours réparé. On peut d'ailleurs profiter de cette propriété pour utiliser un processus de réparation un peu plus lent mais plus performant, comme par exemple choisir le nœud qui va assurer la réparation en fonction de ses ressources

disponibles.

La topologie que nous présentons n'est, par contre, pas du tout résistante à des attaques ciblées sur certains nœuds, plus particulièrement les attaques visant les nœuds ayant les plus grosses capacités. Le fait par exemple de retirer 1% des nœuds les plus connectés sans réparation entraîne une baisse de la couverture très importante : seulement 0.15% des nœuds restent accessibles. Ceci s'explique aisément par le fait que toutes les recherches se font en remontant jusqu'aux pairs de haut degré.

Cette faible résistance aux attaques peut être contrebalancée par une réparation rapide : comme nous l'avons déja mentionné, il devient alors d'autant plus intéressant de ne pas optimiser le maillage trop régulièrement, de sorte que tous les nœuds de haute valence aient quelques voisins de haute capacité mais de valence minimale.

4.5 Synthèse

Le modèle d'architecture P2P que nous avons proposé offre des caractéristiques intéressantes, notamment en terme de couverture réseau. Ce modèle est naturellement adapté à l'hétérogénéité entre les capacités des pairs composant le réseau. Le coût de maintien de la topologie virtuelle reste en moyenne constant, quelle que soit la taille du réseau, car la valence des nœuds reste en moyenne constante. Nous avons proposé une primitive d'exploration, EAR, qui permet de contacter un nombre important de nœuds, sans redondance et sans avoir besoin de cache pour éliminer les requêtes déjà traitées. Ce type de réseau est donc adapté pour le stockage massif d'informations sur lesquelles on souhaite pouvoir réaliser des requêtes complexes. Il est également très bien adapté pour faire de la diffusion multicast.

L'architecture P2P proposée permet d'atteindre un grand nombre de nœuds avec une faible latence (de l'ordre du double de celle de l'inondation). Dans le cas d'un réseau parfaitement équilibré, on peut dans le cas le plus favorable (en partant du nœud de plus haute valence) atteindre un nombre de nœuds $c = TLL$ si $TTL \leq 3$ ou $c = 3 + 3^{TTL-3}$ dans le cas contraire. Il faut dans le cas le plus défavorable un TTL deux fois plus important pour atteindre la même quantité de nœuds, ce qui correspond au fait que le marcheur va remonter au premier niveau de récursivité, sur un nœud de très haute valence pour ensuite explorer le réseau en redescendant.

Les heuristiques étudiées mettent en évidence que les performances des arbres de remplissage sont directement liées aux principes récursifs qui conditionnent la construction de la topologie du réseau. L'heuristique des *k plus vieux triangles*, permet d'approcher une construction du réseau parfaitement équilibrée. L'heuristique du choix du voisin ayant le plus grand 2-voisinage permet la remontée efficace des marcheurs vers les niveaux supérieurs du réseau.

Il est évident que les performances du réseau sont très dépendantes de celles des nœuds de haute valence. Le fait d'en avoir peu (topologie initiale en triangle ou tétrahèdre) permet d'avoir les meilleures performances, mais rend le système plus vulnérable à une panne de ces nœuds.

Il reste encore certains aspects à améliorer, notamment réussir à placer des nœuds

de haute capacité à certains endroits clé du réseau, de sorte qu'ils n'aient que peu de voisins et voient "passer" beaucoup de requêtes. Cela permettrait d'avoir des nœuds capables de prendre en charge la réparation du maillage suite à la panne d'un nœud de haute valence, et que les nœuds voient passer une quantité de requêtes croissante en fonction de leurs capacités. On peut imaginer réaliser cela en mesurant sur les nœuds une "charge" correspondante au nombre de requêtes vues dans un intervalle de temps en fonction de leurs capacités. Un nœud en surcharge pourrait ainsi échanger sa place dans le réseau avec un nœud en sous-charge.

En suivant ce modèle, les nœuds de haute capacité auraient donc accès à plus d'informations que les nœuds de faible capacité. Cet aspect est très intéressant car plus les utilisateurs contribuent au bon fonctionnement du réseau (en allouant des ressources plus importes à cette architecture P2P), plus ils peuvent interroger un nombre de nœuds important quand ils recherchent une information en particulier.

On peut également imaginer enrichir l'architecture P2P que nous proposons avec un choix de triangle lors de la connexion prenant en compte des paramètres comme la latence ou la proximité géographique, ce qui permettrait de réduire encore la latence lors de la recherche d'information.

Chapitre 5

Réplication proactive uniforme

Sommaire

Bien qu'entraînant un surcoût notable en terme de consommation de ressources (notamment mémoire et bande passante), la réplication des données est une opération cruciale dans les systèmes distribués, car elle favorise les propriétés de résistance aux pannes et de passage à grande échelle : si un nœud dans le réseau devient inaccessible et qu'il possède des informations non répliquées, alors celles-ci sont définitivement perdues (sauf bien sûr si le nœud réintègre le réseau). La réplication de ces données les rend accessibles à tout instant, permettant ainsi une tolérance aux pannes. La réplication augmente aussi les capacités de passage à grande échelle : une grande quantité de nœuds peut simultanément accéder à la même information plus facilement si la charge induite par l'accès à cette donnée est répartie entre plusieurs nœuds dans le réseau.

Ce chapitre présente un mécanisme de réplication proactive uniforme basé sur la gestion d'un cache mis à disposition par les utilisateurs (nœuds) du réseau. Ce cache est constitué d'une partie de la capacité de stockage du nœud utilisateur. Il est utilisé par le mécanisme de réplication pour créer ou détruire des réplicas de données, dans le but de garantir une disponibilité maximale soit de toutes les données, soit d'une partie d'entre elles (par exemple les plus importantes).

Bien qu'on puisse très facilement réaliser cette opération en utilisant un serveur ayant une connaissance globale du réseau et des informations qui y sont présentes, une telle approche centralisée ne résiste pas au passage à grande échelle. Nous présentons une stratégie de réplication basée sur une estimation locale des densités de réplicas qui essaie de répliquer les données uniformément de manière dynamique [9]. Nous analysons la mise en œuvre de cette approche selon différentes stratégies d'exploration, en simulant des environnements stables et dynamiques.

5.1 Stratégies de réplication

Contrairement aux architectures P2P structurées dans lesquelles la recherche d'information est très rapide grâce à l'utilisation des propriétés structurelles du réseau, les architectures P2P non structurées mettent en œuvre des stratégies de recherche dans lesquelles les requêtes sont évaluées sur chaque nœud visité, puis propagées sur les nœuds voisins.

Puisque, pour les architectures P2P non-structurées, aucune hypothèse restrictive portant sur la nature des requêtes n'est formulée, il est délicat de proposer des heuristiques de recherche *a priori*. Dans ce cas il est nécessaire d'évaluer la requête sur chaque nœud et le temps de réponse du système sera par conséquent proportionnel à la densité de présence des réplicas des éléments d'information impliqués dans la requête. Plus une donnée est répliquée et plus sa probabilité de présence sur un nœud sélectionné au hasard sera grande. La question du temps de réponse est donc liée ici à un problème de contrôle de densité locale de réplica.

Cette section présente dans un premier temps des approches quantitatives de la réplication et aborde le problème de la réplication de toutes les données en quantité homogène ou non. Dans un second temps, nous présentons les avantages et inconvénients des principe de réplication réactive et proactive.

5.1.1 Quantité de réplicas

Comme mentionné précédemment, la réplication des données consomme des ressources, notamment en terme d'occupation mémoire et de bande passante. On ne peut donc pas répliquer toutes les données sur tous les ordinateurs du réseau et il faut déterminer la quantité et le placement optimal pour chaque réplica. Suivant différents critères de performance, on peut vouloir favoriser une réplication uniforme des données, proportionnelle à leur popularité ou basée sur d'autres critères. Nous présentons ici plusieurs stratégies de réplication étudiées dans [18]. Le critère de performance retenu ici est le temps moyen pour localiser l'information en utilisant une marche aléatoire.

La taille de l'espace de recherche exploré en moyenne est le nombre de nœuds qu'il faut interroger en moyenne pour répondre correctement à une requête. Si l'on souhaite minimiser le temps moyen pour localiser les informations, il faut donc minimiser la taille moyenne de cet espace de recherche.

5.1.1.1 Réplication uniforme

Pour ce type de stratégie de réplication, les informations possèdent toutes le même nombre de réplicas, indépendamment de tout critère, en particulier basé sur la popularité ou sur la rareté de l'information recherchée. Cette stratégie minimise la taille de l'espace de recherche exploré en moyenne quand la quantité de requêtes insolubles (aucune donnée ne satisfait aux critères de la requête) est importante [18].

On peut mettre en œuvre cette stratégie de la manière suivante : chaque fois qu'une information est initialement insérée dans le réseau, on créé un nombre fixe de copies

de cette information, les copies des données pouvant être placées sur d'autres nœuds choisis aléatoirement ou déterminés selon certaines heuristiques.

Cette mise en œuvre est notamment utilisée dans les architectures P2P structurées car les protocoles de routages utilisés permettent de savoir rapidement si l'information insérée dans le réseau est déjà présente ou non. C'est par contre une technique très difficile a mettre en œuvre dans les architectures P2P non structurées car on ne peut jamais être certain que l'information n'est pas déjà présente dans le réseau.

5.1.1.2 Réplication linéairement proportionnelle à la popularité

La réplication proportionnelle consiste à répliquer les informations suivant leur popularité, les informations très demandées devenant beaucoup plus répliquées que les informations peu demandées. Quand il y a peu de requêtes insolubles, la taille de l'espace de recherche exploré en moyenne est la même que dans un cadre de réplication uniforme, quelle que soit la distribution de la popularité des données [18]. Ce mode de réplication induit un gain de temps lors de la recherche des informations populaires, mais des délais plus longs lors de la recherche des informations rares.

Cette stratégie peut être mise en œuvre de la façon suivante : chaque fois qu'une information est demandée, on créé un nombre fixe de copies de cette information. C'est d'ailleurs la stratégie de réplication utilisée dans la majorité des systèmes P2P non structurés : quand les utilisateurs téléchargent une donnée sur leur ordinateur, cela créé un réplica supplémentaire de cette donnée sur leur propre machine.

5.1.1.3 Réplication proportionnelle à la racine carrée de la popularité

Cette stratégie se situe entre les deux précédentes, et c'est celle qui minimise la taille de l'espace de recherche exploré en moyenne quand le nombre de requêtes insolubles est assez faible [18]. Elle est définie de la manière suivante : pour n'importe quelle paire d'informations A et B dans le réseau, le ratio du nombre de copies R_A et R_B de ces informations est la racine carrée du ratio de leur popularité P_A et P_B : $\frac{R_A}{R_B} = \sqrt{\frac{P_A}{P_B}}$.

Cette stratégie peut être mise en œuvre de différentes manières, la plus simple étant la réplication basée sur le chemin parcouru par la requête : tous les nœuds qui ont propagé la requête relative à une donnée reçoivent une copie de cette donnée. Ainsi une donnée très populaire sera localisée très vite, les chemins des requêtes seront donc courts et peu de réplicas seront créés. Au contraire, une requête concernant une information rare générera de long chemins de recherche et beaucoup de réplicas seront créés. La convergence de cette approche n'est pas la plus rapide, mais cette méthode est la plus simple à mettre en œuvre ; elle est notamment utilisée par Freenet [16].

5.1.1.4 Discussion

Nous avons vu différentes stratégies de réplication, celle qui minimise la taille de l'espace de recherche exploré en moyenne quand le nombre de requêtes insolubles est relativement faible étant la réplication en quantité proportionnelle à la racine carrée

de la popularité des données. Dans le cas où la quantité de requêtes insolubles est importante, c'est la stratégie de réplication uniforme qui minimise la taille de l'espace de recherche exploré en moyenne.

Les architecture P2P structurées répliquent en général les données uniformément. A notre connaissance, aucune approche n'a été proposée pour assurer une réplication uniforme, en environnement dynamique, pour les architectures P2P non-structurées. Ceci est d'autant plus limitant que ce type d'architecture permet l'acheminement de requêtes pouvant être très spécifiques, et donc avoir un taux de rejet (quantité de requêtes insolubles) élevé.

5.1.2 Réplication réactive et proactive

La réplication des données peut se faire de deux manières : soit les données sont répliquées lorsque cela est nécessaire, de manière réactive, soit l'on essaye d'anticiper et les données sont répliquées avant que cela ne soit nécessaire, c'est-à-dire de manière proactive. Les deux approches ont bien sûr chacune leurs avantages et inconvénients, qui sont détaillés à la section 5.1.2.3.

5.1.2.1 Réplication réactive

Dans la plupart des systèmes P2P existants, les informations sont répliquées quand un utilisateur décide de télécharger une donnée sur son ordinateur. Il est par ailleurs nécessaire de mettre en œuvre des mécanismes de réplication pour assurer la disponibilité des données à tout moment dans le réseau. Dans la plupart des DHT, ceci est réalisé de manière réactive. Quand un nœud devient inaccessible, puisque la localisation des données est couplée à la topologie du réseau, on peut déterminer quels étaient les fichiers stockés par ce nœud, et donc répliquer ces fichiers, de manière à atteindre le niveau de réplication souhaité.

5.1.2.2 Réplication proactive

Quelques systèmes réalisent la réplication des informations de manière proactive. Une mise en œuvre pour les DHT de ce mode de réplication a notamment été proposé par [68]. Il induit une charge du réseau similaire à celle induite par une approche réactive, mais cette charge est distribuée de manière homogène dans le temps. De plus le système peut éventuellement être paramétré de telle sorte qu'il est possible de fixer des contraintes sur la bande passante utilisée. En réalisant une réplication basée sur le chemin des requêtes, Freenet [16] réalise lui aussi une réplication implicitement proactive.

5.1.2.3 Avantages et inconvénient des deux approches

Les mécanismes de réplication réactifs minimisent le trafic réseau généré dans la mesure où les réplicas ne sont créés que quand cela est nécessaire. Cependant ils peuvent

aussi générer des pics d'activité du réseau quand plusieurs nœuds deviennent inaccessibles simultanément. Ces pics peuvent créer des perturbations et gêner le fonctionnement de certaines applications utilisant l'architecture P2P comme couche basse [68].

D'un autre coté les mécanismes de réplication proactifs engendrent un peu plus de trafic réseau, mais celui-ci est à peu près constant dans le temps. On peut par ailleurs fixer plus facilement certaines contraintes comme l'utilisation de bande passante pour la réplication en tâche de fond [68]. Enfin, ce mécanisme paraît plus adapté aux architectures P2P non structurées pour assurer la disponibilité des données : il est difficile ou coûteux pour de telles architectures de déterminer quelles données étaient stockées sur un nœud devenu inaccessible. Celles-ci doivent donc être répliquées nécessairement avant la panne.

5.2 Réplication par estimation de densité

Les architectures P2P non-structurées permettent de ne faire aucune hypothèse sur le langage de requête utilisé. Nous nous intéressons tout particulièrement ici à des langages de requête permettant la formulation de requêtes spécifiques. Plus les requêtes sont spécifiques, moins il y a de réponses susceptibles de satisfaire aux critères de recherche et plus la quantité de requêtes insolubles augmente. Il faut donc dans ce cas de figure assurer une réplication uniforme des données pour minimiser la taille de l'espace de recherche exploré en moyenne.

Nous proposons un mécanisme de réplication uniforme et proactif pour les architectures P2P non structurées, basé sur un estimateur local de la quantité des réplicas. Ce mécanisme permet de garder pour chaque donnée un nombre de réplicas proportionnel à la taille du réseau, sans posséder la connaissance de cette taille. Le caractère proactif de notre approche permet d'adapter la réplication des données aux quantités de ressources réseau disponibles : les nœuds disposant d'une faible bande passante peuvent répliquer les données plus lentement que ceux qui possèdent cette ressource en plus grande quantité. Nous n'abordons pas dans cette partie la cohérence des réplicas, c'est-à-dire la mise à jour de nouvelles versions des fichiers.

5.2.1 Principe de fonctionnement

Dans l'architecture que nous proposons, un utilisateur (un nœud) n dispose de deux espaces pour stocker les données. Le premier est son espace personnel E_n : c'est dans cet espace que sont stockées les données que cet utilisateur n télécharge, ou met explicitement à disposition des autres utilisateurs. Le second espace, C_n, est un cache sur lequel l'utilisateur n'a aucun contrôle, si ce n'est la spécification de la taille c_n de celui-ci. Le cache est utilisé par le système pour stocker des réplicas de données.

Pour maintenir de manière distribuée la quantité de réplicas désirée, un score est calculé périodiquement en fonction de mesures effectuées pour chaque réplica présent sur le nœud courant (dans le cache et dans l'espace personnel de l'utilisateur). La durée de cette période d'estimation des taux de réplication locale influe sur la réactivité du système et sur la consommation de ressources. Ce score correspond à la quantité

de réplicas localement présents autour du nœud, il est utilisé pour la création et la suppression des réplicas dans le cache. Les données que l'on réplique sont celles qui ont le score le plus faible et celles que l'on supprime sont celles qui ont le score le plus important.

5.2.2 Gestion des réplicas

Nous considérons que chaque information possède un identifiant unique. Chaque nœud n dans le réseau possède une liste L_n de données candidates à la réplication proposée à l'itération précédente par d'autre nœud du réseau à ce nœud n. Chaque liste L_n contient des couples (d, A_m), où d correspond à un identifiant de données et A_m à l'adresse d'un autre nœud m dans le réseau possédant la donnée d. Cette liste L_n est de taille illimité, le nombre moyen d'éléments dans cette liste est le nombre moyen d'éléments proposés à la réplication par chaque nœud à chaque itération.

5.2.2.1 Mesures du score

La mesure du score S_d d'une donnée d sur un nœud correspond à une estimation locale de la densité de réplicas pour cette donnée d. Cette estimation est réalisée en explorant le voisinage du nœud et en comptant le nombre de réplicas rencontrés pour chaque donnée au cours de l'exploration. Les données les plus répliquées sont donc celles qui ont le score le plus élevé.

Il y a deux solutions pour récupérer les résultats d'une exploration : soit les marcheurs qui ont rencontré des nœuds possédant un réplica d'une donnée dont on cherche à calculer le score retournent au nœud source une fois qu'ils ont fini leur exploration, soit un clone du marcheur est renvoyé vers le nœud source dès qu'un réplica d'une information pertinente est rencontré. La première approche optimise la consommation de bande-passante tandis que la seconde est plus fiable.

5.2.2.2 Réplication proactive uniforme

Nous considérons un ensemble de données de taille unitaire (toutes les données sont considérées comme ayant la même taille). L'algorithme de réplication est exécuté périodiquement sur chaque nœud n et se déroule en plusieures étapes. Premièrement, le nœud n détermine les données qu'il doit répliquer et celles qu'il doit supprimer. Ceci est réalisé avec une exploration locale de son voisinage pour quantifier les nombres de réplicas des données que ce nœud possède (dans son espace personnel, son cache et sa liste de données candidates à la réplication).

Pour chaque donnée d candidate à la réplication, si son score est inférieur à la moyenne du score des données dans le cache du nœud n, alors la donnée dans le cache de n ayant le plus haut score est supprimée et la donnée d est téléchargée sur le cache du nœud n. A la fin de cette étape, la liste des données candidates à la réplication est vidée.

Enfin le nœud n sélectionne les k données ayant le plus petit score parmis celles qu'il possède dans son espace personnel et dans son cache. Il contacte k nœuds du

Algorithme 13 : Réplication sur le nœud n par estimation des densités locales des réplicas.

Données : k : nombre de données par nœud proposées à la réplication à chaque itération

tant que *vrai* **faire**

Calcul du score S_d pour toutes les données d dans $C_n \cup E_n \cup L_n$

`// Initiation d'une exploration locale`

`// Attente du résultat`

$\overline{S} \leftarrow \frac{\sum_{d \in C_n} S_d}{|C_n|}$

pour tous les $d \in L_n$ **faire**

si $S_d < \overline{S}$ **alors**

$C_n \leftarrow C_n \cup \{d\}$

$C_n \leftarrow C_n \setminus \{d_{max} \in C_n / \forall d_x \in C_n, S_{d_{max}} \geq S_{d_x}\}$

fin

$L_n \leftarrow \emptyset$

$D_{min} = [d_1..d_k] \leftarrow k$ données qui ont les plus petits scores dans $C_n \cup E_n$

$M = [m_1..m_k] \leftarrow k$ nœuds du réseau contactés au hasard

pour tous les $i \in [1..k]$ **faire**

$L_{m_i} \leftarrow L_{m_i} \cup (d_i, A_n)$

fin

`// Attente de la prochaine itération`

fin

réseau au hasard et propose à la réplication à chaque nœud une donnée différente. Ce processus peut être réalisé en parallèle du téléchargement des données candidates à la réplication pour ce nœud n décrit au paragraphe précédent si les données proposées à la réplication ne viennent pas d'être juste répliquées sur ce nœud (à la précédente itération de l'algorithme).

Ce processus est décrit en détail dans l'algorithme 13. La fréquence d'exécution de cet algorithme peut être réglée différemment selon les nœuds du réseau. On peut notamment paramétrer cette fréquence en fonction des ressources réseau que chaque nœud possède.

5.2.3 Étude expérimentale

Nous évaluons différentes stratégies d'exploration pour la mesure du score : inondation, marche aléatoire et arbres de remplissage (EAR). Nous étudions plusieurs heuristiques pour la mesure du score avec la stratégie EAR. La marche aléatoire est biaisée pour éviter de visiter les nœuds déjà vus. Les réseaux simulés dans cette section ont une taille de 10000 nœuds. Les algorithmes de marche aléatoire et d'inondation sont utilisés sur une topologie en graphe aléatoire suivant une loi de puissance.

Nous utilisons la même répartition de capacité que dans le chapitre précédent pour

déterminer les ressources (CPU, mémoire et bande passante) disponibles sur les nœuds du réseau. Pour simuler une réplication quantitativement hétérogène des données, nous attribuons différents niveaux de popularité aux données (entre 1 et 100). Les nœuds reçoivent ensuite une quantité de données proportionnelle à leur capacité, le tirage aléatoire des données se faisant en utilisant une distribution proportionnelle par rapport à la popularité des données.

Chaque fois qu'un nœud rejoint le réseau, son cache est vide. La taille du cache de chaque nœud correspond au nombre de données qu'il reçoit au début de l'expérimentation. Il est ensuite possible de régler le nombre moyen de replicas des données en paramétrant le nombre de données (plus il y a de données différentes et plus le nombre moyen de replicas est faible).

5.2.3.1 Critère de mesure

Nous étudions l'uniformité de la réplication en mesurant l'écart type des quantités de réplicas de chaque donnée sur tout le réseau. En considérant qu'il y a n données différentes en quantités $x_i, i \in [1..n]$ et que le nombre moyen de réplicas est $\overline{x}$, l'écart type des quantités de réplicas est :

$$\sigma = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(x_i - \overline{x})^2}$$

Pour pouvoir faire varier le taux de réplication des données et pouvoir comparer les différentes approches, nous utilisons l'écart type relatif $\frac{\sigma}{\overline{x}}$.

5.2.3.2 Réseau statique

Dans cette expérimentation, le taux moyen de réplication des données est initialement de 0.6% (le taux de réplication espéré est donc de 1.2%). Nous fixons les TTL pour les différentes approches de manière à avoir une couverture du réseau (nombre de nœuds visités) à peu près semblable, fixée à 20%. Nous obtenons les couvertures décrites dans le tableau 5.1. A chaque itération, les nœuds proposent les $k = 2$ données ayant localement le score le plus bas à d'autres nœuds. Cela permet une convergence plus rapide de l'algorithme qu'en proposant une seule donnée, tout en faisant en sorte que les nœuds ne copient pas trop de données simultanément (l'idéal pour une convergence rapide étant d'avoir un réplica créé par nœud en moyenne à chaque itération)

Stratégie	TTL	Couverture
Inondation	4	19, 65%
Marche aléatoire	2000	19.98%
EAR-2hopNeighboor	13	18.07%
EAR-random	120	19.86%
EAR-smallestNeighborhood	290	20.67%

Tab. 5.1 – Couverture de plusieurs stratégies d'exploration.

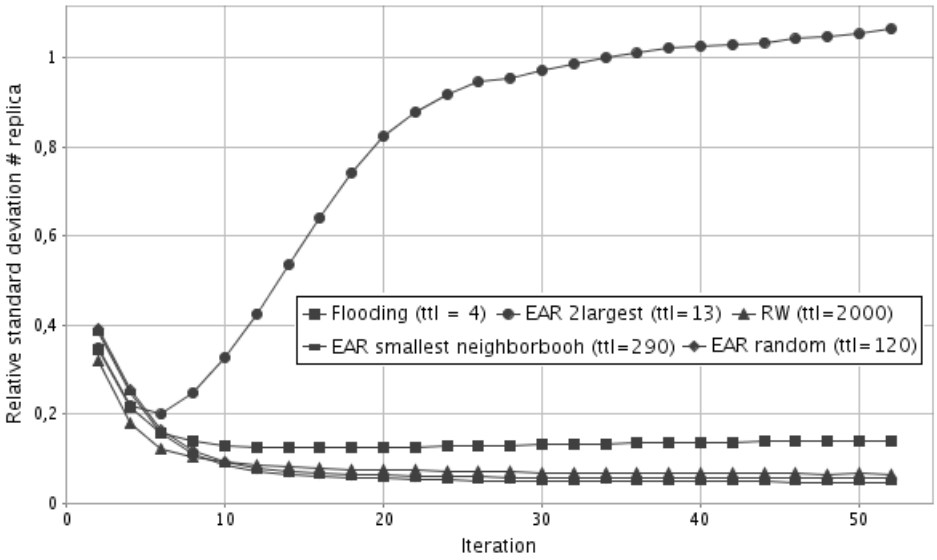


FIG. 5.1 – Évolution de l'écart type relatif de la quantité des réplicas.

La figure 5.1 illustre l'évolution de l'écart type relatif de la quantité de replicas. La marche aléatoire, EAR associé à une heuristique de propagation basée sur un choix aléatoire du voisin sélectionné et EAR associé à l'heuristique de propagation basée sur le choix du voisin de plus faible valence offrent les meilleures performances, avec un léger avantage pour le dernier cité. L'inondation offre des performances raisonnables, de l'ordre de la moitié, par rapport aux trois stratégies précédentes.

Par contre, EAR associé à l'heuristique basée sur le choix du voisin ayant le plus grand 2-voisinage offre les moins bonnes performances : l'écart type relatif de la quantité de réplicas diminue pendant les premières itérations mais il remonte fortement ensuite. L'utilisation d'une heuristique de propagation qui privilégie les nœuds des premiers niveaux (car de plus haute valence) a pour conséquence que les nœuds dans le dernier niveau du maillage ne peuvent pas atteindre d'autres nœuds de ce même niveau en utilisant EAR à TTL réduit. Quelques données possèdent beaucoup de réplicas sur ces nœuds, ce qui entraîne un accroissement de l'écart type relatif de la quantité de réplicas.

La figure 5.2 illustre le nombre de données copiées par nœud à chaque itération. Elle nous permet d'analyser la convergence de l'algorithme en millieu statique. La mesure du score en utilisant une exploration par inondation offre les meilleures performances en terme de convergence, c'est d'ailleurs la seule approche qui converge réellement. Au contraire, l'algorithme de réplication ne converge jamais avec l'utilisation d'une exploration par marche aléatoire pour la mesure du score.

La convergence de l'algorithme de réplication en mesurant le score avec EAR se situe entre les performances obtenues en mesurant le score par inondation et celles obtenues en utilisant la marche aléatoire. La quantité de réplicas déplacés à chaque itération est variable suivant l'heuristique de propagation utilisée pour EAR. Nous pensons que

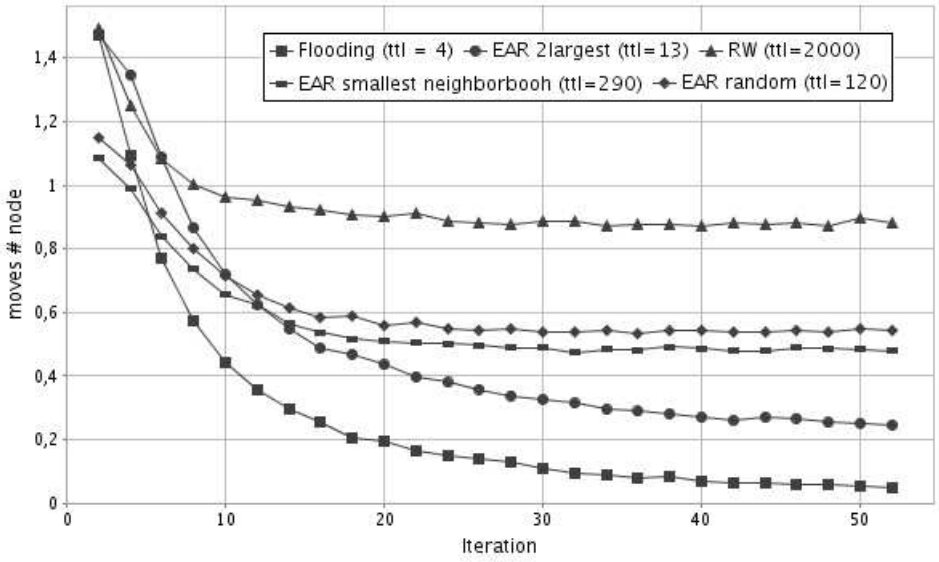


FIG. 5.2 – Évolution du nombre de création de réplicas par nœud à chaque itération.

ceci est dû à la part d'aléatoire utilisée lors de la propagation. Un choix de voisin aléatoire est rarement nécessaire en utilisant l'heuristique du plus grand 2-voisinage (la probabilité que deux nœuds aient exactement la taille de 2-voisinage est assez faible). L'impact de ce choix aléatoire du voisin suivant est plus important pour l'heuristique de la plus petite valence, et encore plus important pour le dernier cas.

Visiter les mêmes nœuds à chaque exploration semble donc être une condition importante à la convergence de l'algorithme en environnement statique. On peut imaginer utiliser comme heuristique de propagation avec la stratégie EAR le choix du voisin ayant le plus petit 2-voisinage, pour réduire la part d'aléatoire par rapport au voisin ayant la plus petite valence. Cependant nous verrons que ce critère de régularité dans l'exploration n'est pas aussi important en environnement dynamique.

5.2.3.3 Impact de l'estimation de densité

D'après la figure 5.2, la quantité de réplicas créés à chaque itération se stabilise au bout de 20 itérations à peu près. Pour déterminer dans quelle condition la réplication uniforme par estimation de densité est viable, nous déroulons donc l'algorithme de réplication dans les mêmes conditions que dans la section précédente, pendant 20 itérations puis nous analysons le résultat obtenu. Le paramètre que nous faisons varier est la quantité moyenne de réplicas des données. Cela nous permet de déterminer combien de réplicas sont détectés en moyenne lors de chaque exploration.

Nous n'avons retenu pour cette étude que deux stratégies : l'inondation car c'est celle qui possède la meilleure convergence en milieu statique, et EAR avec comme heuristique de propagation le choix du voisin de plus faible valence car c'est la stratégie

qui réplique le plus uniformément les données en milieu statique.

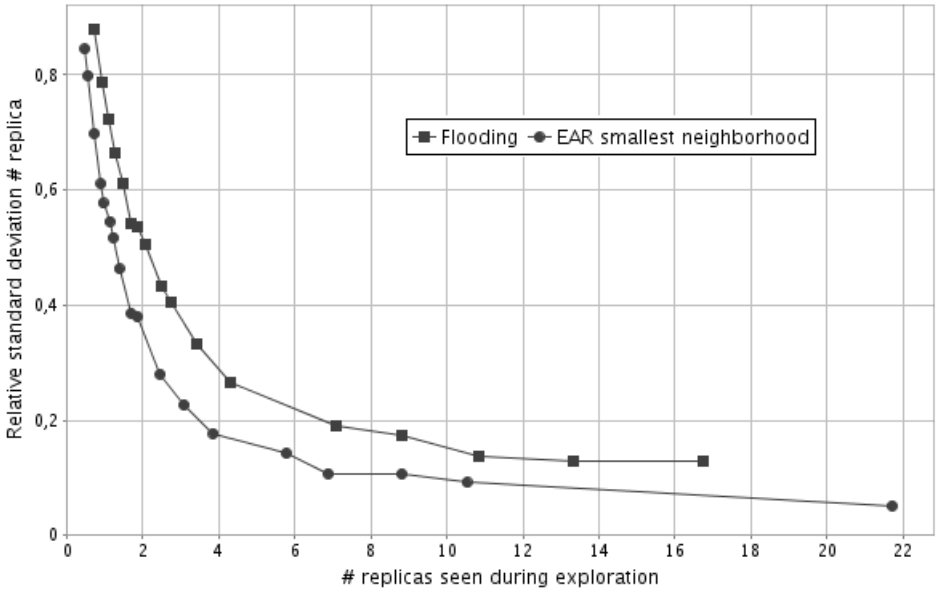


Fig. 5.3 – Écart type relatif obtenu au bout de 20 itérations en fonction du nombre moyen de nœuds possédant un réplica des données dont on mesure le score.

La figure 5.3 illustre l'écart type relatif obtenu à la fin des 20 itérations. Nous pouvons voir que si le nombre des réplicas rencontrés lors de l'exploration est insuffisant, soit parce que le nombre de nœuds visités est trop faible, soit parce que l'information n'est pas assez répliquée, l'approche ne converge pas, et même pire, elle diverge (l'écart type relatif initial se situe entre 0.3 et 0.4, comme illustré sur la figure 5.1).

Alors qu'il faut rencontrer en moyenne 7 réplicas avec l'inondation pour que l'écart type relatif du nombre de réplicas des données passe en dessous de 0.2, EAR associé à l'heuristique de propagation basée sur le choix du voisin de plus faible valence ne nécessite de rencontrer que 4 réplicas. Cette différence pourrait être due à la part d'aléatoire présente dans cette heuristique.

On peut également supposer que la différence de taille du 4-voisinage des différents nœuds de la topologie aléatoire est importante et que des données peuvent être surrépliquées sur les nœuds ayant un petit 4-voisinage, comme dans le cas de la stratégie EAR associée à l'heuristique de propagation basée sur le choix du voisin ayant le plus grand 2-voisinage.

Nous avons réalisé des simulations sur des réseaux de taille relativement faible (10000 nœuds). Cependant, la taille du réseau ne change rien pour une estimation de la densité locale des réplicas : seul le taux de réplication de l'information à un impact sur cette stratégie. En explorant 1000 nœuds en utilisant EAR, il faudrait que l'information soit répliquée à hauteur de 0.4% pour avoir un écart type relatif sur la quantité des réplica qui soit inférieur à 0.2. Un taux de réplication plus faible de 1 pour

10000 nécessite l'exploration de 40000 nœuds en utilisant EAR, et presque deux fois plus en utilisant l'inondation. La réplication uniforme par estimation de densité n'est donc adaptée que pour les réseaux où l'information est relativement bien répliquée.

5.2.3.4 Réseau dynamique

Nous avons enfin voulu tester notre approche en environnement dynamique. Pour simuler cet aspect, nous retirons des nœuds du réseau et nous rajoutons de nouveaux nœuds de sorte que la taille du réseau reste en moyenne constante dans le temps. Les nœuds (initialement présents ou nouvellement ajoutés) reçoivent une quantité de données proportionnelle à leur capacité, le tirage aléatoire des données se faisant de la même manière que ce qui a été décrit au début de cette section. Le cache des nœuds (initialement présents ou nouvellement ajoutés) est initialement vide. L'algorithme de réplication est exécuté sur tous les nœuds du réseau à chaque fois que 10% des nœuds du réseau ont été renouvelés.

Là encore, pour cette expérimentation, nous n'avons retenu que deux stratégies : l'inondation et EAR associé au choix du voisins ayant la plus petite 2-valence. Le taux moyen initial de réplication de l'information est, comme dans la première expérimentation, de 0.6% (le taux de réplication espéré est donc de 1.2%). Contrairement aux deux expérimentations précédentes, les nœuds proposent une seule ($k = 1$) autre donnée à répliquer à chaque itération : cela permet de diminuer le nombre de création de réplicas (et donc de diminuer l'utilisation des ressources du réseau) à chaque itération. En contrepartie la quantité de réplicas créés à chaque itération met plus de temps à se stabiliser.

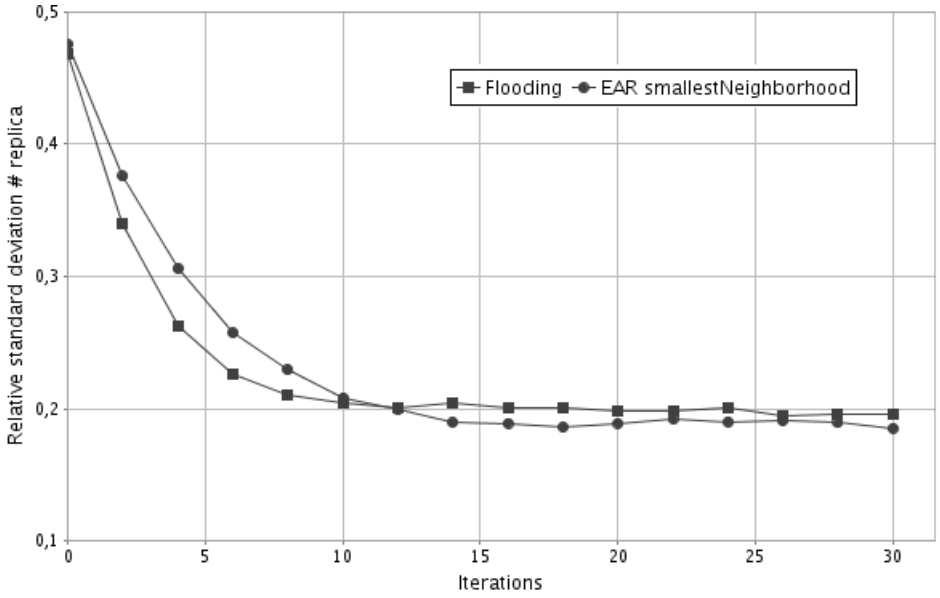


Fig. 5.4 – Évolution de l'écart type relatif de la quantité des réplicas.

La figure 5.4 illustre l'évolution de l'écart type relatif des quantités de réplicas des différentes données. On peut distinguer deux phases : la première correspond à un régime transitoire pendant lequel la quantité de réplicas s'adapte au réseau, puis une seconde phase à partir de la dixième itération qui correspond à un régime de fonctionnement stable. Tandis que le résultat obtenu en utilisant l'inondation converge plus vite pendant la première phase d'adaptation, EAR permet d'obtenir un écart type relatif du même ordre en fonctionnement permanent lors de la deuxième phase.

Dans les deux cas, l'écart type relatif est plus important que dans des réseaux statique : très probablement parce que les nœuds qui se connectent au réseau ont un cache vide.

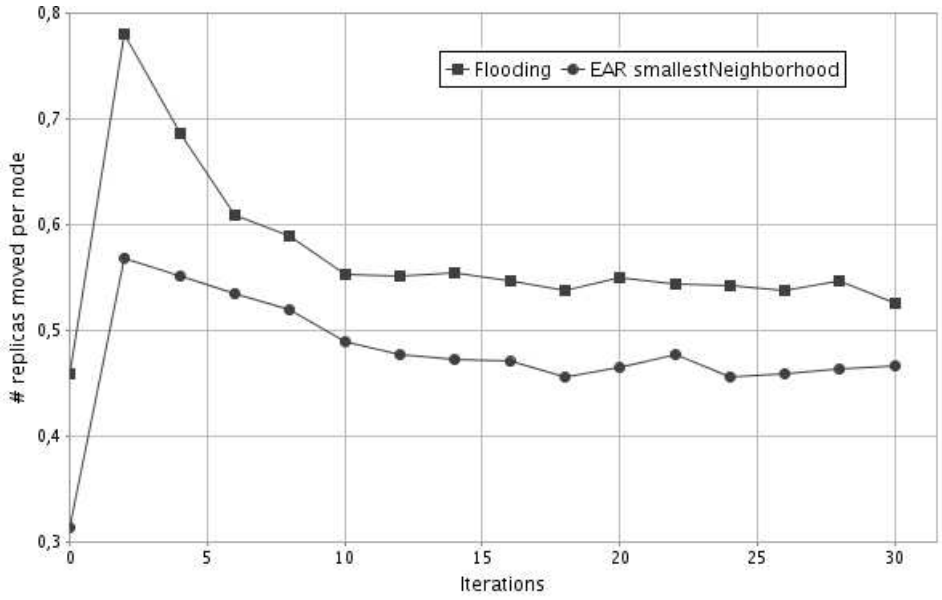


Fig. 5.5 – Évolution du nombre de copie de réplicas par nœud à chaque itération.

La figure 5.5 illustre la quantité de réplicas créés par nœud à chaque itération de l'algorithme. Dans tous les cas, la quantité de copies créées est significativement inférieure en utilisant une exploration par arbre de remplissage par rapport à une exploration par inondation.

Même si réaliser une itération de l'algorithme de réplication tous les 10% de nœuds renouvelés peut paraître important, il faut rappeler que la moitié des nœuds du réseau Gnutella restaient connectés moins d'une heure en 2003 [66].

Nous n'avons pas testé de configuration pour laquelle une itération de l'algorithme de réplication est réalisée par exemple tous les 50% de nœuds renouvelés. Nous pensons qu'un délai trop important entre deux itérations de l'algorithme de réplication pourrait provoquer des pertes de données.

5.2.4 Discussion

Nous avons présenté dans ce chapitre une stratégie de régulation du nombre de réplicas basée sur une estimation locale des densités des réplicas. Cette approche n'exploite pas de mesures de la popularité des données en se basant sur l'analyse des requêtes formulées par les utilisateurs et permet de s'approcher d'une réplication uniforme, ce qui est adapté aux environnements P2P dans lesquels beaucoup de requêtes ne peuvent être résolues, cas d'autant plus fréquent que l'on fait des requêtes complexes et précises.

L'architecture P2P que nous avons proposée au chapitre 4 est bien adaptée à cette stratégie de régulation de la quantité de réplicas basée sur l'estimation de la densité locale. En plus de fournir de meilleures performances comparativement à une exploration par inondation sur un graphe aléatoire, tant du point de vue de l'uniformité de la réplication que de la quantité de trafic réseau générée, elle utilise l'approche EAR qui permet l'exploration sans redondance et sans utilisation de cache supplémentaire sur les nœuds.

Dans la mesure où c'est le nombre moyen de réplicas rencontrés lors d'une exploration qui influe sur les performances de l'algorithme de réplication par estimation de densité des réplicas, il pourrait être intéressant d'adapter la quantité de nœuds explorés (donc le TTL utilisé pour la stratégie d'exploration) pour avoir un nombre moyen de réplicas rencontrés constant (par exemple une dizaine). Les quantités de nœuds visités en lançant une exploration avec les mêmes paramètres de différents nœuds seraient alors plus homogènes, ce qui permettrait d'améliorer les performances de l'algorithme de réplication. Enfin cette adaptation du TTL permettrait de pouvoir utiliser l'algorithme de réplication que nous proposons quel que soit le taux moyen de réplication de l'information : plus le taux de réplication moyen est faible et plus l'algorithme de réplication est coûteux (car il nécessite de visiter plus de nœuds pour l'estimation de la densité des réplicas).

La réplication par estimation de densité locale est, à notre connaissance, la seule stratégie qui permette à ce jour de réaliser une réplication uniforme dans un réseau P2P non structuré. Cette stratégie est relativement facile à mettre en œuvre et ne nécessite aucune connaissance globale du réseau. Elle n'est cependant réalisable que si suffisamment de place est disponible dans les caches pour que les informations puissent être suffisamment répliquées.

Les expérimentations que nous avons réalisées ne constituent qu'une étude préliminaire : nous n'avons par exemple pas exploité la distribution non uniforme de la taille des données pour optimiser la localisation des réplicas. Ainsi, les réplicas des données sont proposés à des nœuds choisis aléatoirement. Il pourrait également être intéressant d'essayer de placer les différents réplicas des données de telle sorte que l'on essaye de minimiser le nombre de nœuds à visiter pour retrouver des données correspondant à un critère de recherche.

Chapitre 6

Conclusion

Sommaire

6.1 Bilan

Les architectures P2P non-structurées sont actuellement relativement bien adaptées pour la recherche et la réplication d'informations populaires, tout en tirant partie des capacités hétérogènes des différents pairs composant le réseau. Parallèlement, les architectures P2P structurées permettent de répliquer et de rechercher efficacement n'importe quelle donnée, quelle que soit sa popularité. Dans ce contexte, les contributions de cette thèse concernent la recherche d'information sans contrainte particulière sur le langage de requêtes utilisé, et la réplication de données peu populaires tout en tirant partie de l'hétérogénéité des capacités des différents pairs composant le réseau.

Bien qu'étant initialement centrée sur un certain type de données (en l'occurrence des fichiers XML) et sur des requêtes dédiées, notre étude s'est ensuite généralisée à tout type de données. Nous avons présenté dans un premier temps une structure de données permettant l'aiguillage de requêtes de chemins XML dans des réseaux P2P non structurés. Cette approche permet de retrouver les documents semi-structurés possédant les propriétés structurelles requises à faible coût (notamment par rapport au nombre de messages générés) mais reste limitée quand à l'expressivité du langage de requêtes employé.

Nous nous sommes donc attachés dans un second temps à ne faire aucune supposition sur la nature des données ni sur le langage de requête employé en se basant sur des architectures P2P susceptibles de contacter rapidement un grand nombre de nœuds par propagation des requêtes. L'inondation permet de réaliser cela, mais elle entraîne en contrepartie un nombre important de messages redondants, et donc inutiles. Nous avons développé une marche en spirale qui permet d'explorer le voisinage d'un nœud sans redondance, mais nous avons aussi montré que cette approche n'était pas viable

dans un environnement hétérogène.

Les arbres de remplissage que nous avons ensuite présentés permettent d'atteindre un grand nombre de nœuds rapidement dans un environnement hétérogène et ne génèrent pas de redondance. Les approches actuelles essaient de limiter la redondance induite par inondation en utilisant un mécanisme de cache pour stocker les requêtes récentes déjà vues. Les arbres de remplissages n'ont pas besoin de cache pour réaliser la propagation de requêtes sans redondance. Ils nécessitent par contre une topologie dédiée : nous avons présentés des protocoles qui permettent de maintenir cette topologie à un coût relativement faible, en moyenne constant quelle que soit la taille du réseau, et qui tirent efficacement partie de l'hétérogénéité des capacités des différents pairs composant le réseau.

Nous avons enfin présenté un mécanisme permettant de réaliser une réplication uniforme des données, adapté aux situations où la proportion de requêtes insolubles est importante, ce qui est d'autant plus à même de se produire que le langage de requêtes utilisé permet une expressivité et surtout une précision importante. Cette approche est basée sur l'exploration locale du voisinage des nœuds, et peut donc être combinée avec les arbres de remplissage pour assurer la disponibilité des informations rares. Nous avons montré que cela permet d'ailleurs d'obtenir de meilleures performances comparativement à une approche basée sur un mécanisme d'inondation sur un graphe aléatoire qui suit une loi de puissance, tant du point de vue de l'uniformité de la réplication que du trafic réseau généré pour l'exploration (pas de redondance) ainsi que du point de vue de la réplication (moins de mouvement de données qu'avec une estimation de densité par inondation).

Nous ne pouvons pas prétendre avoir étudié tous les aspects présentés dans cette thèse de manière exhaustive et empirique. Les études expérimentales que nous avons présentées ont été réalisées à l'aide d'un simulateur relativement simple, et nous avons omis de nombreux paramètres nécessaires à une simulation se rapprochant des conditions réelles. Cette thèse valide des éléments de conception pour un réseau P2P, et si l'étude paraît viable en théorie, il reste à la tester en pratique à grande échelle.

6.2 Perspectives

Comme nous venons de le mentionner, une première prolongation de notre travail consisterait soit à réaliser un simulateur beaucoup plus performant, soit à en utiliser un déjà existant, pour pouvoir prendre en compte des paramètres comme la latence des nœuds, des répartitions de données plus réalistes avec des fichiers de taille hétérogène, etc.

Par ailleurs, si les arbres de remplissage sont bien adaptés à des réseaux P2P dont les nœuds possèdent des capacités hétérogènes, si la topologie virtuelle nécessaire peut être entretenue à faible coût et facilement réparée et si les départs et pannes aléatoires des nœuds n'affectent pas les performances globale du réseau, des attaques coordonnées peuvent par contre poser de gros problèmes : les nœuds de valence élevée ont un rôle

très important dans le bon fonctionnement de l'exploration par arbre de remplissage. La mise hors-service de quelques nœuds à forte valence diminue de manière considérable les performances de l'approche que nous avons présentée. Des efforts sont donc nécessaires pour trouver un moyen de rendre cette approche plus résistante aux attaques.

Il y a deux manières de solutionner ce problème : rendre le système moins dépendant des pairs de haute valence ou faire en sorte que la réparation du système soit rapide et efficace. Il paraît difficile d'améliorer le premier point, car les bonne performances de couverture que nous obtenons sont justement dûes à une exploitation maximale de l'hétérogénéité.

Le second point peut par contre être amélioré. Comme nous l'avons mentionné au chapitre 4, certains pairs de faible valence voient passer beaucoup de requêtes : ces pairs sont directement connectés à un ou plusieurs pairs de haute valence. Nous avons vu aussi que certains pairs ayant beaucoup de ressources sont en sous-charge, même si ce point est discutable car il dépend directement de la distribution des capacités choisie. A des fins d'optimisation, il serait donc intéressant de pouvoir placer les pairs en sous-charge au voisinage immédiat des pairs de haute valence.

Cela permettrait ainsi que les pairs voient passer une quantité de requête proportionnelle à leurs capacités (mémoire, CPU, bande passante) et améliorerait également la réparation du maillage. On pourrait imaginer une possibilité d'échange de position entre les pairs, mais contrairement à ce que nous avons présenté dans la section 4.2.7, l'échange pourrait se faire entre deux pairs distants et dépendrait non plus de la valence mais de la charge des nœuds.

La stratégie de réplication que nous avons présentée manipule des données de taille homogène. En ajoutant à l'identifiant d'une donnée sa taille lors de la proposition d'éléments à répliquer, il serait alors possible de travailler sur un ensemble de données de taille hétérogène. On pourrait aussi fragmenter les données : tous les fragments auraient alors la même taille (ou au moins une taille bornée), tandis que les données de taille différente sont caractérisées par un nombre de fragments différent.

Des mécanismes de fragmentation avancés comme les codes à effacement [55, 56] permettraient sans doute d'accroître les performances du système au niveau de la disponibilité des données (mécanisme plus résistant aux pannes) et de leur accessibilité (possibilité de choisir un fragment de fichier a tranférer parmi plusieurs pairs et choix de celui qui permet le meilleur taux de transfert).

L'étude qui a été réalisée reste encore incomplète, et certains points restent encore à préciser comme notamment la gestion de situations rares mais délicates comme la panne simultanée de plusieurs nœuds voisins. L'architecture proposée n'a été testée qu'en simulation : une étude de cette approche en pratique permettrait de préciser certains points, et de confirmer ou non la viabilité de l'architecture proposée.

Nous pensons aussi que l'algorithme de réplication que nous avons proposé peut être amélioré. L'idée serait de combiner des élément de *crawling* avec ce que nous avons proposé. Ainsi les utilisateurs auraient des descripteurs de leurs centre d'intérêt et l'algorithme de réplication pourrait tenir compte de ces descripteurs pour essayer de

créer des réplicas des données là où elles sont susceptibles d'être le plus utilisées. Cela permettrait d'une part de réduire le trafic réseau, et d'autre part de diminuer les temps d'accès à l'information.

Bibliographie

[1] K. Aberer, P. Cudré-Mauroux, A. Datta, Z. Despotovic, M. Hauswirth, M. Punceva, and R. Schmidt. P-grid : a self-organizing structured p2p system. *SIGMOD Rec.*, 32(3) :29–33, 2003.

[2] N. Andrade, F. Brasileiro, W. Cirne, and M. Mowbray. Discouraging free riding in a peer-to-peer cpu-sharing grid. In *Proc. 13th IEEE Symposium on High Performance Distributed Computing (HPDC'04)*, jun 2004.

[3] S.E. Arnold. The google legacy : How google's internet search is transforming application software., 2005.

[4] A. Awan, R.A. Ferreira, S. Jagannathan, and A. Grama. Unstructured peer-to-peer networks for sharing processor cycles. *Parallel Computing*, 32 :115–135, 2006.

[5] Pirate Bay. Pirate bay homepage, 2006. http ://www.piratebay.org/.

[6] R. Berlich, M. Hardt, M. Kunze, M. Atkinson, and D. Fergusson. Egee : building a pan-european grid training organisation. In *ACSW Frontiers '06 : Proceedings of the 2006 Australasian workshops on Grid computing and e-research*, pages 105–111, Darlinghurst, Australia, Australia, 2006. Australian Computer Society, Inc.

[7] B.H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7) :422–426, 1970.

[8] R. Bolze, F. Cappello, E. Caron, M. Daydé, F. Desprez, E. Jeannot, Y. Jégou, S. Lantéri, J. Leduc, N. Melab, G. Mornet, R. Namyst, P. Primet, B. Quetier, O.Richard, E.G. Talbi, and I.Touche. Grid'5000 : a large scale and highly reconfigurable experimental grid testbed. *International Journal of High Performance Computing Applications*, 20(4) :481–494, nov 2006.

[9] N. Bonnel, G. Ménier, and P.-F. Marteau. Information replication strategy in unstructured peer-to-peer networks using thematic agents. In *7th Int. Conf. on Intelligent Systems Design and Applications (ISDA'07)*, Rio de Janeiro, Brazil, October 2007. IEEE.

[10] N. Bonnel, G. Ménier, and P.-F. Marteau. Path query routing in unstructured peer-to-peer networks. In *13th European Conference on Parallel and Distributed Computing (Euro-par'07)*, Rennes, France, August 2007. Springer.

[11] N. Bonnel, G. Ménier, and P.-F. Marteau. Search in p2p triangular mesh by space filling trees. In *16th IEEE Int. Conf. on Networks (ICON'08)*, New Delhi, India, December 2008. IEEE.

[12] T. Bray, J. Paoli, and C.M. Sperberg-McQueen. Extensible Markup Language (XML) 1.0 - W3C recommendation 10-february-1998. Technical Report REC-xml-19980210, 1998.

[13] D. Chamberlin. Xquery : An xml query language. *IBM Syst. J.*, 41(4) :597–615, 2002.

[14] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable : a distributed storage system for structured data. In *USENIX'06 : Proceedings of the 7th conference on USENIX Symposium on Operating Systems Design and Implementation*, pages 205–218, Berkeley, CA, USA, 2006. USENIX Association.

[15] Y. Chawathe, S. Ratnasamy, L. Breslau, N. Lanham, and S. Shenker. Making gnutella-like p2p systems scalable, aug 2003.

[16] I. Clarke, O. Sandberg, B. Wiley, and T.W. Hong. Freenet : A distributed anonymous information storage and retrieval system. *Lecture Notes in Computer Science*, 2009, 2001.

[17] Clip2. The gnutella protocol specification v0.4, 2002.

[18] E. Cohen and S. Shenker. Replication strategies in unstructured peer-to-peer networks, aug 2002.

[19] L.P. Cox, C.D. Murray, and B.D. Noble. Pastiche : making backup cheap and easy. In *OSDI '02 : Proceedings of the 5th symposium on Operating systems design and implementation*, pages 285–298, New York, NY, USA, 2002. ACM.

[20] R. Cox, A. Muthitacharoen, and R. Morris. Serving dns using a peer-to-peer lookup service, mar 2002.

[21] A. Crainiceanu, P. Linga, J. Gehrke, and J. Shanmugasundaram. Querying peer-to-peer networks using p-trees. In *In WebDB*, pages 25–30, 2004.

[22] A. Crespo and H. Garcia-Molina. Routing indices for peer-to-peer systems, jul 2002.

[23] F. Dabek, M.F. Kaashoek, D. Karger, R. Morris, and I. 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, oct 2001.

[24] distributed.net. http ://www.distributed.net/.

[25] P. Druschel and A. Rowstron. PAST : A persistent and anonymous store. In *HotOS VIII*, may 2001.

[26] L. Fan, P. Cao, J. Almeida, and A. Broder. Summary cache : A scalable wide-area web cache sharing protocol. In *Proceedings of SIGCOMM'98*, Computer Sciences Department, Univ. of Wisconsin-Madison, feb 1998. Technical Report 1361.

[27] S.B. Handurukande, A.M. Kermarrec, F. Le Fessant, L. Massoulié, and S. Patarin. Peer sharing behaviour in the edonkey network, and implications for the design of server-less file sharing systems. *SIGOPS Oper. Syst. Rev.*, 40(4) :359–371, 2006.

[28] F. Harrell, Y. Hu, G. Wang, and H. Xia. Survey of locating & routing in peer-to-peer systems. 2001.

[29] H. Huilongand J.H. Hartman and T.N. Hurst. Data-centric routing in sensor networks using biased walk. *Sensor and Ad Hoc Communications and Networks,SECON '06.*, 1 :1–9, 2006.

[30] O. Heckmann and A. Bock. The edonkey 2000 protocol. Technical report, 2002.

[31] T. Holz, M. Steiner, F. Dahl, E.W. Biersack, and F. Freiling. Measurements and mitigation of peer-to-peer-based botnets : a case study on storm wor. In *LEET'08 : 1st USENIX Workshop on Large-Scale Exploits and Emergent Threats, April 15, 2008, San Francisco, USA*, apr 2008.

[32] Sun Microsystems Inc. Jxta v2.0 protocol specification, 2005.

[33] INSEE. Indice trimestriel du prix de vente industriel des micro-ordinateurs.

[34] Ipoque. The impact of p2p file sharing, voice over ip, skype, joost, instant messaging, one-click hosting and media streaming such as youtube on the internet, oct 2007.

[35] Itwire. Itwire homepage, 2007. http ://www.itwire.com/content/view/10211/53/1/0/.

[36] S. Iyer, A. Rowstron, and P. Druschel. Squirrel : A decentralized peer-to-peer web cache. In *12th ACM Symposium on Principles of Distributed Computing (PODC 2002)*, jul 2002.

[37] H.V. Jagadish, B.C. Ooi, K.L. Tan, Q. H. Vu, and R. Zhang. Speeding up search in peer-to-peer networks with a multi-way tree structure. In *SIGMOD '06 : Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, pages 1–12, New York, NY, USA, 2006. ACM.

[38] H.V. Jagadish, B.C. Ooi, and Q.H. Vu. Baton : A balanced tree structure for peer-to-peer networks. In *In VLDB*, pages 661–672, 2005.

[39] B. Jenkins. Bob jenkin's hash function, 1996. http ://burtle-burtle.net/bob/hash/doobs.html.

[40] S. Jiang, L. Guo, X. Zhang, and H. Wang. Lightflood : Minimizing redundant messages and maximizing scope of peer-to-peer search. *IEEE Transactions on Parallel and Distributed Systems*, 19(5) :601–614, 2008.

[41] R. Al King, A. Hameurlain, and F. Morvan. O-Chord : A Method for Locating Relational Data Sources in a P2P Environment. In *International Conference on Systems, Computing Sciences and Software Engineering (SCSS), University of Bridgeport, Connecticut, 03/12/07-12/12/07*, volume 1 of *Advances in Computer and Information Sciences and Engineering*, pages 416–421, http ://www.springerlink.com, aug 2008. Springer.

[42] G. Koloniari and E. Pitoura. Content-based routing of path queries in peer-to-peer systems. In *Proceedings of the EDBT'04 International Conference, Heraklion, Crete, Greece*, 2004.

[43] J. Kubiatowicz, D. Bindel, Y. Chen, P. Eaton, D. Geels, R. Gummadi, S. Rhea, H. Weatherspoon, W. Weimer, C. Wells, and B. Zhao. Oceanstore : An architecture for global-scale persistent storage. In *Proceedings of ACM ASPLOS*. ACM, nov 2000.

[44] A. Kumar, J. Xu, and E.W. Zegura. Efficient and scalable query routing for unstructured peer-to-peer networks. In *Proceedings of IEEE Infocom*, 2005.

[45] lastfm. lastfm homepage, 2008. http ://www.lastfm.fr/.

[46] LHC. Lhc homepage, 2008. http ://lhc.web.cern.ch/lhc/.

[47] J. Liang, R. Kumar, and K. Ross. The kazaa overlay : A measurement study. 2004.

[48] J. Liang, R. Kumar, and K. Ross. Understanding kazaa, 2004.

[49] B.T. Loo, R. Huebsch, I. Stoica, and J.M. Hellerstein. The case for a hybrid p2p search infrastructure, 2004.

[50] K. Lua, J. Crowcroft, M. Pias, R. Sharma, and S. Lim. A survey and comparison of peer-to-peer overlay network schemes. *Communications Surveys & Tutorials, IEEE*, pages 72–93, 2005.

[51] C. Lv, P. Cao, E. Cohen, K. Li, and S. Shenker. Search and replication in unstructured peer-to-peer networks, 2001.

[52] D. Malkhi, M. Naor, and D. Ratajczak. Viceroy : a scalable and dynamic emulation of the butterfly. In *PODC '02 : Proceedings of the twenty-first annual symposium on Principles of distributed computing*, pages 183–192, New York, NY, USA, 2002. ACM.

[53] P. Maymounkov and D. Mazieres. Kademlia : A peer-to-peer information system based on the xor metric, 2002.

[54] Napster. Napster homepage, 2001. http ://www.napster.com/.

[55] K. Nybom and J. Bjorkqvist. Designing tornado codes as hyper codes for improved error correcting performance. In *AICT-ICIW '06 : Proceedings of the Advanced Int'l Conference on Telecommunications and Int'l Conference on Internet and Web Applications and Services*, page 41, Washington, DC, USA, 2006. IEEE Computer Society.

[56] P. Pakzad and A. Shokrollahi. Design Principles for Raptor Codes. In *Proceedings of the IEEE Information Theory Workshop, 2006*, pages 165–169, 2006.

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

[58] J.A. Pouwelse, P. Garbacki, D.H.J. Epema, and H.J. Sips. The Bittorrent P2P file-sharing system : Measurements and analysis. In *4th Int'l Workshop on Peer-to-Peer Systems (IPTPS)*, feb 2005.

[59] D. Raggett, A. Le Hors, and I. Jacobs. Html 4.01 specification, 1999.

[60] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A scalable content addressable network. Technical Report TR-00-010, Berkeley, CA, 2000.

[61] S.C. Rhea and J. Kubiatowicz. Probabilistic location and routing. In *Proceedings of Infocom*, 2002.

[62] J. Robie, M.F. Fernández, S. Boag, D. Chamberlin, A. Berglund, M. Kay, and J. Siméon. XML path language (XPath) 2.0. W3C proposed reccommendation, W3C, nov 2006. http ://www.w3.org/TR/2006/PR-xpath20-20061121/.

[63] A. Rowstron and P. Druschel. Pastry : Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. *Lecture Notes in Computer Science*, 2218, 2001.

[64] A. Rowstron, A.M. Kermarrec, M. Castro, and P. Druschel. Scribe : The design of a large-scale event notification infrastructure. In Jon Crowcroft and Markus Hofmann, editors, *Networked Group Communication, Third International COST264 Workshop (NGC'2001)*, volume 2233 of *Lecture Notes in Computer Science*, pages 30–43, nov 2001.

[65] H. Sagan. *Space-Filling Curves*. 1994.

[66] S. Saroiu, K. Gummadi, and S. Gribble. Measuring and analyzing the characteristics of napster and gnutella hosts, 2003.

[67] Data Dissemination Shelley. Bayeux : An architecture for scalable and fault-tolerant wide-area.

[68] E. Sit, A. Haeberlen, F. Dabek, B. Chun, H. Weatherspoon, R. Morris, M. Kaashoek, and J. Kubiatowicz. Proactive replication for data durability, 2006.

[69] M. Steiner, T. En-Najjary, and E.W. Biersack. A global view of KAD. In *IMC 2007, ACM SIGCOMM Internet Measurement Conference, October 23-26, 2007, San Diego, USA*, oct 2007.

[70] J. Stillwell. *Geometry of surfaces*. Springer-Verlag, 1992.

[71] I. Stoica, R. Morris, D. Karger, M.F. Kaashoek, and H. Balakrishnan. Chord : A scalable peer-to-peer lookup service for internet applications. In *SIGCOMM '01 : Proceedings of the 2001 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 149–160, New York, NY, USA, 2001. ACM.

[72] W.W. Terpstra, J. Kangasharju, C. Leng, and A.P. Buchmann. Bubblestorm : resilient, probabilistic, and exhaustive peer-to-peer search. In *SIGCOMM '07 : Proceedings of the 2007 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 49–60, New York, NY, USA, 2007. ACM.

[73] B. Yang and H. Garcia-Molina. Improving search in peer-to-peer systems. In *Proceedings of the 22nd International Conference on Distributed Computing Systems (ICDCS)*, 2002.

[74] B. Yang and H. Garcia-Molina. Designing a super-peer network. *icde*, 00 :49, 2003.

[75] 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, apr 2001.

[76] F. Zhou, L. Zhuang, B. Zhao, L. Huang, A. Joseph, and J. Kubiatowicz. Approximate object location and spam filtering on peer-to-peer systems, jun 2003.

Résumé

Depuis quelques décennies, la quantité d'information numérique produite ne cesse de croître exponentiellement, ce qui soulève des difficultés de plus en plus critiques en terme de stockage, d'accessibilité et de disponibilité de cette information. Les architectures logicielles et matérielles construites autour du modèle pair-à-pair (P2P) semblent répondre globalement aux exigences liées au stockage de l'information mais montrent leurs limites en ce qui concerne les exigences d'accessibilité et de disponibilité de l'information.

Nous présentons dans cette thèse différents apports concernant les architectures P2P pour la gestion de grands volumes d'information. Les stratégies algorithmiques que nous proposons exploitent des topologies virtuelles dédiées sur lesquelles nous développons des protocoles de maintenance et de gestion du réseau efficaces. En particulier, pour assurer le passage à grande échelle, nous proposons des solutions pour lesquelles les coûts des opérations de maintenance et de gestion des topologies virtuelles sont constants en moyenne pour chaque noeud du réseau, et ceci, quelle que soit la taille du réseau.

Nous analysons les principaux paradigmes de la répartition d'information sur un réseau P2P, en considérant successivement, le problème de l'accès à de l'information typée (semi-structurée) et le cas général qui dissocie entièrement la nature des requêtes du placement de l'information. Nous proposons une méthode d'aiguillage de requêtes portant sur la structure et le contenu de documents semi-structurés ainsi qu'une technique plus générale dans le cas le plus défavorable où aucune connaissance n'est disponible *a priori* sur la nature des informations stockées ou sur la nature des requêtes.

Dans l'optique de la gestion d'une qualité de service (qui s'exprime en terme de rapidité et de fiabilité), nous nous intéressons également au problème de la disponibilité pérenne de l'information sous l'angle de la réplication des données stockées dans le réseau. Nous proposons une approche originale exploitant une mesure locale de densité de réplicas estimée sur une topologie virtuelle dédiée.

Abstract

In the last few years, the amount of digital information produced has exponentially increased. This raises problems regarding the storage, the access and the availability of this data. Software and hardware architectures based on the peer-to-peer (P2P) paradigm seem to satisfy the needs of data storage but cannot handle efficiently both data accessibility and availability.

We present in this thesis various contributions on P2P architectures for managing large volumes of information. We propose various strategies that operate on dedicated virtual topologies that can be maintained at low cost. More precisely, these topologies scale well because the cost for node arrival and node departure is on average constant, whatever the size of the network.

We analyze the main paradigms of information sharing on a P2P network, considering successively the problem of access to typed information (semi-structured) and the general case that completely separates the nature of the queries and data location. We propose a routing strategy using structure and content of semi-structured information. We also propose startegies that efficiently explore the network when there is no assumption on the nature of data or queries.

In order to manage a quality of service (which is expressed in terms of speed and reliability), we also investigate the problem of information availability, more precisely we replicate data stored in the network. We propose a novel approach exploiting an estimation of local density of data replica.